Leo Prikler, BSc

Design and Implementation of a Method-based Java Card to Native Code Compiler

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Advisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Dipl.-Ing. Stefan Lemsitzer (NXP Semiconductors Austria GmbH und Co KG)

Institute of Technical Informatics

Graz, May 2021

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Abstract

Vendors of modern-day Java Card smartcards provide "accelerators" – methods, which are implemented in native code rather than Java Card bytecode for reasons of performance. The set of methods implemented in such a way naturally includes (parts of) the Java Card API, but is not necessarily limited by it.

However, the usage of accelerators defined outside of any standard may come at the cost of interoperability. Any such accelerator may only be implemented by a particular vendor and calling conventions between two given sets of accelerators can differ drastically. Applet developers can be locked in to (a set of) vendors, may need to keep around several versions of their code – which might exhibit different behaviour from each other as time marches on – and risk a great performance penalty when not making use of them.

The development of accelerators in turn is a time- and labour-intensive task and previous research has shown that Java Card applets can still gain performance through compilation to native code despite them. The benefits of compilation are clear: not only is interoperability kept on a source and bytecode level, it can also be done automatically. However, native applets are also significantly larger than those implemented in bytecode, even if they still fit onto a smartcard. In contrast to an existing applet-based approach, this thesis shows a method-based compilation approach based on an interface to the Java Card operating system that makes it possible to have multiple applets and even libraries with native code attached to them. This approach has less code-size overhead than the previous one while still achieving decent speed-up and can further be tuned to favour time or space in this trade-off.

Kurzfassung

Hersteller moderner Java Card Smartcards bieten “Beschleuniger” (“accelerators”) an – Methoden, die aus Performanzgründen in nativem Code anstelle von Java Card Bytecode implementiert sind. Diese Methoden umfassen natürlich Teile der Java Card API, sind aber nicht notwendigerweise durch sie beschränkt.

Werden Beschleuniger genutzt, die nicht in irgendeinen Standard fallen, so passiert dies auf Kosten der Interoperabilität. So kann es zum Beispiel sein, dass nur ein Hersteller einen bestimmten Beschleuniger bietet, aber auch die Art, wie diese Beschleuniger zu verwenden sind, kann zwischen zwei verschiedenen Herstellern sehr unterschiedlich sein. Applet-Entwickler können somit auf einen oder mehrere Hersteller beschränkt werden und müssen gegebenenfalls mehrere Versionen ihres Applets entwickeln – die sich mit der Zeit in ihrem Verhalten unterscheiden können – oder sie riskieren große Leistungseinbüßungen.

Was die Entwicklung von Beschleunigern betrifft, so ist diese auch zeit- und arbeitsaufwändig, und bisherige Forschung zeigt, dass Java Card Applets trotz ihnen durch Kompilierung weiterhin beschleunigt werden können. Die Vorteile einer Kompilierung sind klar: Nicht nur ist die Interoperabilität auf Quellcode- und Bytecode-Ebene weiterhin erhalten, sie kann zudem automatisch erfolgen. Allerdings sind Applets, die in nativen Code kompiliert wurden signifikant größer als die Bytecode-Variante – selbst wenn sie noch auf die Smartcard passen. Als Kontrast zu einer bereits existierenden Applet-basierten Kompilierung zeigen wir einen methodenbasierten Ansatz zusammen mit einer Schnittstelle, die es ermöglicht, mehrere Applets und sogar Bibliotheken mit nativem Code zu versehen, der weiter verfeinert werden kann, um Geschwindigkeit bzw. Speicherbedarf zu optimieren. Wir stellen fest, dass unser Ansatz zu einem geringeren Overhead und dennoch passabler Beschleunigung eines Applets führt.

Acknowledgement

I would like to thank everyone who made this thesis possible, including in no particular order

- professor **Christian Steger**, my supervisor,
- everyone at NXP Semiconductors Austria GmbH und Co KG who collaborated with me during the implementation, with special thanks going to **Stefan Lemsitzer** for leading the project and **Tobias Rauter** for implementing the runtime environment,
- and my family for the continued support they've given me for the long time it took me to write this thesis.

Had any of them not aided me in the way that they did, this thesis would probably not be able to exist.

Contents

Abstract iii

Kurzfassung iv

Acknowledgement v

1. Introduction 1

2. Background and Related Work 3
2.1. Notation and language . . . 3
2.2. Language Interaction . . . 6
2.3. JNI and variations . . . 8
2.4. Specialised Java Environments . . . 10
2.4.1. Java Card . . . 10
2.4.2. Other resource-constrained devices . . . 12
2.5. Bytecode to native code compilation . . . 13
2.6. Other smartcard platforms . . . 17

3. Design 19
3.1. System Architecture . . . 19
3.2. Interfaces . . . 21
3.3. Build Process . . . 24
3.4. Compiler architecture . . . 27

4. Implementation 29
4.1. Analysis . . . 29
4.1.1. Control Flow . . . 31
4.1.2. Data Flow . . . 34
4.1.3. Symbols . . . 35
4.2. Code Generation . . . 36
4.2.1. Translating Java Card bytecode to C . . . 36
4.2.2. Exception handling . . . 42
4.3. Optimizations . . . 43
4.3.1. Caches . . . 43
4.3.2. Function outlining . . . 48
4.3.3. Function call re-routing . . . 49
4.3.4. Mapping arrays into C runtime . . . 50
4.3.5. Contracting the DDG . . . 51

5. Evaluation **53**
5.1. Benefits and drawbacks of native code . . . 53
5.2. Benefits and drawbacks of optimizations . . . 54
5.2.1. Caches . . . 54
5.2.2. Function outlining . . . 55
5.2.3. Function call re-routing . . . 56
5.2.4. Mapping arrays into C runtime . . . 56
5.3. Overhead . . . 57
5.4. Experimental Results . . . 58
5.5. Limitations . . . 65
5.5.1. Compatibility . . . 65
5.5.2. Lacking Features . . . 66
5.5.3. Known Bugs . . . 66
5.6. Security Considerations . . . 67
5.6.1. Type confusion . . . 67
5.6.2. Object reference manifacturing . . . 67
5.6.3. Arbitrary Code Execution . . . 68
5.6.4. Exception Handling . . . 69
5.6.5. Unintentional data leakage . . . 70
5.6.6. Introducing side channels . . . 70

6. Conclusion **71**

References **72**

A. Equivalence Checking in the DDG **76**

B. Lost in translation: function purity **77**

C. Exception range semantics **78**

List of Tables

3.1. JVM bytecode macros. 21
3.2. Macros used in the main skeleton. 22

4.1. Encoding of expressions. 38
4.2. Stubs for outlined functions. 49

5.1. Test applet size comparison legend 59
5.2. Test applet size comparison 59
5.3. Test applet size comparison cont. 60
5.4. Test applet size comparison cont. 60
5.5. Test applet size comparison cont. 61
5.6. Test applet size summary 61
5.7. RWA1 performance 62
5.8. RWA2 performance 63
5.9. RWA3 performance 63

List of Figures

2.1. Performance of Grimmer's native function interface. 10
2.2. Architecture of the Fiji VM 13
2.3. Files generated by Muller et al. 14
2.4. Compilation process used by Ellul and Martinez. 16

3.1. Architecture of a typical Java Card system. 19
3.2. Architecture of the extended system. 20
3.3. Exception stack. 23
3.4. Build flow for a classic Java Card applet. 24
3.5. Extended build flow of native applets. 26
3.6. Dependencies within the generated code. 27

4.1. CST of a simple assignment. 30
4.2. Data flow graph of the same example. 30
4.3. Creation of a data flow graph. 35
4.4. Structure of a cached field read in the DDG. 46
4.5. Structure of a cached write in the DDG. 46
4.6. Structure of a cached array read in the DDG. 47
4.7. Structure of a cached array read in the DDG. 47

5.1. Size results 64
5.2. Performance results 64

List of Algorithms

1. Procedure branchtarget(C, B, n) 32
2. Generation of the control flow graph. 33
3. Code generation for simple instructions. 38
4. Procedure makelabel(n, o) 40
5. Code generation for branch instructions. 40
6. Code generation for switch instructions. 41

List of Acronyms

ABI application binary interface.

AES Advanced Encryption Standard.

AOT ahead of time.

APDU application protocol data unit.

API application programming interface.

CAP converted applet.

CFG control flow graph.

CLDC Connected Limited Device Configuration.

CPU central processing unit.

CRE C runtime environment.

CST concrete syntax tree.

DDG data flow graph.

FFI foreign function interface.

FPGA field-programmable gate array.

GCC GNU Compiler Collection/GNU C compiler.

GNFI Graal Native Function Interface.

IR intermediate representation.

ISO International Organization for Standardization.

JAR java archive.

JCVM Java Card Virtual Machine.

JIT just in time.

JNA Java Native Access.

JNI Java Native Interface.

JRE Java runtime environment.

JVM Java Virtual Machine.

OS operating system.

VM virtual machine.

WSN wireless sensor network.

1. Introduction

In ye olden days, one was typically forced to write software directly for the hardware one had – a change in hardware, specifically the processor a program was running on – either meant one had to recompile the program all over again or even worse was forced to port the software over. This style of programming is still in use today where it is (assumed to be) required for the sake of performance. On the other hand, software developers have collectively decided that they prefer languages with strong guarantees like Java over those that permit arbitrary bit fiddling like C or those in which a standard-compliant compiler may give you a wrong answer on the question of "Is this a valid program?" like C++. Particularly interesting for the sake of interoperability is the promise of a virtual machine (VM) on which the generated code runs. Unlike C or C++ compilers, which often generate machine code[1], Java and its variants compile to bytecode, which is then run on top of a VM. Porting one's entire codebase to Java may however hurt performance, specifically in computation-intensive tasks. To mitigate these issues, Java allows calling "native" machine code from a Java context and vice versa.

In the same way that Java promises type safety, interoperability, etc. to Java programmers writing desktop or Android applications, Java Card makes those promises towards smartcard application developers. Like Java, it permits the use of native code, but in a more limited context – this room for optimization is exploited by smartcard vendors to provide fast implementations of certain algorithms for particularly interesting use cases. Historically, Java Card failed to live up to the expectations w.r.t. interoperability, as its filesystem and cryptographic application programming interfaces (APIs) were not defined concretely enough to fit in various existing implementations, as pointed out by Baentsch [3]. While that particular issue may have been resolved by now, accelerators pose a similar problem, in that they can force developers to use proprietary APIs, because alternative open ones – or even the ones an applet developer could write for themself – will perform worse.

As the use of smartcards is so varied, one can hardly expect an API to ever satisfy all potential needs. Hence it is also not possible to standardize such an API under the Java Card label, but even vendors, who do develop accelerators beyond the standard have to struggle with the possibility of there being some need outside of what they accounted for.

One solution to the problems raised by accelerators is to use compilation instead. Unlike accelerators, which have to be written and carefully optimized before an applet developer uses them and must also be accounted for by that developer, compilation specifically optimizes the functions they wrote with the intent of being compatible with

[1] This is no hard requirement. A compiler could just as well target a virtual machine, like the WebAssembly virtual machine.

all platforms. Much of the performance gains that would normally be achieved by specifically targetting those platforms, can be harvested by the compiler while interoperability at a source code and even bytecode level is retained. Only the result of the complation process and parts of the toolchain to derive them are platform-specific.

This thesis describes such a compilation approach, in particular one in which each method can be compiled separately much in the same way that vendors would often choose to make certain methods faster through acceleration. It is structured as follows: Chapter 2 lays out background information and shortly discuss related work. Chapter 3 lays out the general design of the compilation approach and how it integrates into the Java Card workflow. Chapter 4 describes the compiler itself and various optimizations. Chapter 5 gives a theoretical and practical evaluation of the compiler and also discusses limitations and security considerations. The conclusion can be found in Chapter 6.

2. Background and Related Work

Before diving head-first into the design and implementation of the compiler, it is necessary (or at least helpful) to form a common basis of understanding. Section 2.1 explains the terms and notations that will be used throughout this thesis. Section 2.2 gives a rough summary of how components written in different programming languages are usually stitched together into one cohesive program. Section 2.3 explains the Java Native Interface and some variations of it. Section 2.4 gives an overview of Java environments that users of the desktop version might not be aware of, particularly with regards to resource-constrained devices. Section 2.5 lists previous compilation approaches targetting some Java platform. Finally, Section 2.6 gives an overview of the smartcard systems that fall outside the scope of this thesis but could still be considered for related research.

2.1. Notation and language

Chapters and sections in this thesis are often structured in a fashion that puts context and other helpful metaphors at their start and more technical details towards their end. With that comes a change in the way things are phrased. While in the first parts, conditionals may be used freely to draw up hypothetical scenarios, many of which are known to occur, or likewise freely omitted them as there is no real change in semantics, in the latter, conditionals such as potentially, possibly, etc. denote occasions in which at that given step in some process the outcome is unsure and the lack thereof implies certainty[1]. This holds especially if qualifiers of potential and lack thereof are often intermixed within or across paragraphs.

This thesis deals in large part with the construction of a compiler and as such uses language commonly associated with compilers and their construction, specifically:

code: a set of instructions forming a program.

- **machine code:** code that can be executed as-is (given some data) on physical machines.
  - **native code:** (from the point of a virtual machine) code for the machine that runs the virtual machine.
- **bytecode:** code that can be executed as-is (given some data) on non-physical (i.e. virtual) machines[2].
- **source code:** code written by a human in a way that can (hopefully) be reasoned about.

[1] at least to the extent that we can believe to be "certain" about things

[2] other literature may also call this p-code or "portable code"

intermediate representation (IR): a representation of code used by a compiler.

machine: a device that executes code.

virtual machine (VM): such a machine that is itself implemented as code for another one.

compiler: a tool (or in rare cases a human) that transforms code from one form to another.

interpreter: a tool that executes source code directly.

As one can see, these definitions are somewhat circular (as all fundamental definitions happen to be). The difference between virtual machines and interpreters is a rather nuanced one, though they are sometimes also conflated – see some of resources cited in this very thesis, which refer to various virtual machines as "bytecode interpreters". By the definitions above, an interpreter can – as it executes code – be considered a machine. Since they are also implemented as code for other machines, they can be considered a strict subset of virtual machines – strict, because virtual machines also include machines that don't execute source code directly.

However, which virtual machines are interpreters and which are not, depends on what one views as source code. As laid out above, source code is defined as code written by humans. But nothing is stopping humans from writing or reasoning about any other form of code as defined above, it is merely easier to express one's thoughts in structured text rather than sequences of bytes[3]. Some also use a different definition of source code, defining it as the input to a compiler. Using a more exact definition of compilers as tools that transform code from a source language into a target language, it makes sense to declare code written in the source language as source code and code written in the target language as target code. From this point of view, other systems that execute the "source code", such as virtual machines, may be considered interpreters.

Following this train of thought to its logical extreme however, every machine could be considered an interpreter, as instead of executing code directly, it could very well compile the code for some other machine – which in turn could compile the code for another, and so on, and so forth ad infinitum. Hence all machines are interpreters, and at the same time all interpreters are also machines, meaning there is no distinction between interpreters and machines. And we haven't even talked about actual hardware that actually translate their own instruction set to some other instruction set at runtime – those do exist. But even if machines and interpreters are in some sense the same philosophically and the lines might get blurred in practice as well, there is a point at which a line has to be drawn, because clearly this does not work for all potential angles. Therefore the word "interpreter" will in this thesis not be used to describe machines that execute compiled code, including bytecode.

In most existing software systems, particularly larger ones, source code will at some point be split into multiple files, each of which can be compiled separately (when a

[3]By these definitions, an assembler would be considered a compiler as well.

compiler exists) or interpreted "on their own", under the assumption, that dependencies between those files are somehow resolved. Several styles of such splits can be distinguished, including

- **method-based** or **function-based** splits, in which a file contains at least one method (or function)
- **class-based** splits, in which a file contains at least one class (i.e. a data structure with associated methods), and
- **module-based** splits, in which a file contains one logically connected module (a set of classes and functions).

Some programming languages, like Python or Scheme encourage module-based splitting, although in both examples it would still technically be possible to construct modules for single classes and single functions. Java is explicitly class-based, as no function may exist outside of a class.

A compiler will of course always dissect its input, e.g. a module becomes a set of classes and functions, a class becomes a data structure and a set of functions, and functions become sets of basic blocks. However, it makes sense to consider the smallest part that a compiler will typically translate on its own, even if its input may generally be larger than that. Here it is no longer a matter of programming languages, but really the compiler itself. For instance, in interpreted languages one may decide to compile code blocks or expressions, which offers even finer granularity than the splits discussed above, a compiler for Java may use a method-based approach and translate methods into another language while leaving others alone, and so on, and so forth.

Within this thesis, the word *macro* might sometimes pop up. With the existence of almost as many definitions of this term as there are programming languages featuring them[4], one might qualify it as "not really well-defined", and indeed, even within this thesis multiple definitions are used as-needed, not aiding the situation in the slightest. The simplest definition of a macro we can give is "code that expands to other code, usually at compile time". While this in-place expansion usually explains why macros are used instead of functions or global variables, it is also not very helpful. Within the context of C, Stallman and Zachary [30] distinguish between object-like macros, which to the reader of the code appear as if they were variables and function-like macros, which to the reader of the code appear as function calls. Most of the time, when the term macro is used within this thesis, it refers to a function-like macro that expands to an expression, in which each of its arguments is used at most once. While this looks very limiting, considering what macros *can* do in C, it suffices in many cases[5]. In those cases in which it does not suffice, the expansion shall be explicitly stated, so that it will be clear from the context that it is not simply an expression.

[4] the fact that C has two kinds of macros doesn't really help here either

[5] Considering that Stallman and Zachary also mention duplication of side effects as a pitfall and GCC provides "compound statements" to build "safe" macros, not expanding arguments twice is already good practise on its own.

Some terms or notations may be borrowed from fields not necessarily linked to compiler construction, including:

Graph Theory: Graphs just so happen to be the most important data structure in this compilation process and thus language will be borrowed from this field. Expect nodes and edges, but also cuts, specifically the cut $\delta_G^-(n)$, which are the ingoing edges of n in graph G. However, the language used within this thesis may also conflict some of the terms. In specific instances, where confusion is otherwise likely, an ad-hoc definition should clear those up.

Design Patterns: Since this is not a thesis on design patterns, they will not be mentioned explicitly unless particularly noteworthy – such as the excessive use of visitors – but as patterns may be found anywhere, surely you may also find them here.

2.2. Language Interaction

Many programs are not written in only one, but many languages, the reasons for doing so often being part of the design. For instance, one could write the core of a text editor in C, while writing almost everything else – including configuration – in some dialect of Lisp. Programs written primarily in C may from time to time include (inline) assembly for reasons of performance, or to use special instructions that a C compiler usually does not emit. Command line applications, which on their own may have dozens upon dozens of options may be wrapped in shell scripts, whose options are easier to understand, and so on and so on.

Whenever a mix of programming languages is used within a given software project, some interaction of parts written in some language with parts written in another is implied by principle of association. Or rather, they wouldn't be part of the same program if they did not somehow interact. But for those parts to interact one first requires a way of facilitating said interaction. Indeed, this even holds for programs written in just one language.

Consider function calls, particularly within the C programming language. A library written (primarily) in C, compiled with some compiler and some flags, should be usable in another library or another program possibly compiled with different compilers or flags, assuming that both compilers at least somewhat adhere to some specification, which is usually referred to as the application binary interface (ABI). A commonly known part of such an ABI are *calling conventions*, which state how functions are to be called in a given context. For instance, the `cdecl` convention is to push arguments in reverse order onto the stack and then use the `call` instruction, which itself pushes program counter and stack pointer before jumping to the destination. In contrast Java has a calling convention in which the arguments are pushed onto the stack in "normal" order, followed by an `invoke` bytecode, which is analogous to `call`, except that it carries some information about the method that it tries to invoke.

Along with calling conventions, *data representation* also varies across platforms. This already starts at the endianness, which affects multi-byte integral data types, such as

`int` and `short`. While the Java Virtual Machine is big-endian, many non-virtual machines are little-endian. The packing of structures with multiple fields may cause further problems, and so on and so forth.

A program may be split into multiple parts – across different languages even – but can still be compiled and linked into a single executable, a long as the target platform for each of those parts is the same or can be made the same. Think back to the earlier example of mixing C with Assembly. It turns out that as long as compiler warnings are ignored (the specific one in this case being `-Wimplicit-function-declarations`), one can use arbitrary functions declared anywhere – the linker will eventually resolve those references and construct an executable that makes the right calls. Depending on the inner workings of the assembler, the same may hold for it as well. Of course, this is not the "correct" way of doing that. In C, we write *header files*, for example *stdio.h*, which declare that a function – e.g. `printf` – exists, but provide no implementation. In lack of such a header, one can also write an `extern` function declaration into a header or into the source code directly. A similar directive exists in Assembly as well. Once the compiler or assembler is made aware of the functions to call, everything should behave as one would reasonably expect.

Until now, everything works fine, because symbols can be resolved at link time and can thus things work. The worst thing that can happen, is dynamic linkage, i.e. the binding of a symbol to an address at runtime, which itself is not a big deal, because lookup tables are known data structures. But now comes a program written in a totally bizarre language that is certainly not C, which is interpreted by or compiled for some other target platform. Not only are symbols from that language nonexistent, not visible, or sometimes even just obscured – if one were to somehow find them and then try to call such functions from C or Assembly, the result would be a failure. These functions are foreign in the same way as some human languages appear foreign to the speakers of others. Even when assuming a common alphabet, trying to pronounce foreign words without knowledge of the language they come from will end up weird at best and disastrous at worst, with a tendency towards disaster whenver machines are involved.

The *foreign function interface (FFI)* is a part of any language runtime that expects to coexist with another and has several responsibilities. The biggest is of course making functions from one **safely** callable by the other, but in order to facilitate this, it needs to ensure

- a common calling convention or conversion between uncommon ones,
- implicit conversion of data types to their respective counterparts, or at least a method of explicit transformation[6],

[6] Oftentimes one would want implicit conversion, especially with numeric data types. However, this is not always straightforward. Consider for instance a language such as C, which only has fixed-size numeric data types, and then take a language such as Scheme, which has arbitrarily large numerical types as well as exact fractions. Converting from the C representation to the Scheme one is no problem, but going back without additional information is hard, which is why this conversion to e.g. `int` will always need to be made an explicit one at some point in user-written code and can not "simply" be abstracted.

- common memory management[7],
- and a bidirectional symbol lookup[8] through which parties learn of interfaces implemented by the other

among other things. In the case of interpreted languages, there may also be a function (usually called `eval`), which takes code in that language as a string and evaluates it. In fact, `eval` alone can in conjunction with string manipulation functions and other ways of marshalling data be the entire FFI. An example for that would be the `system` function, which thus provides an FFI to the underlying system's shell interpreter, although most people who only use it to spawn a process would rarely if ever refer to it as such.[9]

The tackling of these problems has given rise to a few patterns and a widely used library (`libffi`) on which many such interfaces are based. One such pattern shall be given by example. C++, Rust, and Vala are – while they still compile to the same machine code as C – in some sense foreign to C, as each have their own naming schemes, with C++ being an especially notorious example with its compiler-dependant name mangling. However, all of these languages provide some way of giving functions a readable name in C context, multiple even in the case of Vala. The pattern here is a language-level interface towards naming conventions, allowing direct (or sometimes indirect, as both are possible in Vala) control over a generated function's name at a lower level. Another pattern concerns the way data is laid out and managed and will be be discussed at a later point within this thesis, when it is used for optimization purposes.

With a basic understanding of what a foreign function interfaces are and why they exist, we can now proceed further towards the actual core of this thesis.

2.3. JNI and variations

One foreign function interface and the default for Java is the so-called Java Native Interface (JNI), which – along with Java itself – has been subject to analysis, especially with respect to performance, pretty much since the dawn of Java. As running a virtual machine always results in some overhead, people have sought to eliminate that by the use of native code, and the more native the code, the better performance should get – at least in theory. In praxis, performance comes at the sake of compatibility and code size, which in turn causes people to still rely on the benefits the Java Virtual Machine (JVM) provides in *some* sections and thus raises the problem of efficiency and usability of the FFI.

[7] At the very least both parties must agree which data not to free, which on its own already shatters (wrongly held) ideas about ownership, and sometimes also when to actually free data, if there is potentially high memory usage or critical timings.

[8] This mechanism can in theory also be implemented as one lookup methods per direction.

[9] In a similar manner, interfaces to some language from C are sometimes referred to as the "C API". While this term is technically correct in most situations and useful in manuals to distinguish between the different directions of an interface, on a more philosophical level there really is no difference between the two; there only appears to be a difference due to different viewpoints.

Kurzyniec and Sunderam [15] provided benchmarks for the JNI and highlight various kinds of performance overhead, specifically

- the overhead of calling native methods from Java,
- the overhead of calling Java methods from native methods,
- the overhead of field accesses in native methods,
- the overhead of array and string accesses in native methods,
- and the overhead of exception handling.

Kurzyniec and Sunderam found that all of the above perform significantly worse than their Java counterparts. While their exact numbers may differ from the ones found in this thesis, these results are nontheless in accordance with each other, only excluding strings as a source of overhead, as they do not exist in Java Card-based technologies.

As far as usability is concerned, the JNI does not fare great either, requiring much boilerplate code to call an already existing native method. This is rather unfortunate in situations where native code is not (only) used for performance reasons. If one wanted to optimize a routine that would otherwise be implemented in Java using native code, one is still writing that code (although the code itself may now be different) *to be used* in Java, whereas one couldn't care less about Java if the same algorithm was implemented for the use in a C(-like) environment. Writing wrappers for such code is boring and repetitive, especially if one does it for a large library, e.g. all of POSIX, and as with all copy-paste style coding, the chance of introducing an error somewhere rises with each copy and each derivation. This is where Java Native Access (JNA) [14] comes to the rescue. Instead of using the JNI directly, with JNA a native library is wrapped in an interface with individual function wrappers being automatically generated using `libffi`. These functions can then be called as if they were functions of any other object.

Grimmer et al. [12] implemented their own native interface (dubbed Graal Native Function Interface (GNFI)) in the Graal VM, which is used by OpenJDK. The GNFI is designed around the concept of handles – a programmer first creates a library handle and then uses that handle to explicitly instantiate a function handle, similar to how a C programmer would use `dlopen` and `dlsym` to get a function pointer. A programmer also has to explicitly pack arguments into an object array and unpack the return value into whatever type was expected. The function handle itself is implemented by synthesizing a Graal intermediate representation (IR) graph, which is turned into native code using facilities of the Graal VM. As a result, all parts of a native function call can at some point be optimized by the VMs they were meant for. This reportedly makes the GNFI significantly faster and more flexible as the JNI, which itself is – according to the same report – also faster than using JNA (whose only benefit therefore is the flexibility). One of their comparisons is shown in Figure 2.1. As can be seen, throughput is significantly higher with their native interface compared to the other ones.

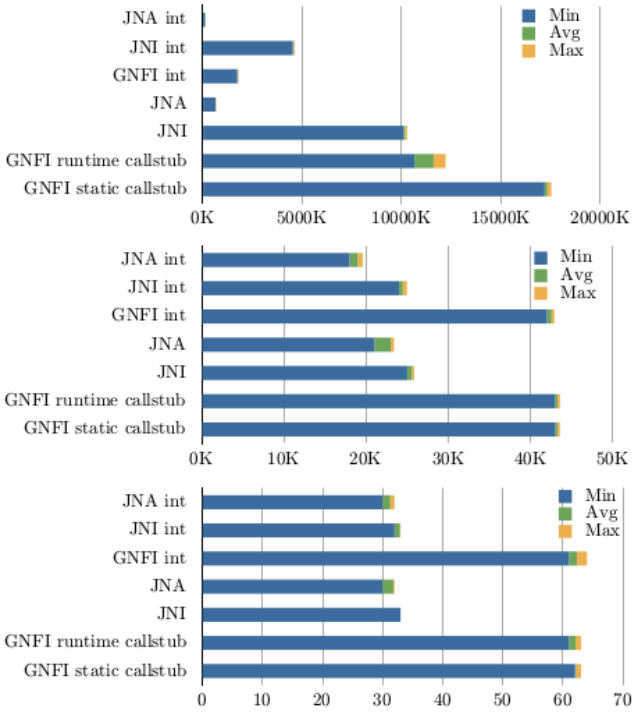

Figure 2.1.: Number of matrix multiplications, that can be done within 10 seconds for matrices of size 10^2, 100^2 and 1000^2 through various Java FFIs, taken from [12].

2.4. Specialised Java Environments

While the standard edition of Java satisfies the needs of most desktop users, there also exist other platforms with special needs that some edition of Java caters to. Of particular interest for this thesis is of course Java Card, as it serves as technological foundation, but there are also resource-constrained devices that differ from smartcards, which are worth taking a short look at.

2.4.1. Java Card

Java Card is a specialised Java environment targetting smartcards. It differs from desktop Java in a few key areas, e.g. the instruction set, the lack of a 32-bit integer (or to be more accurate, support for 32-bit integers is optional), the lack of strings, the lack of threads, the lack of wrapper types such as `Integer`, and so on, and so forth. Oracle [22] claims Java Card – specifically the Java Card Virtual Machine (JCVM) – to be a subset of the JVM, and while this is true to some extent, it is at the same time false to another. Looking only at the Java code and ignoring that it's using somewhat atypical packages, one could conclude that Java Card is indeed a subset of Java. This kind of subset – a behavioral subset – is what they meant with that claim. Were one to

get to the bones and consider e.g. specialized bytecodes, it would be more appropriate to call it a distinct version of Java altogether. Nitpicks aside, let us look at some more substantial differences.

A big architectural difference between Java and Java Card is the difference between their object files. Java packs single classes into class files, whereas Java Card packs entire packages into converted applet (CAP) files.[10] CAP files again are essentially java archive (JAR) files structured in a special way. Rather than packing folders and files into a container as one would normally do, the segments of a typical Java class file are packed into components, with additional components added to tie them together. For instance the `ConstantPoolComponent` contains all the constant pools, the `MethodComponent` the methods, and so on, with the `DescriptorComponent` "describing" classes, methods and fields, i.e. linking an index to their respective location elsewhere as well as providing access and type information, the `AppletComponent` listing applets, and so on, and so forth.

Another difference is the limited support for native methods, which are only allowed for packages "located in the card mask". The term *card mask* is never again used in any Java Card related manual, but since it's explained that "masking" refers to the embedding of the virtual machine, runtime environment and applets into the read-only memory of a smartcard, it becomes somewhat clear that this mechanism is only meant to be used by vendor-provided packages. This is where the idea of accelerators is born. A vendor will first try to make as many packages related to the Java Card framework native, since that will make applets run much faster on their cards than the competition's. Once that is done, however, there are still computationally intensive algorithms not covered, e.g. crypto algorithms, which an applet developer would otherwise implement in Java. As one might imagine, running such algorithms in a virtual machine has a large runtime impact, so vendors seek to provide their own methods, which are significantly faster due to the fact that they run on the hardware itself. Reliance on a single set of such methods leads to vendor lock-in, whereas supporting multiple ones requires some sort of wrapper if the applet developer does not want to rewrite all the code using any of them. In either case some of the benefits provided by the Java Card platform are done away with. In the former, applet code will not be portable, in the latter it is portable at the expense of the maintenance of platform-specific wrapper code.[11]

Of course, there are not only differences with regards to the virtual machine, but also with regards to the runtime and framework. While the typical Java program is written around one class containing a `public static void main(String[] args)`, Java Card is centered around applets. These extend `javacard.framework.Applet`, implementing methods such as `install` and `process`. The former is used in lieu of a constructor to allow complex instantiation of an applet. The latter could be seen as a typical iteration of a read-eval-print loop, in which an applet takes a command from the environment, processes it and writes the result back. This loop – specifically the "reading" and "print-

[10] Contrary to what this name suggests, a CAP file can have any number of applets, as long as "any number" fits into an unsigned byte. A more accurate name would therefore be "converted and compressed package" or CCP.

[11] Note that this is in no meaningful way different from the portability of C code.

ing" aspects of it – is performed by the smartcards on one or multiple so-called *logical channels*. An applet may be bound to (or in the language of Java Card *selected* on) any such channel, even multiple ones if it implements the `Multiselectable` interface.

Within the runtime environment of Java Card the Applet Firewall prevents cross-package[12] object accesses, except for the case in which applets are explicitly shared between two packages (or "contexts") and global arrays. If that sounds awfully similar to the way operating systems separate processes, but at the same time provide mechanisms for inter-process communication, you are not alone.

Not quite satisfied with the division of install and process, many real world applet developers actually distinguish three phases, `install` in which the applet (plus dependencies are installed), `perso` in which initial settings are set, and `process` in which the actual work is performed. These phases are not to be confused with the methods of `javacard.framework.Applet`. While both `install` and `process` involve the respective methods, they may also include other things. For instance, a part of the `process` phase could be selecting the applet. The `perso` or "personalization" phase is realized a series of calls to the `process` method, which involves commands that are otherwise not used.

2.4.2. Other resource-constrained devices

Java Card is of course not the only Java environment for resource-constrained devices. Other devices have different restrictions and needs, leading to the creation of special environments for them. Many of them use a so-called "split VM architecture", meaning that the Java bytecode is first analyzed and compiled to a different representation before being loaded onto the device.

Simon et al. [29] proposed "Squawk", a JVM written in Java for the Sun SPOT devices, which were at the time supposed to serve as sensor nodes in wireless sensor networks (WSNs). This VM had a large set of features, including garbage collection and threads, neither of which are supported by Java Card, and on top of that allowed the isolation of applications, so that they could start, pause, resume, stop, and even be transferred onto other devices.

Brouwers et al. [5] proposed "Darjeeling", a virtual machine for sensor nodes, which implements a subset of the JVM and handles native code with the `native` keyword just as Java would, only that said native code has a different interface to the Darjeeling VM. Darjeeling also supports garbage collection and threads.

Aslam et al. [2] proposed "Takatuka", a virtual machine for sensor nodes (which they refer to as *motes*), which aims at minimal memory requirements while being fully CLDC-compliant[13].

Maye and Maaser [18] compare Darjeeling and Takatuka in several qualitative and quantitative aspects, arriving at the conclusion that Takatuka is more feature-rich and uses less memory in most cases, whereas Darjeeling has better runtime performance and as a result consumes less power.

[12] And again it's cross-package, not cross-applet. Why do we even care?

[13] Connected Limited Device Configuration (CLDC) is the specification of a framework for the Micro edition of Java.

Armbruster et al. [1] constructed a real-time JVM for the use in avionics. However, as Pizlo et al. [25] note, their approach is not sufficient in a large number of areas with really resource-constrained devices. Hence, they propose "Fiji", which rather than some kind of bytecode runs native code in a special environment, really stretching the split VM architecture. As can be seen in Figure 2.2, they first transform Java bytecode into ANSI C before translating that into native code through the GNU C Compiler (GCC). This segues neatly into the next section.

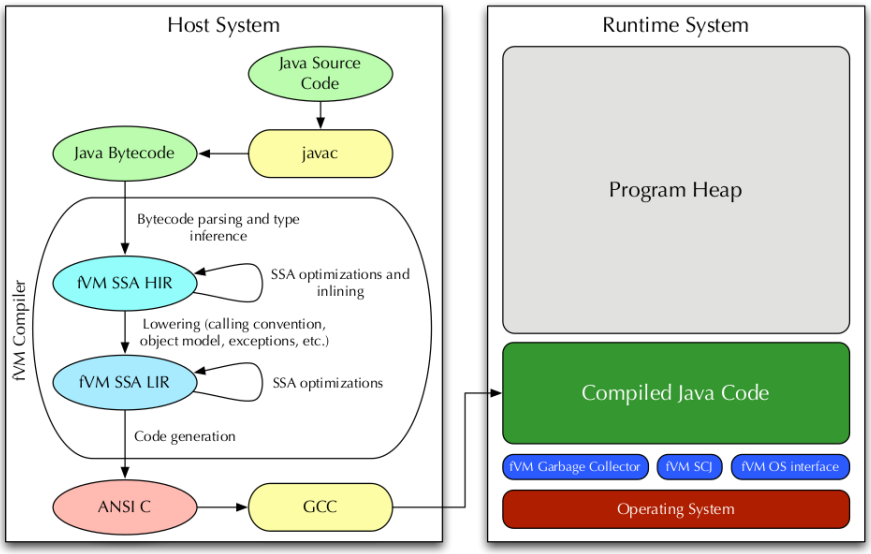

Figure 2.2.: Architecture of the Fiji VM proposed by Pizlo et al. [25].

2.5. Bytecode to native code compilation

Ever since bytecode was first used as a compilation target, it was the compilation source to someone else. In some cases the virtual machine, which it is supposed to run on, compiles it to native code "on the fly", or as others would say *just in time (JIT)*. In other cases people, who sought to do away with the overhead of interpreters or virtual machines, compiled it to native code *ahead of time (AOT)*, i.e. before the program is run, possibly dropping the infrastructure that would be required to run it in favor of a new executable while doing so.

There are a few key differences between JIT compilation and AOT compilation that are worth pointing out. First of all, AOT compilers are programs on their own, whereas JIT compilers exist (to be) embedded in an interpreter or virtual machine. Secondly, there are different constraints placed upon them. For instance the time it takes to compile some given code is crucial in JIT settings, but mostly irrelevant in AOT settings. Thirdly, AOT compilers are (if programmed to do so) capable of cross-compilation, i.e. a compiler running on one machine can produce code for another (kind of) machine.

JIT compilers on the other hand always compile for the machine they are running on. As a corollary of the second and the third point, JIT compilers will have to produce the target machine code directly without invoking another compiler. In an AOT setting, the chaining (or "pipelining") of compilers is not only theoretically possible, but also practically useful.

Muller et al. [20] designed a Java environment including a class-based AOT compiler as well as runtime library integrated JVM. As far as compilation is concerned, there are a few similarities to the approach discussed in this thesis, although they are merely accidental. Their compiler takes a `MainClass` (i.e. a class that implements a public, static `main` method) and generates two source files from them, a source file for the class itself and a `main` file, which contains the C `main` method. See Figure 2.3 for the full set of for the full set of files generated by their compiler. As will be explored in Section 3.3, the separation into sources and main methods in different files remains, but the input is a package (as CAP file) and it need not necessarily contain a `MainClass` (or rather an applet). They also generate sources for all classes related to this main class, resolving those dependencies statically. While the compiler described in this thesis and the system around it also depend on static dependency analysis, there is no assumption that all code is present at compile time. In fact, this case is quite rare.

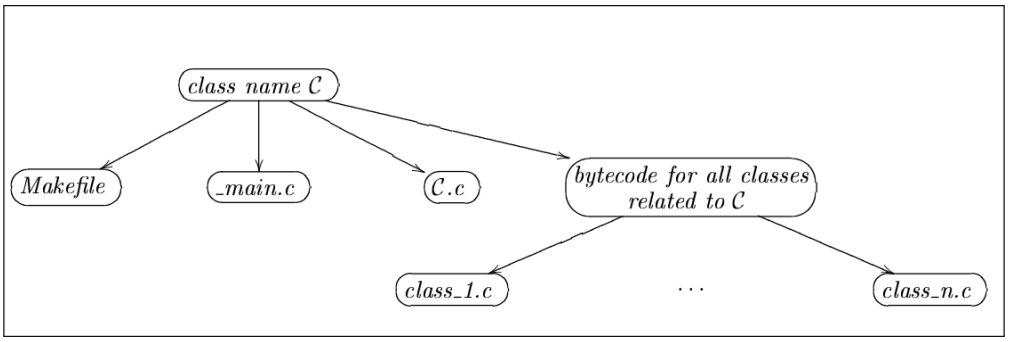

Figure 2.3.: Files generated by Muller et al.'s compiler [20].

Muller et al.'s runtime architecture is the inverse of ours. Since their VM is made into an interpreter that is part of a library, it can be controlled by an application, much more so than it is the case in the scenario assumed within this thesis. Such a design requires – at the very least – that the application boots the VM before it actually runs, often allowing the application programmer to directly interact with the VM to a certain extent. This is an approach often taken by so-called *extension languages*, such as Guile [9, 5 Programming in C] and Lua [13, 4 – The Application Program Interface], perhaps because of its flexibility, and it certainly constitutes good interoperability. However, it also puts certain amounts of trust in the programmer of said application – a level of trust that is hard to argue for in the context of Java Card. This might not be a security risk in the scenario of Muller et al., but linking an arbitrary application to the JVM and have the former control the latter runs counter to the design goals of both Java and Java Card, with especially the latter imposing strict rules for some notion of security.

As complying with said rules was a requirement for interoperability, this aspect had to be reflected in the design process.

Gal et al. [8] constructed a JIT compiler for resource-constrained devices. They use a trace-based approach, meaning they compile sequences of already executed instructions. Using this approach, they optimize loops, which are executed often (so-called "hotspots" or "hotpaths" as they call them on bytecode level). Within a trace, the only branch that is taken is the unconditional branch to the loop header. Conditional branches that occur within the trace are therefore not taken, and the native code is generated in a way that the corresponding check causes the JVM to resume at the bytecode in case of failure. To make this possible, Gal et al. retain a mapping between registers and altered locals, which are written back to their respective location before control is returned to the JVM.

Ellul and Martinez [7] constructed a "run-time" compiler for resource-constrained devices, which somewhat stretches the distinction between AOT and JIT. While the transformations they make are not too complicated to be implemented as a JIT compiler – they do not even map stack or locals to registers, keeping them "as-is" on the runtime stack – they still compile the whole program before execution, so that they can do away with the virtual machine, making the process AOT. As they retain the stack, each operation that pushes on top of the Java stack therefore becomes a set of instructions that does the same and any pop done by an instruction becomes a pop instruction. This causes many gratuitous push and pop instructions, some of which Ellul [6] later eliminates. Ellul [6] also considers mixing AOT and JIT compilation. Reijers et al. [26] expanded upon their work, improving the optimizations Ellul has made while adding their own.

Wang et al. [31] implemented a mixed-mode ahead-of-time compiler for Android applications. Their framework identifies hot methods based on a static profiling model, which are then compiled to native code and linked to the remaining parts of the application while their bytecode is replaced with a redirection. They had quite interesting findings with partial compilation, i.e. leaving cold methods in the bytecode, while compiling hot ones to native code. While the compiler described in this thesis does not come with a static profiler, it allows methods to be filtered. With this mechanism any static or dynamic analysis can be done prior to compilation. Some of their optimizations also have limited applicability in Java Card systems, for instance the *cloning* of methods. It reduces the amount of context switches at the cost of essentially doubling the size of cloned functions. When size is very limited, overhead of that scale should already be called into question if it arises from the compilation itself, requiring it for such purpose seems unreasonable.

Gressl [11] has built a Java Card cross-compilation framework based on an already existing ahead-of-time compilation framework. They use a class-based approach, representing Java classes as C `structs`, complete with method table and everything. They also pack the Java Card framework classes together with the applets they're actually compiling, creating a program that runs like any other program without the need for a JVM. Their approach comes with severe limitations. While the applets themselves may be able to run in such a manner, only one can do so at a time, and it is unclear if and how these applets would interact with the rest of Java Card. This interaction may not

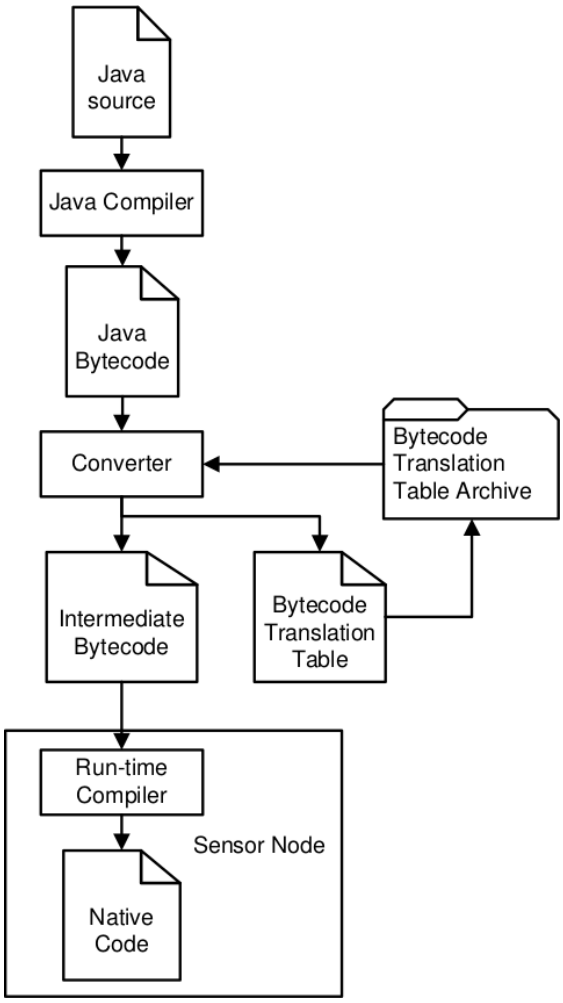

Figure 2.4.: Compilation process used by Ellul and Martinez [7]. Note, that the last step of the process – the compilation of intermediate bytecode to native code – is performed on the sensor node.

be needed, as all dependencies are compiled into the program as well, but it limits the scope of what the framework can reasonably be used for – specifically it would exclude libraries if there indeed was no FFI. It also leads to worse overhead, as everything must be compiled regardless of whether it is meaningful to do so.

The compiler described in this thesis only needs the CAP file of the applet to convert and the export files of its dependencies, with the virtual machine handling runtime dependencies. Gressl on the other hand needs to have all dependencies – both explicit and implicit – in their resulting binary. This leads to a dependency resolution method dubbed *automated greenlisting*. Their framework allows for the specification of a set of classes, which shall be the only ones to be included in the output – this set is called *greenlist*. Gressl computes their greenlists starting from a basic initial one by compiling and running the program and including the class whose exclusion caused a failure in this process. This is repeated until no such failure occurs.[14] It might be worth noting that a simpler dependency resolution exists for their case, as Java Card is not reflective. Starting from a set of classes that one wants to include in one's target output, the recursive resolution of any constant pool item referenced by the methods of those classes will lead to a minimal working greenlist for all possible execution paths. As Gressl already needs the CAP files of all dependencies to extract their bytecode, this should be possible in their setting.

2.6. Other smartcard platforms

Other Java Card systems typically work like the one that serves as basis for this thesis. However, there are instances of Java Card systems that use a completely different architecture as well as competition to Java Card itself.

When talking about other Java Card platforms, one optimization strategy that does not depend on native code at all, would be the usage of dedicated hardware, particularly a processor, which is able to execute Java Card bytecode. Said bytecode would then be native to that processor, making it the most efficient code that could be executed on it. While some blueprints, such as the ones by Zhang et al. [33] or Golatowski et al. [10] do exist, it is not accompanied by research suggesting that they can outperform other processors running native code. Similar research was also done for the Java Virtual Machine to create a Java processor – Schoeberl [28] shows that they are indeed outperformed even by a JVM, as long as they compete on the grounds of performance; only when putting chip size into consideration their system becomes better.

An overview of competing architectures is given by Sauveron [27] or Markantonakis and Akram [16]. Specifically, they list MULTOS, Smartcard.NET, Multiapplication BasicCard and Windows for Smartcard. MULTOS smartcards use a variant of Pascal p-codes, with compilers available for C, Java, and other languages (Sauveron specifically lists Basic and Modula-2). While the availability of more compilers appears to be liberating, it is in fact just yet another virtual machine and on top of the advantages

[14] Let us assume, even though one may doubt it that an execution only succeeds if the requirements for any execution path through the applet are met.

and drawbacks that Sauveron points out, it stands to reason that performance could be gained through native code there as well. Smartcard.NET supports a subset of the .NET framework with a larger array of features than Java Card, and allows applications to be written in any language suitable for the framework. Multiapplication BasicCard is the continuation of an older product – BasicCard – but with support for multiple applications, as the name implies. BasicCard itself is built around the Basic programming language and the cards execute some variant of p-code (neither Sauveron nor Markantonakis and Akram are more specific). Lastly, Windows for Smart Card is an abandoned project, whose goal was to bring the Windows Operating System to smartcards.

Of the still existing smartcard platforms listed above, all follow a similar structure to Java Card as far as the core is concerned. A virtual machine runs some kind of bytecode and provides a certain set of features – the exact feature set depending on the architecture and possibly the vendor. If one further wishes to be able to program in Java, only MULTOS (Sauveron [27]) and BasicCard (Markantonakis and Akram [16]) offer compatibility layers on their own. There does however exist at least one compiler that allows for the interoperability of Java and .NET, namely the one that Gressl bases their thesis on. In any case, it appears as if smart card system designers can all agree that they want to use some kind of virtual machine for whichever reason (usually something along the lines of security, portability, etc.), but are divided on the question of which of them to take. Given such an ecosystem, all of them could profit in some way or another from native compilation and/or dedicated hardware.

3. Design

The following sections outline the design of the compiler and the infrastructure around it. Section 3.1 describes the system that the generated code is supposed to run on. Section 3.2 describes interfaces – i.e. functions and macros – to said system, which the compiler may rely on existing. Section 3.3 describes the compilation process itself. Section 3.4 gives a general overview of the design that went into the compiler itself, whose inner workings are described in Chapter 4.

3.1. System Architecture

The Java Card specification covers two big parts: the JCVM and the runtime environment for applets. Anything beyond that is outside of its scope. However, that is not everything necessary to facilitate a complete Java Card system. Of course, some hardware is required, but there is also a reliance on (software) components that handle communication or deal with resources. These components must exist alongside or below the JVM for them to be available to the Java runtime environment (JRE). In computer science, such components are usually combined into an entity, that is called the "operating system (OS)". Hence, we have a system that looks like Figure 3.1.

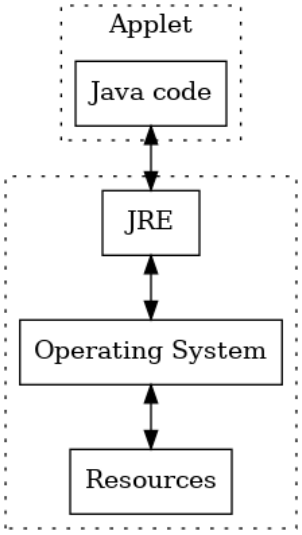

Figure 3.1.: Architecture of a typical Java Card system.

It does not really matter, whether the JVM is part of the OS or part of the JRE, whether one applet or more is/are running, or whether the JRE lives inside its own box that is encapsulated both from the OS and from the applet or just from the applet – the Java Card specification only requires a strict separation between different applets and between applets and the JRE. This is the system that will be expanded.

It is extended by adding a C runtime environment (CRE), as shown in Figure 3.2. Native code is treated as separate from bytecode, but is still part of the same package. For instance, both parts of Applet 2 in the figure refer to the same constant pool – that way method invocations are facilitated among other things. The CRE is set up inside of a sandbox and communicates with the JRE through system calls. Each of these system calls corresponds to some Java Card bytecode, and the same checks are expected to run for both – e.g. a `BALOAD` from the JRE or CRE shall in either case check that indeed an array is accessed and that the index is valid. That way, the CRE is perhaps not more efficient than the JRE for such accesses, but an important security property is gained, as any attack using this system calls would also be possible by abusing the JRE in some way.

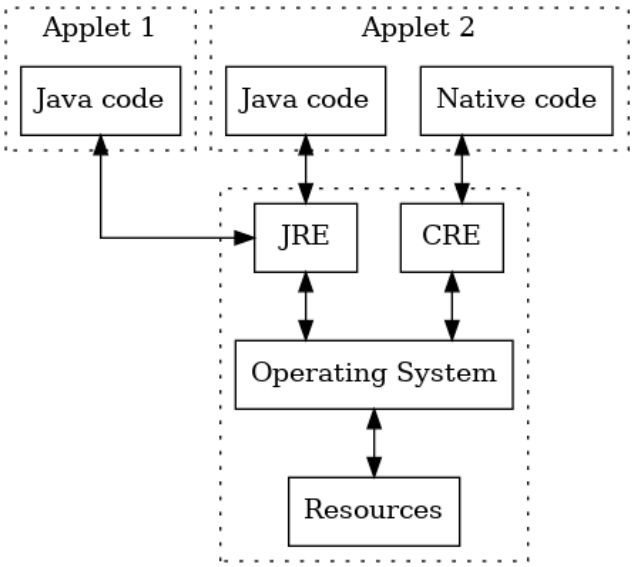

Figure 3.2.: Architecture of the extended system.

To allow arbitrary recursion up to the limits imposed by memory, the OS shall maintain a call stack that is used by both the JRE and the CRE. When a method call resolves to a function implemented in Java, a Java frame shall be pushed on top and populated with necessary data (e.g. reserve space for stack and locals, initialize locals from arguments). When a method call resolves to a function implemented in native code, a native frame shall be pushed on top.

3.2. Interfaces

The compiler described in this thesis shall make as few assumptions on the underlying system as possible – that way, the generated code will be portable to an extent. However, some additional interfaces are necessary, given that the functionalities that are to be implemented, exist partly beyond the scope of plain C.

First of all, the bytecodes given in Table 3.1 must somehow be made available through system calls. UPPER_SNAKE_CASE is used for them as opposed to the lower_snake_case used in the Java Card specification to indicate that these can be regarded as macros. A 1:1 correspondence between bytecode and system call is not required. For instance, one could merge `LOAD` and `STORE` type calls or `CHECKCAST` and `INSTANCEOF` into one with an additional parameter. Order of arguments can also be swapped around to the extent allowed by the C preprocessor, and so on, and so forth.

Bytecode	Comments
`<T>ALOAD`	
`<T>ASTORE`	
`GETSTATIC_<T>`	
`PUTSTATIC_<T>`	
`GETFIELD_<T>`	
`PUTFIELD_<T>`	
`INVOKE<T>`	A `nargs` parameter is added to those invokes, which do not already have them. This parameter can be used to determine the number of arguments to push onto the JVM stack before resolving the invocation, as the JVM would handle bytecode with similar amounts of precognition.
`NEW`	
`NEWARRAY`	
`ANEWARRAY`	
`ARRAYLENGTH`	
`ATHROW`	
`CHECKCAST`	
`INSTANCEOF`	

Table 3.1.: JVM bytecodes, which lack native handling. Generally speaking, all bytecodes, which dereference object references, as well as all communication with other packages, are to be handled by the JVM.

There are two entry points from the OS. The first one is `main`, which is called when a method is invoked. This may happen for one of two reasons.

1. A Java method invokes a native method for the first time.
2. A C method invokes – directly or indirectly through some Java method(s) – another C method.

In either case, `main` resolves the method call and calls the function corresponding to it, returning its return value (if any) to the OS. The second is `catch`, which is used for exception handling. Native methods may both raise and handle exceptions in the same way that Java methods do. This, however, creates an interesting problem, as there is no direct link between positions in Java bytecode and offsets to the machine code produced by the C compiler. However, those positions are essential to the JVM exception handling, both to indicate ranges of a try block and the position of an exception handler. The CRE instead uses a `longjmp`-esque convention for `catch`. If the exception is unhandled, it returns to the OS normally, otherwise it does not return. Both `main` and `catch` may be renamed as the OS desires.

Since `catch` can not return normally if the exception is handled, `main` can not return normally either. An additional syscall, `SYS_RETURN(bool,short)`[1], is therefore necessary to signal a return from `main`, wherein the first argument is false for `void` methods to indicate the lack of a return value. The name of this syscall as well as all implementation details surrounding it, are however hidden by the compiler, so a platform may freely choose to supply a different way of handling exceptions and returns. To aid portability to a wide array of platforms, the `main` method is instead built via a skeleton using the macros defined in Table 3.2.

Macro	Expansion
`MAIN_TYPE`	Return type of the main function, such as `int`.
`MAIN_FUNCION`	Name of the main function.
`MAIN_ARGS`	Arguments supplied to the main function by the platform.
`MAIN_SETUP(methodOffset)`	Platform-specific setup code. After this code is executed, `methodOffset` is expected to hold the offset of the method that is to be invoked.
`MAIN_RUN(func, ...)`	Code to run `func(...)` and store its result.
`MAIN_RUN_VOID(func, ...)`	Code to run `func(...)` without storing its result
`MAIN_ARG(type, i)`	The *i*th argument, cast to `type`
`MAIN_TEAR_DOWN()`	Platform specific tear-down code, including the return to the caller.

Table 3.2.: Macros used in the main skeleton.

The `catch` procedure can be defined in any file that is compiled and linked together with the results of the compiler. The exception handler uses three non-functional macros that need to be defined in a header.

[1]Systems that support 32-bit integers (`int`) may also use `SYS_RETURN(short,short,...)`, in which the first argument specifies the length of the return value. Note however, that the compiler does not produce code that requires such systems yet. See Section 5.5.

1. `DECLARE_OF_EXCEPTION(idx)` expands to the declaration of the variables used by exception range `idx`.

2. `ENTER_EXCEPTION_RANGE(idx, type, handler)` will be called when entering an exception range. Here, `type` will be the index of an exception class in the constant pool (which may be 0 in case of a "finally") and handler will be a label to jump to when the exception is thrown. This makes it so that `setjmp` (or functions like it) can be used inside the expansion, along with a code snippet à la `if (exception_thrown) goto handler;`.

3. `EXIT_EXCEPTION_RANGE(idx)` will be called when exiting an exception range.

Together, these macros help maintain an exception (handler) stack in the native components. These stacks are again kept on an exception frame stack, through the code inside `MAIN_SETUP` and `MAIN_TEAR_DOWN`, which together looks roughly like Figure 3.3. The workflow is as follows:

1. `MAIN_SETUP` creates a new frame and pushes it on top of the frame stack, which exists on a well-known address.

2. `ENTER_EXCEPTION_RANGE` creates a new handler and pushes it on top of the frame.

3. `EXIT_EXCEPTION_RANGE` pops the current handler from the frame.

4. `MAIN_TEAR_DOWN` pops the current exception frame.

These steps come in pairs – i.e. there is always a `MAIN_TEAR_DOWN` for a `MAIN_SETUP` and and `EXIT_EXCEPTION_RANGE` for an `ENTER_EXCEPTION_RANGE` – and they may be nested and repeated up to the amount permitted by memory. In addition there is `catch`, which pops as many exception frames as needed, until the frame that was popped is the one which handles the exception and then performs a `longjmp` (or an operation similar to it) to resume execution there. A `catch` therefore acts as multiple implicit exception range exits.

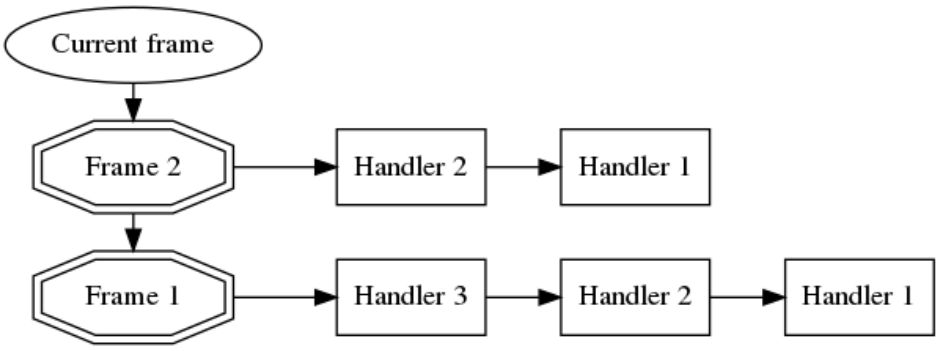

Figure 3.3.: Exception stack corresponding to a recursive invocation of depth 1. Between the two C frames, there can be an arbitrary number of invisible Java frames.

3.3. Build Process

Having a good enough abstraction of the underlying system and the interfaces to it, one can start to write code that would at least compile in some way. However, doing so programmatically requires some kind of build flow. Typically, a Java Card applet would be built as demonstrated in Figure 3.4. One or several Java source files are compiled to class files as they would be for a Java application, which are then used in conjunction with export files to produce a converted applet (CAP) file. A JCVM implementation can then install an applet from this CAP file. In the case of libraries, this process may also yield export files for the build flow of other applets.

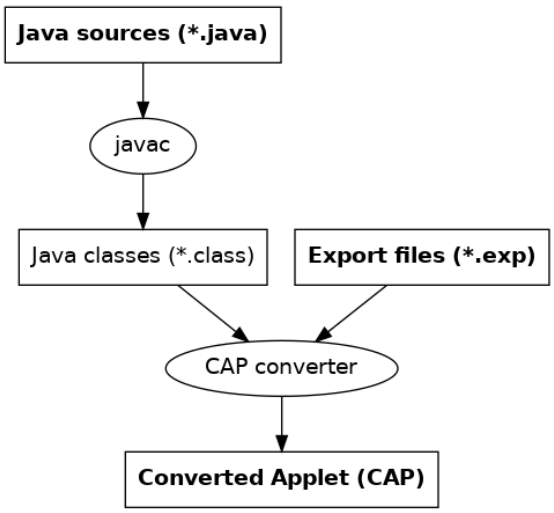

Figure 3.4.: Build flow for a classic Java Card applet.

This flow is extended as shown in Figure 3.5. On the outside, a CAP file and the export files that were involved in its generation are taken as inputs and a CAP file extended with (custom) native components is produced as output. Inside that process, three kinds of files are to be distinguished. The first are headers, also referred to as ***methods.h***. These are quite literally C header files and contain a declaration to each method that is to be compiled from Java Card bytecode to C. The second are the sources with the actual compiled methods, referred to as ***methods.c***. They contain the C code for the methods listed in the headers. Finally, there is a so-called *main* file, also referred to as ***main.c***. This is the place, where the `main` function is defined. Specifically, it is defined as in Listing 3.1, with methods from ***methods.h*** added to the switch.

```
#include ...

MAIN_TYPE MAIN_FUNCTION(MAIN_ARGS)
{
  MAIN_SETUP(methodOffset);
  switch(methodOffset)
  {
  case __offset_to_function:
    MAIN_RUN(__function_name, MAIN_ARG(__arg_type, 0), ...);
    break;
  ...
  }
  MAIN_TEAR_DOWN();
}
```

Listing 3.1: Skeleton main file.

In each of the compilation steps – *header*, *compile* and *main* – methods can be filtered based on regular expresion. This allows one to partially compile a CAP file to C or to use different compilation switches for individual methods.

The dependencies of the files generated by the compiler are shown in Figure 3.6. In addition to internal dependencies, there are

- *platform type headers*, which define basic types, such as `inttypes.h` and `stdbool.h` if depending only on the ISO C standard,
- *platform interface headers*, in which the system calls of Section 3.2 are defined, and
- *platform main macros*, which in a rather self-explanatory way are the headers that define the macros used by `main`.

Both internal and external headers must be specified on the command line through options – `-sys-include=FILE` for an angled bracket include directive and `-include=FILE` for one with quotes.

In a standard setting, one would require both `inttypes.h` and `stdbool.h` to translate all primitive types used in Java Card applications to primitive types in C. However, non-standard conventions are not unheard of, particularly not within the C environment, with platforms, frameworks, etc. each adding their own `typedefs` to the mix. Ironically, the JNI would be one of them, and it would define all the types that are needed, but as the `native` keyword and the JNI were "dropped" from Java Card – at least as far applet development is concerned – these definitions are missing in a Java Card environment. In order not to define yet another set of types, a so-called "type definition file" is used to override the internal data type definitions. The headers, which define those types towards C code, are included as *platform type headers*.

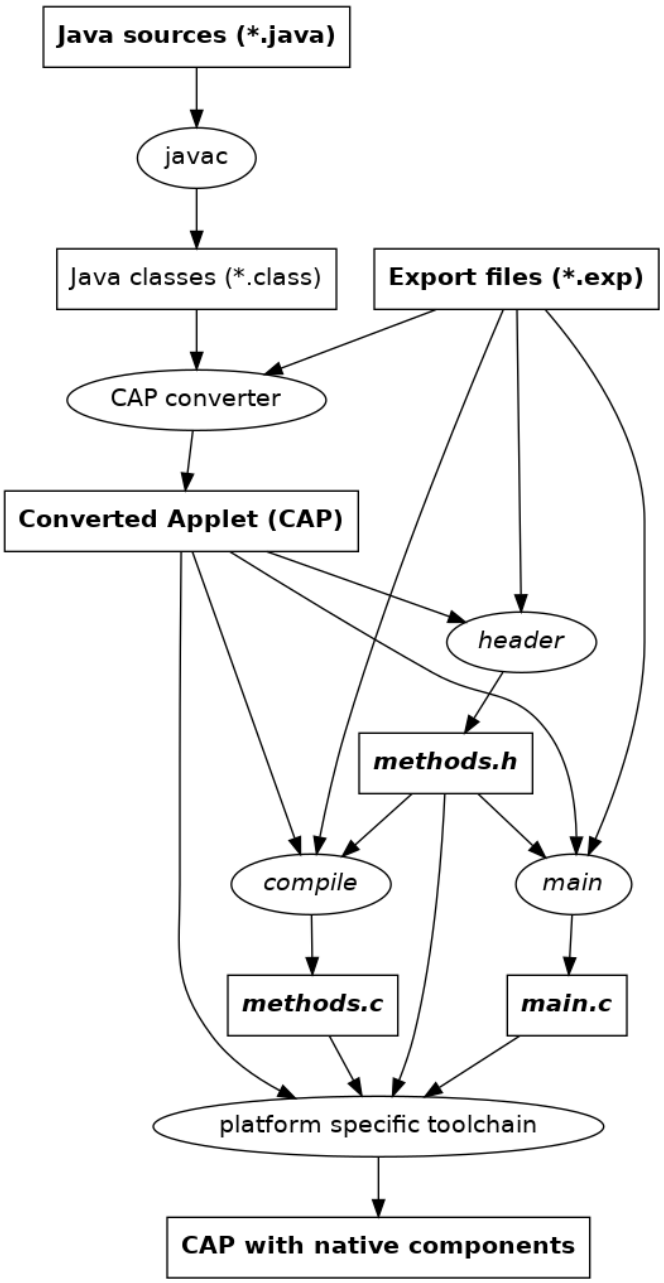

Figure 3.5.: Extension of the classic Java Card applet to an applet with native components. The commands *header*, *compile* and *main* are implemented by the compiler. The files ***methods.h***, ***methods.c*** and ***main.c*** refer to one or multiple outputs of said commands.

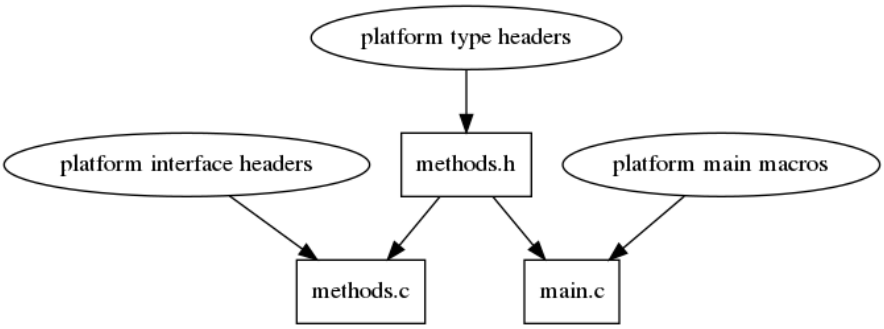

Figure 3.6.: Dependencies within the generated code.

3.4. Compiler architecture

Compilers are usually structured into three components,

- a *front end*, consisting of parser and lexer, which transforms a file written in the source language into an intermediate representation (IR),
- a set of optimizations performed on that IR, sometimes grouped together under the name *middle-end*, and
- a *back-end*, which transforms this IR into target-dependant code.

At least, such is the case for compilers, whose input is assumed to conform to some programming language. The input to the compiler of this thesis is a CAP file in the broadest sense, and a stream of instructions with various metadata scattered around said CAP file in the narrowest sense. As such, its *front end* is an ad-hoc constructed binary analysis framework and the *middle* and *back* ends are one procedure with a series of hooks allowing for optimizations. Since C is not only the output of the compiler described in this thesis, but also the input to some toolchain that does the actual compilation to native code, C itself can be seen as an IR here. Using C as an IR is however not in any way a new and exciting idea – it has existed at least since C++, if not before it.

Before the actual work of the compiler starts, some metadata needs to be extracted from the CAP file. This is boring work, parsing the file according to the specification, and will thus not be discussed in greater detail. Once that is done, we are left (among other things) with a set of classes and methods, the latter of which are to be compiled. The compilation of a method begins with its disassembly it, i.e. then turning the bytecode into a list of instructions. Thereafter follows the creation of a control flow graph (CFG) from this list and some metadata, and a data flow graph (DDG)[2] from the CFG and some metadata. Next, this CFG and DDG are used to construct a mapping from instructions to a set of equivalent statements (and labels) in C. Finally, these statements are collected

[2]Normally, one would abbreviate "data flow graph" as DFG, but internally the term DDG (actually means "data dependency graph") was used until it got stuck. As those terms are only as meaningful as we decide as developers, we settled on the somewhat weird combination of data flow graph and DDG.

in the same order as the instructions – as well as the labels, putting the latter before the statements they point to – and after being decorated with variable declarations and a function signature, the result can be called a C function.

Repeating this process for each method will allow us to compile the entire CAP file. As will be seen later, some optimizations will also produce additional functions. This transforms the simple loop used before to a more complex one able to receive events from the innermost parts of the compiler through a global queue, but the core idea remains the same: the compiler takes Java methods as input and produces C functions as output, preferably printing them in a “pretty” manner.

4. Implementation

This chapter elaborates on the implementation details of the compiler. The raw structure has been outlined in Section 3.4: First, the binary is to be analysed, so as to produce a CFG and a DDG, from which then C code can be generated with various optimizations being applied along the way. This is also the structure, in which the following sections are laid out. The rest of this section is structured just like that: first comes bytecode analysis, then code generations, then optimizations.

4.1. Analysis

As a binary is to be analysed – particularly a binary for the JCVM – the methods through which it is analysed are binary analysis methods, not the lexical or syntactical analysis one would typically expect from a compiler. The intermediate representation, which this analysis yields, consists of two graphs: the control flow graph and the data flow graph. While normal compilers use an abstract syntax tree to represent their input, this compiler uses the data flow graph as a union of concrete inverted syntax trees, each encoding an expression. This formulation might at first look a bit intimidating, but it is a lot easier to understand it by example.

Consider a simple assignment, such as

```
short foo = 2 + 3;
```

The corresponding concrete syntax tree (CST) would look like Figure 4.1. A non-optimizing Java Card compiler may generate the following bytecode out of it:

```
SCONST_2
SCONST_3
SADD
SSTORE_1
```

The corresponding data flow graph looks like Figure 4.2. Clearly, the same information (modulo interpretation) is present in both, it's just that the direction of the data flow is inverted w.r.t. the CST. Therefore, to generate a syntax tree from the data flow, one simply needs to invert the flow back into its original direction.

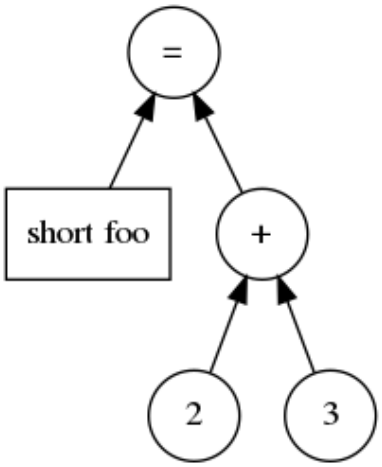

Figure 4.1.: CST of a simple assignment.

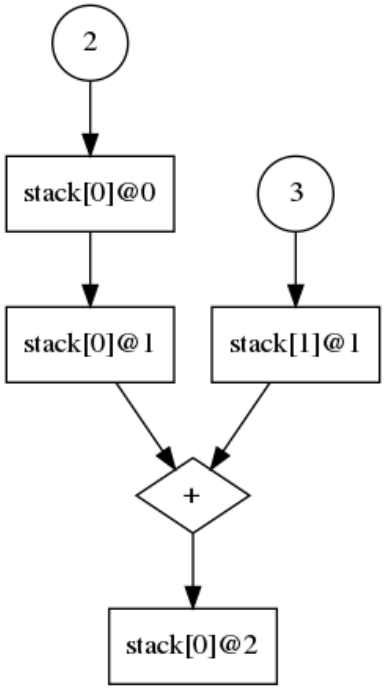

Figure 4.2.: Data flow example of a the same assignment as Figure 4.1.

4.1.1. Control Flow

The analysis of the control flow starts from a disassembly of the bytecode of a method represented as a list of instructions and generate a graph, which has

- one node per instruction representing said instruction, and
- one node per target of a branch instruction representing a label.

In addition to that, one might have to deal with exception handlers, which are given to as a set of integers, each corresponding to the offset of a handler. Their entry points also get a node each, which also serves as an additional root[1] in the CFG, the main root being the main entry to the function, i.e. its first instruction.

The structure of the control flow graph was chosen in such a way to make processing easier, especially in the generation of the data flow graph, but also within code generation. It constructs views, in which each instruction can be seen as somewhat independent of previous and subsequent ones (while still being connected), and the implementation of any other method becomes a Visitor of the control flow graph with a switch on the instruction code.

With a quick overview of the structure to create and the reasoning as to why it is to be created in that way in mind let us proceed towards the process of its generation. As previously mentioned, the starting point is a list of instructions. Each instruction in said list has a *position*, a *code* and optionally *arguments* that have been encoded in the bytecode itself rather than being taken from the stack. None of that matters in the case of instructions that do not alter the control flow. In that case, the next instruction will be the one that comes afterwards in the instruction list. Branching instructions may alter the control flow in the following ways:

1. They might have an edge to some target instruction that would otherwise not be reached in such a manner.
2. They might not lead to the next instruction.
3. They might do all of the above (e.g. `GOTO`).

In order to deal with this complexity, a two-step process is used. First, one node is created per instruction and then the edges are filled in, with a mapping B tracking all redirections due to branches and a node n_p tracking the previous instruction node. The pseudocode for the generation process is given in Algorithm 2.

By convention, each node holds at most one instruction. To make it simpler to deal with nodes that have none, dummy instructions are also added for branch targets and exception handlers with special "opcodes" that does not map to any JCVM bytecode marking them as such.

[1] Note that the concept of a "root" differs from the roots of a tree in graph theory. Since control flow graphs are directed, but not necessarily acyclic, it makes no sense to apply such concepts here. In this case, a root is much more an entry point to the function. A comparison might be made to state machines, whose graphical representation have an initial state as their "root", which is often depicted as a node having an ingoing edge with only one endpoint, the other endpoint lying somewhere in the ether (though the ether itself might be a node on its own for some, as Graphviz surely would agree).

Input: CFG C, branch target mapping B, CFG node n
Output: CFG node t
begin
  if $\exists t : (n, t) \in B$ **then return** t;
  else
    Allocate a new node t;
    Set $V(C) = V(C) \cup \{t\}$, $E(C) = E(C) \cup \{(t, n)\}$, $B = B \cup \{(n, t)\}$;
    if $\exists n' : (n', n) \in E(C)$ **then**
      Set $E(C) = E(C) \cup \{(n', t)\} \setminus \{(n', n)\}$;
    end
    return t;
  end
end

Procedure 1: branchtarget(C, B, n)

Input: Instructions I, Exception handler offsets H
Output: CFG C
begin
  Allocate a CFG C;
  Let R be the roots of C and L be its leaves;
  Allocate a map of integer to nodes N;
  foreach *instruction* $i \in I$ **do**
    Construct CFG node n_i and set $V(C) = V(C) \cup \{n_i\}$;
    if $R = \varnothing$ **then** Set $R = \{n_i\}$;
    Set $N = N \cup \{(p_i, n_i)\}$, where p_i is the position of i;
  end
  Allocate a map of nodes to nodes B;
  Let $n_p =$ `null`;
  foreach *position-node pair* $(p_i, n_i) \in N$ *sorted by* p_i **do**
    if $n_p \neq$ `null` **then**
      if $\exists t : (n_i, t) \in B$ **then** set $E(C) = E(C) \cup \{(n_p, t)\}$;
      else set $E(C) = E(C) \cup \{(n_p, n_i)\}$;
    end
    foreach *offset* o *denoting a branch* **do**
      Let $(p_i + o, n') \in N$ be the position of the targeted instruction and its node;
      Let $t :=$ `branchtarget`(C, B, n');
      Set $E(C) = E(C) \cup \{(n_i, t)\}$;
    end
    if i *is a returning instruction* **then** set $L = L \cup \{n_i\}$, $n_p =$ `null` ;
    else if i *is a* `goto` **then** set $n_p =$ `null`;
    else set $n_p = n_i$;
    if $p_i \in H$ **then**
      Allocate a new node n_h;
      Set $V(C) = V(C) \cup \{n_h\}, R = R \cup \{n_h\}, E(C) = E(C) \cup \{(n_h, n_i)\}$;
    end
  end
  Return C;
end

Algorithm 2: Generation of the control flow graph.

4.1.2. Data Flow

The analysis of the data flow starts from the freshly created control flow graph and generates a graph, in which stack and local nodes as well as their assignments are represented. For each node in the CFG, there will be a number of stack and local nodes (henceforth summarized as *variable nodes*), some of which may be created or overwritten depending on the current instruction. For two CFG nodes that share an edge, there will be paths from variable nodes corresponding to the source to variable nodes corresponding to the destination, which may or may not have *non-variable nodes* – e.g. operator nodes, which represent an operation, or function nodes, which represent a method invocation – on their way.

The creation of the data flow graph is a three step process, as illustrated in Figure 4.3. In the first step, all stack nodes are created and some edges, such as those for mathematical operators are already added. The stack can be created inductively from the roots of the CFG, at which it is either empty if the node is the function's entry or holds the exception if the node is the first instruction of an exception handler. Each instruction pops a number of values from the stack (e.g. `SADD` always pops 2) and pushes a number of values to the stack (e.g. `SADD` always pushes 1), and that number is either statically known as in the case of operator nodes, or can be inferred from arguments, such as in the case of method invocations through any invoke instruction. With both the stack depth of an already reached node and the changes made to it being known, the depth upon reaching the next instruction can be determined. Of course, one has to assume (and verify) that the stack depth of an instruction is invariant across all the routes by which it can be reached.

In the second step all local nodes, i.e. nodes that correspond to the method's arguments as well as variables assigned by the programmer, are created. In order to do this, the stack creator first needs to be modified, such that it also reports local *definitions* (i.e. assignments) and *references*. Bytecodes that use locals can be categorized into

- bytecodes that push locals onto the stack,
- bytecodes that pop the top of the stack into a local,
- or bytecodes that modify a local in-place

according to their op-code. With the additional information of which local they operate on, corresponding nodes and edges can be computed from the previous state as was done for the stack.

The third step processes "special" bytecodes that have so far not been dealt with. This means filling in everything that has to do with field or array accesses, method invocations and adding return nodes that highlight returns from the function as well as *decision* nodes that highlight the value based on which branches are made.

For each node that is created in the DDG, regardless of the step in which it was created, the CFG node, which was being handled at the time of its creation will be remembered as its "parent" – or to avoid confusion with terminology from graph theory its *linked CFG node*. In addition to that, each node also has a name, which is a string loosely

associated with its meaning. For instance, stack nodes will be named "stack[%d]", with "%d" replaced by their index, function nodes have the name of the function, operator nodes the operator, and so on, and so forth. This name is not purely cosmetic (i.e. solely related to the nodes "label" within the graph's visualization), but will also be used during code generation.

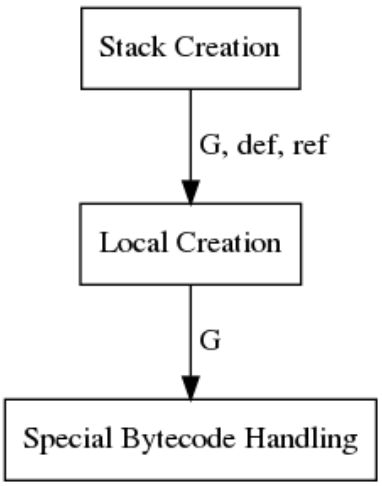

Figure 4.3.: Steps during the creation of the data flow graph. In the first step, the stack nodes are created and their relation to each other represented by edges. In the second step, local nodes are created and connected to the stack. In the third step, "special" nodes are created for bytecodes that deal with decisions or data other than the stack and locals, such as fields and arrays.

4.1.3. Symbols

Symbols are needed in various places, one of the first ones being the data flow analysis, which needs to know the number of arguments of functions as well as their return type to pop/push the right amount of variables from the stack. Within the CAP file all references that are made to such symbols are encoded as indices to the `ConstantPoolComponent`, which mainly consists of class references and tokens. In combination with the `DebugComponent` of a file (or alternatively the `DescriptorComponent` in combination with some clever naming scheme), as well as the export files of all imported packages, these references can be resolved. We uncreatively term them *resolved (ConstantPool) item* and regard them as a tuple of *name* referring to the fully qualified name of the item, *type* referring to its data type and *attributes* referring to some flags that are deemed important. The symbol table is thus replaced by such a lookup mechanism, though it can of course be made a table explicitly by first looking up all indices and writing them to some kind of memory.

4.2. Code Generation

Having created both control flow and data flow graphs, it is finally time for some actual C code generation. As already hinted at, the code generator will visit the CFG and associate each instruction with a (list of) statement(s)[2] and an optional label. The function that is created then consists of a signature that has already been looked up and a body that is about to be constructed. This body consists of a set of variables and a list of statements interleaved with labels. On a higher level, a Java Card instruction is associated with at most one label, an arbitrary amount of statements and an arbitrary amount of exception handlers. When putting together the method body exception handlers are added first together with any code associated with them (if any), followed by the optional label and then the statement(s) of the instruction. These statements labels and statements are of course only printed after all variables have been declared.

We will look at the somewhat naïve core of the code generator, which translates a single Java Card instruction to C in Section 4.2.1, also adding an association with labels in the process. With each instruction having an equivalent "block" of C code and a way of stringing them together as described above, a first prototype can be constructed that is able to translate method-like Java Card bytecode constructs (including Java Card methods themselves, which are extract from the `MethodComponent` of a CAP file) to the body of a C function. The only exception to its feature set are exceptions, which will be discussed in Section 4.2.2.

4.2.1. Translating Java Card bytecode to C

Most bytecodes, which do nothing special, have a straightforward representation already within the DDG. If an instruction with such a bytecode is encountered, Algorithm 3 can be used to create a list of statements. For now, this list will hold at most one element, which is an assignment to either the top of the stack or some local. However, formulating the algorithm in this way later permit certain contractions within the DDG while retaining all of the functionality of the generated code.

The procedure `expression` is defined recursively. If d is a stack or a local node, it yields a C variable corresponding to the same variable within the Java Card context. Otherwise it recursively turns the node's inputs (nodes with an outgoing edge towards d) into expressions and then returns an expression corresponding to Table 4.1.

node type	inputs	expression	notes
stack	any	`Variable`("stack", i)	i is the index part of `name`(d)
local	any	`Variable`("local", i)	i is the index part of `name`(d)
constant	none	`Constant`(c)	

Continued on next page

[2] We regard a sequence of statements as a single statement since either way the output will be a simplified, almost assembler-esque subset of C with labels and `goto`.

Continued from previous page

node type	inputs	expression	notes
operation	d'	`Operation`(n, d')	n is the operation name
operation	d', d''	`Operation`(n, d', d'')	n is the operation name
cast	d'	`Cast`(t, d')	t is the type to which d' is cast. The only relevant cast in this thesis is `short` to `byte`
array read	a, i	`FunctionCall`(m, a, i)	m is the mnemonic of the linked CFG node's bytecode
array write	a, i, v	`FunctionCall`(m, a, i, v)	m is the mnemonic of the linked CFG node's bytecode
field read	none	`FunctionCall`(m, i)	m is the shortened[3] mnemonic of the linked CFG node's bytecode, i is the constant pool index of the field
field read	o	`FunctionCall`(m, o, i)	m is the shortened[3] mnemonic of the linked CFG node's bytecode, i is the constant pool index of the field
field write	v	`FunctionCall`(m, i, v)	m is the shortened[3] mnemonic of the linked CFG node's bytecode, i is the constant pool index of the field
field write	o, v	`FunctionCall`(m, o, i, v)	m is the shortened[3] mnemonic of the linked CFG node's bytecode, i is the constant pool index of the field
function call	$a...$	`FunctionCall`$(m, i, n, a...)$	m is the mnemonic of the linked CFG node's bytecode, i is the constant pool index of the function

Continued on next page

Continued from previous page

node type	inputs	expression	notes
interface function call	$a...$	`FunctionCall`$(m, n, i_C, i_F, a...)$	m is the mnemonic of the linked CFG node's bytecode, i_C is the constant pool index of the interface, i_F is the method id given in the bytecode

Table 4.1.: DDG nodes and their corresponding expressions. Parameters mentioned as inputs or in the notes are implicitly converted to expressions themselves.

Input: CFG C, DDG D, CFG node n
Output: List of statements S
begin
  Allocate a new statement list S;
  foreach *DDG node d corresponding to the stack or locals at n* **do**
    Let (d', d) be the first edge in $\delta_D^-(d)$[4];
    if `name`$(d) \neq$ `name`(d') **then**
      Let $s :=$ `assignment`(`expression`(d) as variable, `expression`(d'));
      Set $S = S \cup \{s\}$;
    end
  end
  return S;
end

Algorithm 3: Default algorithm, which is able to handle most bytecodes naturally.

With all such "simple" expressions out of the way, control flow operations (conditional and unconditional jumps, as well as switches, also various ways of returning from the function) and the special bytecodes `DUP_X` and `SWAP_X` remain to be handled. Thanks to the control flow graph all *potential* successors of a given instruction are known, but this is not enough for code generation, as the conditions under which such branches are taken need to be accounted for. Those are encoded either in the DDG or in the

[3]Some bytecodes are variants of others with suffixes such as "_W" signaling a short-sized parameter instead of a byte-sized one, or in the case of non-static field accesses, "_THIS", which implies that the local 0 (the `this` pointer as it were) shall be taken as first argument. All these cases to a uniform one, as it decreases the number of functions that need implementation.

[4]Note the implicit assumption that the first input is the same as any other. This follows from the way the gloss(cfg) and gloss(ddg) were constructed earlier. Any difference that could be made due to branching is eliminated by having the results first written to some stack or local variable. In a similar manner, exception handlers do not affect correctness either, because the stack is initially empty and the locals are unchanged or in the case of the local holding the exception itself, invisibly changed. If such an instruction is at all reachable from normal code, it is also quite likely the target of a `goto` instruction, due to that only being the case in `finally` blocks.

instruction itself. While one could label the edges of the CFG with those conditions, these labels are missing until the creation of the DDG, for which the CFG is already needed, creating a circular dependency. Instead of recomputing the CFG, it makes sense to instead recompute the target instruction of a jump as one walks through the already exiting one and to that said target is indeed a successor of the currently visited node. This is done in the procedure `makelabel`(n, o).

Now we have all prerequisites to handle conditional and unconditional jumps, which are handled by Algorithm 5, as well as switches, which are handled by Algorithm 6. Returns do not require that much overhead. In case of a return without value, one simply emit the statement `return;`, otherwise `return val;` is emitted, where `val` is the single input to the single return value node for n formatted according to Table 4.1.

Lastly, there are the special bytecodes `DUP_X` and `SWAP_X`, special in the sense that a "direct" translation from the DDG to C code leads to errors due to the lack of simultaneous assignments. Instead, one has to consider which sequence of assignments would yield the same result. `DUP_X` is surprisingly similar to Algorithm 3, but with a few key differences:

1. The list of assignments starts at the top of the new stack and goes downwards.
2. A map needs to be kept containing values that were already written to new locations. If one of them ends up on the right hand side of an assignment, the newly-written one is to be used, as that one will not change as the assignment "loop" continues.

This way, the previously described problems can be averted. For `SWAP_X`, let n be the amount of swapped variables. Then:

1. allocate n temporary variables,
2. assign the inputs of the lower n changed variables to the temporaries,
3. assign the other inputs to their variables as we'd normally do,
4. assign the temporaries to their respective variables.

Indeed, this sequence of operations is no different to the typical swap routine that you'd find in countless places, perhaps with the difference of explicit unrolling.

Input: CFG node n, offset o
Output: Label l
begin
  Let p be the position of n's instruction ;
  Let $q := p + o$;
  Assert that n has a successor with position q ;
  Let i be the instruction of that successor ;
  if *i is already associated with a label l* **then**
    return l ;
  end
  else
    Generate a new label l and associate i with it;
    return l;
  end
end

Procedure 4: makelabel(n, o)

Input: CFG C, DDG D, CFG node n
Output: Statement s
begin
  Let o be the offset encoded in the jump;
  Let $g :=$ `goto`(`makelabel`(n, o));
  if *n is a conditional jump* **then**
    Let d be the single operator node for n;
    return `if`(`expression`$(d), g$);
  end
  else
    return g;
  end
end

Algorithm 5: Algorithm for translating unconditional jump instructions, such as `goto`, as well as unconditional jumps, such as `ifeq`, `ifneq`, etc.

Input: CFG C, DDG D, CFG node n
Output: Statement s
begin
  Let d be the default target of the switch;
  Let x be "stack[i]", where i is the number of stack nodes at n;
  Let $s :=$ `switch`(x) ;
  if *n is a table switch* **then**
    Let i, j be lower and upper bound of the table;
    for $i \leq k < j$ **do**
      Let o be the $k - i$th target offset of n ;
      Add `case`$(k,$ `goto`$($`makelabel`$(n, o)))$ to s;
    end
  end
  else if *n is a lookup switch* **then**
    foreach *k, o in n's lookup table* **do**
      Add `case`$(k,$ `goto`$($`makelabel`$(n, o)))$ to s;
    end
  end
  else
    There is no such switch;
  end
  Add `default`$($`goto`$($`makelabel`$(n, d)))$ to s;
  return s;
end

Algorithm 6: Algorithm for translating switch instructions. Note that the generation of x is not an off-by-one error. As the instruction at n pops x from the stack, the DDG, which always encodes results, no longer references it, but it is still valid at exactly this point.

4.2.2. Exception handling

In order to handle exceptions, the exception handler needs to

- have its bytecode translated, and
- be registered in some way or another.

Translating the bytecode is easy, as all exception handlers are in fact part of the bytecode of the method they belong to. In order to ensure that their instructions are reachable from the CFG, a root has already been added, which points at its first instruction.

To register exception handlers recall the discussion of exception frames in Section 3.2. To comply with this specification, one has to generate the `ENTER_EXCEPTION_RANGE` and `EXIT_EXCEPTION_RANGE` "macro calls". For this purpose, two classes of bytecodes need distinction:

1. those which affect control flow within the current method, and
2. those that do not.

For the latter, it suffices to emit one set of `ENTER` macros before the translated instruction code and one set of `EXIT` macros afterwards. In case of the former, this cannot be done, as the code after the branch will not be reached before the next instruction, if at all. Both `ENTER` and `EXIT` macros must come before it, this time first exiting the old range(s) and then entering the new one(s). In addition, an instruction may end up jumping directly to the start of (a) new exception range(s). In this case, the corresponding `ENTER` macros are already contained behind the label it jumps to and should therefore not be emitted. This can be achieved by checking that the target instruction position is strictly greater than the start and less than or equal to the end of a given exception range.

To accurately handle returns, the code is blown up a little. First, a new variable `ret` is introduced, and when there is a valued return, it is assigned to the value of whatever variable would be returned. Next, all open exception ranges are exited and finally `ret` is returned from the function.

With this, all bytecodes are now covered, but not yet all execution flows. Just like code generation was implemented without thinking about exceptions before, here exceptions are implemented without thinking about exceptions. Specifically the case of multiple exception handlers for the same exception range remains to be handled. This happens when a `try` block has more than one handler and requires some extra magic when gluing together the code. When a certain exception handler is reached, it needs to exit all exception handlers that correspond to the same range and have not yet been exited. The final code then looks like Listing 4.1. This strategy makes no assumptions about the code other than the assumptions about exception handling that are already given. Most importantly the original bytecode may jump freely into handler code on its own without interference.

```
exception_handler_1:
  goto L1234;
exception_handler_2:
  EXIT_EXCEPTION_RANGE(X);
  goto L1234;
exception_handler_3:
  EXIT_EXCEPTION_RANGE(Y);
  EXIT_EXCEPTION_RANGE(Z);
  goto L1234;
L1234:
/* handler code */
```

Listing 4.1: Example of multiple exception handlers in different ranges pointing to the same code block.

4.3. Optimizations

As a matter of principle, I regard compilers – especially compilers for the C language – as smarter than myself when it comes to optimizing C code. This naturally also holds when tasked to output particularly efficient C code. Nevertheless, there are situations in which the heuristics employed by the C compiler do not suffice, or the C compiler is unable to understand what powers really are at play in a given situation due to lacking metadata. The following discusses a few ways of tweaking the compiler (and runtime) in hope for better performance, code size, or some other nice property.

4.3.1. Caches

Oftentimes in computing, instructions get repeated… a lot. At least that is what one would think from a fairly high-level perspective, if one was e.g. looking at the source code of a program[5] or at an even higher level at the things people use their computers for. In practice, there are ways to mitigate this repetition, and one of these ways – conveniently the being discussed here – are caches. To give an example, there are many pages on the web, some of which are very popular and thus accessed quite frequently. If the host of such a page needed to go through all the steps required to computing it each and every time, it would be quite a load as well as quite a loading time for the user. To keep this from happening, there are large numbers of caches, from the ones embedded in browsers for the sake of user experience to the ones employed by the hoster mostly for load balancing, but maybe also user experience, as having bad user experience would otherwise negatively affect the site's popularity.

Originally, (hardware) caches were a way of making up for the increasing performance gap between accessing data in a register and loading or storing the same data to the

[5]Even without outright repetitive code of the "copy & paste" variant, specific functions will be called way more often than others.

main memory of a computer. Similar ideas have also been applied elsewhere, as indicated by aforementioned web caches. Applying caches to the idea of programming itself also yields interesting results. In functional programming, the application of a function is considered the same as its result. Knowing that, an interpreter might want to remember both function call and result for some function that takes long to compute, so that if the same function is called again with the same arguments, the result can be returned more quickly. The real function is effectively only called once. Returning from this fantasy lambda world to the real one with only bare metal and C, we find that compilers use similar reasoning for optimizations, if they know functions to be pure. GCC actually has two attributes with similar semantics, the first being *pure*, meaning that a function has no visible side effects and the second being *const*, implying *pure* and also stating that no variables other than those passed to the function are used to compute its return value [30]. In the programs handled by this compiler, there exist two classes of functions, which might fit the criteria of the former, but also might not[6]. Since one can not simply declare them to be what they are, and the C compiler lacks the information it would need to check whether underlying resources *did* change, a potential opportunity for optimizations will be missed. Caches can (partially) recreate the effects of some of these optimizations.[7]

For this compiler, a cache is a variable that stores the value read from or written to specific places via references to objects "belonging to the JVM" that is specifically arrays and fields, both of which will also be mentioned separately later. Additional nodes are inserted into the DDG, which we dub *cache nodes* (short: caches) and connect the value as an ingoing edge. For the sake of brevity and also clarity, cache nodes for read accesses and write accesses are referred to as *read caches* and *write caches* respectively, and to the specific access of a given cache as *their access*. We say that a cache is *active*, if at a given point in the CFG, their value is both known and known to be correct. That is, their access must happen before the instruction at that point is executed, and the underlying memory must not possibly[8] have been changed since. We also say that two caches (potentially) *share data* if a specific subset of the inputs to their accesses are (potentially) the same. This is done by checking for equivalence in the DDG (see Appendix A).

A read cache may be replaced by another cache, if the other cache is active and they share data. Finding such a cache is a two-step process. First, a list of active caches for a given DDG node is computed, then one can check whether they share data with the cache in question.

[6]The GCC manual mentions "functions, which access system resources that might change between calls" as "interesting non-pure functions". In our architecture, arrays and fields are considered system resources, which sure enough might change between two calls depending on what other functions are called.

[7]GCC performs both common subexpression elimination and loop optimizations. With this admittedly limited approach, one can do some of the former.

[8]Thorough analysis of whether the underlying data actually changed might be quite complicated when taking function calls into account, touching the realm of symbolic execution. This is beyond the scope of this thesis. Instead, all caches will as of currently assume such invocations to be destructive (see their entries).

The list of active caches per CFG node can be built incrementally. A cache is active, if it is it was active in all predecessors and is not overwritten in the current instruction. A cache becomes active, if its access is being made in that instruction. Given that information, a map of CFG nodes to list of active caches can be created by visiting the CFG.

This leads to an easy implementation of caches. First, caches are created for all accesses, then as many accesses as possible are replaced with cached ones. Finally, caches that are no longer used (replaced caches) or only used once (i.e. only at the point of insertion, not in any replacement) are deleted. The details for the specific kinds of caches handled by the compiler, i.e. when they share data and when they are overwritten, will now be discussed.

Field caches

A cached non-static field read is shown in Figure 4.4 and a cached non-static field write in Figure 4.5. Static accesses appear similar, but lack the first argument, which in a non-static access is the object whose field is being accessed.

Two static field caches share data, when their access involves the same field, non-static caches require their access to involve both the same object and the same field to share data. Fields are potentially overwritten at function invocations (i.e. all `INVOKE` bytecodes) and if a write potentially shares data with an active cache.

Array caches

Array caches are structurally similar to field caches, but with different argument placement and semantics. Read and write accesses are shown in Figure 4.6 and Figure 4.7 respectively.

Two array caches share data, when their access involves the same array at the same index. Array caches are potentially overwritten by a write to potentially shared data or function invocations.

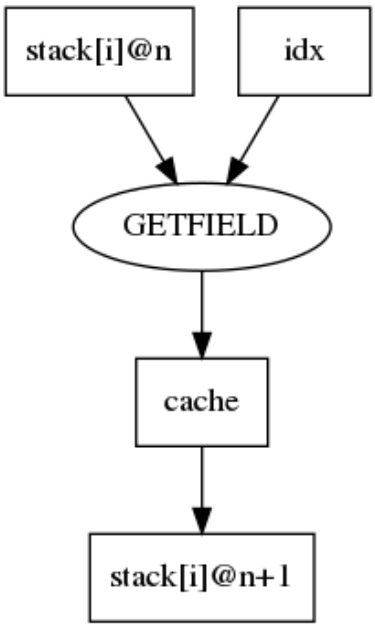

Figure 4.4.: Structure of a cached non-static field read in the DDG "stack[i]@n" is the object being accessed, "idx" the field, which is encoded in the instruction as an index to the Constant Pool, making it a unique `short` per field, and "stack[i]@n+1" is the result.

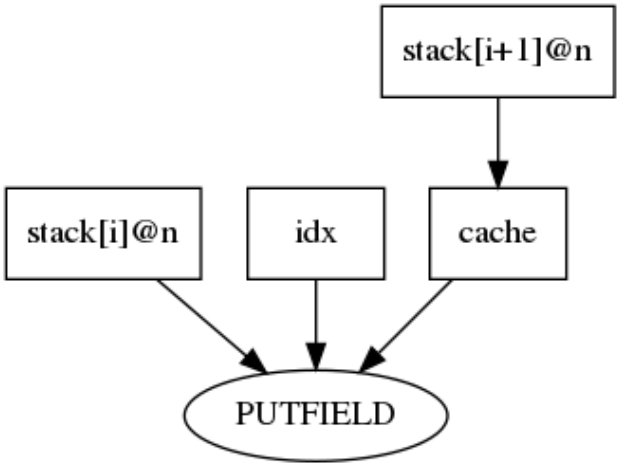

Figure 4.5.: Structure of a cached non-static field write in the DDG. The arguments "stack[i]@n" and "idx" are just as in read accesses, while "stack[i+1]@n" is the value.

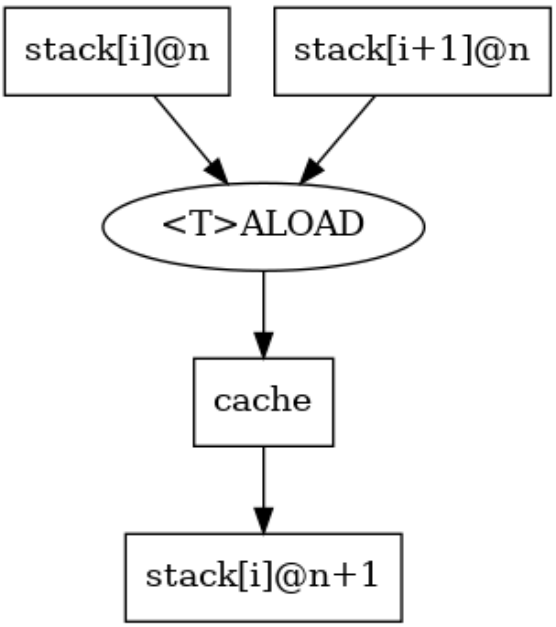

Figure 4.6.: Structure of a cached array read in the DDG. "stack[i]@n" is the array being accessed, "stack[i+1]@n" is the index at which it is accessed, and "stack[i]@n+1" is the result.

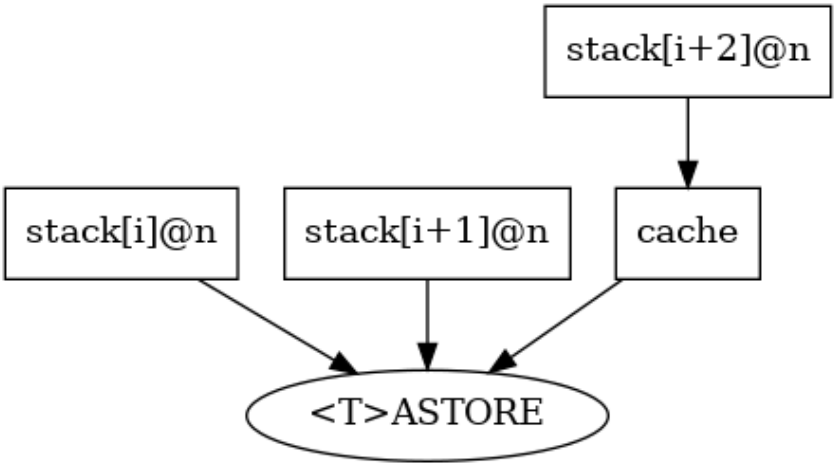

Figure 4.7.: Structure of a cached array write in the DDG. "stack[i]@n" and "stack[i+1]@n" are array and index, as with read accesses, "stack[i+2]@n" is the value.

4.3.2. Function outlining

As pointed out back in Section 3.2, an additional parameter is added to all `INVOKE` bytecodes when turning them into system calls to have a clear way of passing around arguments. However, it turns out that adding said argument to a function call quickly blows up the size of the assembly outputted by the C compiler, as shifting around arguments passed by registers is anything but elegant. The same holds for field accesses as well, and even with array accesses an additional argument can be saved when the array or index is statically known.

Compilers which optimize for speed will try to inline functions (i.e. expand their code with respect to arguments at a given point) if they deem the cost of doing so to be less than the cost of invoking the function. In contrast, compilers which optimize for size will search for common code snippets to outline, i.e. to make a separate function of, even if the programmer didn't explicitly ask for it. In either case, the compiler will leave the code as-is in cases where it is not too sure about the way a transformation affects its output, and it seems the C compiler is largely indifferent to the code bloat introduced by the compiler of this thesis. We wish to rectify that by outlining those functions inside the compiler itself.

In order to do so, several things need to be done. Firstly, code must be generated for the outlined function – both method header and body – and secondly said code needs to be invoked. The latter is not so difficult, one simply overrides the `expression` procedure, so that the outlined function is called rather than the generic macro. The former can also be done lazily – in fact at the same time as the latter – as one just needs to create a function call and perhaps wrap it in a return statement to create the entire method body. In the method header, arguments that are still needed will be retained, and those that can be done without are dropped and instead added to the name of the function in some format that avoids naming conflicts. For instance, a `BALOAD` with a constant index of `42` turns into `BALOAD_42`. To avoid duplicates, these functions are stored in a map, which can be considered part of the symbol table[9], with the C name of the function as key and a *stub* as value. This stub can act as a method signature, which is of minor importance now, but somewhat relevant when we want to write out the function. More importantly right now, these stubs are used to create the overrides for the `expression` procedure mentioned before, which are given in Table 4.2.

node type	inputs	expression	notes
array read	a, i	`FunctionCall`(s, a)	s encodes the offset
array write	a, i, v	`FunctionCall`(s, a, v)	s encodes the offset
field read	none	`FunctionCall`(s)	s encodes the field name
field read	o	`FunctionCall`(s, o)	s encodes the field name
field write	v	`FunctionCall`(s, v)	s encodes the field name
field write	o, v	`FunctionCall`(s, o, v)	s encodes the field name

Continued on next page

[9] Yes, this compiler has a decentralized symbol table. No, it has not lead to conflicts… yet.

Continued from previous page

node type	inputs	expression	notes
function call	$a...$	`FunctionCall`$(s, a...)$	s encodes the original function signature
interface function call	$a...$	`FunctionCall`$(s, a...)$	s encodes the original function signature

Table 4.2.: Stubs for outlined functions. In all of these function calls, s is the name of the stub to be generated as a C function.

Now we have almost everything to outline functions. All that remains to be done is for the compiler to write out the actual function definition at some point before it is used. However, this puts us in a somewhat odd position. The method-based approach that has been used so far expected a 1:1 mapping of Java methods to C functions and can thus simply be implemented by "returning" the C function definition. This 1:n mapping on the other hand requires the introduction of an event queue. Treating the creation of functions as events, one event is queued for each new outlined function. The compiler can then first empty that queue before printing the function that uses any definition within it, and the C compiler will signal no error.

4.3.3. Function call re-routing

One quite obvious difference between C and Java is the difference between a *function* and a *method*. In C, a function is little more than a name and a pointer to some machine code underneath said name. In Java, a method is a signature consisting of name and parameters, which at runtime turns into an index by which to lookup the code for said method in a table. Since the function pointer in the latter case is generally speaking not obvious until the method is invoked at runtime, the way these two languages handle invocations is quite different. Method invocations are – until now – handled by simply asking the JVM to look up the correct method and call it. This is necessary for functions implemented in other applets and also required for functions we've translated, as method overriding might cause a function to be invoked that is not the same as the one specified in the bytecode. If we're compiling a library, a public function may even be overridden in a subclass from another library or an application.

As anyone should be able to tell after a primer on object-oriented programming, Java methods are actually collections of functions that may be called, instead of one function pointer that will be called. While Java programs could certainly be written in a way that a 1:1 relation of method to function can be established, being able to *not* do that is considered a feature, and we will not make ourselves many friends by imposing such a restriction. However, there is a range of *effectively final* methods, i.e. methods which will not resolve to overrides upon invocation, and identifying those might already lessen the need of runtime lookups. These include of course `final` methods, i.e. methods declared final through the keyword with the same name, but also `static` and `private` ones as well. In fact, even the Java Card spec is somewhat aware of optimizations

that can be made if the function that is to be called at runtime is statically known. In Java Card bytecode, *static method references* are simply an offset into the bytecode pointing at the header of said method.[10] Using a similar approach, an effectively final method can be called directly if its name as a C function is known – which it is during header creation. In order to make this information available during the `method` part of the build flow (see Figure 3.5), annotations are added to the headers created by `header` as comments. During compilation, this header is read and the annotations parsed to create a mapping from Java method signature to C function names. An override of the `expression` procedure then emits a call to the corresponding C function if one such effectively final method is encountered.

More generally, whoever invokes the compiler could decide to override certain procedures themselves, creating faster implementations of certain methods as part of the CRE – somewhat similar to accelerators. A similar set of annotations can manually be written into the header rather than automatically created, and may take an optional *fallback* argument As was done previously, a mapping is created from Java method signature to C function names, but a fallback function with the name specified in the fallback argument is also generated, so that the function may call it if the override is not applicable in certain situations. This process is almost identical to that of outlined functions, the only difference being that whoever writes the header also controls the name of the function.

4.3.4. Mapping arrays into C runtime

Going back to the background chapter, one might recall that this is by far not the first source-to-source compiler and also not the first FFI. One can look back to earlier compilers and FFIs in search for certain patterns, particularly ones that would not even require modification of the compiler. It is time to talk about *direct memory access*[11]. The JNI [24] documents direct access for arrays and strings, Guile's arrays directly map to C arrays [9] whereas strings can only be converted, and Ruby strings can also be directly accessed from C [17] whereas structs and arrays can not or at least should not be accessed directly. Guile and Ruby are especially interesting here, because they also allow direct access in the other direction, i.e. of C pointers via wrapper types in the respective language.

Having proven direct memory access to be a pattern by giving at least three examples, it is time to make use of it. In order to directly access the underlying data of an array, the OS needs to expose said data in some fashion to the CRE. Since the output of the compiler is generic and should at least in theory be able to run on any machine, system dependant hacks that would require it to be modified are not of interest here, though they may be an interesting idea for future work. Instead, one can modify the system call handlers and the macros defined in Section 3.2 for the bytecode instructions that are already translated to make use of direct memory access. Within a system call handler

[10] Ironically, not even Java Card is consistent with their terminology here. Static method references are used for static methods, private methods and constructors.

[11] No, not *that* direct memory access, which allows components other than the CPU access to the physical memory, although it is similar in principle.

the OS may decide to map an array into the runtime and if it does, the mapping will be recorded in an array on some well-known address. The macros in turn resolve to functions, which first check whether the array is mapped into the runtime by iterating over this array and directly accesses it if so. Otherwise they call the corresponding handler.

From the perspective of the generated C code, the access is thus still indirect. However, this is just as intended, because it allows the OS or CRE to freely unmap/remap arrays between calls. This might become necessary, if for instance a call is made to a function in another library – were this data to remain mapped during this call, it would otherwise be leaked to a potential attacker.

4.3.5. Contracting the DDG

At the very beginning of Section 4.2.1, after having defined the `expression` procedure, a generic way of dealing with almost any "basic" bytecode was constructed. It was also mentioned that this would later allow certain contractions fo the data flow graph. People with a background in graph theory will notice that this results in the construction of a *minor* of the DDG. However, this does not sound very flashy as an optimization technique. Instead, we call the software component that performs this operation, the `GraphReducer`, as it reduces the number of nodes in a graph (specifically the DDG).[12]

As you might imagine after reading Section 4.1.2, many gratuitous nodes are inserted into the data flow graph, the most obvious offender being variable assignments of the kind `stack[x] = stack[x]` or `local[x] = local[x]`. It turns out that all such assignments can be dropped without any change in semantics whatsoever until a node is hit with assignments from multiple sources, i.e. one of the nodes specifically inserted for branch targets, also called a "branch marker" for short.

More aggressively, one can (partially) eliminate the stack. Again, branch markers serve as limitation, but assignments can be delayed until they hit a marker, local or special node (i.e. function calls, arrays, fields, returns, etc.). This optimization may cause assignments to occur in control flow statements, which is not ideal, because the code generator ignores them as of now. To fix this, a round of assignments à la Algorithm 3 is inserted, in which only stack nodes need to be taken care of.

At the end of this, simple expressions like `return (a + b) / 2` can be reconstructed to the quite similar `return (locals[a] + locals[b]) / 2`, with `a` and `b` in the latter being the local indices that those variables were translated to during compilation. The ternary operator is not handled well, because it has no equivalent on bytecode level – instead it translates to a normal jump.

Speaking of jumps, they serve as a hard synchronisation barrier until now, when this might not be needed. Consider for instance the procedure `max` shown in Listing 4.2. With the current reduction, the `GraphReducer` manages to produce code similar to Listing 4.3. Don't worry about the inequality operator changing its sign that happens outside of our influence. More important is the fact that locals still need to be assigned to even though

[12]The name `GraphReducer` is my own invention and has not been approved by the marketing team of NXP Semiconductors Austria GmbH und Co KG, nor do I seek approval.

they are always taken as argument values. If at this point the C compiler stopped caring about the value being stored in `locals`, it would produce vastly inferior code. One can actually propagate arguments further than that, as long as one makes sure that they are the only value that the local takes in a given branch, see Listing 4.4. In a similar fashion, one can try to propagate constants and variables through branches, but not other operations, as that could lead to duplication of side effects.

```
short max (short a, short b)
{
  return a > b ? a : b;
}
```

Listing 4.2: Simple Java code for the maximum operator.

```
short max (short arg0, short arg1)
{
    short stack[2];
    short locals[2];
    locals[0] = arg0;
    locals[1] = arg1;
    if (locals[0] <= locals[1])
        goto L1;
    stack[0] = locals[0];
    goto L2;
  L1:
    stack[0] = locals[1];
  L2:
    return stack[0];
}
```

Listing 4.3: `max` after translation to C with stack elimination.

```
short max (short arg0, short arg1)
{
    short stack[2];
    if (arg0 <= arg1)
        goto L1;
    stack[0] = arg0;
    goto L2;
  L1:
    stack[0] = arg1;
  L2:
    return stack[0];
}
```

Listing 4.4: `max` after translation to C with stack elimination and argument propagation.

5. Evaluation

We evaluate the code size and performance of some applets when compiled for the mystery architecture X (henceforth MAX) provided by NXP Semiconductors Austria GmbH und Co KG. Testing is done on a sample specially created for that occasion in case of one applet, where it was made possible by NXP Semiconductors Austria GmbH und Co KG, and on a(n) field-programmable gate array (FPGA), which runs the code at exactly $\frac{1}{4}$th of the speed of the actual hardware, in the other cases. Whether actual hardware or the FPGA is used, does not matter, though, because execution times are compared in the relation to the time the same operation would take when not compiled.

5.1. Benefits and drawbacks of native code

In general, there are many advantages of native code, such as

- better control over memory management,
- better control of execution time,
- overall shorter execution times, AND
- interaction with specific hardware (either directly or through the OS).

Virtual machines usually claim the memory and have their own allocators for the objects they create. Depending on the virtual machine, a user may be able to freely allocate and deallocate space, but this is not the case in the JVM – while byte arrays of any size may be allocated at will, those arrays cannot be repurposed, e.g. for holding objects of varying kind.

The execution of code in a virtual machine follows a scheme that is similar to the one already performed by simple hardware. Put simply, the machine fetches an instruction (i.e. reads it from memory), executes it and adjusts the instruction pointer to point towards the next instruction. Doing this in code rather than relying on hardware creates a big runtime overhead – the execution of a simple bytecode instruction takes 20 to 30 native ones. Native code can be executed directly and thus does not have this overhead. This is the source of performance gain for every native compilation scheme.

Lastly, if a system offers specialized hardware of any kind, it must be possible to talk to that hardware through native code somehow. Virtual machine "drivers" for such hardware too require native code, which in many cases must also be adjusted to some type system. If the VM does not come equipped with such drivers, they need to be written by the user utilizing some FFI. However, it is very likely that native

applications and libraries are already built around this driver. Hence if an FFI is already a requirement, those libraries can be wrapped instead.

Of those benefits, this compiler can at best achieve shorter execution time. In terms of the memory model both it and more generally the CRE are bound to the restrictions imposed by the JVM and the OS, the former applying to objects, the latter applying to the stack and exception stack. As far as "control" over said execution time is concerned, there is little to none. Given the almost literal translation of bytecode to C code, there are few liberties to enforce certain timing behaviour, and one is instead subject to the whims of the JVM in the FFI sections and the underlying hardware everywhere else. Lastly, interaction with specific hardware would require function rerouting to implement and hence presupposes already existing bindings, which are exported to Java, meaning no functionality would be gained that was not already present before.

Native code also has several drawbacks. A big one is portability. While source code may be portable to an arbitrary extent – e.g. requiring only a POSIX-compatible system and a compiler capable of handling a specific language version – binaries are hardware and dependency-specific. We've taken some steps to ensure that generated C sources are portable through macro expansion – see Section 3.2 – but implicitly rely on some undefined behaviour to not be that undefined – see Section 5.5. As a result, code generated by the compiler of this thesis is certainly less portable than the bytecode it was compiled from. Memory management is also widely known to be error-prone. Thankfully, most data structures needing to be allocated by or for the compiled code, can be allocated on the stack, which is harmless for sufficiently large stacks or sufficiently small amounts of allocations – implying recursion limits among other things. However, the biggest drawback for resource-constrained devices such as smartcards is the increase of code size. This will be discussed in greater detail in Section 5.3.

5.2. Benefits and drawbacks of optimizations

The following sections give a quick theoretical view of the benefits and drawbacks that the implemented optimizations may have. As many of these implicitly rely on the C compiler backend and further the targeted architecture, your mileage may vary in multiple ways, but it still doesn't hurt to justify them on theoretical backgrounds before diving into the raw numbers that will be provided by Section 5.4.

5.2.1. Caches

The code size and performance costs of a cache can be described as $C = c_i + kc_a$ with c_i being the cost of initializing the cache once and c_a being the variable cost of reading the cache instead of performing the actual action. A cache only makes sense, if the cost of accessing it is less than the cost of performing the cached operation, or in other words if c_a is negative. A cache is overall beneficial if C is negative.

For code size, the value of C is measured in bytes, k is the amount of cache accesses visible in the code, and c_i may be

- the size of the code storing some value in an additional register, OR
- the cost of the code storing some value on the stack.

At the same time, c_a is the difference between the size of the code for a register/memory read against the size of the code for a system call with two to three arguments or – in case that outlined functions also need to be accounted for – a function call with up to two arguments. These should in the worst case break even, the worst case being a function call with the correct values already assigned to their registers. Otherwise, c_a is a small negative number like -2.

For performance, C is measured in seconds or a fraction thereof, and is determined by stacking the time required to make k register/stack accesses against the time to it takes for a system call that performs various checks before eventually returning some value that hopefully is still within the main memory. It is much clearer for this case that c_a is negative and that even a single access is beneficial.

Note that k in these two cases are not necessarily equal. There might only be a single cache, that is accessed within a loop, which may account or 10 or 100 read accesses or 0 depending on whether the correct branch was hit.

5.2.2. Function outlining

The theoretical benefits of grouping code blocks into functions are well known – in fact, programmers do it all the time to avoid errors that come with repeated typing of the same code block. As a result, the size of the source code is greatly reduced. Leaving compiler optimizations aside, the same also holds for any code produced from it. The otherwise repeated instructions become a single call instruction. When a compiler performs such a transformation on the programmer's behalf (in an attempt to optimize for size), we call it outlining. Within this thesis, the code being outlined consists of sequences of instructions that add one argument to the argument list of another function call.

Were we to use the `cdecl` calling convention, this would be a nearly pointless exercise. After all, pushing one argument in front of the list is a trivial operation – although to be fair it does still cost one additional operation instead of simply a call instruction and thus the optimization could still be beneficial. However, the MAX calling convention is one that passes the first n parameters through registers, as do many other calling conventions out there. Only in the code directly surrounding the `INVOKE<T>` system call, it is transformed into a stack-based calling convention as a way of handling variadic arguments[1], which means that any code calling system call wrappers still has to repack registers to make things fit.

In either case, the size cost is n times calling a function plus the definition of the function against n times the inlined function. Given that the function itself is by definition larger than a function call, the comparison of the two is almost always in the favour of

[1] `INVOKE<T>` as defined in Section 3.2 is of type (`short`, `short`, `short`…) → `short`, the system call handler is of type (`short`, `short`, `short*`) → `short`. This definition is used to support a stack and locals consisting of variables instead of arrays – a trivial distinction thanks to optimizations in the C compiler – but can also be exploited by the `GraphReducer`.

the outlined version. In the few cases that it is not, e.g. trivially $n = 1$, the C compiler has both a performance and size reason for inlining the outlined function and will thus very likely weed out any improper outlines.

5.2.3. Function call re-routing

Re-routed functions in the specialized case do not have any cost on their own, safe for perhaps the additional work that needs to be done in the compiler, which is not a concern. Instead, they are beneficial in terms of code size, since they eliminate the trouble of constructing a system call, and in the runtime none is performed. Code size *may* however take a hit, when re-routed and non-rerouted functions are mixed, since these conform to somewhat different calling conventions. In re-routed functions, the first argument is the first argument, non-rerouted functions may (if they are not outlined) however take the number of arguments as first argument and move the actual argument list to the second and so on. An optimizing compiler that detects this, may turn out better code for the case in which the second kind of call is made uniformly.

In the general case, the overhead of implementing another version of an already existing function is added, which is certainly less than ideal. (Recall the harsh words used towards Wang et al.'s method cloning.) Worse yet, if multiple packages all use the same optimization, that code is duplicated over and over. Function call re-routing seems to accidentally reintroduce a problem, that has once been solved by shared libraries, but until shared libraries are discovered once again, those are the side effects, that one must be prepared for. Then again, since this code is vendor-controlled, nothing stops them from using shared libraries as long as they find ways to map them into the CRE, or perhaps reuse code that is implemented in the OS with some function call. It should be noted that this does eliminate some of the benefits of this optimization, however. The more work is required to map a shared library into the CRE or worse yet call an optimized function rather the one that would normally be called, the less beneficial the rerouting will be. In other words, we have a typical time-memory tradeoff.

5.2.4. Mapping arrays into C runtime

While direct memory access can trivially be proven to be faster than indirect access, it is hard to make statements that are not very specific to some benchmark. There are also implementation-specific limitations to consider. Neither Java Card nor the compiler of this thesis knows such a thing as an explicit construction of an array view, whereas the JNI provides a large array of such functions (pun intended). Even if the compiler did know, it would require great amounts of speculation about repeated accesses – similar to the one done for caches, but needing to find much clearer patterns of repeated use – and additional code would need to be emitted to deal with the (de-)allocation of necessary data structures. Not only are those not implemented, they contradict the design goal of generating fairly generic code. In lieu of the compiler the OS and CRE manage the memory mappings. This comes at the cost of having the OS speculate, which array is going to be used or not – a speculation that is replaced in the particular implementation

tested here by the assumption that the array *will* be reused, hence always mapping the array into runtime.

Whether a mapping is produced or not, there is considerable overhead both in the decision of whether or not to map and in establishing the mapping itself. Furthermore, when switching between multiple libraries, one may have to unmap arrays when calling into another native library so as to not cause information leakage. As a result, while fast subsequent accesses to the same array are the goal and likely result of this optimization, it may end up inadvertently slowing down applets that had not asked for the optimization in the first place. A solution to this dilemma would again be an explicit API, which in turn comes at the cost of portability and additional compilation overhead.

5.3. Overhead

Generally speaking, one can expect code size to increase during the transformation from Java Card bytecode to machine code understood by the platform itself. This is to be expected, as machines are constructed in a way that caters to the language that is most likely used to write programs for it. In the case of the JVM (and the JCVM by extension) there are lots of bytecodes referring to *objects*, their *fields* and *methods*. To give another – although completely unrelated – example, the Guile VM has an even larger amount of bytecodes dedicated to various forms of calls (including tail calls) and checking the number of arguments to a procedure, as well as various bytecodes to deal with "basic" types such as numbers, lists, vectors or strings. CPU designers on the other hand expect code to be written in Assembly or C, maybe C++ or Rust, and are thus much more concerned with pointers. This holds especially for the smartcards Java Card is run on. Even though the main purpose is running Java code, a virtual machine implemented in C or C++ is still closer to the hardware, and this fact would not change if said virtual machine was implemented in another language. If one wanted to mitigate the size overhead caused by this catering towards a language through compilation alone, the compiler would have to translate paradigms into more suitable ones rather than simply reimplementing them. This may be a subtle difference, but not only do modern compilers lack the creativity for such a task, this kind of translation would likely not be acceptable when interoperability with particular pre-existing components is a requirement.

There are several specific sources of overhead worth addressing. First would be variable instruction size, seen not only in Java Card, but also many modern CPUs, specifically the x86 architectures. In Java Card bytecode, the shortest instruction – which hopefully for Shannon are the most common ones – have a size of 1. MAX on the other hand has a "fixed" instruction size of 2, with some pseudo-instructions affecting the way the next instruction is handled, essentially creating instructions of size 4. As often the exact same operations are done only with registers rather than a stack, code size is effectively doubled in such methods. Even more code is required to push values onto or pop values from the C stack. Interestingly, we will find later that many functions grow by a factor smaller than two. In other words, native functions – while being larger in terms of size – are smaller in terms of instructions. Put differently, it would

seem that individual instructions carry more meaning in maX than in Java Card. A similar phenomenon can be found when comparing natural languages with compounds (such as German) to those without. While some German sentences may very well be longer than their English translation in terms of characters, they may contain less words. For instance, the *Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz* (beef labeling supervision duties delegation law) has 63 characters as opposed to 47 (including spaces), but is one word instead of 6.

An additional factor would be glue code. The macros specified in Section 3.2 themselves often require functions to be implemented (or at least would benefit from being implemented as functions), which also need to be included in the resulting binary and are not immediately visible when inspecting the code of a single translated function. (This issue would somewhat be mitigated by shared libraries.) The same holds for some optimizations more or less. Last, but not least there are the entry points to the compiled applet with additional glue code. First would be the resolution of Java methods to C functions, which is implemented in the file ***main.c*** (see Section 3.3) as a large switch. The OS only provides a key for this lookup. In a similar fashion `catch` exists as entry point for exception handling. Provided with the thrown exception, this function needs to find the corresponding handler and "call" it, or otherwise return to the OS.

When optimizing a binary for size, link-time optimizations which can completely erase methods by pulling them into others, are often beneficial. However, this comes with the drawback of no longer being able to easily inspect that function by itself, which is important in the process of developing and evaluating other optimizations. For instance, the heavy cost associated with adding one parameter to each call, which inspired the function outlining optimization[2] would likely have gone unnoticed without a disassembly with debug information, which contained both the generated C code and the Java Card bytecode from which it was generated as comments.

5.4. Experimental Results

Various experiments have been made with test and real-world applets to back up the claims made earlier. We refer to them by using a prefix – "T" for test and "RWA" for real world applets – and a 1-based index.

The first test applet has two purposes. First, it tracks the development of the compiler by adding one testcase per feature and some for bugs. Second, it highlights specific factors, that contribute towards size overhead of compiled methods w.r.t. their Java Card counterparts. These should allow (perhaps limited) reasoning about the effects that compilation will have on larger applets, particularly those appearing in the real world.

For a better comparison, optimizations are grouped into sets. A set is either empty $\varnothing$ or a list of identifiers denoting a given optimization. The interpretation of these

[2] One additional idea we initially had was to actually encode this added parameter by literally inserting it after the system call instruction and having the affected system call adjust the program counter so that this dummy instruction would be skipped. This attempt failed at the code generation step for unclear reasons, presumably magic in the C compiler.

symbols is given in Table 5.1. We will compare $\varnothing$, (R), (R, C_A), (R, C_F), (R, O) and (R, C_A, C_F, O) to the Java version. Note that O obscures some of the size of a method by removing the size of the outlined functions completely. However, since outlined functions can be shared by all of the methods, counting them to any one of them would be even more dishonest than reporting a slightly smaller size.

Key	Meaning
$\varnothing$	Should mean "no optimizations", but actually does include the rerouting of effectively final methods. The flags for that are automatically added by the build system used to wrap compiler calls.
R	DDG contractions (`GraphReducer`).
C_A	Array cache.
C_F	Field cache.
O	Function outlining.

Table 5.1.: Optimization symbols and their meaning.

The first set of comparisons concerns methods relevant to the applet lifecycle, those methods being `<init>` – also known as the constructor –, `install` and `process`. The sizes for those methods can be seen in Table 5.2. It can be seen, that no optimization affects `install`, even though it calls `<init>`. This is because a constructor is effectively final and can thus be called directly. `<init>` uses some array and field operations, but only touches each of them once and thus is only affected by outlining. `process` does heavier work and apparently touches some fields at least twice, thus benefitting from the field cache.

Method	`<init>()V`	`install([BSB)V`	`process(<APDU>)V`
Java	151	24	746
Native $\varnothing$	328	40	1194
Native (R)	328	40	1178
Native (R, C_A)	328	40	1178
Native (R, C_F)	328	40	1122
Native (R, O)	296	40	950
Native (R, C_A, C_F, O)	296	40	910

Table 5.2.: Java Card runtime related functions and their sizes when compiled with different options. `<APDU>` is an abbreviation of `Ljavacard/framework/APDU;`.

The next set of comparisons concerns various variants of the Fibonacci microbenchmark. `fib` is the iterative version, `fibrec` the recursive, and `fibrec_s` is also the recursive version, but declared static. Their sizes can be seen in Table 5.3 and differences in their implementation are immediately obvious. `fibrec_s` never shrinks, because it is effectively final and the recursion is therefore as effective as it can be, `fibrec` is shrunk through outlining to the same size, supporting the claim about the cost of reordering arguments on MAX – note that a stub still needs to be generated, though – and `fib` is

only slightly improved through DDG reduction. A better version of this compiler could through reasoning about `this` also make `fibrec` as effective as `fibrec_s`, but such an optimization is not implemented.

Method	`fib(S)S`	`fibrec(S)S`	`fibrec_s(S)S`
Java	24	27	25
Native ∅	46	56	40
Native (R)	42	56	40
Native (R, C_A)	42	56	40
Native (R, C_F)	42	56	40
Native (R, O)	42	40	40
Native (R, C_A, C_F, O)	42	40	40

Table 5.3.: Variants of the Fibonacci microbenchmark and their sizes when compiled with different options.

Apart from the Fibonacci function, there are further mathematical benchmarks, shown in Table 5.4. `crc16_ISO3309` is an implementation of a cyclic redundancy check according to the mentioned ISO standard. Funnily enough, the generated C code is already better than the Java code it was generated from. The effects of outlining are negligible in this case – no other method uses these fields. `max0` is our idea of how to troll compiler developers. The normally straightforward implementation of `max` using the ternary operator is wrapped in an identity preserving sum (or in plain English, 0 is added to the result). Since the Java compiler performs no optimizations whatsoever, the C compiler reaches the point of not caring if no optimizations are performed by the compiler of this thesis. Luckily enough, contracting the DDG fixes this.

Method	`crc16_ISO3309(S[BS)S`	`max0(SS)S`
Java	139	11
Native ∅	136	28
Native (R)	136	10
Native (R, C_A)	136	10
Native (R, C_F)	136	10
Native (R, O)	132	10
Native (R, C_A, C_F, O)	132	10

Table 5.4.: Other mathematical functions. Note that `crc16_ISO3309` also initializes a lookup table on the first run and the code for that is inside the function itself.

Finally, some benchmarks are given for the operations on fields. Both `addMember` and `subtractMember` load two fields to perform the respective operations that is addition or subtraction upon them. The results are shown in Table 5.5 and the numbers speak for themselves – even with optimizations, they are still horrible.

While other test applets exist (e.g. one that mimics the “echo” command has been used as a proof of concept elsewhere), only the above was used for the purpose of

Method	`addMember()S`	`subtractMember()S`
Java	6	6
Native ∅	18	20
Native (R)	18	20
Native (R, C_A)	18	20
Native (R, C_F)	18	20
Native (R, O)	14	16
Native (R, C_A, C_F, O)	14	16

Table 5.5.: Trivial operations on fields.

Method	Java	Native worst	Native best	Ratio worst	Ratio best
`<init>`	151	328	296	2.1721854	1.9602649
`install`	24	40	40	1.6666667	1.6666667
`process`	746	1194	910	1.6005362	1.2198391
`fib`	24	46	42	1.9166667	1.75
`fibrec`	27	56	40	2.0740741	1.4814815
`fibrec_s`	25	40	40	1.6	1.6
`crc16`	139	136	132	0.97841727	0.94964029
`max0`	11	28	10	2.5454545	0.90909091
`addMember`	6	18	14	3	2.3333333
`subtractMember`	6	20	16	3.3333333	2.6666667

Table 5.6.: Summary of size comparisons on T1. As can be seen, the best results are achieved with mathematical operations and moderately good results when arrays are added to the mix. Object oriented programming primitives, such as field accesses, however, vastly increase the size of a method, even with optimizations.

benchmarking. All other results therefore come from real world applets, which are exactly the kinds of applets to be optimized.

The first real-world applet performs some file system operations interleaved with an AES-based authentication scheme as part of a larger transaction. The performance for one such transaction is compared in Table 5.7. As indicated in the table, the native version is 17% faster than the Java version overall. Performance is significantly improved in all major parts except the authentication, which might be an indicator, that it defers to an already existing accelerator or calls into some library to do the main work.

Command	Java [us]	Native [us]	Difference[ms]	Ratio Native:Java
1.select()	20926	19985	0.941	0.9550320176
2.read(A, 20)	25777	23363	2.414	0.9063506226
3.auth(K_A)	207216	206706	0.51	0.9975388001
	53497	52803	0.694	0.9870273099
4.read(B, 32)	174357	138450	35.907	0.7940604622
5.read(C, 12)	143754	123501	20.253	0.8591134855
6.read(D, 37)	167003	126738	40.265	0.7588965468
7.read(E, 32)	162478	126162	36.316	0.7764866628
8.read(F, 35)	165292	126448	38.844	0.764997701
9.read(G, 35)	165217	126511	38.706	0.7657262872
10.read(H, 48)	177179	128026	49.153	0.7225799897
11.read(I, 13)	145046	123988	21.058	0.8548184714
12.read(J, 24)	155126	125203	29.923	0.8071051919
13.read(K, 13)	145081	123965	21.116	0.854453719
14.auth(K_B)	199313	198300	1.013	0.9949175418
	53572	52816	0.756	0.9858881505
15.write(L, 35)	195373	149835	45.538	0.7669176396
16.write(G, 35)	184547	139080	45.467	0.7536291568
17.write(D, 37)	186445	139262	47.183	0.746933412
18.write(C, 12)	163303	136480	26.823	0.8357470469
19.write(B, 32)	183711	141292	42.419	0.7690992918
20.commit()	213839	201358	12.481	0.941633659
sum [ms]	**3288.052**	**2730.272**	**557.78**	**0.8303615636**

Table 5.7.: A single transaction in RWA1. Note that the sum only carries commands 3 to 20, as the first two are only required once.

The second real-word applet is a large banking applet that was heavily abused for feature testing in the initial phase. In fact, it is so large that it requires all of size-related optimizations to fit on the card with all features implemented. Most test results for this applet are from an earlier stage of development, in which not all features were available. For instance, this applet has some exception handlers that greatly reduce its size were they to be dropped from the binary (as they were before exceptions were implemented). These results are therefore of little meaning. Once the compiler was

fully implemented, however, this applet could be tested on real hardware, producing the results in Table 5.8. As can be seen in the table, the room for improvement is rather small, because the applet relies strongly on other components (it is barely a fifth of the total runtime). Still, its performance was decently improved.

	Time [ms]	Difference	Ratio
Java	108.56	0	1
(applet)	20		
Native	100.74	7.82	0.928
Native + short accessors	99.32	9.24	0.915

Table 5.8.: A single test run of RWA2. The Java version is compared against two native versions: one with all optimizations applicable to just the applet and with arrays mapped into the C runtime, and one where additionally the short accessors for byte arrays in `javacard.util` have been rerouted to an optimized version.

The third real-world applet is intended for the long-time storage of data and readout at request. There are two "phases", which can be distinguished in this applet: personalization or *perso*, in which data is written, and *read* in which data is only read. Note that perso also carries out a read to check whether the data is actually written as expected. The comparison of the applets with respect to these phases are done in Table 5.9. As can be seen, performance only improves slightly – by less than 5% to be exact – which was expected to be the case due to the applet's heavy use of accelerators.

Phase	Java [ms]	Native [ms]	Difference [ms]	Ratio Native:Java
Perso	3886.48641975	3800.68119975	85.80522	0.9779221614
Read	818.5	784	34.5	0.9578497251

Table 5.9.: Comparison of Java and native versions of RWA3 with corrected values.

The size overhead compilation induces on RWA1 and RWA2 is also evaluated. Both grow to about 150% of their original size, but at least in the case of RWA2, this overhead can be dropped to 15% through partial compilation.[3]

Size results for both testing and real world applets are summarized in Figure 5.1 and performance results in Figure 5.2. As can be seen from either the figures or the tables, the best performance result is a runtime reduction of roughly 17%. Compared to Gressl's results, who reports a runtime reduction of 27% with their reference applet, these improvements might seem rather disappointing. However, while this approach has an expected size increase of roughly 150% on average and up to 300% in exceptionally bad situations, their reference applet grows beyond three times in size, so one can still consider this contribution a success.

[3]This number might change w.r.t. to the way methods are filtered. For this test, any method invoked after the personalization phase was kept and the rest dropped. It may very well be that this does not include all possible post-perso commands.

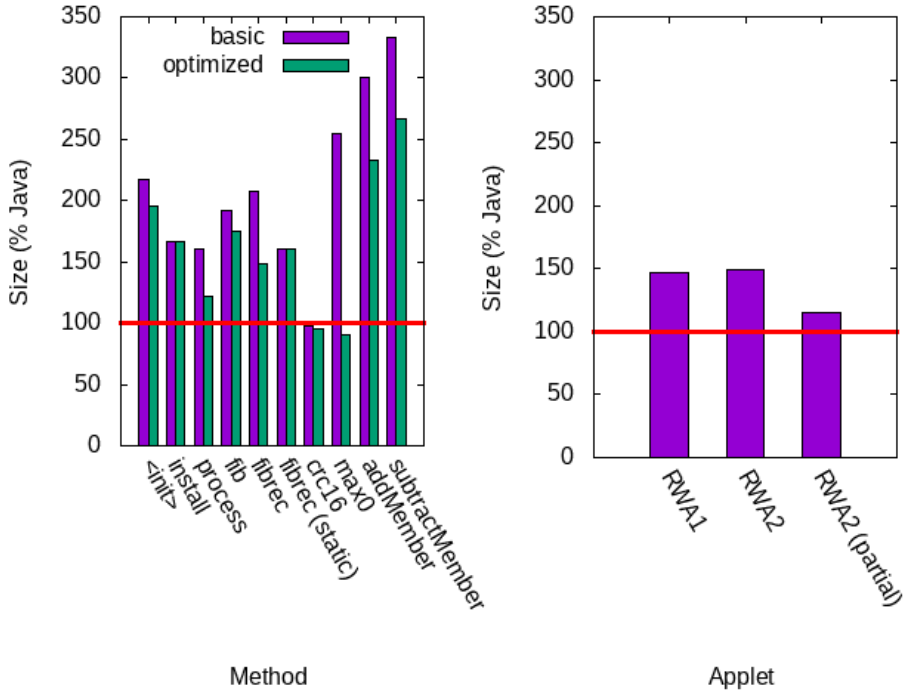

Figure 5.1.: Summary of size results. On the left individual functions are measured without additional overhead, on the right total applet size is measured without considering individual functions.

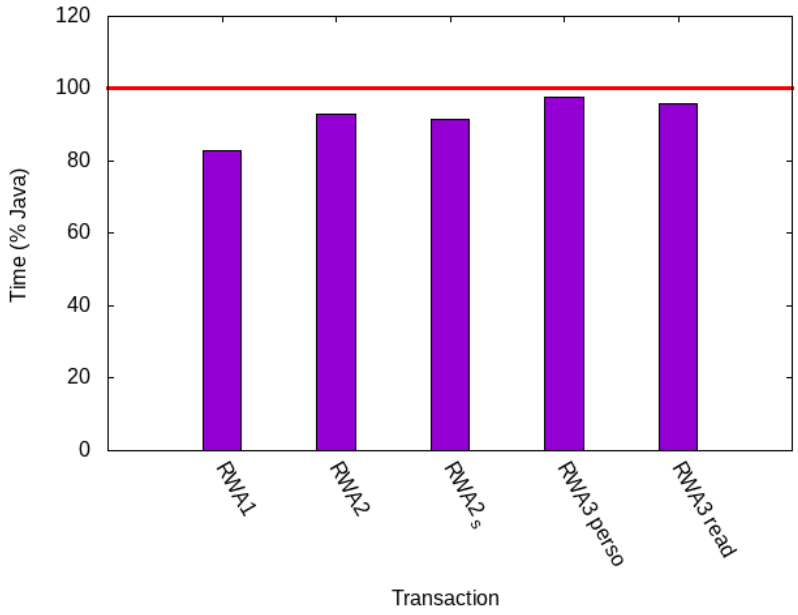

Figure 5.2.: Results of the performance evaluation.

5.5. Limitations

Nothing is without limits, but this compiler has so many, that they can be split into three categories. Firstly, there are compatibility limitations with regards to the applets that can be compiled and run as well as platform requirements. Second, the compiler lacks certain features that a "decent" (for some value of decent) compiler would have. And last but not least, there are bugs.

5.5.1. Compatibility

It goes without saying that enabling native code for applets, rather than just system libraries at the vendor's discretion is not quite conforming to the Java Card standard. However, there are further issues to be pointed out.

- The supporting infrastructure of the compiler was written with Java Card specifications of the version 3.0.5 in mind. As of January 2019, Oracle [23] has published a new version, which among other things updates the CAP file format. At the time of writing, this format is not yet supported.
- While not explicitly stated in the rest of this document, a few of the algorithms discussed in Section 4.1.2 and Section 4.2.1 implicitly assume `short` (equivalently `int16_t`) to be the only or largest data type that is to be handled, among others specifically
  - Algorithm 3 assumes this to be the case, as both the stack and locals are `short` arrays. Conversions to `byte` are implicitly followed by a conversion back to `short`, just like the JVM would do it.
  - Algorithm 6 assumes this to be the case, as it only regards `stack[i]` for some i, which itself is a `short` value.

  This of course makes the current version of the compiler incompatible with applets that require the 32-bit extension of Java Card.
- Regarding the hardware the code is supposed to run on, there are additional requirements.
  - To achieve a separation between JRE and CRE, their memories need to be separated as well. In other words, virtual memory is required.
  - To achieve a separation of runtime environments and resources, the OS needs to be running with different privileges than said environments, specifically the privilege to access such resources.
- As the original bytecode of compiled methods and constant pool indices inlined into the generated code, the `RefLocationComponent` (or Reference Location Component, whichever you prefer), no longer holds *meaningful* values after the transformation. In fact, they are cleared even if some methods are retained, so as to not create a false sense of "correctness". In any case, the contents of this component

are not to be trusted once native code is present. In a similar manner, the contents of the `ConstantPoolComponent` shall not be "optimized" after compilation, as that would break the generated code.

- Java Card allows vendors to add their own bytecodes to the machine. Such extensions are not handled by this compiler.
- As Pizlo et al. [25] note, directly translating Java Card bytecode to C code can cause problems due to undefined behaviour. For instance, signed integer overflow in additions is undefined (but usually causes wraparound), whereas in Java wraparound is required. Unlike them, we are not cautious about these cases, since most non-exotic compilers do tend towards the same behaviour as Java. In cases, where this does not hold, users may have to alter the compiler to emit safe instructions or – if barred the ability to do so – manually or automated through scripts alter the resulting C code.

5.5.2. Lacking Features

Rather than "Future Work", this is work that would under normal circumstances be part of a compiler, but aren't included in this compiler for reasons that should perhaps not be questioned too strongly.

- The compiler currently lacks type checking, and what checks are made can be disabled for "optimization".
  - Neither is any analysis on class relations performed. Instead, objects are all conflated to a single type.
- The compiler also doesn't check for compliance of the bytecode with any requirement that may be mandated by Java Card. This check (as well as type checks) are to be done by a bytecode verifier instead.
- The compiler does not understand the concept of a "pure" function and thus also does not infer purity, making some optimizations more conservative than they need to be.

5.5.3. Known Bugs

All software contains bugs, and this compiler is no exception. This section points out notable ones.

- Equivalence checks in the DDG confuse "maybe" and "no" with each other (for more details, see Appendix A, which documents how it *should* be, rather than how it is). This is not a "big" deal, since the two are often conflated later on, but it may end up causing troubles sometime. However, there are also instances, in which "maybe" is interpreted as "potentially yes", which might cause significantly more problems for the poor soul who wants to fix this bug.

- The DDG reduction as described in Section 4.3.5 actually led to a strange bug with some programs, in which branches would be translated wrongly. We still don't fully understand this bug, but rather circumvented it by not replacing stack nodes which end in a decision node. This detail is hidden from the reader in the given examples to make them easier understandable.
- The DDG reduction is also oblivious to exception ranges and may move some operations into or out from them as a result. This is usually not a problem, since applets tend to be written (and compiled) in a way that storage of the result of an operation (if any) is still within the same range(s) as that operation.
- The Java Card OS used for evaluation currently ignores some timeouts while in the CRE, so they had to be reconfigure. The exact issue will likely not be replicated by other systems, though there may be other causes for the same results, including, but not limited to, the deliberate deactivation of such timeouts when the CRE can not be preempted. (The OS in question **should** be able to preempt the CRE, however, so it seems that timeouts are merely ignored by accident.)

5.6. Security Considerations

As already noted at the very start of this thesis, letting arbitrary C code run freely on a smartcard is a potential security risk. In this section, a few ways are listed by which the compiler or supporting architecture might be abused for various attacks. Most of these attacks are purely hypothetical, as the platform-specific resources required to even compile an applet for their smartcards, let alone run it on them, are handled by the vendor, who would only shoot themselves into their knees, if they maliciously introduced bugs in their own hardware based on the writings below.

5.6.1. Type confusion

As the compiler sees everything as `short` (see Section 5.5), it would certainly be possible to carry out some of the attacks described by Mostowski et al.[19] at least partly. The compiler would happily emit code, which unsafely casts an object or array to any other type. However, when it comes to using the object as one of the new type – whether it is from the JRE or CRE no longer matters at this point – both have to pass through a barrier (the OS) to access the data of this object. Ideally, the checks within this barrier should be sufficient to detect such malicious behaviour, in other words, the operating system is aware of Java Card data types and (correctly) checks an object's real type.

5.6.2. Object reference manifacturing

In a similar fashion to the previous attack, one might cast arbitrary shorts to objects. Since an "object" on the Java Card stack is little more than an index in some hidden list, with said indices very likely simply increasing as more objects get created, an attacking applet can try to cast a range of values to references of a given type (using `checkcast`

or `instanceof`). An attacker who knows the order in which objects are created by their victim applet, might also after obtaining just one object from said applet infer the references of the rest, and so on, and so forth. Such casts of shorts to objects are obviously not allowed in the Java language and they might also be detected by the VM depending on how it is implemented. Within C code however, all type information that could be used to prevent this part of the attack is lost, forcing the system to treat the shorts as if they were genuine objects.

It turns out, however that the potential harm of this attack is minuscule in the presence of the Java Card applet firewall[21, pp. 43–48]. As specifically outlined in [21, 6.1.4. Object Access], any usage of the bytecodes

> `getfield`, `putfield`, `invokevirtual`, `invokeinterface`, `athrow`, `<T>aload`, `<T>astore`, `arraylength`, `checkcast` [and] `instanceof`

shall only be performed if the "owning context [of the object] is the currently active context". Assuming that this firewall is implemented correctly, the attack mentioned above will always fail with a `SecurityException` unless the applet would already be allowed to access the object. This not only prevents an attacker from illegally accessing objects, but it might as well prevent them from learning anything about the underlying object that they do not already know, unless there was an additional side channel, e.g. the exception is thrown faster in the case of arrays or nonexistent objects.

Of course, for this to work, one has to assume that implementing the applet firewall is still possible with all extensions. However, only one comes to mind that could cause potential problems, said extension being the mapping of arrays (or objects generally for that matter) into the runtime as described in Section 4.3.4. That being said, it would not be unreasonable to simply unmap the corresponding objects if another context becomes owner of the object.

5.6.3. Arbitrary Code Execution

Function rerouting in its more general version (Section 4.3.3) allows, nay, requires arbitrary code execution to work at all. As such, an attacker could write a malicious function replacing some generic function that is likely to be called, like `javacard.util.getShort` or something within `javacard.framework` and perform anything they want with it. Due to the separation of runtimes and applets, the effects are limited to whatever can be done with Java Card bytecode on a given card. However, this does not exclude malicious behaviour, as an applet could certainly read one of its own secret keys into some byte array and share that array with another applet or send it over an application protocol data unit (APDU).

Since this is the *intended* use of this feature, the only way around its problems would be not using it. This decision ultimately lies with the one who compiles the binary, i.e. the vendor, who would already be able to inject arbitrary Java Card bytecode into any of the applets running on their smartcard. As trust in the vendor – unfounded though it may be – would first need to be established before handing any applet to them, this is hardly a real attack surface.

In a similar manner, the loading mechanism that a smartcard would need to have to load extended CAP files, allows for some degree of arbitrary native code execution. In the setting used for evaluation, this is mitigated by preloading the applet onto the smartcard – in other words, on-the-fly installation of applets with native components, which is still possible for normal Java Card applets, is prohibited for those with native components. There are potentially other solutions to this issue that make use of code signing.

5.6.4. Exception Handling

As we've described in Section 3.1, the CRE exception stack – i.e. the part of the exception stack that is used by functions lying in the CRE – lies in the CRE, more importantly in directly accessible memory from the view of such a function. This fact is partly obscured by two things: first the fact that Java Card bytecode is compiled, which itself makes no reference to the exception stack whatsoever, second that the only code emitted in order to deal with the stack are macros, which hide platform-dependant details. One such detail would for instance be the well-known address of the pointer to the top of the exception stack.

Due to the above, it is highly unlikely that code is generated which messes up the exception stack. However, an attacker with access to it and the ability to run arbitrary code in an (implied) dependency of a given applet, may perform a denial of service attack by crafting a malicious stack frame, which – depending on the course of action that the program takes – causes an access violation, which ends up in a security exception or a card reset. For instance, they might insert an empty frame, which would cause a read violation once the exception range hits its end. They might also fake a handler with bogus values, which causes a similar violation.

A more sophisticated attacker may also set up handlers in a way that they don't catch the right exception types, possibly altering the control flow in-applet or themselves catching the exception in another applet they prepared. Finally, an attacker could set up their own exception handler in such a way that execution jumps to an arbitrary point in the victim's binary, which may even be used for some return oriented programming.

All of these attacks can of course be averted by handling the exception stack in the OS, as none of these attacks are possible without direct access to it. Since the compiler relies on platform-dependant macros to handle exceptions, this can be done simply by rewriting the header that defines them. (Of course, said functionality must first be implemented in the OS.) That being said, the cost of expensive system calls can also be saved by not giving access to the exception stack to arbitrary code. One would be to not allow any applets other than the ones compiled from Java Card bytecode access to the exception stack. Sadly, this would then turn into a problem of securely marking such applets, or one simply couldn't install any applets except those compiled by some "trusted" compiler. (Remember that it is still the chip vendor who makes this decision.) Another method would be to use code analysis tools to determine whether an applet accesses global variables in a context in which it shouldn't be allowed to. How accurately such tools can find malicious accesses is however a topic for papers in other fields.

5.6.5. Unintentional data leakage

Recall that in C, functions put data onto the stack once it becomes too large to fit into a register and also uses the same stack to manage call frames. This is important, when considering cross-package function calls with a shared stack (or registers). "Accidental" access of some information is already prevented by restricting the stack of a function to exclude the stack of its caller(s). However, a malicious applet might call some function of another applet which uses secret information and then inspect the called functions' stack to extract e.g. crypto keys. This can be mitigated by clearing the stack (either with zeros or random data) of the called function. In a similar fashion, it needs to be ensured that the call of another function changes no register in the calling code safe for those designated to capture return values.

Note that this does not exclude any data leakage that occurs from functions called directly, since they do not travel through the OS to arrive at their target. However, the impact of such "leakage" is minimal, as the attacker would attack their own applet (or rather package), which they may freely do if they so wish.

5.6.6. Introducing side channels

It is worth pointing out that the compiler pays no attention to potential side channels it might create through optimizations. For instance, using the array cache might make one code path faster, which would allow an attacker to infer (parts of) a key. Optimizations performed by the C compiler can also produce a side channel involuntarily.

If you plan on compiling such code with this compiler, we strongly suggest to only use function outlining and redirection and to tune the C compiler to not perform any optimizations that interfere with timing either – that is, assuming that your code was already written with timing in mind. Consider also passing the resulting program to a side-channel eliminating compiler, such as the one proposed by Wu et al. [32] – although this specific one might be of little use, because the resulting code is still obfuscated to a point, where the actual secret is not apparent – how exactly should their compiler deal with `BALOAD`, for instance?

In theory, compilation to C also makes it possible to link a (small) crypto library against the application, which would not suffer from the above weaknesses. However, doing so in a meaningful way would require the compiler to be handed to the application developer rather than being kept behind closed doors with the hardware vendor, which would open up much larger potential for exploitation (see earlier sections). Again, a software developer writing crypto code in Java has to hope that the vendor either does not transform their code at all or performs security-aware transformations. This too is little different from the implicit trust already put into the hardware vendor that some given Java Card bytecode has no side-channels.

6. Conclusion

A compiler was constructed together with a runtime environment and an FFI to perform a method-based compilation of Java Card bytecode into C, and subsequently native code. Some optimizations were added to said compiler to improve the performance of the generated code, and arrive at decent speedup compared to the original code, while keeping size overhead minimal by adding the generated code as a shared object to the original CAP file.

The potential for future work is quite large in the realms of optimization and security. In terms of optimizations, a fair number of techniques were left out, including the extraction of invariants, loop unrolling and other operations on basic blocks. These could be of interest, as the C backend lacks relevant information that this compiler would have. In a similar manner, the compiler could infer function purity, which could also prove useful (see Appendix B for motivation). Methods could be (partially) resolved at compile time to save runtime spent doing lookups in the JCVM. The overall size usage of a set of applets utilizing native components could potentially be reduced by allowing dynamic linkage between native components, though the Applet Firewall would need to be considered in such a design as well.

In terms of security, the compiler should be modified, so as to not invoke undefined behaviour, and may on top be modified to better integrate with side-channel eliminating techniques. Systems designed to run applets containing native code may in turn want to employ their own rules when it comes to loading and sandboxing such applets.

Bibliography

[1] Austin Armbruster et al. "A real-time Java virtual machine with applications in avionics". In: *ACM Trans. Embedded Comput. Syst.* 7.1 (2007), 5:1–5:49. DOI: 10.1145/1324969.1324974. URL: https://doi.org/10.1145/1324969.1324974.

[2] Faisal Aslam et al. "Optimized Java Binary and Virtual Machine for Tiny Motes". In: *Distributed Computing in Sensor Systems, 6th IEEE International Conference, DCOSS 2010, Santa Barbara, CA, USA, June 21-23, 2010. Proceedings*. 2010, pp. 15–30. DOI: 10.1007/978-3-642-13651-1_2. URL: https://doi.org/10.1007/978-3-642-13651-1%5C_2.

[3] Michael Baentsch et al. "JavaCard-from hype to reality". In: *IEEE Concurrency* 7.4 (1999), pp. 36–43.

[4] Per Bothner. *Kawa Reference Manual*. URL: https://www.gnu.org/software/kawa.

[5] Niels Brouwers, Koen Langendoen, and Peter Corke. "Darjeeling, a feature-rich VM for the resource poor". In: *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems, SenSys 2009, Berkeley, California, USA, November 4-6, 2009*. 2009, pp. 169–182. DOI: 10.1145/1644038.1644056. URL: https://doi.org/10.1145/1644038.1644056.

[6] Joshua Ellul. "Run-time compilation techniques for wireless sensor networks". PhD thesis. University of Southampton, 2012.

[7] Joshua Ellul and Kirk Martinez. "Run-time compilation of bytecode in wireless sensor networks". In: *Proceedings of the 9th International Conference on Information Processing in Sensor Networks, IPSN 2010, April 12-16, 2010, Stockholm, Sweden*. 2010, pp. 422–423. DOI: 10.1145/1791212.1791286. URL: https://doi.org/10.1145/1791212.1791286.

[8] Andreas Gal, Christian W. Probst, and Michael Franz. "HotpathVM: an effective JIT compiler for resource-constrained devices". In: *Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE 2006, Ottawa, Ontario, Canada, June 14-16, 2006*. 2006, pp. 144–153. DOI: 10.1145/1134760.1134780. URL: https://doi.org/10.1145/1134760.1134780.

[9] Mark Galassi et al. *Guile Reference Manual*. Version 2.2.3. Free Software Organization. Dec. 2017.

[10] Frank Golatowski et al. "JSM: A small Java processor core for smart cards and embedded systems". In: *Target* 2.10 (2002), p. 11.

[11] Lukas Gressl. "Design and Implementation of a Java Card Cross-Compilation Framework". MA thesis. Graz University of Technology, 2016.

[12] Matthias Grimmer et al. "An Efficient Native Function Interface for Java". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '13. Stuttgart, Germany: ACM, 2013, pp. 35–44. ISBN: 978-1-4503-2111-2. DOI: 10.1145/2500828.2500832. URL: http://doi.acm.org/10.1145/2500828.2500832.

[13] Roberto Ierusalimschy and Luiz Henrique. *Lua Reference Manual*. Version 5.3. 2017.

[14] *Java Native Access*. https://github.com/java-native-access/jna.

[15] Dawid Kurzyniec and Vaidy Sunderam. "Efficient cooperation between Java and native codes–JNI performance benchmark". In: *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*. Jan. 2001.

[16] Konstantinos Markantonakis and Raja Naeem Akram. "Multi-Application Smart Card Platforms and Operating Systems". In: *Smart Cards, Tokens, Security and Applications*. Ed. by Keith Mayes and Konstantinos Markantonakis. Cham: Springer International Publishing, 2017, pp. 59–92. ISBN: 978-3-319-50500-8. DOI: 10.1007/978-3-319-50500-8_3. URL: https://doi.org/10.1007/978-3-319-50500-8_3.

[17] Yukihiro Matsumoto. *Creating Extension Libraries for Ruby*. URL: https://docs.ruby-lang.org/en/trunk/extension_rdoc.html.

[18] Oliver Maye and Michael Maaser. "Comparing Java Virtual Machines for Sensor Nodes - First Glance: Takatuka and Darjeeling". In: *Grid and Pervasive Computing - 8th International Conference, GPC 2013 and Colocated Workshops, Seoul, Korea, May 9-11, 2013. Proceedings*. 2013, pp. 181–188. DOI: 10.1007/978-3-642-38027-3_19. URL: https://doi.org/10.1007/978-3-642-38027-3%5C_19.

[19] Wojciech Mostowski and Erik Poll. "Malicious code on Java Card smartcards: Attacks and countermeasures". In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2008, pp. 1–16.

[20] Gilles Muller, Fabrice Bellard, and Charles Consel. "Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code". In: *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*. Usenix, 1996, pp. 1–20.

[21] Oracle. *Java Card 3 Platform Runtime Environment Specification*. Classic. Version 3.0.5. Oracle. May 2015.

[22] Oracle. *Java Card 3 Platform Virtual Machine Specification*. Classic. Version 3.0.5. Oracle. May 2015.

[23] Oracle. *Java Card 3 Platform Virtual Machine Specification*. Classic. Version 3.1.0. Oracle. Jan. 2019.

[24] Oracle. *Java Native Interface Specification*. Version 11. Oracle. 2017. URL: https://docs.oracle.com/en/java/javase/11/docs/specs/jni.

[25] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. "Real time Java on resource-constrained platforms with Fiji VM". In: *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009*. 2009, pp. 110–119. DOI: 10.1145/1620405.1620421. URL: https://doi.org/10.1145/1620405.1620421.

[26] Niels Reijers and Chi-Sheng Shih. "Ahead-of-Time Compilation of Stack-Based JVM Bytecode on Resource-Constrained Devices". In: *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks, EWSN 2017, Uppsala, Sweden, February 20-22, 2017*. 2017, pp. 84–95. URL: http://dl.acm.org/citation.cfm?id=3108022.

[27] Damien Sauveron. "Multiapplication Smart Card: Towards an Open Smart Card?" In: *Information Security Technical Report* 14 (May 2009), pp. 70–78. DOI: 10.1016/j.istr.2009.06.007.

[28] Martin Schoeberl. "JOP: A Java Optimized Processor for Embedded Real-Time Systems". PhD thesis. Vienna University of Technology, 2005. URL: http://www.jopdesign.com/thesis/thesis.pdf.

[29] Doug Simon et al. "Java™ on the bare metal of wireless sensor devices: the Squawk Java virtual machine". In: *Proceedings of the 2nd international conference on Virtual execution environments*. ACM. 2006, pp. 78–88.

[30] Richard M Stallman and Zachary Weinberg. *GCC Reference Manual*. Version 8.3. Free Software Organization. 2018.

[31] Chih-Sheng Wang et al. "A Method-based Ahead-of-time Compiler for Android Applications". In: *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '11. Taipei, Taiwan: ACM, 2011, pp. 15–24. ISBN: 978-1-4503-0713-0. DOI: 10.1145/2038698.2038704. URL: http://doi.acm.org/10.1145/2038698.2038704.

[32] Meng Wu et al. "Eliminating timing side-channel leaks using program repair". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. 2018, pp. 15–26. DOI: 10.1145/3213846.3213851. URL: https://doi.org/10.1145/3213846.3213851.

[33] Zhang Jianjie et al. "A Java processor suitable for applications of smart card". In: *ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No.01TH8549)*. Oct. 2001, pp. 736–739. DOI: 10.1109/ICASIC.2001.982668.

Appendix

A. Equivalence Checking in the DDG

Some optimizations require equivalence checks in the DDG. When checking two nodes in the DDG for equivalence, an answer is searched to the question of whether or not they contain the same data. This is a trivalent question with a large room for "maybe"s.

In order to answer this question and leave as little room to "maybe"s as possible, this compiler uses three heuristics, each more powerful than the one before while also being more costly.

- Trivial equivalence check
- Equivalence modulo basic blocks
- Identical *sources*

The trivial equivalence check is – as its name suggests, trivial. The compiler checks that the two nodes are either the same, or in the case of constants that their contents are the same.

In the equivalence check modulo basic blocks, it follows the predecessors of variable nodes until they are either more than one or no longer variables, and then appliies the trivial equivalence check on these two.

Should both of these tests fail, it looks for the *sources* of a node. A *source* is in this case either a non-variable node or a variable node that is a root[1] of the DDG, whichever is reachable first through its predecessors. As branches occur, a node may have multiple predecessors, which do not necessarily share a source, hence it is possible for a node to have multiple sources. In the case that they do have only one source, and the sources of two nodes are the same, they can consider them equal. The same holds if the sources were constants with the same content, but this check is somehow missing in the current version. Otherwise, the sources of two nodes may overlap or not. If there is no overlap, it is highly unlikely that they are the same. There could potentially be two different nodes in the DDG encoding the same operation, which would make them equal, but the compiler does not eliminate common subexpressions yet, so they are considered distinct. If there is some overlap, then it may be possible for both nodes to have been assigned to the same source. However, this cannot be asserted without venturing into symbolic execution land, and therefore the result has to be considered a "maybe".

[1] Although the DDG is certainly not acyclic just like the CFG, here the notion of a root as a node with no ingoing edge actually applies.

B. Lost in translation: function purity

Currently the compiler has to assume that any function call may in theory modify each and every object or array in existence. We already know that a firewall exists to prevent unauthorized modification, but that would not prevent someone from deliberately calling a function that modifies all objects within the scope of an applet anyway. Still, it is silly to assume that all functions do that when we in fact know better. Sadly, it is not really possible to annotate the C code in a way that the compiler understands which modifications are or are not made. Instead, the compiler of this thesis would need to take them into account.

Consider the following example. An applet uses a `byte[]` buffer to read and store data. Some of that data is actually of type `short`, which can be extracted from the buffer using `javacard.framework.Util.getShort`. Maybe some data is copied into another buffer using `javacard.framework.Util.arrayCopy`. In either case, it is worth noting that the source buffer is not modified in those calls, and neither are any other objects safe for the destination buffer of `arrayCopy`.

Another example would be the typical Java programmer's habit of wrapping field accesses in (frankly meaningless) accessor methods. No getter will ever change anything in any object and no setter should ever modify any data other than that associated with its corresponding getter. On the plus side, the additional cost of writing a function call each time to access some data will often convince the programmer to store its return value in a local variable and use that for the rest of the function. However, if one was to implement loop optimizations, it would not be clear whether that getter could be pulled out from a loop or not. It would also not be clear whether any field access could be pulled out from a loop in the presence of any accessor method without information of what that method modifies (if it does).

If any of the above are mixed with field/array accesses and one decided to cache such accesses – because we know that many programmers prefer to write code in a way that it makes the access rather than storing the data in to them useless temporary variables – caches would too quickly get invalidated for them to become useful. A lot of problems would be alleviated by some kind of `const` keyword. Were the buffer in `getShort` to be declared const, one would be able to assume that it is not modified within. However, infering whether or not *other* data is modified remains a problem – an even bigger one in fact. Such information would need to be taken from documentation or by actually inspecting the code and putting it into some database or perhaps into an "extended" export file. Once this database is established, however, one would be able to strengthen the assumptions about invoked methods. Sadly, none of this is done yet.

C. Exception range semantics

In Section 5.5.3, it was pointed out that operations may move across exception boundaries, but it was also claimed that this rarely turns into an issue. This section delves deeper into that claim and also somewhat discredits it, but before doing so, it is worth pointing out, that the setup for this analysis differs from the one used in the main portion. Namely, it consists OpenJDK version 1.8.0_212, Kawa 3.0 (3.0-0-g39797ea), and Jasmin 2.4. To make things fair for platforms, which are not supported by Java Card anyway, the desktop JVM is used instead of the JCVM.

```
public class NthJava
{
    public static short nth(short[] arr, short i) {
        try {
            return arr[i];
        } catch (IndexOutOfBoundsException ex) {
            return 0;
        }
    }
}
```

Listing C.1: Example code for a class with a static method that returns the nth element of an array or 0 if it doesn't exist.

Consider the class `NthJava` as described in Listing C.1. It is obvious that the access `arr[i]` is covered by the `try` block, but so would be the return statement – at least visually. When compiling this to bytecode, one might be surprised to see that this is not the case.

```
public static short nth(short[], short);
  Code:
     0: aload_0
     1: iload_1
     2: saload
     3: ireturn
     4: astore_2
     5: iconst_0
     6: ireturn
  Exception table:
     from    to  target type
         0     3     4   Class java/lang/IndexOutOfBoundsException
```

As can be seen from the exception table, the range covered by the try block is from 0 (inclusive) to 3 (exclusive). In other words, the return statement is for all intents and purposes considered "outside" of the exception range. This is the painful observation that everyone will make when trying to mix `return` and `finally` in the wrong way. Code written in such a manner would trigger the mentioned bug, but in reality, a lot of code is written like Listing C.2 instead. The explicit return value prevents all errors that would come from mixing `try` and `return`, both the bug described in Section 5.5.3 and the infamous `try-finally` behaviour.

```
public class NthJava2
{
    public static short nth(short[] arr, short i) {
        short ret;
        try {
            ret = arr[i];
        } catch (IndexOutOfBoundsException ex) {
            ret = 0;
        }
        return ret;
    }
}
```

Listing C.2: `NthJava` with explicit return value.

The bytecode it produces becomes larger, but the assignment to the local variable that contains the return value is now inside the exception range, just as expected by the compiler.

```
public static short nth(short[], short);
  Code:
     0: aload_0
     1: iload_1
     2: saload
     3: istore_2
     4: goto          10
     7: astore_3
     8: iconst_0
     9: istore_2
    10: iload_2
    11: ireturn
  Exception table:
     from    to  target type
         0     4     7   Class java/lang/IndexOutOfBoundsException
```

But why limit ourselves to Java? Well, for one, the Java Card environment is very limited and it would not make sense to use another language built on top of the JVM

that requires an even larger runtime environment – to which your favourite programming language likely belongs if it has one. On top of that many languages have quite similar exception semantics to Java, even if their syntax may differ.

Okay, you might say, but what about languages with *different* exception semantics, reminding us that there are languages with completely different concepts for everything, exceptions included. Consider for example the Kawa scheme implementation. Rather than writing statements and statement blocks as one would in other languages, Lisp dialects – among them Scheme – are built upon expressions, and blocks are simply a sequence of expressions that are evaluated in order.[1] Behold the Kawa implementation of `nth`, which can be seen in Listing C.3.

```
(define (nth (array ::short[]) (i ::short)) ::short
  (try-catch (array i)
             (x java.lang.IndexOutOfBoundsException 0)))
```

Listing C.3: Kawa implementation of `nth`

The semantics of `try-catch` are outlined in the manual[4], but as that does not seem to completely line up with how things *actually* work (at least at the time of writing), they are also described here. The first argument is evaluated as an expression and if it succeeds without an exception, then its result is returned. Otherwise, the rest of the arguments is scanned for a handler given as `(var cls stmt ...)` where the currently thrown exception is a subclass of `cls`. If so, it is bound to `var` and the statements are evaluated with the last one being the returned value.

Having achieved knowledge of how Kawa exceptions are supposed to work in theory, one can now verify, that this theory holds by running the compiler.

```
public static short nth(short[], short);
  Code:
     0: aload_0
     1: iload_1
     2: saload
     3: istore_2
     4: goto          10
     7: astore_3
     8: iconst_0
     9: istore_2
    10: iload_2
    11: ireturn
  Exception table:
     from    to  target type
         0     7     7   Class java/lang/IndexOutOfBoundsException
```

[1] This distinction is actually more contentious than it is made to seem. At the time of writing Kawa sees expressions as a subcategory of statements, Guile sees statements as expressions used for side-effects only and the Emacs Lisp reference uses both words without clearly defining either, sometimes interchangeably.

Oh wait, that's the same as the second Java version. To understand why, one could dissect the compiler, but let's not do that and instead think about this on a more conceptual level. Kawa's `try-catch` evaluates to the result of the body or the last statement in a handler. Since an expression and the value it evaluates to are considered the same in functional programming, the interpreter/compiler is tasked to allocate a new value, store the given result in it and hand it to the outer expression. In conclusion, yes, Kawa has different exception semantics from Java, but these are exactly the semantics that can be considered sane, that cause no trouble. This might inspire some people to write all their applets in Kawa, but sadly this is not all of the code generated by the above snippet. There is more boilerplate generated to make this method callable from Scheme, which relies on additional infrastructure, making it difficult to port this example to Java Card without first porting the interpreter itself.

Moving on to the next language, there is another candidate that could be used to actually write applets in: Jasmin, i.e. an assembly language for the Java VM. Since it's assembly, a developer can apply human intelligence to write exactly the code wanted by the compiler while also turning it to a shorter and better version of the first one.

```
.method public static nth([SS)S
.limit stack 2
.limit locals 2
START:
  aload_0
  iload_1
  saload
  ireturn
OOPS:
  iconst_0
  ireturn
.catch java/lang/IndexOutOfBoundsException from START to OOPS using OOPS
.end method
```

Again, the resulting Java bytecode can be inspected to verify, that it matches expectations.

```
public static short nth(short[], short);
  Code:
     0: aload_0
     1: iload_1
     2: saload
     3: ireturn
     4: iconst_0
     5: ireturn
  Exception table:
     from    to  target type
         0     4     4   Class java/lang/IndexOutOfBoundsException
```

Just as promised, one can write code that is both shorter and captures expected exception semantics better than the initial Java version – although the gains achieved in doing so don't really translate that well to C. (Recall that the compiler generates the variable `ret` similar to how it is manually inserted in the second Java version.) In a similar fashion, however, a very nasty programmer could decide to only wrap the `saload` in an exception range just to spite the compiler, not that this decision would end up creating "better" C code even if exceptions were to be handled correctly.

Of course, these experiments are neither very thorough nor are the results from them guaranteed to stay the same over time. In fact, one might (fearfully or hopefully) expect future compilers to reason away variables explicitly created for the purpose of serving as temporary return values, to do away with caught exceptions whose value is never used, and so on, and so forth, especially when considering size-constrained environments like Java Card. Hopefully this section has served as a small glimpse into how we currently expect code to be written.