KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Secure Compilation of ML Modules

Matthias van der Hallen

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Artificiële intelligentie

Promotor:
Prof. dr. ir. F. Piessens

Assessoren:
Dr. ir. W. Meert
Dr. D. Devriese

Begeleiders:
M. Patrignani
R. Strackx

Academiejaar 2013 – 2014

Preface

I would like to thank everyone who helped me to produce this master thesis. First, I would like to thank my promoter, prof. dr. ir. Frank Piessens, for providing me with this opportunity. Next, many thanks go out to my supervisors, M. Patrignani and R. Strackx for supporting me throughout the year.

Furthermore, I would like to thank my friends, classmates, and especially my girlfriend Elke Sekeris, with whom many moments of joy or frustration could be shared. Finally, I would like to thank my family for the encouragement and moral support I was lucky to receive.

Matthias van der Hallen

Contents

Preface **i**

List of Figures **iv**

List of Listings **vi**

Abstract **vii**

Samenvatting **viii**

1 Introduction **1**
1.1 Secure Compilation . . . 2
1.2 Protected Module Architecture . . . 3
1.3 Thesis Structure . . . 5

2 A Compilation Example **7**
2.1 MiniML . . . 7
2.2 A Cipher In MiniML . . . 8
2.3 LLVM and the LLVM Intermediate Representation . . . 10
2.4 Translation Example: The Caesar Cipher . . . 15
2.5 Lessons learned . . . 21

3 Formal Specification **23**
3.1 MiniML . . . 23
3.2 LLVM Intermediate Representation . . . 35
3.3 Formalized Compiler . . . 37
3.4 Conclusion . . . 40

4 Advanced ML Concepts **41**
4.1 Higher-Order Functions . . . 41
4.2 Functors . . . 46
4.3 Conclusion . . . 60

5 Advanced Concepts: Formalization **61**
5.1 Formal Specification of MiniML . . . 61
5.2 Formalized Compiler . . . 66

6 Proving Full Abstraction **73**
6.1 Formal Proof Techniques . . . 73
6.2 Conclusion . . . 75

7 Conclusion **77**
7.1 Related Work 77
7.2 Future Work 78
7.3 Conclusion 79

A LLVM Code **81**
A.1 @mask, @unmask & @unmasktype 81
A.2 @tyvarcheck 84
A.3 Polymorphic Example 86

B Paper **93**

C Poster **101**

Bibliography **103**

List of Figures

1.1 PCBAC Semantics 4

2.1 LLVM Types 12

3.1 Syntax: Core Language 24
3.2 Syntax: Module Language 26
3.3 Syntax: Program Language 26
3.4 Contexts 27
3.5 Typing Judgements 27
3.6 Type Schemes 28
3.7 Typing Rules: Program Language 30
3.8 Typing Rules: Core Language 31
3.9 Typing Rules: Module Language 32
3.10 Operational Semantic Entities 33
3.11 MiniML Evaluation Rules 34
3.12 Syntax: LLVM 36

4.1 MiniML Structure Syntax(excerpt) 50
4.2 Frame Representation Example: StringEqual 51
4.3 Frame Representation Example:StringDict 52
4.4 Type Representations For Meta-frames 55
4.5 Meta-frame Elements 55
4.6 Comparison of execution 59

5.1 Extended Core Syntax 62
5.2 Extended Module Syntax 63
5.3 Updated Contexts 63
5.4 Updated Typing Rules: Core Language 63
5.5 Updated Typing Rules: Module Language 64
5.6 Updated Value Concept 64
5.7 Environment entity 65
5.8 Updated Module Table Concept 65
5.9 Closure Evaluation Rules 65
5.10 Functor Evaluation Rules 66

6.1 Syntax for trace semantics. 74

List of Listings

2.1 Dictionary Definition Example 7
2.2 Dictionary Declaration Example 8
2.3 Caesar Cipher Example 8
2.4 Variable Reassignment 12
2.5 SSA Representation 1 12
2.6 SSA Representation 2 13
2.7 Alternative Dictionary Definition 17
2.8 LLVM Translation Caesar Cipher Example 18
4.1 Lexical Scoping 41
4.2 Predefined Function Passing 42
4.3 Predefined Function Passing: Unsugared 42
4.4 Copy Free Variables 43
4.5 Dictionary Functor Example 46
4.6 Alternative Dictionary Functor 47
4.7 Secure Code Fragment 52
4.8 Insecure Context Fragment 53
4.9 Functor Application or Static Definition: 1 58
4.10 Functor Application or Static Definition: 2 58
A.1 LLVM mask, unmask and unmasktype functions 81
A.2 LLVM tyvarcheck function 84
A.3 Pair structure MiniML 86
A.4 Pair Structure: LLVM 86

Abstract

Malware infects many new computers each day. Some estimates suggest that up to 30% of all computers are in fact infected. Malware can be transmitted by exploiting bugs in computer software. In many cases, these bugs abused by the exploit originate from the disparity between the computing model presented by high-level source language and the effective model used by the low-level target language.

These bugs cannot be prevented by analyzing the source language, for example using formal software verification tools, because they are effectively 'introduced' by the process of compilation from source language to target language. Instead, they must be prevented by strengthening the compilation process in a way that reduces the power of low-level attackers to that of high-level attackers. Compilers that achieve this provide 'secure compilation'.

This work uses the notions of full abstraction and contextual equivalence to formalize the requirements for a secure compilation scheme, and shows how secure compilation can be achieved for MiniML, a subset of the ML language. As a prerequisite however, the secure compilation scheme assumes that the result of compilation runs on an architecture that provides program counter based access control, called a protected module architecture.

The source language for this secure compilation scheme, MiniML, uses a module system with the powerful notion of a functor to provide modularization of code. This contrasts with the object oriented approach used by many other languages. The compilation scheme targets the LLVM Intermediate Representation, or LLVM IR, as a target language. A formalization of this MiniML source language and the LLVM IR target language is presented, enabling a formalization of the secure compilation scheme to be given as well.

Samenvatting

Malware infecteert dagelijks nieuwe computers. Volgens sommige schattingen zou maar liefst 30% van alle computers besmet zijn met malware. Malware kan zich verspreiden door het exploiteren van bugs in computer software. In veel gevallen zijn de bugs die deze malware misbruikt een gevolg van verschil tussen het computer model dat een hoge programmeertaal aanbiedt, en het effectieve model gebruikt door de lage niveau programmeertaal.

Zulke bugs kunnen niet ontdekt worden door analyse van de code op het niveau de hoge programmeertaal, bijvoorbeeld met behulp van formele software verificatie programma's. De bugs worden namelijk 'geïntroduceerd' door het compilatieproces dat de hoge programmeertaal vertaalt naar de lage programmeertaal. In plaats daarvan moeten deze bugs voorkomen worden door het compilatieproces aan te passen, zodat een aanvaller op het niveau van de lage programmeertaal geen mogelijkheden extra heeft ten opzichte van een aanvaller op het hoge niveau. Een compiler die deze garantie bereikt, biedt 'veilige compilatie' aan.

Dit werk gebruikt de concepten volledige abstractie en contextuele equivalentie om de doelstellingen van een veilige compiler formeel te omschrijven. Bovendien wordt getoond hoe veilige compilatie mogelijk is voor MiniML, een kleine taal gebaseerd op de programmeertaal ML. Een van de vereiste voor de correct werking van de veilige compiler is dat het resultaat van de compilatie draait op een computer architectuur die instructiewijzer afhankelijke toegangscontrole aanbiedt. Zo'n architectuur wordt een veilige modulenarchitectuur genoemd.

De hoge programmeertaal gebruikt door dit veilig compilatieschema, MiniML, biedt de mogelijkheid om modulaire code te schrijven dankzij een modulesysteem dat de krachtige notie van een functor bevat. Dit staat in contrast met de object geörienteerde aanpak die door vele andere hoge programmeertalen genomen wordt. Als lage programmeertaal gebruikt deze veilige compiler de LLVM Intermediate Representation, ofwel LLVM IR. Dit werk levert een formele specificatie van de talen MiniML en LLVM IR aan, om vervolgens op basis hiervan ook een formele beschrijving van de veilige compiler te geven.

Chapter

1 Introduction

In today's technology-driven world, computer software is used by nearly everyone on a daily basis. The near omnipresence of malware trying to infect computers makes *safety* a continuous concern for anyone involved in the creation of this computer software. Partly, this can be done by verifying that no bugs exist in the source code of the software, for example using verification software [JP08, JSP10].

However, a formal verification of the source code can only show that no source-level bugs exist. After a program's source code has been written, it is usually compiled to a target language such as assembly that can be executed. This target language most often is substantially different from the source language. These differences are a direct consequence of the simplified computing model that high-level languages usually offer to the programmer.

For example, high-level languages often hide the fact that values have to be represented using target language concepts, saved in computer memory. Another concern hidden from the programmer is how the flow of control, i.e. determining the next command to be executed, is managed. In the high-level language, a function execution is an atomic operation that can only be executed as a whole, and never partially. This is less evident in the low-level language. The compilation process reintroduces these concerns.

In many instances, malware exploits target-level bugs introduced by this compilation process [EYP10, YJP12]. These bugs do not show up during formal verification of the source code, as they are only introduced when the abstract computing model used on source-level is traded in for the concrete target-level computing model.

To provide security, the compilation process itself must be strengthened, so that it guarantees that any source-level security guarantees are preserved throughout the compilation step from source language to target language. If this is achieved, then any software that is attack-free on the source-level is attack-free as well on the target-level. A compilation scheme able to provide these guarantees is rightly called secure. Effectively, such a compilation scheme reduces the capabilities of a target-level attack to those of a source-level attack. The aim of this thesis is to describe how secure compilation can be achieved for a *functional* language that implements an *ML-style* module system.

1.1 Secure Compilation

Software programs or libraries are usually written in a source language, to be followed by compilation to a target language. Often, the main reason for this distinction is that it is easier to reason about the program in the source language than at the target language. This is because the source language works at a higher level of abstraction than the target language.

For example, a high-level source language such as Java abstracts away all concerns a programmer might have about how to handle computer memory. The representation of an object defined by the programmer in memory, as well as the location of this object is all hidden away. Effectively, the source language takes the burden of handling these daunting tasks away from the programmer.

This is a form of *data abstraction*: objects are described by their properties and the functionality they offer, not by their implementations. Here data abstraction concerns hiding the implementation of basic types by bits in memory, but most high-level languages offer the same software principle by allowing abstract data types. An abstract data type is implemented using basic types offered by the source language, but whenever a value of this type is used, only its described properties and abstract functionality are available.

Data abstraction is part of an abstraction paradigm provided by a source language, called *information hiding*. Information hiding happens any time that some value, function or datatype is implemented, but its use is restricted: it is hidden.

Another abstraction paradigm that the source language can provide is called *modularization*. Modularization offers a way to group closely related functionality together. For example, in Java this would correspond to a class or a group of classes, called a package.

Modularization and information hiding are closely related and affect each other to allow for *encapsulation*, where the internal representation is hidden so that it influences only a small, identifiable region of the program [Pie02]. The information hiding mechanisms of Java for example, the `private, public, protected` and `default` access modifiers, differ from one another in how they handle the different modularization mechanisms: classes and packages. Public functionality can be used everywhere, private functionality only within the same class, whereas the default access modifier allows access from any class within the same package.

These abstractions not only make it easier for the programmer to reason about the correctness of software, they also provide a way to describe security properties. When a value or function is marked with an access modifier, any programmer rightly assumes these access rights are indeed enforced. A value marked `private` can only be read of modified by certain parts of the code. In security terms these access modifiers provide *confidentiality* and *integrity*. Even formal verification tools to keep the software free from bugs assume the enforcement of these access rights.

The target language, however, does not always offer these abstractions. For example, when executing software, the values created by this software must be saved in memory, using a certain representation. The platform on which the target language runs might not provide access control for this memory, which means values saved in memory might be readable by any code running on the platform.

Software code must be saved somewhere as well, and doing this might make it

possible to corrupt control flow, for example by overwriting a return address [EYP10]. Such an attack might result in executing functionality that was supposed to be hidden, because the atomicity of function execution can be broken.

Full Abstraction

Secure compilation is a compilation process that does preserve these source-level security properties when compiling to the target language. In this work, this property of a compiler is formalized as *full abstraction* [Aba99]. Full abstraction uses the idea of *contextual equivalence* to formalize security guarantees.

Contextual equivalence is an equivalence relation on programs. The contextual equivalence relation $O_1 \simeq O_2$ expresses that two programs O_1 and O_2 are indistinguishable from each other, even when running them in combination with any other program O_C called the context. Informally, this corresponds with there being no *observable* differences between O_1 and O_2. Formally, $O_1 \simeq O_2$ means:

$$\forall O_C : O_C[O_1] \to c \iff O_C[O_2] \to c$$

where $O_C[.]$ is a program where a certain component is unspecified. $O_C[O_1]$ is the program that results from linking O_C with O_1, where O_1 is used as the unspecified component.

Note that contextual equivalence does indeed imply security guarantees are enforced. Suppose that two programs O_1 and O_2 differ only by a value that is supposedly confidential. If it were possible for any context to read or modify this value, then this context provides a counter example for the statement that two programs that use a different hidden value are contextually equivalent.

As contextual equivalence implies that security guarantees are enforced, secure compilation can be formalized as providing *full abstraction*, meaning contextual equivalence is preserved and reflected when compiling a program O_1 to its corresponding target language program $O_1^{\downarrow}$. Formally:

$$O_1 \simeq O_2 \iff O_1^{\downarrow} \simeq O_2^{\downarrow}$$

In the remainder of this text, the program O_1 or O_2 presents the secure code, an encapsulated entity for which some security guarantees hold. The secure compilation of this encapsulated entity is called the self-protected module or *SPM*. The secure code is provided and compiled to an *SPM*, and is linked to an insecure target level context $O_C^{\downarrow}$

1.2 Protected Module Architecture

As explained in Section 1.1, a secure compilation scheme is a compilation scheme that preserves source level security guarantees in the target language.

The secure compilation scheme compiles security sensitive parts of an application into a self-protected module or *SPM* [iDRG]. Such an *SPM* operates in isolation of the other parts of the application. The high-level security guarantees of an *SPM* with respect to the other parts of the application (i.e. the insecure context) are preserved

in the low-level. In other words, any part of the application outside the *SPM* can only operate on the *SPM* in ways specified by the *SPM*'s public high-level API.

Such a preservation of security guarantees is only possible if some form of access controlled memory is available. Indeed, if no protection of any memory where possible, no confidentiality of values within the *SPM* with respect to other parts of the application could ever be preserved, as values would always be readable directly from memory.

An *SPM* requires a specific access control model: *SPM*s are split into a protected code and a protected data section. Protected code contains the security sensitive part of the application's code. All other memory, containing data and code corresponding to other parts of the application, is considered to be the unprotected memory.

Metadata in the *SPM* also specifies a list of entry points. This list specifies the only memory locations in the protected code section to which instructions located in unprotected memory can jump.

The semantics of the access control required by the SPM are summarized in Fig. 1.1. This work uses the same memory access control model as given by Agten et al. [ASJP12]. As shown by Patrignani et al. [PCP13] the same access control model can be used to provide secure compilation for more advanced concepts object oriented programming concepts.

From \To	Protected			Unprotected
	Entry Point	Code	Data	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

Figure 1.1: Program counter based access control semantics as specified in Agten et al. [ASJP12].

The architecture running the SPM enforces that execution can only enter the *SPM* by jumping to an entry point. Instructions in unprotected memory are not allowed to jump to other memory locations inside protected memory than those mentioned in the entry points list. This protects the atomicity of function execution with respect to the insecure code. Instructions inside the protected code section can jump to any other memory location in the protected code section or any location in unprotected memory.

Besides being limited to jumping to protected code locations listed in the *SPM*'s entry point list, instructions in the unprotected memory are not allowed read or write access to any location in the protected code or protected data section. Instead, instructions in unprotected memory can only read from or write to other unprotected memory. Instructions in the protected code section can read from or write to any memory location in unprotected memory, as well as memory locations inside the protected data section.

To enforce such access control semantics, the SPM must execute on an architecture that provides *program counter based access control* [iDRG] or *PCBAC*. Such an architecture is called a *Protected Module Architecture*. The term *program counter based access control* refers to the fact that the validity of a memory access depends on the current location of the program counter.

There are already a few architectures that support these program counter based access control semantics, or a variation on them [SP12, ASAP13, NAD+13, MAB+13].

1.3 Thesis Structure

This thesis aims to show how secure compilation can be achieved for a *functional* language that implements an *ML-style* module system. For this, a source language, *MiniML*, is defined as a subset of the functionality provided by the ML programming language.

A target language to which MiniML will be compiled has to be chosen as well. In this thesis, the *LLVM Intermediate Representation* is used. The LLVM Intermediate Representation is a language used in the LLVM compiler project. As a language it is only slightly more abstract than assembly, and it is specifically designed as an intermediate step in the compilation of different high-level source languages to assembly.

This offers the benefit that compilation to the intermediate language can be performed, and afterwards transformations can be done on the intermediate language to optimize it for different architectures. Afterwards, the LLVM Intermediate representation can be compiled to assembly code for a specific architecture, for example an architecture providing the necessary access control sketched in Section 1.2.

The following list gives a roadmap of how each chapter contributes to the thesis goal of showing how secure compilation of a language with an *ML-style* module system is possible.

- Chapter 2 describes a first version of the MiniML language, mimicking some of the basic functionality provided by the ML language and its module system, using an example: An implementation of a Caesar cipher.

  It also describes the target language, LLVM IR, and shows how compilation from MiniML to LLVM might normally occur.

  It concludes by describing the security issues that must be solved and shows how secure compilation would address these using the example.

- Chapter 3 gives a formalization of both the MiniML source language and the LLVM IR target language. It then formally describes a secure compiler for this first version of MiniML.

- Chapter 4 introduces some more advanced ML concepts to MiniML. Specifically, *higher order functions* and *functors*.

- Chapter 5 extends the formalization of MiniML given in Chapter 3 to include the advanced concepts that were introduced to MiniML in Chapter 4.

  It proceeds by extending the formalization of the secure compiler of Chapter 3.

- Chapter 6 sketches how full abstraction of a compilation scheme can be proven.

- Chapter 7 gives an overview of possible improvements or extensions of the work presented here. It also formulates a conclusion to this thesis.

Chapter

2 A Compilation Example

This chapter firstly informally describes the MiniML source language (Section 2.1), a subset of the ML language whose syntax and semantics are reminiscent of those of Standard ML. Section 2.2 then introduces an example program (a Caesar cipher implementation) that will be used to show the secure compilation scheme. This chapter continues by describing the LLVM intermediate language to which the first translation occurs (Section 2.3). This chapter concludes by translating the earlier proposed example program, showing the resulting LLVM code (Section 2.4).

2.1 MiniML

The ML language is a functional programming language that is well known for its module system. This module system aims to group data and code together into coherent entities, called modules.

A *structure* is the most basic type of module. It can be defined using the `struct` construct and provides a set of bindings for types, values and functions. A structure specifies a name for the binding and the corresponding value, called *implementation*. Structures provide the possibility of grouping related code and data, fulfilling the need of *modularization* in software. However, the need for *data abstraction* is not yet fulfilled. Listing 2.1 shows how a dictionary of strings to strings might be implemented by a module.

Listing 2.1: An example structure showing the definition of a dictionary in ML.

```
1 structure Dictionary =
2     struct
3         type dictionary = (string * string) list
4         val emptyDictionary = []
5         fun insert d, x, y = (x,y)::d
6     end
```

Listing 2.1 firstly defines a type dictionary, which is defined to be a type synonym for a list of string pairs, a value representing the empty dictionary (`emptyDictionary`) and a function for inserting data into the dictionary (`insert`).

As it stands, the dictionary type is a type synonym for lists of string pairs, and any such list could be used where a dictionary is expected. However, the concept of a

dictionary does not require users to know that the dictionary type is implemented as a list of string pairs. According to the principle of data abstraction, it is favorable to hide this information from the user of the dictionary module.

Here the idea of a signature comes into play. A signature groups a set of types, values and functions without providing an implementation. It provides a way of abstracting over structures that implement the same logical concept using a different implementation. A possible signature for dictionaries is shown in Listing 2.2.

Listing 2.2: An example signature showing the declaration of a dictionary in ML.

```
1 signature DICTIONARYSIGNATURE =
2     sig
3         type dictionary
4         val emptyDictionary : dictionary
5         val insert: dictionary -> string -> string -> dictionary
6     end
```

A signature guarantees that two implementations of the same logical concepts are interchangeable for each other by standardizing the way an implementation communicates with the other code. It can also abstract the fact that the current implementation for dictionaries uses lists, as well as obscuring any helper methods that the specific implementation defines in order to simplify its internal workings. This last functionality of a signature is a way to perform *information hiding*.

The MiniML fragment discussed here was chosen to incorporate only the idea of structures, more complex language features will be added later (Chapter 4).

2.2 A Cipher In MiniML

Listing 2.3 presents a simple example program that consists of the definition of a signature that represents symmetric cyphers, a concept used in cryptography. This example was chosen since the modules related to cryptography are usually under more scrutiny with regards to the privacy of their internal values. The code in Listing 2.3 defines a signature `SYMMETRICCIPHER`. This signature describes the common traits between modules that implement a symmetric cipher. In order to implement a symmetric cipher, one must have a credential, i.e. the key, and two functions, `encrypt` and `decrypt`, which take data and credentials. The `encrypt` function takes the raw data and encodes it in a way only those with knowledge of the correct credentials can later use the `decrypt` function to transform the encoded data back into the raw data.

Listing 2.3: Example of a security sensitive module specifying and implementing a symmetric cypher.

```
1 signature SYMMETRICCIPHER =
2     sig
3         type cred
4         val newcredentials : cred
5         val encrypt: int -> cred -> int
6         val decrypt: int -> cred -> int
7     end
8
```

```
9  structure Caesar :> SYMMETRICCIPHER =
10     struct
11         type cred = int
12         fun newcredentials = rand
13         fun encrypt a cred = (a + cred)%26
14         fun decrypt a cred = (a - cred)%26
15         val seed = 3
16         fun rand = time.now * seed
17     end
```

From line 8 of Listing 2.3 onwards the definition of a structure called `Caesar` is given. `Caesar` implements the `SYMMETRICCYPHER` signature. In this context, `Caesar` provides the `newcredential`, `encrypt` and `decrypt` functions. For internal use it also possesses the necessary characteristics of a pseudorandom number generator, namely a seed value and a rand function that provides a pseudorandom number. It is necessary to hide the seed value from users since this would allow attackers to predict the output of the pseudorandom number generator.

The `Caesar` structure is forced to conform to the signature `SYMMETRICCIPHER` by means of *ascription*(`:>`). Ascription not only forces the module to implement all the necessary elements of the signature, but it also restricts the means of interaction with the module to those elements that are explicitly mentioned in the interface. It is this notion of *ascription* that dictates what it means for this module to be secure.

In this case, the ascription of the `Caesar` structure with signature `SYMMETRICCIPHER` is done opaque (`:>`), as opposed to transparent (`:`). The difference between opaque and transparent ascription is as follows:

Opaque ascription Ascribing a structure using opaque ascription `:>` means any declaration, be it type or value, in the signature must have a corresponding definition in the structure. The ascription hides any `val` or `fun` definition for which there is no corresponding `val` declaration inside the signature.

For types declared in the signature, but without a specific implementation, the implementation of the type is unspecified. In other words, for code outside the structure, the type and its implementation are not synonymous.

Transparent ascription Ascribing a structure using transparent ascription `:` hides the same definitions as opaque ascription.

However, for types declared in the signature, but without a specific implementation, the implementation that the structure provides for the type is specified. In other words, for code outside the structure, the type and its implementation are synonymous, and interchangeable.

Concretely, to be secure, this module hides its `rand` function and its seed value from any outside code, only allowing the code internal to the structure to access this value or call the function. Because the ascription is opaque, the implementation of `cred` as an `int` is hidden as well. This makes it impossible for code outside the structure to use a value of type `int` where a value of type `cred` is expected, even though they are type synonyms inside the `Caesar` structure.

2.2.1 Compilation of MiniML

Compilation of MiniML reflects the compilation of the ML language. As in ML, the MiniML compilation process takes a MiniML program as a collection of *files* that contain signatures and structure definitions. Conventionally, MiniML processes each file in order of the collection, and separately, as a unit of compilation.

Within each file it processes the definitions in the order that they are listed in the file. It expects the resulting order in which it processes the definitions to correspond to a *directed acyclic graph* or *DAG*. This means that a definition can only use elements that were defined *before* its own definition.

When talking about the secure compilation of MiniML, the remainder of this text will expect that a file containing all secure signature and structure definitions is passed to the compiler first. This represents the secure module or *SPM*, which contains all secure code.

Next, all code representing the context is compiled separately and later linked to the result of the compilation of the secure code. As a result, the definitions of structures in secure code can not directly depend on elements that are defined only later, in the insecure context.

2.3 LLVM and the LLVM Intermediate Representation

This section introduces the LLVM and its intermediate representation. It also specifies the expected LLVM Intermediate Representation code for the example in Listing 2.3.

2.3.1 LLVM

LLVM, short for *Low Level Virtual Machine* is the name of a project providing many different and closely affiliated utilities concerned with the compilation process. Created by Chris Lattner [Lat02] in 2000, the project was picked up by Apple and work on the LLVM project has continued up to this date.

The main sub-project of LLVM is the LLVM Core, which combines code generation and optimization for many platforms. Because the generation of code for a specific platform and the optimization of code is generally very difficult work, the LLVM Core is built around the LLVM Intermediate Representation, or LLVM IR.

This intermediate representation attempts to provide a shared abstraction that the compilers of many source-level languages can use. The idea is that any source-level language can be compiled to the LLVM IR, using the LLVM project as a *backend* for its own compilation. Once a source-level program is compiled to LLVM IR, any form of optimization can be done on the LLVM Intermediate Representation, and thus optimizations are shared between the different source-level languages.

When all required optimizations are performed, it is possible to compile the intermediate language into machine code and perform linking of all necessary code. Any special modification necessary to run on specific target platforms is shared across the different source languages as well, because the code generation uses the LLVM IR as its input.

Providing a compiler for a source-level language is now confined to providing a compilation to the LLVM IR. When such a compiler exists, the full power of the

LLVM optimizer is accessible for the source language, and compilation is possible to every one of the multitude of platforms compatible with LLVM.

2.3.2 LLVM IR

This work uses LLVM as a *backend*. The secure compiler for MiniML will translate MiniML code into the LLVM Intermediate Representation. In order to focus on the security of the compilation, the more aggressive optimization capabilities of LLVM will not be used.

It is possible to write a program in this LLVM Intermediate Representation using one of three different and equivalent encodings, according to the *LLVM Language Reference* [LA05]:

- A bitcode format
- Textual assembly language
- A symbolic representation, manipulated by the LLVM API

This text will use the textual assembly language as representation for LLVM IR programs, because this makes examples and results more understandable and human readable. While it is possible to generate LLVM IR programs using the LLVM API, this work chooses to generate the human readable intermediate code itself, because it offers more direct control over the resulting translation.

The benefit of LLVM Intermediate Representation is not limited to the points mentioned above. The LLVM Intermediate Representation works at a higher level of abstraction than standard assembly does. Some important additional aspects make the intermediate representation used by LLVM of a higher level of abstraction than standard assembly code:

Type System More information about the program is captured by LLVM than when using regular assembly, using LLVM's type system. This type system helps the optimalization process.

The type system, as given by the *LLVM Language Reference* [LA05] consists of the types shown in Fig. 2.1. Not all types are shown, the vector type and opaque types are omitted.

A shorthand for types can be defined using `%Name = type`.

Register Limitations The LLVM Intermediate Representation abstracts away the fact that real architectures have only a given amount of registers. Instead, one can write a program assuming an infinite amount of *virtual registers*. This results in many more variables in use than available registers. In a later compilation stage this consequence is remedied using the technique of *spilling*. Variable spilling occurs by mapping the variables in use to the smaller set of available registers, saving all variables that could not be assigned to a register in RAM memory in the stack.

LLVM Basic Types	
iN	Arbitrary width integer. The width is specified by N.
Void	The void type. Like the void type in Java, this represents no value. The void value has no size.
type∗	The pointer type. It specifies a specific memory location. The memory location must contain a value with the correct type, as specified by *type*. It is considered by the *LLVM Language Reference* to be a basic type.
label	The label type. This specifies a pointer to a label. This is equivalent to i8*.
LLVM Derived Types	
[N x *type*]	The Array type. This represents N elements of type *type* ordered sequentially in memory.
{*typelist*}	The structure type. This represents a sequence of values in memory of type *type*. The elements are in the order of the list. The list is comma separated.
$type_1$(*typelist*)	The function type. This represents a function that returns a value of type $type_1$, and takes arguments with the types specified in the type list.

Figure 2.1: The LLVM types, with vector types and opaque types omitted.

SSA For optimization purposes, the LLVM IR adheres to the *static single assignment* paradigm, or *SSA*. This implies that every register can be assigned a value only once.

For example, the code in Listing 2.4 reassigns the value saved in register x from 3 to 4. In SSA, this would be represented by Listing 2.5.

Listing 2.4: Reassigning a variable.

```
Block:
    %x = 3;
;    br i1 %cond, label %Cond, label %Ret
Cond:
    %x = add i32 %x, 1;
Ret:
    ret i32 %x;
```

Listing 2.5: Code in SSA form.

```
Block:
    %x1 = 3;
;    br i1 %cond, label %Cond, label %Ret
Cond:
    %x2 = add i32 %x1, 1;
Ret:
    ret i32 %x2;
```

This essentially provides a versioning postfix to the variable identifier. The problem with SSA becomes worse however if the modification of x depends on control flow. For example by uncommenting the conditional statement in Listing 2.4, the eventual value of `\%x` depends on the value of `\%cond`.

The return statement in Listing 2.5 should now return either `%x1` or `%x2`. But how can the code in SSA form decide which of the two registers should be returned? LLVM IR solves this problem by providing the Φ function [AP03].

The Φ function 'merges' a set of variables. It assigns a new variable the value of one of a number of old values, where the choice of the old value depends on the control flow. In Listing 2.6, the SSA-representation of Listing 2.4 with the condition uncommented is shown.

Listing 2.6: Code in SSA form with function.

```
Block:
    %x1 = 3;
    br i1 %cond, label %Cond, label %Ret

Cond:
    %x2 = add i32 %x1, 1;

Ret:
    %x3 = phi i32 [%x1 Block] [%x2 Cond]
    return %x3;
```

2.3.3 Translating MiniML concepts to LLVM

Compiling from MiniML to LLVM IR means the high-level abstractions made in MiniML, for example *signatures* and *structures*, must be mapped to lower-level constructs that are available in the LLVM Intermediate Representation. This section presents these different mappings.

File A file with ML structures and signatures inside it is a separate unit of compilation. Its signatures and structures are type checked and compiled together. When a file is compiled, there can only be dependencies on values that were defined in an earlier unit of compilation, since the compilation of MiniML expects a directed acyclic graph, as explained in Section 2.2.1.

LLVM already provides the concept of a module as a separate unit of compilation. This means each LLVM module is compiled to a single different object file. Firstly, an LLVM module declares which external functions will be provided by other code, and then continues by defining and implementing its own code and data. These definitions and external references are compiled into a single object file. LLVM poses no restrictions on the access of data and functions within a single module. This is poses no problem for files containing multiple secure structures. Since they are first type checked together before being compiled together, a secure structure will never access a hidden component of another secure structure.

The access of data and functions from within other modules *is* restricted by LLVM: the object file to which a module is compiled is accompanied by a link table. This link table specifies which methods are defined and made externally visible, which is used to match the declaration of external functions with their implementation in other object files. The link tables represent what other code knows about the definitions contained within the compiled module.

Structures Structures are a collection of types and value definitions. As such they can be compiled by compiling every one of their components.

Structures also provide namespaces: 2 structures str_1 and str_2 can both define a value t without resulting in a clash of names. When compiling structures by compiling every one of their components, this situation should not introduce a clash of names either. This is guaranteed by identifier prefixing: When a component x within str_1 is compiled, the compilation is not bound to x but to $\text{str}_1.x$ instead.

Functions LLVM provides the concept of a function as a set of basic blocks of code. These LLVM IR functions allow the programmer to modify the visibility of a function using the linkage keyword.

When linking the object files that result from the compilation of different modules, LLVM looks for the implementation of externally declared functions in the different object files, keeping into account whether or not the code was in fact declared to be visible outside the module. It is possible to map ML functions directly to their LLVM counterpart.

Signatures While structures can be mapped directly onto the concept of a module in LLVM, the information provided in signatures will mainly affect metadata in the resulting code or influence the specific implementation of different elements inside the modules.

When a structure is opaquely ascribed, or *sealed* by a signature, we must make sure that the values and functions defined in the structure but not specified in the signature are not externally visible. The first step in protecting these internal functions consists of marking these members as private in the module corresponding to the ML structure. This will filter these members from the object files link table.

Value The translations of a value is a getter function without arguments. This is necessary because

- The value might not be defined as a constant, but as the result of a call to a pure function. This does not violate the constraint that a value must represent an immutable value, because the result of a pure function call will always result in the same result. When translated, this function must be executed however and the result must be passed.
- These values are readable by any code. As shown later, when it is read by the insecure context, certain precautions must be taken. This is impossible when values are translated to constants.

2.4 Translation Example: The Caesar Cipher

In order to study the compilation scheme, the example ML code in Listing 2.3 is translated to LLVM IR.

The code given to the compiler is treated as a single file containing all secure or trusted structures. In the case of Listing 2.3, this consists of only the `Caesar` structure. As mentioned in Section 2.3.3, this is compiled to a single LLVM module.

Before discussing the translation of Listing 2.3, the security concerns that MiniML introduces have to be discussed.

2.4.1 MiniML-specific security concerns

Ordering of Fields and Module Expressions

The necessity of reordering field definitions in a target-level object was shown by Agten et al. [ASJP12]. In MiniML the same argument holds, not only for fields, but for the order of structure definitions as well.

The order in which structures and signatures are defined, or the order of fields within a structure is to a large extent unimportant for the behavior of a MiniML program. As long as no structure is used before it is defined, differently ordered MiniML programs are contextually equivalent. The order of field definitions within a structure has no result on the behavior of the program either.

The ordering of structure or field definitions in LLVM IR might be leaked however, when examining the pointers to different fields. If contextually equivalent but differently ordered MiniML programs would be translated to differently ordered LLVM IR programs, the results would not be contextually equivalent.

To guarantee *full abstraction*, it is thus necessary to ensure that all contextually equivalent but differently ordered MiniML programs result in the same ordering of definitions in LLVM IR. This can be accomplished by alphabetically ordering structures, and the fields within their structures, when outputting the LLVM IR program.

Stack Switching

As described by Agten et al. [ASJP12], when code execution switches from protected code to unprotected code or back, the security of the run-time stack must be protected. Therefore the stack is split into two parts, the secure stack and the insecure stack. These are respectively located in secure and insecure memory. The following precautions, as described by Agten [ASJP12] are then taken:

- When the secure code is entered in an entry point, the stack pointer is modified to point to the secure stack, whose location is saved in a field in the data section. This field is called the *shadow stack pointer*. There is another field reserved in the data section as well, to keep track of the value of the stack pointer before modification.

- At each exit point, the stack is restored to its previous address in unprotected memory, and the location of the *shadow stack pointer* is updated.

Because this is only necessary when functions are called from the unprotected code, these security measures are taken care of by a low-level stub that wraps around the low-level translation of the high-level publicly available function. This stub will then forward execution to an internal function (which is not publicly available) which will strictly focus on the low-level implementation of the high-level function itself. This internal function can thus assume that any necessary security precautions are taken. Within the secure code, calls to these internal functions can happen directly, without passing through the stub.

Register Clearing

Any communication in MiniML normally only occurs through information passed along by returns and method calls. In the case of low-level architecture however, all communication occurs through the unprotected memory, through flags and through the CPU registers. This means that a low-level attacker can try to perform side channel communication.

In order to prevent the leaking of protected data through side channels, it is important to ensure that the information passed through unprotected memory, flags and registers is restricted to that information being passed by the returns and method calls in MiniML. Thus, any callback or return to untrusted code must:

- Clear the flags
- Clear the registers, except those being used to pass a parameter or a return value.

If these registers and flags, which were possibly modified by the protected code, are not reset when returning execution to unprotected memory, code running in the unprotected memory will be able to read the values in these registers and flags, breaking confidentiality. The code could even modify these values, resulting in a modification of control flow if execution is later returned to the protected code while expecting these values to be uncorrupted.

However, since the LLVM Intermediate Representation does not allow multiple assignments to the same registers, it is impossible to clear a register simply by overwriting its value. Furthermore, the LLVM IR does not know how many registers it can use, instead assuming an infinite amount of registers, as mentioned earlier. This means that clearing these registers must happen later on in the compilation process, introducing an extra LLVM pass.

This clearing can once again happen inside the stub, keeping the internal functions free from any code related to security concerns.

Opaque types

The MiniML language allows a programmer to define his own types using the type keyword. Taking another look at the Dictionary example of Listing 2.1, this happens in line 3. The module's internal representation of a dictionary is a list of string pairs, but clearly this is some internal choice that the programmer does not want to make explicit, which is why the type synonym `dictionary` is kept opaque. This can be

seen in Listing 2.2, line 3, where the external specification of a type is not revealed to be a list of string pairs.

This is a means of information hiding, if someone were to later rewrite this dictionary structure in such a way that its internal representation changes, e.g. it becomes a pair of string lists as in Listing 2.7, this would be a perfectly valid change. This change should not result in any changes to the external code, since the specific implementation choice for the dictionary type was not made explicit.

Listing 2.7: An alternative structure defining a dictionary.

```
structure Dictionary =
    struct
        type dictionary = (string list * string list)
        val emptyDictionary = ([],[])
        fun insert((fst,snd), x, y) = (x::fst, y::snd)
    end
```

Even more so, one would expect the two programs/modules to be contextually equivalent! However, clearly, if no checking is performed within functions expecting something of type dictionary, a low-level attacker could discriminate between the two using the following tactic: call a function expecting a dictionary argument with a self-created list of string pairs. If it gives the expected results, the dictionary implementation being used is the one given in Listing 2.1. If not, it is the module described in Listing 2.7, expecting a pair of string lists. This breaks contextual equivalence.

The code must assure that any object, passing as an argument of type dictionary, was in fact created by the module code itself (using the `emptydictionary` method). If not, it should raise an error (even if the object has the correct type according to the synonyms) in order to prevent an attack on contextual equivalence in the same spirit as the one described above.

This restriction can be enforced by a technique called masking, introduced by Patrignani et al. [PCP13]. Masking works by ensuring that the code creating a dictionary never returns a pointer to this dictionary itself, instead saving this dictionary pointer in an internal list located within protected memory. The index in this list can then be returned and used as a mask for the real object. Once again, this masking can be done within the stub function that wraps around the internal function that returns the value.

When a stub represents a high level function that expects an argument of an opaque type, it will receive the index of the desired value in the internal masking list. It can use the internal masking list to retrieve the pointer associated with the index, and to check whether it points to a value of the right type.

Entry points

Instructions in the insecure code can only jump to memory locations marked as an entry point by the *SPM*'s metadata. This compilation scheme adds an entry point to the *SPM* for every function available in the high-level language. Because the security checks are performed inside the stubs, it might seem logical to use the memory locations of their first instructions as the entry points for the *SPM*.

However, because the size of these functions can be derived from the distance between two successive entry points, and this size might leak information about the *SPM*, the memory locations of these stubs are not listed as entry points. Instead, a small entry function that calls these stubs is created, in the low-level this corresponds to a simple tail call. These small entry functions can be grouped together in the secure code section, and take up a constant amount of space. The memory locations corresponding to these entry functions can then be listed in metadata as entry points for the *SPM*.

2.4.2 Translation

The translation to LLVM Intermediate Representation is given in code in Listing 2.8.

Listing 2.8: LLVM IR for the example.

```
1  %int = type i64
2  %Caesar.cred = type {%int}
3
4  declare i8* @malloc(%int)
5  declare void @free(i8*)
6  declare void @exit(i32)
7
8  define %int @Caesar.decrypt(%int %arg0, %int %arg1) {
9     %ret = tail call %int @Caesar.decrypt_stub(%int %arg0, %int %arg1)
10    ret %int %ret
11 }
12
13 define %int @Caesar.encrypt(%int %arg0, %int %arg1) {
14    %ret = tail call %int @Caesar.encrypt_stub(%int %arg0, %int %arg1)
15    ret %int %ret
16 }
17
18 define private %int @Caesar.newcredentials() {
19    %ret = tail call %int @Caesar.newcredentials_stub()
20    ret %int %ret
21 }
22
23 define private %int @Caesar.decrypt_stub(%int %arg0, %int %arg1)
       noinline {
24    %arg1int = call %int @unmask(%int %arg1)
25    %arg1type = call %int @unmasktype(%int %arg1)
26    %check1 = icmp eq %int %arg1type, 4
27    br i1 %check1, label %Continue, label %Error
28
29    Continue:
30    %arg1ptr = inttoptr %int %arg1int to %Caesar.cred*
31    %ret = call %int @Caesar.decrypt_internal(%int %arg0, %Caesar.cred*
        %arg1ptr)
32    ret %int %ret
```

```
33
34    Error:
35    call void @exit(i32 -1)
36    unreachable
37  }
38
39  define private %int @Caesar.decrypt_internal(%int %a, %Caesar.cred* %
        credptr) {
40    %loccred = getelementptr inbounds %Caesar.cred* %credptr, i32 0,
        i32 0
41    %cred = load %int* %loccred
42    %x = sub %int %a, %cred
43    %y = urem %int 26, %x
44    ret %int %y
45  }
46
47  define private %int @Caesar.encrypt_stub(%int %arg0, %int %arg1)
        noinline {
48    %arg1int = call %int @unmask(%int %arg1)
49    %arg1type = call %int @unmasktype(%int %arg1)
50    %check1 = icmp eq %int %arg1type, 4
51    br i1 %check1, label %Continue, label %Error
52
53    Continue:
54    %arg1ptr = inttoptr %int %arg1int to %Caesar.cred*
55    %ret = call %int @Caesar.encrypt_internal(%int %arg0, %Caesar.cred*
        %arg1ptr)
56    ret %int %ret
57
58    Error:
59    call void @exit(i32 -1)
60    unreachable
61  }
62
63  define private %int @Caesar.encrypt_internal(%int %a, %Caesar.cred* %
        credptr) {
64    %loccred = getelementptr inbounds %Caesar.cred* %credptr, i32 0,
        i32 0
65    %cred = load %int* %loccred
66    %x = add %int %a, %cred
67    %y = urem %int 26, %x
68    ret %int %y
69  }
70
71  define private %int @Caesar.newcredentials_stub() noinline {
72    %credptr = call %Caesar.cred* @Caesar.newcredentials_internal()
73    %credint = ptrtoint %Caesar.cred* %credptr to %int
74    %ret = call %int @mask(%int %credint,%int 4)
```

```
75    ret %int %ret
76 }
77
78 define private %Caesar.cred* @Caesar.newcredentials_internal(){
79    %rand = call %int @Caesar.rand()
80    %ptr1 = call i8* @malloc(%int 16)
81    %ptr = bitcast i8* %ptr1 to %Caesar.cred*
82    %locval = getelementptr inbounds %Caesar.cred* %ptr, i32 0, i32 0
83    store %int %rand,%int* %locval
84    ret %Caesar.cred* %ptr
85 }
86
87 define private %int @Caesar.rand_internal() {
88    %t = call %int @Caesar.seed()
89    ret %int %t
90 }
91
92 define private %int @Caesar.seed_internal() {
93    ret %int 3
94 }
```

This code, that implicitly is part of a module, consists of a list of *global values*, denoted by the @ sign. Every value, be it a global function or a global variable, has a *linkage* type associated with it. Linkage types control the accessibility of of variables and functions. The two linkage types in use are `private`, which makes a value only accessible by objects inside the same module, and the default linkage type, `external`.

The code in Listing 2.8 specifies 11 different global values, corresponding to the 5 definitions in the `Caesar` structure, their three stubs and the three entry functions. It also defines one type, corresponding to the `cred` type. As mentioned earlier, the ordering of these functions is alphabetical.

cred type To start the translation, a type definition for the `cred` type is given on line 2. It is represented as a structure, where first the effective structure is represented, and then an `%int` is reserved to keep track of the type. Here, the number 4 is chosen to correspond with the `cred` type.

encrypt & decrypt The first translated functions are the `encrypt` en `decrypt` functions. The translation starts with the definition of `@Caesar.decrypt_internal` on line 39 in the code of Listing 2.8, and that of `@Caesar.encrypt_internal` on line 63. Both definitions use linkage type private.

Since these functions are available in ML for untrusted code, stubs are provided where security precautions can be made. These can be found on lines 23-37 and 47-61. The definitions of these stubs take only `%int` values as arguments: these are either real integer values, or masked indices.

The stubs are marked as private as well, and entry point functions are created that perform a tail call to these stubs. These can be seen on lines 8-11 and 13-16. These entry points specify no linkage type explicitly, which means linkage type `external` is used.

When calling these functions from insecure code, execution starts at the entry functions, as their memory locations are listed as entry points by the *SPM*. These entry functions perform a tail call to the stubs.

The stubs know that the second argument to the internal function must be of type `%Caesar.cred*`, so they use the `@unmask` function to get the corresponding `%Caesar.cred*` value, and type check it.

The definitions of the internal functions take one `%int` and one `%Caesar.cred*` as argument. They can read the integer value that this `cred` represents by accessing the representation, using the `getelementptr` function in combination with the pointer `%Caesar.cred*`.

The body of the internal function uses the effective values in two calls to arithmetical assembly functions and returns the result.

newcredentials Next is the `newcredentials` function. An entry function for this publicly available function is defined on lines 18-21, performing a single tail call to the stub. On lines 71-76 in the code of Listing 2.8, a stub for the function is defined and its return type is declared to be `%int`, since it returns an index in the masking list.

On line 72, the stub calls the internal function, which returns a pointer value of type `%Caesar.cred*`. The stub casts this pointer to an `%int`, and saves the `%int` as well as type information in the masking list using a call to `@mask`. This returns an index which can be returned to the context.

The body of the internal function `@Caesar.newcredentials_internal` implements the functionality of the ML `newcredentials` function. `@Caesar.newcredentials_internal` performs a call to the rand function, saving the resulting return value and creating a pointer to it. This pointer is returned.

rand The function `rand` is defined starting line 87 and onwards in Listing 2.8. As the structure `Caesar` is opaquely ascribed by the signature `SYMMETRICCIPHER`, and the value `rand` is not specified in this signature, it should be hidden from any outside components. This means the linkage type is set to `private` It also follows that no stub nor entry function is necessary for the `rand` function.

In its body, it returns the value of the `seed`.

seed The value `seed` is translated to a getter function. For the same reasons as the variable `rand`, its linkage type is set to `private`.

The definition of `seed` can be seen on lines 92-94 in the code of listing Listing 2.8.

2.5 Lessons learned

This chapter gave an introduction to the MiniML language, which was chosen because of its special module system and how it specifies security aspects. Formalizing the MiniML language semantics will be the topic of Chapter 3.

Furthermore, LLVM and its intermediate representation was introduced as the target language for the described compilation scheme.

The chapter proceeded by describing the compilation scheme, showing how *structures* can be mapped to LLVM *modules*, followed by a translation of its *fields* and *functions* to global variables and LLVM *functions*.

Signatures were shown to have no translation to a single LLVM concept, instead having an influence on the translation of the modules that it *seals* in linkage types and more subtle ways.

The chapter concluded with a discussion about preventing side channel communication, which makes it necessary to clear registers and flags when calling or returning to any external code.

The existence of opaque types in MiniML means that objects of these opaque types can only be created by methods inside the module that declares the type synonym. To ensure this, the use of masking and the tracking of type information proved necessary.

Chapter

3 Formal Specification

First, this chapter formally specifies the MiniML source language (Section 3.1). This formal specification is broken down in three parts: a description of the MiniML syntax (Section 3.1.1), the typing rules to which a correct program must conform (Section 3.1.2), and the operational semantics that specify how a program executes (Section 3.1.3).

In the same way, Section 3.2 formalizes the syntax of the LLVM Intermediate Representation. This chapter then concludes by formalizing a secure compilation scheme for the MiniML source language (Section 3.3).

3.1 MiniML

3.1.1 Syntax

First this section introduce the syntax of the MiniML language, as seen in Fig. 3.1, Fig. 3.2 and Fig. 3.3. Like ML, the MiniML syntax is composed of three parts [MTM97]: the *Core* language, a *Module* language, and the concept of a *Program*. The three languages, *Core*, *Module* and *Program* each have their corresponding valid *expression*s. The division between *Core* and *Module* mostly correlates to the concepts of 'programming in the small' and 'programming in the large' [MTM97, DK75] respectively. This separation of programming in the large and programming in the small aims to help programmers to introduce the right degree of modularity in their software.

Core language

The *Core* mainly consists of *value expressions* e. These express the manipulation of values and execution of functions to implement small algorithms or control logic. Every value expression e needs to have a corresponding type τ, otherwise the expression is not *sound*. The *Core* language syntax is shown in Fig. 3.1.

Module language

The *Module* language uses *module expressions* me to specify how the small parts of *Core* expressions can be 'glued together' or *composed* into larger, working programs. Its syntax is shown in Fig. 3.2.

Val Exp

e	$::= num\ n$	(natural number)
	$\mid id$	(value identifier)
	$\mid StrId.id$	(struct value)
	$\mid (e_1 \overline{e_2})$	(function application)
	$\mid \texttt{let}\ x : \tau\ =\ e_1\ \texttt{in}\ e_2$	(let binding)
	$\mid (e_1, e_2)$	(pair)
	$\mid e.\#1 \mid e.\#2$	(pair projection)
	$\mid []$	(empty list)
	$\mid e :: e$	(list concatenation)

Identifiers

x	$::= id$	(value identifier)
	$\mid StrId$	(structure identifier)

Types

τ	$::= \texttt{int}$	(int type)
	$\mid StrId.t\ \overline{\tau}$	(struct type)
	$\mid \overline{\tau_1} \rightarrow \tau_2$	(function type)
	$\mid \alpha$	(type variable)
	$\mid \tau_1 \times \tau_2$	(pair type)
	$\mid [\tau]$	(array type)

Figure 3.1: Core language syntax.

The *Module* language brings encapsulation and namespaces to MiniML. The *Module* language does this by introducing the concept of a *structure*, as was informally explained in Section 2.1. A structure is defined using a `struct` expression, and bound to an identifier *StrId* using the `structure` expression. It consists of a body of definitions which is denoted in the syntax as $\overline{\delta}$, using the bar notation for lists[1]. In its body, a structure *StrId* can bind values *id* to a value expression e, or can define a new type t.

Just as types in the *Core* language limit the number of valid or *sound* value expressions, signatures restrict the number of well-typed module expressions. Signatures are named and bound to their identifier using the `signature` expression. They are

[1] The bar notation uses $\emptyset$ as the empty set and the comma (,) as the prepend operator. For example:

$$\begin{aligned} \overline{\delta} &::= \emptyset \\ &\mid e : \tau, \overline{\delta} \end{aligned}$$

defined using the `sig` expression, and their body consists of abstract type definitions, type synonyms and value declarations.

A structure can be ascribed a signature using *transparant* (:) or *opaque* (:>) ascription.

Transparant ascription Transparant ascription *StrId* : *SigId* lets the implementation of type definitions in the underlying structure *StrId* propagate through, while hiding from external view any values that were not declared in the signature *SigId*.

Opaque ascription Opaque ascription *StrId* :> *SigId* restricts the external view of the structure to the values and types declared inside the signature *SigId*. If a type t is declared abstract in *SigId*, its implementation is not known to any code not local to the structure.

In order to obtain a *more simple* language to study, MiniML signatures restrict their declarations to a subset T of the *Core* types τ. The subset T contains only *opaque* types defined by structures, the `int` and function types. As a result, any value that is accessible from outside the structure itself can only get parameters and return values of type `int`, of an opaque type or of a function combination of those types.

This results in arrays $[\tau]$ and pairs $\tau_1 \times \tau_2$ not being primitive types for module expressions. Only within a structure can a value be treated as an array or pair, if the value is of an opaque type that is defined:

1. Within the same structure.
2. With an implementation containing the array or pair type.

Program

A MiniML program consists of a set of *Module expressions*. It is then concluded by a single naked value expression e, functioning as the main of the program. This description of a program allows us to first specify a set of signatures as well as a set of structures conforming to those signatures. The syntax of a Program is shown in Fig. 3.3

3.1.2 Type system

Having defined the syntax for MiniML and its parts, this section formalizes the static semantics, also called its *type system*. As stated in Section 3.1.1, types and signatures restrict the world of possible programs to those that consist of *sound* or *well-typed* module and value expressions. It is the type system that formalizes when exactly a module expression *me* is of the correct signature *SigId*, or a value expression e of the correct type τ.

The MiniML language type system is based on the Hindley-Milner type system [Hin69, Mil78]. While the Standard ML implementation provides type inference using the Damas-Milner type inference algorithm [DM82], the MiniML language assumes type annotations are available, written by the programmer. This is by no means a fundamental restriction, the MiniML language does not deviate from ML enough to prohibit the use of type inference.

Mod Exp

$me ::= \texttt{structure}\ StrId : SigId = \texttt{struct}\ \overline{d}\ \texttt{end}$ (trans. struct. binding)

$\mid \texttt{structure}\ StrId :> SigId = \texttt{struct}\ \overline{d}\ \texttt{end}$ (opaque struct. binding)

$\mid \texttt{signature}\ SigId = \texttt{sig}\ \Sigma\ \texttt{end}$ (sig. binding)

Definition

$d ::= \texttt{val}\ id = e : \tau$ (value def.)

$\mid \texttt{fun}\ id\ \overline{x}\ = e : \tau$ (value def.)

$\mid \texttt{type}\ \overline{\alpha}\ t = \tau$ (type def.)

Signature body

$\Sigma ::= \overline{\delta}$ (signature body)

Declaration

$\delta ::= \texttt{type}\ \overline{\alpha}\ t$ (abstract type declaration.)

$\mid \texttt{type}\ \overline{\alpha}\ t = T$ (type synonym)

$\mid \texttt{val}\ id : T$ (value declaration)

Shared type

$T ::= \texttt{int}$ (type int)

$\mid StrId.t\ \overline{\tau}$ (struct type)

$\mid \overline{T_1} \rightarrow T_2$ (function type)

$\mid \alpha$ (type variable)

Figure 3.2: Module language syntax.

Program

$P ::= me, P$ (module expression prefix)

$\mid e$ (program entry point)

Figure 3.3: Program language syntax.

Type judgements and Contexts

The type system validates programs by checking that a program P consists solely of *sound* or *well-typed* module expressions me and value expressions e. This checking is done using *typing judgements* that state whether a single expression e or me is well-typed. The typing judgement of an expression e is symbolized by:

$$\vdash e : \tau$$

The well-typedness of an expression e however does not depend on the expression e in isolation. Instead, parts of the program processed earlier by the type checker can have an influence on the well-typedness of the single expression e. This dependence

of an expression e on earlier expressions is formalized using the idea of a *context* Γ [Pie02].

This context keeps track of the type assumptions and type definitions as well as structure and signature definitions made earlier. The structure of a context Γ is shown in Fig. 3.4.

To access the mappings from *StrId* or *SigId* to its signature, its definitions or its declarations, the context allows for projections. For example $\Gamma[\mathit{StrId}].\overline{d}$ will look up the mapping $(\mathit{StrId} \mapsto \{\Sigma, \overline{d}\})$ in Γ and project this to the $\overline{d}$ specified in the mapping. A lookup will *fail* if the identifier has no mapping in the context.

Context

$$
\begin{aligned}
\Gamma ::= {} & \emptyset && \text{(empty context)} \\
| {} & (id : \sigma), \Gamma && \text{(identifier type assumption)} \\
| {} & (t = \tau), \Gamma && \text{(type definition)} \\
| {} & (\mathit{StrId} \mapsto \{\Sigma, \overline{d}\}), \Gamma && \text{(structure definition)} \\
| {} & (\mathit{SigId} \mapsto \Sigma), \Gamma && \text{(signature definition)}
\end{aligned}
$$

Figure 3.4: Contexts in the MiniML type system.

With the addition of contexts, type judgements become relations between a context Γ and expressions *me* or e, formalized as $\Gamma \vdash e : \tau$. This states that an expression or other part of the syntax is well-typed within the specified context Γ.

When typing judgement of a structure and its definitions occurs, the type system returns a new typing context Γ' to perform subsequent typing judgements. In this resulting context, all subsequent expressions must be well-typed.

The $\Gamma \vdash \Diamond$ judgement is a statement of well-formedness of a context Γ. A context is well-formed if the keyset of the lookup table it represents conforms to the standard notion of a set, meaning every key is used only once. This makes it invalid to rebind structures or signatures.

The possible typing judgements can be seen in Fig. 3.5.

$$
\begin{aligned}
\text{ExpressionTyping} &::= \Gamma \vdash e : \sigma \\
\text{ModuleTyping} &::= \Gamma \vdash me \rightarrow \Gamma' \\
\text{Well-formedness} &::= \Gamma \vdash \Diamond
\end{aligned}
$$

Figure 3.5: Typing judgements in the MiniML type system.

Type-schemes

The typing judgements, in their contexts, do not type an expression using a type τ. Instead they use an extension on the regular type τ called a type-scheme σ. The concept of a type-scheme is shown in Fig. 3.6.

$$\begin{aligned}\text{Type-Scheme } \sigma \ ::= \ & \tau \\ & \mid \forall\alpha.\sigma\end{aligned}$$

Figure 3.6: Type-schemes in the MiniML type system.

A type-scheme, sometimes called polytype, introduces a type of polymorphism called let-polymorphism [Pie02] by taking a type variable α in the definition of τ, and quantifying it with the universal quantifier $\forall$. This allows any concrete types τ to 'match' to the type variable, whereas an unquantified type variable a is unique and only matches itself. Note that the definition of a type-scheme assures that the resulting type-scheme is in *prenex normal form*, i.e. a string of quantifiers concluded by a quantifier-free ending.

Free variables Looking at a type-scheme σ in Fig. 3.6, it is clear that some of the type variables α are quantified, and others are not. Those type variables that are not quantified are called 'free' variables. The set of free variables within a type-scheme σ is denoted by the predicate $\mathit{free}(\sigma)$, and the value of this predicate is computed as follows:

$$\begin{aligned}\mathit{free}(\mathit{int}) &= \emptyset \\ \mathit{free}(\mathit{StrId}.t\ \overline{\tau}) &= \bigcup_{\tau\in\overline{\tau}} \mathit{free}(\tau) \\ \mathit{free}(\alpha) &= \{\alpha\} \\ \mathit{free}([\tau]) &= \mathit{free}(\tau) \\ \mathit{free}(\tau_1 \times \tau_2) &= \mathit{free}(\tau_1) \cup \mathit{free}(\tau_2) \\ \mathit{free}(\overline{\tau_1} \to \tau_2) &= \mathit{free}(\tau_2) \cup \bigcup_{\tau\in\overline{\tau_1}} \mathit{free}(\tau)\end{aligned}$$

The type-scheme introduces polymorphism because it possible for two type-schemes to match, even if they are not exactly the same. For example, the identity function `id` is typed $id : \forall\alpha.\alpha \to \alpha$. Because any type can be accepted as being of type α one can use the same `id` function everywhere regardless of the arguments type.

The way that two type-schemes match, even if they are not the same, is given by the two relations called *specialization/generalization*. Instinctively, the relations corresponds to the idea that a type variable α can be exchanged for any type τ, but that this must happen in a consistent way.

Type-scheme specialization: The specialization relation $\sigma_1 \geq \sigma_2$ expresses that σ_2 is more specialized than σ_1. This means that the following rule[2] determines the specialization relation.

[2] A rule uses the notation $\frac{\mathit{premise}}{\mathit{conclusion}}$ where *premise* is the set of conditions for that must hold for the conclusion to hold.

$$\frac{\tau_2 = [\alpha_i \mapsto \tau_i]\tau_1 \qquad \beta_i \notin \mathit{free}(\forall\alpha_1 ... \forall\alpha_n.\tau_1)}{\forall\alpha_1 ... \forall\alpha_n.\tau_1 \geq \forall\beta_i ... \forall\beta_m\tau_2'} \qquad \text{(specialization)}$$

In other words, a more specialized type-scheme can be obtained by consistently replacing quantified type variables α_i in the more general type-scheme by a type τ_i. This type τ_i is allowed to contain type variables itself. However, no type variable that was free in the more general type-scheme can become quantified in the more specialized type-scheme.

The first condition gives one the possibility to specify the type of a type variable. This second condition forbids one to *rescope* a type variable in the process.

Type-scheme generalization: Type-scheme generalization is the opposite process of type-scheme specialization. However, whereas specialization can be expressed independent of the context, whether or not one is allowed to generalize, is dependent on the context. Generalization allows one to quantify an unquantified variable, as long as it does not appear unquantified in any type assumption already made in the current context.

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathit{free}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma} \qquad \text{(generalization)}$$

Signature Matching

Just like set of valid value expressions e is restricted by the type system using types, the set of valid structure expressions is restricted using *signature ascription*. As mentioned earlier, these ascriptions can be either *transparant* or *opaque*. This subsection explains how a structure expression is restricted by a signature using the *signature matching* relation.

Every structure expression has a corresponding *principal signature* [Pie04], symbolized by $PS(\overline{d})$, where $\overline{d}$ is the body of the `struct` expression. The principal signature of a structure expression consists of all type specifications and all values with their corresponding types.

When a structure is ascribed with a signature, its principal signature must *match* with the ascribed signature. In this matching, the principal signature is called the *candidate*, while the ascribed signature is called the *target*. The signature matching relation is formalized using $\Sigma_{target} \succeq \Sigma_{candidate}$.

For a *candidate* signature to match its *target*, any value or type specification in the *target* signature must have an equivalent specification in the *candidate* signature.

1. For types, this means that for each type in the *target*, there is a type in the *candidate* with the same name. If *target* carries a definition for the type, *candidate* must provide an equivalent definition.

2. For values, this means that for each value in the *target*, there exists a value with the same name in the *candidate* whose type is a subtype [3] of the corresponding type in the *target*.

[3] Liskov substitution principle is used to determine subtypes i.e. arguments are allowed to generalize, return types can be specialized.

With these signature matching constraints, a structure can have a principal signature that is broader than its ascribed signature. The structure can define more types or values, and provide existing values with subtypes. These types and values can be used locally within the structure, but not in non-local code.

The effective signature of a structure *StrId* with body $\overline{d}$ is denoted with $ES(\overline{d})$, which corresponds to

- all type definitions present in $\overline{d}$, and the value declarations present in $SigId \Leftrightarrow StrId$ was transparantly ascribed with signature *SigId*.
- The signature bound to *SigId* if and only if *StrId* was opaquely ascribed with signature *SigId*.

Type definitions and value declarations in a signature can contain type variables. However, all type variables are assumed be quantified universally. This means, for example, that a type definition written as *type* $\overline{\alpha}$ $t = \tau$ contained in $\overline{d}$ results in a type definition $t\ \overline{\alpha} = \forall\alpha_1 ... \forall\alpha_n.\tau$ as an element of $ES(\overline{d})$.

Whether or not a structure's principal signature matches with a signature *Sig*, depends only on the structural characteristics *Sig* requires, and not on whether the structure was explicitly ascribed with *Sig*. For this reason, signature matching is a form of *structural typing*, as opposed to *nominal typing* [Pie04].

Note that signatures are in fact a set of type bindings and type definitions. Since a context can contain type bindings and type definitions as well, it is possible to take the union of a signature and a context Γ.

Type Rules

Using the type judgements specified earlier, it is possible to create a set of rules that specify when an expression e or me is valid. Some expressions are well-typed unconditionally and in any context, for example rule T-Num in the typing rules in Fig. 3.8. Others are only typed correctly if for example a certain subexpression is correctly typed. In this case, the typing rule becomes of the form shown in rule T-App.

The typing rules are divided in rules concerning the core language in Fig. 3.8, rules concerning the *Module* language in Fig. 3.9 and rules for the *Program* language in Fig. 3.7.

$$\frac{\Gamma \vdash me \to \Gamma' \qquad \Gamma' \vdash \Diamond \qquad \Gamma' \vdash P}{\Gamma \vdash me, P} \qquad \text{(T-Program)}$$

Figure 3.7: Typing rules for the Program language.

$$\Gamma \vdash num\ n : nat \qquad \text{(T-Num)}$$

$$\frac{\sigma_2 \geq \sigma_1 \qquad id : \sigma_2 \in \Gamma}{\Gamma \vdash id : \sigma_1} \qquad \text{(T-Mono)}$$

$$\frac{\Gamma \vdash e_1 : \overline{\tau_2} \to \tau_1 \qquad \Gamma \vdash \overline{e_2} : \overline{\tau_2} \qquad \tau_1 \neq \tau_3 \to \tau_4 \qquad \forall \tau \in \overline{\tau_2} : \tau \neq \tau_3 \to \tau_4}{\Gamma \vdash e_1 \overline{e_2} : \tau_1} \qquad \text{(T-App)}$$

$$\frac{\Gamma \vdash e_2 : \sigma \qquad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash let\ p\ =\ e_2\ in\ e_1 : \tau} \qquad \text{(T-Let)}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin free(\Gamma)}{\Gamma \vdash e : \forall \alpha . \sigma} \qquad \text{(T-Gen)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2)}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash e_1 :: e_2 : [\tau]} \qquad \text{(T-List)}$$

$$\Gamma [] : \forall \alpha . [\alpha] \qquad \text{(T-EmptyList)}$$

$$\frac{\Gamma \vdash e : \forall \alpha . \tau \times \alpha}{\Gamma \vdash e.\#1 : \tau} \qquad \text{(T-PairLeft)}$$

$$\frac{\Gamma \vdash e : \forall \alpha . \alpha \times \tau}{\Gamma \vdash e.\#2 : \tau} \qquad \text{(T-PairLeft)}$$

Figure 3.8: Typing rules for the Core language.

$$\Gamma \vdash SigId = \Sigma \to (SigId \mapsto \Sigma), \Gamma \qquad \text{(T-Signature)}$$

$$\frac{\Gamma[SigId].\Sigma \succeq PS(\overline{d}) \qquad \forall d_1 \in \overline{d} : PS(\overline{d} \setminus \{d_1\}) \cup \Gamma \vdash d_1}{\Gamma \vdash StrId : SigId = \overline{d} \to (StrId \mapsto \{ES(\overline{d}), \overline{d}\}), \Gamma} \qquad \text{(T-Structure)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{val}\ id = e : \tau} \qquad \text{(T-ValDef)}$$

$$\frac{\overline{(x = \tau_2)} \cup \Gamma\ \vdash e : \tau_1}{\Gamma \vdash \texttt{fun}\ id\ \overline{x} = e : \overline{\tau_2} \to \tau_1} \qquad \text{(T-FunDef)}$$

$$\frac{\overline{\alpha} = free(\tau)}{\Gamma \vdash \texttt{type}\ \overline{\alpha}\ t = \tau} \qquad \text{(T-TypeDef)}$$

$$\frac{StrId.id : \sigma \in \Gamma[StrId].\Sigma}{\Gamma \vdash StrId.id : \tau} \qquad \text{(T-ModVar)}$$

$$\frac{\Gamma \vdash e : \sigma_1 \qquad t\ \overline{\alpha} = \sigma_2 \in \Gamma[StrId].\Sigma \qquad \sigma_2 \geq \sigma_1}{\Gamma \vdash e : StrId.t\ \overline{\tau}} \qquad \text{(T-ModTransType)}$$

$$\frac{\Gamma \vdash e : StrId.t\ \overline{\tau} \qquad t\ \overline{\alpha} = \sigma_2 \in \Gamma[StrId].\Sigma \qquad [\bigcup_{\tau_i \in \overline{\tau}} \alpha_i \mapsto \tau_i]\sigma_2 = \sigma}{\Gamma \vdash e : \sigma} \qquad \text{(T-ModTransType2)}$$

Figure 3.9: Typing rules for the Module language.

3.1.3 Operational semantics

$$
\begin{aligned}
\text{Value } v &::= num\ n \\
&\mid (v, v) \\
&\mid [v] \\
\text{Module Table } T &::= \emptyset \\
&\mid (StrId \mapsto \overline{d}), T \\
\text{Evaluation} &::= T \vdash P \rightarrow T \vdash P'
\end{aligned}
$$

Figure 3.10: Relations and entities of the operational semantics.

The operational semantics defines a module table T, and the evaluation relation, shown in Fig. 3.10. This module table contains mappings from the module identifiers to their definition.

The module table T allows looking up the definition behind a certain identifier and accessing a certain part of it using projection. $T[StrId].SigId$ will give access to the *SigId* in the definition of *StrId*.

The evaluation relation allows the evaluation of a program P, which consists of expressions e and module expressions me, to a (simpler) Program P', while potentially making a lookup in T.

Rules

In Fig. 3.11, the evaluation rules of a MiniML program are shown. It first collects evaluates module expressions, skipping `signature`-bindings and adding `structure`-bindings to the table T. When a field inside a structure is referenced, the reference can be evaluated to its definition, with the exception that all references to fields y defined within the module are correctly substituted with $StrId.y$.

$$\frac{T \vdash e_1 \to T \vdash e_1'}{T \vdash (e_1, e_2) \to T \vdash (e_1', e_2)} \quad \text{(E-PairLeft)}$$

$$\frac{T \vdash e_2 \to T \vdash e_2'}{T \vdash (e_1, e_2) \to T \vdash (e_1, e_2')} \quad \text{(E-PairRight)}$$

$$\frac{T \vdash e_1 \to T \vdash e_1'}{T \vdash \mathit{let}\ p\ =\ e_1\ \mathit{in}\ e_2 \to T \vdash \mathit{let}\ p\ =\ e_1'\ \mathit{in}\ e_2} \quad \text{(E-Let)}$$

$$T \vdash \mathit{let}\ \mathit{id}\ =\ v\ \mathit{in}\ e \to T \vdash [\mathit{id} \mapsto v]e \quad \text{(E-LetV)}$$

$$\frac{T \vdash e_1 \to T \vdash e_1'}{T \vdash e_1 \overline{v} \to T \vdash [\overline{x} \mapsto \overline{v}]e_1'} \quad \text{(E-App1)}$$

$$\frac{T \vdash \overline{e_2} \to T \vdash \overline{v}}{T \vdash e_1\ \overline{e_2} \to T \vdash e_1; \overline{v}} \quad \text{(E-App2)}$$

$$\frac{(x = e' : \tau) \in T[\mathit{StrId}].\overline{d} \qquad e = [\mathit{this.y} \mapsto \mathit{StrId.y}]e'\ \forall(\mathit{this.y} \in e')}{T \vdash \mathit{StrId.x} \to T \vdash e} \quad \text{(E-ModField)}$$

$$T \vdash \mathit{StrId} : \mathit{SigId} = \overline{d}, P \to (\mathit{StrId} \mapsto \overline{d}), T \vdash P \quad \text{(E-StructDef)}$$

$$T \vdash \mathit{SigId} = \Sigma, P \to T \vdash P \quad \text{(E-SigExp)}$$

Figure 3.11: MiniML evaluation rules.

3.2 LLVM Intermediate Representation

The LLVM Intermediate Representation is a language very reminiscent of assembly. In contrast to assembly, it is strongly typed, and operates with a slightly higher level of abstraction.

This section largely builds on the work of Jianzhou Zhao et al. in formalizing the LLVM IR [ZNMZ12]. Their paper formalizes a reduced part of the LLVM IR in order to create mathematically verified program transformations. It provides a language syntax as well as static and dynamic semantics.

3.2.1 Syntax

In Fig. 3.12, the reduced syntax of LLVM IR is given, with some small changes from the work of Jianzhou Zhao et al. [ZNMZ12].

The syntax shows the idea of a module, which corresponds to a unit that can be compiled separately from other modules. Later, all modules are put together in a stage called *linking*.

A module consists, among others, of definitions and declarations:

Definitions are named values, whose name is unique within a module, defined within the module itself. They can be defined with linkage type `private`, which means that the linkage phase will not allow other modules to address these values.

Declarations represent dependencies on values defined externally. They can then be used within the module. Later, the linking phase will couple this declaration to an implementation provided by a different module.

$$
\begin{aligned}
\text{Modules } \mathit{mod} &::= \overline{\mathit{prod}} \\
\text{Products } \mathit{prod} &::= \mathit{id} = \mathit{link}\ \texttt{global}\ \mathit{typ}\ \mathit{const}\ \mathit{align} \mid \mathit{link}\ \texttt{define}\ \mathit{typ}\ \mathit{id}(\overline{\mathit{arg}})\{\overline{b}\} \\
&\quad \mid \texttt{declare}\ \mathit{typ}\ \mathit{id}(\overline{\mathit{arg}}) \mid \mathit{link}\ \texttt{constant}\ \mathit{typ}\ \mathit{const}\ \mathit{align} \\
\text{Linkage } \mathit{link} &::= \texttt{private} \mid \texttt{external} \\
\text{Types } \mathit{typ} &::= \texttt{i}\mathit{sz} \mid \texttt{void} \mid \mathit{typ}* \mid [\mathit{sz} \times \mathit{typ}] \mid \{\ \overline{\mathit{typ}_j}^{\,j}\ \} \mid \mathit{typ}\ \overline{\mathit{typ}_j}^{\,j}\ \} \mid \mathit{id} \\
\text{Values } \mathit{val} &::= \mathit{id} \mid \mathit{cnst} \\
\text{Binops } \mathit{bop} &::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{udiv} \mid \texttt{sdiv} \mid \texttt{urem} \mid \texttt{srem} \mid \texttt{shl} \mid \texttt{lshr} \mid \texttt{and} \\
&\quad \mid \texttt{or} \mid \texttt{xor} \\
\text{Extension } \mathit{eop} &::= \texttt{zext} \mid \texttt{sext} \mid \texttt{fpext} \\
\text{Cast op } \mathit{cop} &::= \texttt{ptrtoint} \mid \texttt{inttoptr} \mid \texttt{bitcast} \\
\text{Trunc op } \mathit{trop} &::= \texttt{trunc}_{int} \mid \texttt{trunc}_{fp} \\
\text{Constants } \mathit{cnst} &::= \texttt{i}\mathit{sz}\ \mathit{Int} \mid \mathit{typ}\ *\ \mathit{id} \mid (\mathit{typ}*)\ \texttt{null} \mid \mathit{typ}\ \texttt{zeroinitializer} \\
&\quad \mid \mathit{typ}\overline{[\mathit{cnst}_j]}^{\,j} \mid \overline{\{\mathit{cnst}_j\}}^{\,j} \mid \mathit{typ}\ \texttt{undef} \mid \mathit{bop}\ \mathit{cnst}_1\ \mathit{cnst}_2 \\
&\quad \mid \mathit{fbop}\ \mathit{cnst}_1\ \mathit{cnst}_2 \mid \mathit{trop}\ \mathit{cnst}\ \texttt{to}\ \mathit{typ} \mid \mathit{eop}\ \mathit{cnst}\ \texttt{to}\ \mathit{typ} \\
&\quad \mid \mathit{cop}\ \mathit{cnst}\ \texttt{to}\ \mathit{typ} \mid \texttt{getelementptr}\ \mathit{cnst}\ \overline{\mathit{cst}_j}^{\,j} \\
&\quad \mid \texttt{select}\ \mathit{cnst}_0\ \mathit{cnst}_1\ \mathit{cnst}_2 \mid \texttt{icmp}\ \mathit{cond}\ \mathit{cnst}_1\ \mathit{cnst}_2 \\
&\quad \mid \texttt{fcmp}\ \mathit{fcond}\ \mathit{cnst}_1\ \mathit{cnst}_2 \\
\text{Blocks } b &::= l\ \overline{\phi}\ \overline{c}\ \mathit{tmn} \\
\phi \text{ nodes } \phi &::= \mathit{id} = \texttt{phi}\ \mathit{typ}\overline{[\mathit{val}_j, l_j]}^{\,j} \\
\text{Label } l &::= \mathit{id}\texttt{:} \\
\text{Tmns } \mathit{tmn} &::= \texttt{br}\ \mathit{val}\ l_1\ l_2 \mid \texttt{br}\ l \mid \texttt{ret}\ \mathit{typ}\ \mathit{val} \mid \texttt{ret}\ \mathit{void} \mid \texttt{unreachable} \\
\text{Commands } c &::= \mathit{id}\ =\ \mathit{bop}(\mathit{int}\ \mathit{sz})\ \mathit{val}_1\ \mathit{val}_2 \mid \mid \mathit{id}\ = \texttt{load}(\mathit{typ}*)\ \mathit{val}_1\ \mathit{align} \\
&\quad \mid \texttt{store}\ \mathit{typ}\ \mathit{val}_1\ \mathit{val}_2\ \mathit{align} \mid \mathit{id}\ = \texttt{malloc}\ \mathit{typ}\ \mathit{val}\ \mathit{align} \\
&\quad \mid \texttt{free}(\mathit{typ}*)\ \mathit{val} \mid \mathit{id}\ = \texttt{alloca}\ \mathit{typ}\ \mathit{val}\ \mathit{align} \\
&\quad \mid \mathit{id}\ =\ \mathit{trop}\ \mathit{typ}_1\ \mathit{val}\ \texttt{to}\ \mathit{typ}_2 \mid \mathit{id}\ =\ \mathit{eop}\ \mathit{typ}_1\ \mathit{val}\ \texttt{to}\ \mathit{typ}_2 \\
&\quad \mid \mathit{id}\ =\ \mathit{cop}\ \mathit{typ}_1\ \mathit{val}\ \texttt{to}\ \mathit{typ}_2 \mid \mathit{id}\ = \texttt{icmp}\ \mathit{cond}\ \mathit{typ}\ \mathit{val}_1\ \mathit{val}_2 \\
&\quad \mid \mathit{id}\ = \texttt{select}\ \mathit{val0}\ \mathit{typ}\ \mathit{val}_1\ \mathit{val}_2 \mid \langle \mathit{id} \rangle\ = \texttt{call}\ \mathit{typ}_0\ \mathit{val}_0\ \overline{\mathit{param}} \\
&\quad \mid \mathit{id}\ = \texttt{getelementptr}(\mathit{typ}*)\ \mathit{val}\ \overline{\mathit{val}_j}^{\,j}
\end{aligned}
$$

Figure 3.12: The reduced LLVM Syntax, largely built upon Jianzhou Zhao et al. [ZNMZ12]

3.3 Formalized Compiler

This section formalizes the secure compilation process. The compilation process is a translation of a program P expressed in MiniML to a program written in the LLVM IR. It is symbolized as the translation function $[\![\bullet]\!]$.

The compiler formalized here aims to be a secure compiler, it takes a set of module expressions and translates them to LLVM IR to allow for fully abstract [Aba99, ASJP12] binary protected modules or *SPM*s. As explained in Section 1.1, fully abstract compilation uses the notion of *contextual equivalence* ($\simeq$) [ASJP12] to describe the requirements of a secure compiler.

These requirements of the compiler are interpreted as follows: The program P given to the compiler is a set of module expressions *me* that represent the secure code. This set of module expressions *me* is compiled separately from any other, insecure code to an LLVM IR program $P^{\downarrow}$ which can be processed to a fully abstract *SPM*. This means that if this $P^{\downarrow}$ is later compiled further to an object file and linked together with a context C, there exists no context C that can tell from $P^{\downarrow}$ from the compilation $P'^{\downarrow}$, where $P'^{\downarrow}$ is the compilation result of a source-level contextually equivalent MiniML program P'.

The result of the translation is expected to run as an *SPM* on a low-level machine model as used in [ASJP12, PCP13]. This model is a Protected Module Architecture or *PMA*, with access control semantics as explained in Section 1.2.

The result of the translation will be loaded within the protected code, and its data section will correspond to the protected data section of the *PMA*. The formalization of the translation function $[\![\bullet]\!]$ will provide clear annotations for additional actions that can not be specified within the LLVM IR language. These are actions such as the specification of entry points in the *SPM* layout or the clearing of low-level registers and flags.

3.3.1 Formalization

First, before translating the program, 4 LLVM IR types are defined:

```
%int = type i64 ; 1
%tyvar = type {%int, %int} ; 2
%pair = type {%tyvar*, %tyvar*} ; 3
%array = type {%int, [0 x %tyvar*]} ;4
```

These will be used to represent the MiniML int, α, (τ, τ) and $[\tau]$ types.

Type variables are represented by a tuple of integers. The first integer is a pointer to the value, cast to an int. The second integer is an integer that identifies the effective type of the value stored inside the type variable.

To do this, every MiniML type τ must correspond to an integer τ_{int}. The int, α, (τ, τ) and $[\tau]$ types are defined to correspond with integers 0 to 4.

Next, the real translation of the program, which corresponds to a set of module expressions *me*, starts.

All module expressions are sorted based on the name used in the binding. Two programs with differently ordered module expressions *me* are contextually equivalent as long as long as no module expression *me* has unresolved dependencies on other module expressions when it is bound. Sorting module expressions based on the name

used in the binding makes sure that all MiniML programs that provide the same set of bindings result in an LLVM IR program with the corresponding bindings happening in the same order.

$$[\![\overline{me}]\!] = [\![me_1, me_2, \ldots, me_n]\!] \quad \rightarrow \quad \text{sort}([\![me_1]\!], [\![me_2]\!], \ldots, [\![me_n]\!])$$

Then, all module expressions can be translated individually. The signature *SigId* used in the ascription of *StrId* is saved:

$$\begin{array}{lll} [\![me]\!] & \rightarrow & [\![StrId : SigId = \overline{d}]\!] \\ [\![StrId : SigId = \overline{d}]\!] & \rightarrow & [\![\overline{d}]\!]^{SigId} \end{array}$$

All fields in a module are sorted alphabetically as well. This is done for the same reason as the sorting of module expressions: two MiniML programs P differing only in the order of their fields are contextually equivalent. To prevent leakage of this ordering in the target-level, outputting the fields in the target language happens in a fixed ordering.

$$[\![\overline{d}]\!]^{SigId} = [\![d_1, \ldots, d_n]\!]^{SigId} \quad \rightarrow \quad \text{sort}([\![d_1]\!]^{SigId}, \ldots, [\![d_n]\!]^{SigId})$$

Now, every definition is translated separately. The translation of a definition depends on a number of things:

- Whether the definition is that of a type or that of a field.
- Whether or not the value is local to the structure *StrId*. This depends on the value being declared in the signature *SigId*.

$[\![\texttt{type } \overline{\alpha}\ t = \tau]\!] \rightarrow$ `%t = type` $\{\tau\}$
[The type and its definition are tracked. The type is assigned a unique integer τ_{int} to identify it. This way, when a masked pointer is received in an entry stub, its type can be checked.]

$[\![\texttt{val } id = e : \tau]\!]^{SigId} \rightarrow$ `define private` τ`* @`id`_internal(){` $[\![e]\!]^{SigId}$
`}`
[The next part is only included if the value is declared in the signature]

```
define private %int @id_stub() noinline{
    [ Switch stack, move parameters.]
    %0 = call τ* @id_internal()
    %1 = ptrtoint τ* %0 to %int
    %2 = call %int @mask(%int %1,%int
 τ_int)
    [Switch stack, clear registers.]
    ret %int %2
}
define %int @id(){
```

[Add entry point in SPM metadata for @id, pointing to this location.]
`%ret = tail call %int @`id`()`
`ret %int %ret`
`}`

$[\![\texttt{fun } id\ \overline{x} = e : \tau]\!]^{SigId} \rightarrow$

`define private` τ`* @`id`_internal(`$\overline{\tau_x*\ x}$`){`
$[\![e]\!]^{SigId}$
`}`
[The next part is only included if the value is declared in the signature]
`define private %int @`id`_stub(`$\overline{\%int\ y}$`)`
`noinline{`
[Switch stack, move parameters.]
$\overline{\%z}$ `= call %int @unmask(`$\overline{\%int\ \%y}$`)`
$\overline{\%t}$ `= call %int @unmasktype(`$\overline{\%int\ \%y}$`)`
$\overline{\%c}$ `= icmp eq %int` $\overline{\%t\ \%\tau_{x,int}}$
`br i1` $\overline{\%c}$`, label %Continue, label %Error`

`Continue:`
$\overline{\%x}$ `= inttoptr %int` $\overline{\%z}$ `to` $\overline{\tau_x*}$
`%0 = call` τ`* @`id`_internal(`$\overline{\tau_x*\ \%x}$`)`
`%1 = ptrtoint` τ`* %0 to %int`
`%2 = call %int @mask(%int %1, %int` τ_{int}`)`
[Switch stack, clear registers and flags.]
`ret %int %2`

`Error:`
`call void @exit(i32 -1)`
`unreachable`
`}`
`define %int @`id`(`$\overline{\%int\ y}$`){`
[Add entry point in SPM metadata for @id, pointing to this location.]
`%ret = tail call %int @`id`(`$\overline{\%int\ y}$`)`
`ret %int %ret`
`}`

The code of the `@mask`, `@unmask` and `@unmasktype` functions is given in Listing A.1 in App. A. It provides a naive implementation of `@mask`, `@unmask` and `@unmasktype` functions using linked lists.

If a function receives polymorphic arguments, for example `createPair :: a -> a -> Pair a`, the internal functions arguments are of type `%tyvar*`. When the function is called from insecure code, the stub loads the parameters from the masking list and wraps them in a `%tyvar`.
Before passing these arguments to the internal function, the compiler inserts code to check the consistency of the underlying type equation, using the `@tyvarcheck` function. The code implementing `@tyvarcheck` is given in Listing A.2 in App. A.
As an example, Section A.3 in App. A provides an example of the compilation of a polymorphic function.

The translation of an expression poses no additional security concerns and is compiled to the function or value body. Since Section 2.2.1 states the module expressions making up the *SPM* are compiled in a separate file, before the context, nothing in these expressions can depend on things defined in insecure code. This is not a fundamental restriction, it is possible to have these expressions depend on outside code by either:

- Allowing dependencies on the context to be hardcoded, which would correspond to the high-level MiniML interpretation that the context is split up in two parts. The first part is compiled before the file containing the secure module expressions, the second part after the secure module expressions.

  The resulting security issues could by resolved by introducing a returnback entry point into the *SPM*, as described by Agten et al. [ASJP12].

- Adding the functionality of callbacks to the language, using closures.

3.3.2 Implementation

An implementation of this formal compiler was written in Scala. This secure compiler takes an abstract syntax tree (AST) representing a MiniML program as its input. The nodes of this AST correspond to MiniML's syntactical constructs, as specified in Section 3.1.1. It assumes that the program represented by this AST is correctly typed, i.e. it has passed a type check as specified by the type system in Section 3.1.2.

As a result of the compilation, it outputs a file with extension .ll, containing the LLVM IR translation of the MiniML program. An example output of this compiler was shown in Section 2.4.2.

3.4 Conclusion

This chapter formally specified a simple version of the MiniML source language and the LLVM IR target language. It then gave a formal representation of the secure compiler from MiniML to LLVM IR.

The MiniML language as sketched here does not yet contain the more advanced concepts found in ML's well-known module language. The next chapter aims to rectify this by introducing both *higher-order functions* and *functors* to the MiniML language.

Chapter

4 Advanced ML Concepts

This chapter extends the capabilities of the subset of the ML language. Section 4.1 introduces *higher-order functions* and discusses how they can be compiled securely. Afterwards, Section 4.2 explains *functors* and their addition to MiniML.

4.1 Higher-Order Functions

A *higher-order function* is a function that allows other functions to be given as input or returns a function as output. MiniML by treats functions as first-class values, meaning that functions represent an entity that can be passed around as a parameter or return value and can be assigned to a variable.

An example of a higher-order function is shown in Listing 4.1. The function `addCurried` takes an argument `x` and as a result returns another function. Therefore, `addCurried` is a higher-order function.

Since MiniML uses what in literature is known as *lexical scoping*, the function that addCurried returns is allowed access to the non-local or free variable `x`, even though it is not defined within the local scope of the function, because its defined scope *lexically surrounds* the definition of the function.

Listing 4.1: The use of lexical scoping calls for closures.

```
1 fun addCurried x =
2   let innerFunction y = x + y in innerFunction
```

When a function is created using a call to `addCurried` (and possibly saved in a variable to call it later on) this function must be able to access the variable `x` that was given as a parameter to `addCurried`. The function entity created and possibly saved in a variable in other words must keep track of the non-local variables it has access to. This is implemented in MiniML using the concept of *closures* [AP03]. A closure consists of a simple function reference together with a list of all the free variables and their values.This list of non-local variables and their values is called the *referencing environment*.

It is possible for these free variables to be complex data such as arrays or abstract data types, for example the Dictionary from Listing 2.1.

Security concerns regarding higher-order functions present themselves in several different ways:

- The code of the function can be defined within trusted code or untrusted code.
- When is a closure saved within trusted memory or within untrusted memory?
- Every free variable of a higher order function can originate from trusted or untrusted code alike. How do the origin of the free variable and the location of the closure interact?
- Closures can be passed around as a value, enabling them to cross the security boundary.

The compilation scheme presented here requires that the referencing environment are saved in the trusted memory when the closure is created in secure code, and in untrusted memory when it is created in untrusted code. The creation of a closure happens when a function is used as a value. For example, Listing 4.1 returns the `innerFunction` that it defines as a value. If Listing 4.1 is located in secure code, its referencing environment and pointer is saved in trusted memory. If it were located in insecure code, the closure would be located in untrusted code. The location of a closure in memory defines the security status of a closure: A secure closure is one that is saved in trusted memory.

For named functions, whose name can be used to pass them as a value, the code where their name is used defines the security status. For example in Listing 4.2, the closure that represents the add function is considered to be created by the code of Listing 4.2, even if this is the insecure context.

This is explained by looking at the unsugared version of this code, as shown in Listing 4.3. In this desugared form, it is clear that the code presented in Listing 4.3 creates the closure. The desugared form finds its justification in examples where the function is already passed one or more of its parameters while leaving the remaining parameters unspecified, a technique called *currying*. This technique allows for the function insert, defined by the dictionary code of Listing 2.1, to be passed as a value with the dictionary into which the key-value pair must be inserted already defined using this code: `let closureValue = insert emptyDictionary`.

Listing 4.2: Passing a predefined function.

```
1 foldl (add) emptyList values
```

Listing 4.3: Passing a predefined function, unsugared.

```
1 foldl (\x y -> add x y) emptyList values
```

Now that the security status of closures had properly been defined and the closure value itself is assigned to trusted or untrusted memory, it is possible to look at the other problems of higher-order function handling, such as the origin of free variables, or the passing of closures across the security boundary.

4.1.1 Secure Free Variables For A Secure Closure

In this case, the code that is executed when the closure is called is located in the secure code section. The *function pointer* and the *referencing environment* are both in secure memory and are protected from any tampering by the target-level access control model and the checking performed upon compilation.

4.1.2 Insecure Free Variables For A Secure Closure

If a closure is created by trusted code with free variables stemming from untrusted code, the values of these free variables should be copied to the referencing environment inside the closure. This is in contrast to the more straightforward low-level solution of having a pointer in the referencing environment to the original value in insecure memory. This need for copying arises because the low-level code could otherwise change the value of the free variable at any time, as it would be located inside untrusted memory, even mid-execution of the higher-order function. Listing 4.4 gives an example of two functions that are vulnerable to this kind of attack.

Listing 4.4: Changing free variables inside untrusted memory mid-execution can break contextual equivalence.

```
1 fun generateClosure1 freevar =
2   let innerFunction x =
3     (let b = freevar
4     in x + callback 2 + b)
5   in innerFunction
6
7 fun generateClosure2 freevar =
8   let innerFunction x = x + callback 2 + freevar
9   in innerFunction
```

The two functions shown in Listing 4.4 are contextually equivalent in MiniML. Their compiled versions however are not contextually equivalent unless the free variables from untrusted code are copied to the trusted memory. If no copying of free variables occurs, an attacker could set distinguish a module using `generateClosure1` from one using `generateClosure2` using the following attack:

Call-by-value Attack

1. After calling the closure, the attacker forces execution to temporarily be passed back to the unsafe code. This is achieved in Listing 4.4 by means of the callback function. Even when no callbacks are available this type of attack remains possible. For example, in a more powerful language providing a multithreaded environment, this can be forced using a context switch.

2. The context uses the fact that `freevar` is saved in unprotected memory to change the value of `freevar`. This not possible in the source language, but the target language provides no guarantees against modification of the unprotected memory.

3. If the implementation of `generateClosure1` was used, then the result of the closure application depends on when exactly execution was temporarily switched to the attacker's context. If this happened after copying the value of `freevar` to `b`, which is saved in protected memory, then the result of the closure will not reflect the change of value of `freevar` by the attacker's context. If the execution switch happens before this copying of `freevar` into protected memory, then the result will be computed using the changed value of `freevar`.

In contrast, a version using `generateClosure2` never copies the value of `freevar` into protected memory. Therefore the result of the closure will always be computed using the changed value of `freevar`.

The attack described above shows that it is possible to manipulate the low-level versions of the code of Listing 4.4 in such a way that the two different functions do not give the same result. Consequently, this means that the contextual equivalence of the compiled code is broken, whereas the source code is unaffected by the attack. After all, the source language does not allow the memory location of the parameter that was passed to be changed. As a result, step 2 is only possible in the low-level language.

This problem is effectively mitigated by ensuring that upon *application* of the closure every value is copied to the secure environment, conforming to a value-passing call semantic.

However, the MiniML language as described in this work restricts the values that can pass the border between secure code and insecure context to values of either type int or an opaque type. Int types are always passed using their effective value. Opaque types are not susceptible to the attack described above because they can only be manipulated using functions from the structure that declared the opaque type.

4.1.3 Insecure Free Variables For An Insecure Closure

This combination entails no interaction between the attacker's context and the *SPM* beyond the regular means of function calling. This means that it is not necessary to introduce security measures beyond those discussed in the previous chapters. How closures and their values are represented is only important to the extent to which implementation for this case and the other cases might be shared by tackling the problem in a generic way.

4.1.4 Secure Free Variables For An Insecure Closure

As stated in Chapter 2, values originating from secure code should not leave the safety of secure memory provided by the low-level acces control model. Instead, these values passed must be masked and represented in the insecure code by their masking index. These same measures are necessary when working with closures. As long as we obey these measures, and represent secure free variables in the referencing environment by their masking index, they are not susceptible to tampering, illegal disclosure or any other manipulation attacks. The only way to interact with these secure free variables is by passing them as parameters to functions within the *SPM*. This shows that the added power of closures does not require any new security measures for this interaction between *SPM* and the attacker's context.

4.1.5 Cross-boundary Passing Of A Secure Closure

Now that it is possible to have a secure closure as a first class value, it can be passed as an object across the boundary between *SPM* and the attacker's context. This section

explores the security measures that must be taken when this occurs. If an *SPM* would pass a closure generated inside its body to the untrusted code, the untrusted code could change the value of the function pointer and inspect or change the values saved in the referencing environment.

Each of these actions break contextual equivalence:

Change pointer value. The pointer to the code might be changed to code that makes certain assumptions about the structure used to implement abstract data types. This way, the context could discern between contextually equivalent *SPM*s in which the assumptions may or may not hold.

Inspect environment values. Assumptions about the way abstract data types are structured can be checked by inspecting the environment values.

Change environment values. Changing values in the reference environment allows an attacker to discern between a higher order function that first copies its free variables and an equivalent higher order functions that does not, as shown in Listing 4.4.

As a result, the closure entity itself should not be passed across the boundary between the *SPM* and the trusted code. Instead, the closure should be masked, just like any value, as explained in Section 2.4.1. When a closure should be passed to the untrusted code, its corresponding index in the masking map of the *SPM* is passed instead. In order to later execute the closure, the *SPM* should offer a *closure-evaluation entry point*. This entry point first takes a pointer to a closure in the masking map. Then, because it is located within the *SPM*, the entry point is able to jump to the function pointer specified by the closure and run the function code.

4.1.6 The Closure-Evaluation Entry Point

Because closures can be passed freely, it is possible for insecure code to obtain a closure value. Of course this insecure code is allowed to make use of the closure, and execute the underlying function. Execution of a closure is also called the *evaluation* of the closure. Application of a secure closure by the code in this context is not possible in a direct and straightforward way: the context is not capable of jumping to the code representing the closure because the code that corresponds to the closure might not be located at an entry point of the *SPM* and because it has only knowledge of the masking index of the closure.

To allow insecure code to execute a closure, the *SPM* offers one generic *closure-evaluation entry point*. This entry point should:

1. Take a masked index as a parameter
2. Allow for an unspecified amount of other parameters to be passed as well, which will be relayed as parameters to the function represented by the closure.
3. Copy the parameters provided by the attacker's context in order to prevent *call-by-value attacks* as shown in Listing 4.4.
4. Look up the masked index and retrieve the closure being referenced, i.e. the function pointer and the referencing environment containing the free variables.

5. Jump to the function pointer providing copies of any parameters it might require, as well as the referencing environment.

6. When execution of the closure is finished, the closure-evaluation entry point provides cleanup of the copied values and performs the tasks necessary for all entry points to the *SPM* code such as register and flag emptying.

4.2 Functors

This section introduces the MiniML concept of functors. Functors can be seen as *functions* from structures to structures. They accept a structure that conforms to a given signature and create a different structure as a result.

Functors are a way of parametrizing structures. They can be used to implement a structure that depends on behavior provided by another structure, making only limited assumptions regarding the way this behavior is implemented. This method of abstraction can be used to create generic data structures.

For example, using functors the dictionary example from Chapter 2 can be made into a more generic data structure. The dictionary structure as given in Listing 2.1 is not type independent since only values of type string are allowed to be used as keys.

Ideally however, how a dictionary works would be described in a generic way that does not care whether the key is of type string, int or any other type. Instead, it needs only the assurance that some specific functionality is provided by the type. The generic dictionary using functors is shown in Listing 4.5. Its alternative implementation is shown as well in Listing 4.6.

Listing 4.5: A generic dictionary that can take any type of the EQUAL typeclass as its key.

```
1  signature DICTIONARY = sig
2          type key
3          type 'a dictionary
4          val emptyDictionary : 'a dictionary
5          val insert : 'a dictionary -> key -> 'a -> 'a dictionary
6          val lookup : 'a dictionary -> key -> 'a
7      end
8  signature EQUAL = sig
9          type t
10         val equal : t -> t -> bool
11     end
12 structure StringEqual: EQUAL = struct
13         type t = string
14         fun equal t1 t2  = case String.compare(t1,t2)
15                                of EQUAL => true
16                                | _ => false
17     end
18 functor DictionaryFn (KeyStruct:EQUAL) :> DICTIONARY where type key =
       KeyStruct.t = struct
19         type key = KeyStruct.t
20         type 'a dictionary = (key * 'a) list
```

```
21          val emptyDictionary = []
22          fun insert d x y = (x,y)::d
23          fun lookup |[] x = error
24                     |(key,value):ds x = if(KeyStruct.equal key x)
25                                         then value
26                                         else (lookup ds x)
27      end
28  structure StringDict = DictionaryFn(StringEqual);
```

Listing 4.6: The alternative implementations of the dictionary.

```
1   functor DictionaryFn(KeyStruct:EQUAL) :> DICTIONARY where type key =
        KeyStruct.t = struct
2           type key = KeyStruct.t
3           type 'a dictionary = (key list * 'a list)
4           val emptyDictionary = ([],[])
5           fun insert (a,b) x y = (x::a,y::b)
6           fun lookup |[] x = error
7                      |(key:ks, value:vs) x = if(KeyStruct.equal key x)
8                                              then val
9                                              else (lookup (ks,vs) x)
10      end
```

In order to parametrize over the type of key, the dictionary signature of Listing 2.2 was expanded with an extra type named `key`, as shown in line 3 of Listing 4.5. On top of that, lines 10-14 specify a signature `EQUAL` that defines the interface to which a type must comply in order to be a possible key for a dictionary. Lines 15-21 specify a structure `StringEqual` that conforms to the signature needed to act as a key type for a dictionary.

Finally, lines 24-34 show how to define a functor in MiniML. First the argument structures and their corresponding signatures are specified, in this case `KeyStruct`, whose signature must match signature `EQUAL`. Next, the structure resulting from functor application is bound to the `DICTIONARY` signature using opaque ascription. The choice for opaque ascription ensures that the specific implementation of dictionaries, i.e. as a list of pairs (Listing 4.5) or a pair of lists (Listing 4.6), remains hidden. As a consequence, the dictionary type is an abstract type whose values only can be created by the dictionary structure that results from the functor application.

However, the signature `DICTIONARY` hides the implementation of the `key` type. This is a problem, since a user must be able to create values of type `key` and pass them to insert function. The type of values that is put inside the dictionary does not suffer from the same problem as its type is specified parametrically as type variable `'a` in the type definition `type 'a dictionary`.

This problem can not be solved by specifying the type inside the `DICTIONARY` signature, since then it loses the flexibility of allowing multiple different types of keys. Nor can it be made a type variable in the type definition `type 'a dictionary` because type variables are *universally quantified*. This means any type can replace them, whereas a dictionary only makes sense if an equality check is possible on the type of its keys. In other words, specifying the key as a type variable (`type ('key, '`

`a) dictionary`) is unsatisfactory because then the ability to put constraints on the chosen type is lost.

Instead, the problem is solved by modifying our opaque ascription when it is applied to the structure resulting from the functor `DictionaryFn`, by specifying that they `key` type is equal to the type `t` specified in the `KeyStruct` that was passed as the functor's argument (Line 24). This modification makes the type of key freely available, yet still dependent on the specific key structure that was used when the dictionary structure was created using the functor.

The last line of Listing 4.5 shows how the functor `DictionaryFn` is eventually applied to the `StringEqual` structure, resulting in a dictionary structure saved as `StringDict` that uses `string` as type for its keys, and compares them using the regular `String.compare` function. This way of specifying a functor allows for very easy change of the `key` type, as well as the way that they are compared. For example, it would require only a minimal change to create a dictionary that disregards capitalization when comparing its `key`s of type `string`, and it is even possible to cleanly use this alternative implementation in conjunction with the case-sensitive implementation.

Note that in the example above, the structure `StringEqual` was a valid parameter to functor `DictionaryFn` because `StringEqual`s signature matched to the `EQUAL` signature that the functor expected. In the example of Listing 4.5 this is straightforward, the transparent ascription of `StringEqual` with signature `EQUAL` already assures this. However, the signature matching relation allows the *candidate* signature to be broader than the *target* signature. For functor application this means that the argument structure that is passed can define more types or members (values or functions) than the functor expects. The existence of these types and members however is not assured and the functor can not use them in its body.

As was shown in line 36 of Listing 4.5, where the `DictionaryFn` functor is applied, the addition of functors requires that structures become values, so that they can be passed to functors to create new structures. However, they are not first-class values: they are bound to names using the special `structure` keyword, and can only be returned by functors, not by any *Core* language expression. As all conditional expressions are part of the *Core* language, structure bindings are always unconditional, which means every structure binding is determined fully before compile time.

4.2.1 Security Status of the Resulting Structure

Now that functors are added to the MiniML language, it is necessary to determine the security status of functors. When is the output structure of a functor considered to be part of the *SPM*, and when is it part of the insecure context? This question has a simple, yet perhaps surprising answer: The output structure resulting from a functor has its security status determined by the location of the code defining the functor.

To see why this is the logical choice, consider the example definitions of a dictionary functor in Listing 4.5 and Listing 4.6, and the two possible locations for the functor source code:

In secure code: Suppose `DictionaryFn` is defined in secure code, for example when it is part of a library that is offered as a protected module. This library could choose to implement the `DictionaryFn` as in Listing 4.5 (line 24-34) or as in Listing 4.6. Regardless of whether its argument, `KeyStruct`, was defined in

secure code or in insecure code, the specific implementation of this dictionary is supposed to be abstract. The resulting dictionary structure is secure. This means that values of types that are defined by the functor can only be returned to the context as masked values. In its implementation, the functor can freely call functions of the (possibly insecure) structure, it must however take the same precautions as any callback to insecure code.

In insecure code: Suppose `DictionaryFn` is defined in insecure code. Now the choices made when implementing the dictionary are not expected to be abstract. There is no implementation hiding in this case.

This is supported by the consideration that one could replace the functor code with the non-parametrized structure that would result from functor application. This structure would then be a part of insecure code, and of course one would expect that the security status of structures does not change simply due to the use of functors.

A possibly surprising conclusion is that this argument holds, regardless of whether the functor application was done within the secure code or within the insecure code. As a consequence, application of a functor inside the attacker's context can result in the creation of an additional secure structure.

4.2.2 Compilation of Functors

With the security status of functors determined in Section 4.2.1, this section aims to give an overview of the possible methods for implementing functors.

Compilation to the target language has two distinct possibilities of handling structures and functors:

- Structures, signatures, functors and functor applications can be compiled away, a technique called *Static Interpretation*. [Els99] This approach is taken by MLKit[1]. As a result, multiple applications of the same functor result in multiple generations of the functor's body, specialized for every combination of parameters.
- Structures can be compiled, keeping the members defined by the structures together in the target language in a record like manner called a *frame*. Functors can then be compiled using generic code, which receives a reference to the frame representing the structure that is passed as parameter. The code of these functors dynamically calls the functions inside the structure received as an argument.

The second option is chosen, since it does not duplicate code and is the more standard way of implementing compilation of the Module language of ML. Also, if functors are compiled away, the code for each resulting structure can only be generated when compiling the functor application. This would result in either:

- losing the ability to apply a secure functor outside of secure code
- having the code representing secure functors outside of secure memory.

[1] Available at http://www.elsman.com/mlkit/

Choosing the second option means that all members (values or functions) of functors are implemented as stubs that take a pointer to a frame as an additional argument.

The stubs can implement any generic code, and use the information inside the passed frame to call any necessary member defined by the argument structure.

Referencing structures differently in the source and target language raises new security issues when considering functors. In the source language, structures can simply be referenced by their name. In the target language however, structures are translated to frames and are referenced by indicating the location of the frame in memory loc_{frame}.

Section 4.2.3 expands on the representation and references of structures in the target language, and discusses whether this different representation can be exploited by an attacker.

4.2.3 Target Representation of Structures

In order to pass structures to functors as an argument, structures are summarized in the target language as a tuple of structure name, and a list of pointers that represent the structure members. This target representations are called *frames*.

These frames reside in the memory section corresponding to their security status, for example structures corresponding to the *SPM* have their frame saved in the protected memory. This prevents tampering with the frames that correspond to secure structures by the attacker's context, such as changing the pointers to the different structure members or changing their order.

The Structure of Frames

Structures in MiniML are either defined using the `struct` construct or are the result of functor applications to a structure (Fig. 4.1). Both cases must be represented in the target language using frames.

$$\begin{aligned} \mathit{Structure}\ S ::= {} & \mathit{structure}\ \mathit{id} = \mathit{struct}\ \cdots\ \mathit{end} \\ & |\mathit{structure}\ \mathit{id} = F(S) \end{aligned}$$

$$\mathit{Functor}\ F ::= \mathit{functor}\ \mathit{id} = \mathit{struct}\ \cdots\ \mathit{end}$$

Figure 4.1: Excerpt from the MiniML syntax related to structures.

In their most basic form, when defined using the `struct` construct, structures are a collection of values and functions. Frames represent these structures as a combination of structure name and a list of pointers to these values and functions, sorted in alphabetic order. For example, the frame representation of the `StringEqual` structure is shown in Fig. 4.2. Values or functions inside these structures are now accessible using knowledge of the frame location and the index of the accessed value in the list of pointers.

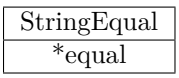


Figure 4.2: The frame representation of the `StringEqual` structure.

To represent functors and the output structure of their application, more information has to be saved. For one, functors have an argument, another structure on which it can depend. To represent the result of a functor application, one needs to track which structure was passed as an argument, i.e. its frame location, as well as the values and functions defined by the functor. The functor has access to values or functions inside the argument structure using the combination of frame location (loc_{frame}) and the index of the necessary member in the list of pointers inside that frame ($index_{mem}$).

When accessing these values or functions, the functor can only compute this index based on the members specified in the expected argument signature. However, as mentioned in Section 4.2, the argument structure can define more members than specified in the argument signature. Clearly the functor should not access these members. Yet the index of an expected value or function as computed from the argument signature could be different from the effective index of the corresponding member in the frame.

As a result, the view of the argument structure to the functor must be *trimmed* to that specified the expected signature. Because the signature of the argument structure as well as the expected signature are known at compile time, this *trimming* process can be performed at compile time. The results of this trimming process can be saved in the mapping $i_{expected} \mapsto i_{effective}$.

Concluding, frames that represent the results of functor applications consist of:

1. Functor name
2. A pointer to the frame of the argument structure
3. The trimming map
4. The list of pointers to the output structure's values and functions, sorted alphabetically.

The frame representing `StringDict` from Listing 4.5 is shown in Fig. 4.3.

Because the argument structure is represented by a pointer to its corresponding frame, a functor can easily be applied to a structure that already results from functor application. This allows iterative function application without any additional modifications.

In order to distinguish frames that represent simple structures from those that represent structures resulting from functor application, the type of frame is tracked using a single bit right at the start of the frame. For clarity, this value is not shown in the figures representing frames.

4.2.4 Creating the Frames

Section 4.2.3 specified how frames can represent structures. It also concluded that in order to avoid tampering with the representation of secure frames, secure frames

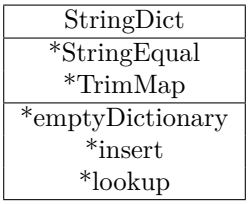


Figure 4.3: The frame representation of the `StringDict` structure, the output of applying functor `DictionaryFn` to structure `StringEqual`.

must be located inside the protected memory of the *SPM*. This section shows that this restriction means not all frames representing structures can be created at compile time (statically), but instead some must be created dynamically.

Problem Example

When compilation is done within the context of security all secure code is compiled first, and only afterwards is the insecure context compiled. This *separate compilation* means that the compiled result of secure code is created before compilation of the insecure context starts.

For example, the earlier example of Listing 4.5 could be split up in a secure part and an insecure part as shown in Listing 4.7 and Listing 4.8 respectively. The code representing the functor is compiled first and separately from the code in Listing 4.8. The *frame* representation for any structures defined inside Listing 4.7 can be created when compiling this file. However, as Section 4.2.1 explains, any structure that results from the application of `DictionaryFn` is supposed to be secure as well, meaning their corresponding frames should be saved in secure memory. Not all these applications are known when compiling Listing 4.7. The application of `DictionaryFn` to `StringEqual` inside Listing 4.8 results in a new secure structure being created *after* the secure code was compiled.

Listing 4.7: Secure code fragment: secure.ml.

```
1  structure StringEqual: EQUAL = struct
2          type t = string
3          fun equal t1 t2  = case String.compare(t1,t2)
4                                  of EQUAL => true
5                                  | _ => false
6      end
7  functor DictionaryFn(KeyStruct:EQUAL) :> DICTIONARY where type key =
       KeyStruct.t = struct
8          type key = KeyStruct.t
9          type 'a dictionary = (key * 'a) list
10         val emptyDictionary = []
11         fun insert d x y = (x,y)::d
12         fun lookup |[] x = error
13                    |(key, value):ds x = if(KeyStruct.equal key x)
14                                         then value
```

```
15                                          else (lookup ds x)
16     end
```

Listing 4.8: Fragment of the insecure context: context.ml.

```
1  ...
2  structure StringDict = DictionaryFn(StringEqual);
```

As a consequence, not all frames can be created statically and added to memory at compile time. Instead they must be added dynamically to preserve the invariant that *all frames representing secure structures are located in secure memory.* The locations of these dynamically created frames in the *SPM* are kept in a list of frames, called the *f-list*.

Dynamically Creating Frames

The problem described above only poses itself with functor applications: the binding of secure `struct` expressions to a name always happens in secure code. It's the application of a secure functor, which can happen in insecure code as well, that creates the problem. How the dynamic creation of frames is handled depends on the location in the source language of the binding of the structure they represent to an identifier.

Binding in secure code: When compiling the secure code, it is possible to create frames for all structure bindings inside the secure code, regardless whether they correspond to `struct` expressions or functor applications. Their frames can be added dynamically using preprocessing code located inside the *SPM*. Since they are added by code inside the *SPM*, they are guaranteed to be free from tampering.

Binding in insecure code: The bindings in insecure code require more attention when processing. When the structure is defined using the `struct` expression or the application of an insecure functor, the result will be an insecure structure, whose frame is allowed to be located in insecure memory. This means that the frame can be added dynamically using preprocessing code inside the attacker's context without any further considerations.

When the structure is defined by the application of a secure functor, the result is a secure structure. This means that the frame must be saved in secure memory. The output of compilation of the secure code is already created, so the compilation must provide a function that the context can call to create the frame. This function corresponds nicely with the idea of `DictionaryFn(StringEqual)` being a function from structure to structure.

For each secure functor *Fn*, the compiler adds an entry point to the *SPM* to a function that allows the context to create a frame that would result from applying that functor. This function expects two arguments:

- The index in the f-list that corresponds to the frame that represents the argument of the functor, or the memory location of this frame if the argument is an insecure structure

- The trimming map that is applied to the structure.

Using this information, the function validates its input, creates the frame in secure memory and adds it to the *f-list* before returning.

The validation step is necessary because the preprocessing code is located in insecure memory. Because this code section is not protected, these preprocessing function calls could be manipulated. An attacker could for example apply the functor *Fn* to a secure argument structure S_a that does not match the required signature for functor *Fn*. This would result in the ill-typed output structure S_o.

Structure S_a can now access members created by S_o directly, because both are located in secure code. By choosing the unsound type assumptions that structure S_a makes carefully, this could leak information about the implementation of functor *Fn*.

Summarizing, when compiling a secure functor the compiler must:

- Add an entry point in the *SPM*'s metadata corresponding to a function that can create frames representing applications of the functor.
- Equip this function with the necessary checks to assure that when the argument structure is secure, the arguments signature also matches the expected signature specified by the functor definition. In other words, code must be added that reads type information about the argument structure and checks it, using *structural typing* as specified in Section 3.1.2. This is done to ensure that functor application on secure structures only succeeds if the secure structure signature matches with the target signature.

By preserving the order in which functors are applied when adding the frames dynamically, the compiler knows at compilation time the position of every frame in the *SPM*'s *f-list*. This way, the index in the f-list is a valid substitute for the real location loc_{frame} in memory.

Implementing Structural Typing Checks

In order to implement *structural typing checks* when applying secure functors, knowledge about the signatures of structures must be available at runtime. Because every secure functor has its own dedicated function that creates the frame and performs the necessary type checks, any necessary knowledge about the target signature can be hardcoded.

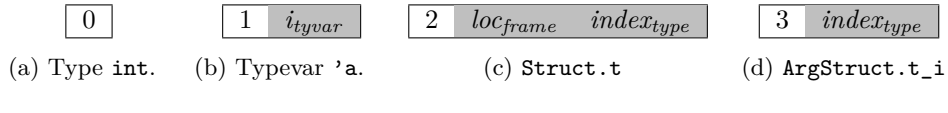
The signatures of structures within the *SPM* are saved as meta-information linked together with the frame, called the *meta-frame*. This meta-frame tracks all type definitions and value or function definitions present in the signature. It needs to formalize a target language representation of types, type definitions and value or function definitions.

The representation of types within the *SPM* is structured as shown in Fig. 4.4. It consists of representations for the primitive type *int*, for type variables $'a$, for types defined in other known structures and for types within the argument structure of a functor.

- The primitive type `int` does not need any additional information for identification.

- Type variables `'a` are denoted only by an index, which says when it was declared. This identifies the type variables uniquely, and since type variables are matched with *alpha-equivalence*, their name is not important.
- Types `Struct.t` defined in other known structures S are identified by specifying the location of the corresponding frame F_S and its index in this frames type definitions.
- Types `ArgStruct.t` defined in the argument structure of a functor are identified in a analogous way to types `Struct.t`. They have a different identifier to mark that they depend on the argument structure, and the location of the frame in which they are defined is known only at runtime, so it is not provided.

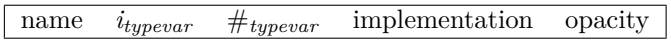
Then, the representation of two composite types is shown, arrays `[]` in Fig. 4.4e and pairs `(,)` in Fig. 4.4f.



(a) Type `int`. (b) Typevar `'a`. (c) `Struct.t` (d) `ArgStruct.t_i`

(e) Type `[]`. (f) Type `(,)`.

Figure 4.4: This figure shows the representation of all different types



(a) Type Definition

(b) Value Definition

Figure 4.5: The representation of type **(a)** and member definitions **(b)**.

The representation of type definitions is shown in Fig. 4.5a, and that of member definitions in Fig. 4.5b.

For the signature of an argument frame to match the target-signature, every type definition in the target signature and every member must have a match in the meta-frame, as explained in Section 3.1.2:

Matching type definitions First, a match is searched for all type definitions within the target signature: Searching inside the meta-information for a matching type definition can be done by scanning the meta-information for a type definition with the same name. If no type definition with the same name is available, the two signatures do *not* match.

If such a type definition is available, check the number of type variables $\#_{typevar}$ it expects. If the type definition in *target* and *candidate* signature take a different number of type variables, the two signatures do *not* match.

If the matching type definitions take the same number of type variables, then the opacity of the type definition in the *target* signature is of importance. If the type is *not* defined opaque, the implementation of the two matching type definition is matched to ensure the same implementation.

Matching members Next, for all value or function definitions inside the *target* signature, a matching definition inside the *candidate* signature is identified using the names. In this matching, only the members which correspond to a result of the *trimming map* are considered.

When a name match is found, its type is checked. When creating a meta-frame, any type used in the typing of members is substituted by its implementation until the first opaque type is encountered. This ensures that checking whether the type of two members match can be done by checking that either the type stated in the candidate signature is exactly the same as the type in the target signature, or it is a type variable that substitutes *consistently* for the type listed in the target signature.

Creating Meta-Frames

Implementing *structural typing* required knowledge about the signatures of structures to be available at runtime. It introduced *meta-frame*s as a type of meta-information linked together in memory with the frame. These *meta-frame*s track all type definitions and member (value or function) definitions present in the structures signature.

The content of these *meta-frame*s can mostly be determined at compile time. This surely is the case for structures, but it also is the case for functors that are not modified by a `where` expression. This means that the function that creates a frame corresponding to a functor application can largely hardcode the content of the corresponding meta-frame. When creating a meta-frame, any type used in the typing of a member is substituted by its implementation until it contains only opaque types, counting `int` as an opaque type.

Handling where Expressions Only `where` expressions of the template `where type t = ArgStruct.t'` can not be processed at compile time. For these expressions, the compiler must produce the structural typing checks so that they check:

1. That the implementation of type `t` is in fact `ArgStruct.t`.
2. What the opaqueness and implementation of type `ArgStruct.t` is, so that any reference to type `t` or `ArguStruct.t` can be substituted by its implementation.

4.2.5 Calling Structure Members

Access to any member of a source language structure (i.e. one of its values or functions) is uniquely determined in the target language using the location (loc_{frame}) of the frame in memory and the index of the member ($index_{mem}$) in its list of pointers.

In the source language, all members are either accessible across the security boundary or strictly accessible only within the same structure, so the same must hold for the target language. It is easy to make members accessible only from within the

same structure: they are not given an entry within the list of member pointers, and their location is not made into an entry point for the *SPM*.

All other members are accessible across the security boundary. However, as frames are saved in the memory corresponding to its security status, the attacker's context is not able to access frames corresponding to structures within the *SPM* directly. Worse, the attacker's context is not even allowed to know the exact location loc_{frame} of these secure frames.

This is solved by inserting all loc_{frame} into the *SPM*'s list of frames (*f-list*) that was introduced in Section 4.2.4.

Frames that were created in secure code are sorted alphabetically by the identifier they were bound to and put in the *f-list* first. This alphabetical ordering is necessary because changing the order of structure bindings does not break contextual equivalence at the high-level MiniML language. The frame locations of frames created in insecure code are appended in the order of their definitions. This means that the specific mapping of an identifier to the value of its index is known statically.

Anywhere a secure structure is used in the source language, it is represented in the target language by the loc_{frame} pointer or its masking index in the *f-list*.

To allow context code to access members from secure structures, a getter function is provided that is located inside the *SPM*. This getter function corresponds to an *entry point* to the *SPM*. The context can now pass an index in the *f-list* $index_{f-list}$ and an index for the structure member $index_{mem}$, as well as any arguments that should to be passed to the member to this getter function. This getter, because it is located in secure code, can access the *f-list* and the frames themselves. It then does the following things:

- Check whether $index_{f-list}$ corresponds to a real frame by using the expected length of the list. Since structures resulting from a functor application are added dynamically, a situation could arise where the corresponding frame has not yet been added, or where something went wrong while adding the frame. In this case the getter function must not allow the $index_{f-list}$ to be used.

- Use $index_{f-list}$ to get a loc_{frame} for the correct frame.

- Check whether the frame represents a simple structure or the result of functor application.

- Look up $index_{mem}$ in the list of pointers within the frame to get a pointer to the accessed member.

- Call this member with any relevant arguments. In case the frame represents the result of functor application, the pointer to the frame must be provided as a parameter when calling the member. The function implementing the member will uses this frame to determine the argument frame.

- Return any results to the caller.

The getter functions only as an additional level of indirection, so it presumes that any clearing of flags, registers, masking of values or other precautionary measures have already been taken care of by the functions that it passes the call to.

Since all structure values are accessible through this getter function, the separate structure value stubs do *not* have to be entry points in the *SPM*. The getter function will always be necessary, because insecure functors that call a member in their argument structure can not hardwire a specific entry point in their generic code.

Indeed, the stubs corresponding to structure members *cannot* be made entry points in the *SPM*, because the source language context is oblivious to whether a structure bound in secure code was the result of a functor application, or was statically defined. This corresponds to Listing 4.9 being contextually equivalent to Listing 4.10.

Listing 4.9: Binding StrA and StrB using functor application.

```
1 functor F(ArgStruct:ArgSig) :> OutputSig = struct
2         val x = ArgStruct.f x
3     end
4 structure StrA = F(Arg1)
5 structure StrB = F(Arg2)
```

Listing 4.10: Binding StrA and StrB using both static definition and functor application.

```
1 functor F(ArgStruct:ArgSig) :> OutputSig = struct
2         val x = ArgStruct.f x
3     end
4 structure StrA :> OutputSig = struct
5         val x = Arg1.f x
6     end
7 structure StrB = F(Arg2)
```

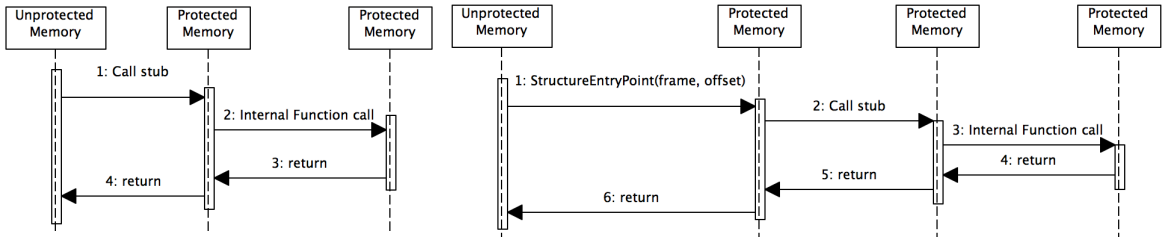
If stubs were made entry points in the target language, the difference between static definition and function application would break contextual equivalence.

- Compiling Listing 4.9 means calling value `StrA.x` would use the same entry point as calling `StrB.x`.
- Compiling Listing 4.10 means calling value `StrA.x` would use an entry point different from the one used when calling `StrB.x`

Clearly, this would result in Listing 4.9 and Listing 4.10 not being contextually equivalent in the target language.

The addition of a generic getter function that is used to access any value or function defined on a structure adds an additional level of indirection in calls between the insecure context and the secure code.

To illustrate this, Fig. 4.6a shows the flow of execution corresponding to a function call in the simple MiniML language or the work of Agten et al. [ASJP12]. As a comparison, Fig. 4.6b shows the flow of execution for that same call in the MiniML language with advanced concepts.



(a) Execution flow in simple MiniML. (b) Execution flow in the MiniML language.

Figure 4.6: Comparison of execution

4.2.6 Security Consequences

In order to explore the possibility of contextual equivalence breaking due to the representation of structures differing between source and target language, the possible combinations of functor and structure security and the relevant security measures are recapped here case by case.

Secure functor and secure structure The *frame* representing a structure is located in the same part of memory as the structure was defined. Since in this combination both the functor code and the frame are located inside protected memory, no tampering is possible. When functor code is called with a reference to the frame in secure memory, the functor code is able to read the frame directly, because the functor is part of the *SPM*. It can then call any member of the argument structure using the pointer provided by the frame.

If the application of the secure functor is done inside insecure code, the frame creating function corresponding to this functor must be called, and the types are checked dynamically using type information in the *meta-frame*.

Secure functor with insecure structure When a secure functor is applied to an insecure structure, its result is a secure structure that calls functions provided by the insecure structure. The resulting structure *depends* on the insecure structure. It must call these functions using the same precautions as regular callbacks to the attacker's context. Of course the pointers that represent these functions can be manipulated by the attacker because the frame is now located in insecure memory. However, because contextually equivalent implementations of the functor must perform the same callbacks with the same arguments in the source language in the same order, their target language versions will always use the same pointers provided by the insecure frame, with the same parameters in the same order. As a result, the regular precautions, such as clearing registers, switching the stack and masking values that pass from secure code to the insecure context, are sufficient to ensure preservation of contextual equivalence of the functors in the target language.

Additionally, when calling the code in insecure context, the real return address should be saved in secure memory, and the return address entry in the insecure stack should be modified to point to the returnback entry point [ASJP12]. This returnback entry point is listed in the entry points of the *SPM*. The returnback

entry point retrieves the last real return address saved in memory, and jumps to this address.

If the application of the secure functor is done inside insecure code, the frame creating function corresponding to this functor must be called, and the types are checked dynamically using type information in the *meta-frame*.

Insecure functor with insecure structure In this case, both the functor code and the frame are located inside unprotected memory. The structure that results from the application is considered to be part of the attacker's context. The use of members from the insecure structure by code inside the insecure functor can happen without any security precautions, as the expected behavior is simply equivalent to that of an unparametrized structure in the attacker's context that depends on another structure of this same context.

Insecure functor with secure structure Because the code of the functor is now located outside of the *SPM*, special care needs to be taken when the functor depends on members of the structure. Because the frame Fr_s representing the structure is now in secure memory, the functor cannot manipulate this frame or perform reads on this frame. In order for insecure functors to call these functions, an entry point is provided by the *SPM*, corresponding to a generic getter that takes as argument a loc_{frame} and an $index_{mem}$, as well as any arguments that would otherwise be passed directly. As a result, the generic getter returns the result of the call to its caller.

4.3 Conclusion

In this chapter, some more advanced concepts of ML were added to the MiniML language: higher-order functions and functors.

Higher order functions, and the closely related concept of closures, can be introduced if certain sensitive information, such as environment values and pointer values are confined to the secure memory. This is done by reusing the earlier concept of *masking* [PCP13], and applying it to closures. In order to execute these closures, a *closure-evaluation entry point* was introduced.

Functors were added by introducing the *target* language concept of a *frame* that represents structures. Implementing values defined by functors using stubs that expect an argument representing the parameter structure makes generic translation of these values in the *target* language possible. The dichotomy between the security status of an output structure and where the functor application generating the structure is located results in the *dynamic* creation of *frames* and the added difficulty of *structural type checking* functor applications. This is solved by adding meta information in so called *meta-frame*s capturing any necessary information of types.

Because functors can exist in insecure code, they can be passed frames. These frames are *masked* in the same way as closures, which calls for another *getter function* to be made available as an entry point into the *SPM*, so that the values or functions within frames are callable.

Additionally, because functors can perform callbacks to the insecure context, a returnback entry point is added that can handle returns from these callbacks.

Chapter

5 Advanced Concepts: Formalization

First, Section 5.1 of this chapter extends the formalization of MiniML made in Section 3.1 with the additions described in Chapter 4. Later, Section 5.2 extends upon the secure compiler as it was formalized in Section 3.3.

5.1 Formal Specification of MiniML

The MiniML language as specified in Chapter 3 consisted of a very limited subset of the ML language. In Chapter 4, this issue was addressed by adding two more concepts to the subset:

Higher-order functions The concept of *higher-order functions* allows that functions receive other functions as their input, or return functions as their output. By allowing them to be assigned as a value to a variable, functions essentially become *first-class values*, going by the name *closure*.

To implement closures, the ability to create anonymous functions using the λ notation is added. This addition makes it possible to define a function as: $\lambda(\overline{x} : \overline{\sigma}).e$. This expression is considered as the construction of a closure.

These expressions are also the *only* way of constructing a closure, as explained in Section 4.1. Essentially, when a named function is used as a closure, this is considered to be syntactic sugar. Its desugared form is an expression in which a closure is explicitly constructed using the λ-notation to wrap around the named function.

Functors The idea of functors is that they act as functions from structures to structures. They provide a way to parametrizing structures.

A parametrized structure is a functor F that, instead of depending on another structure Str_2 by name, receives this structure as an argument. To ensure that the structure given as an argument defines the necessary bindings, the argument structure is restricted by a signature Sig. The functor definition defines all bindings of the parametrized structure. In short, the definition of a functor corresponds to `functor` id`(`$ArgStr : Sig_1$`):>`Sig_2 `= struct` $\overline{d}$ `end`.

The application of a functor F to a Str_1, written as $F(Str_1)$ then results in a new structure Str_2 that can be bound to an identifier. This resulting structure,

or *output* structure, can be used like any regular structure. Any time the functor bindings uses $ArgStr.x$, Str_2 uses $Str_1.x$.

These additions have important consequences for the way MiniML is compiled. To be able to describe these consequences, the formalization of MiniML as given in Chapter 3 is revisited.

5.1.1 Syntax

The additional concepts introduce some new syntactical constructs to the MiniML syntax. The additions to the Core syntax are seen in Fig. 5.1, additions to the Module syntax in Fig. 5.2. The Program syntax remains unchanged by the additions from Chapter 4.

Closures

Closures introduce the λ-expression to the Core syntax, Fig. 5.1. It specifies a number of argument, their identifiers and their types, as well as the expression that represents the function body.

Val Exp

$e ::= \ldots$

$\quad \mid \lambda\, \overline{x}.e : (\overline{\tau_1} \to \tau_2)$ (anonymous function)

Figure 5.1: The Core syntax, extending Fig. 3.1.

Functors

Functors introduce the functor definition and functor application as module expressions to the Module syntax Fig. 5.2.

A functor definition specifies:

- the functor identifier,
- the identifier *ArgStr* for the argument structure,
- the signature Sig_1 to which the argument structure must comply,
- the signature Sig_2 to which the functor's body must comply,
- and the body $\overline{d}$.

A functor application specifies the identifier to which the resulting structure is bound, the functor identifier, and the identifier for the structure that is passed as an argument.

Mod Exp

$me ::= \ldots$

$\mid$ `functor` $FunId$`(`$ArgStr : SigId_1$`):>`$SigId_2$ `= struct` $\overline{d}$ `end` (opaque functor)

$\mid$ `functor` $FunId$`(`$ArgStr : SigId_1$`):`$SigId_2$ `= struct` $\overline{d}$ `end` (trans. functor)

$\mid$ `structure` $StrId_1 = FunId(StrId_2)$ (functor application)

Figure 5.2: The Module syntax, extending Fig. 3.2.

5.1.2 Type system

The type system also needs some minor modifications to account for the new concepts. As earlier, due to its simplicity the Program type system sees no modifications.

First, the concept of a *context* Γ is extended in Fig. 5.3, so that it can contain assumptions related to functors: The signature it conforms to, its body of definitions, and the name of the signature that its argument must conform to.

Context

$\Gamma ::= \ldots$

$\mid (FunId \mapsto \{\Sigma, \overline{d}, SigId\}), \Gamma$ (functor definition)

Figure 5.3: Contexts in the MiniML type system, extending Fig. 3.4.

Next, the typing rules are updated, as shown in Fig. 5.4 and Fig. 5.5.

Closures

The addition of closures, as a Core concept, modifies the function application rule T-App inside the Core typing system in Fig. 5.4, so that it allows arguments or return values to be of the function type. It also adds the λ-expression typing rule T-Lambda. This rule describes that a λ-expression is well typed if its defining expression can be typed to the result type, assuming each argument is of the stated argument type.

$$\frac{\Gamma \vdash e_1 : \overline{\tau_2} \to \tau_1 \qquad \Gamma \vdash \overline{e_2} : \overline{\tau_2}}{\Gamma \vdash e_1 \overline{e_2} : \tau_1} \quad \text{(T-App)}$$

$$\frac{\overline{(x : \tau_2)} \cup \Gamma \vdash e : \tau_1}{\Gamma \vdash \lambda\ \overline{x}.e : \overline{\tau_2} \to \tau_1} \quad \text{(T-Lambda)}$$

Figure 5.4: Updated typing rules for the Core language, extending Fig. 3.8.

Functors

Functors are strictly a Module language concept, which means modifications are limited to Module language, updated in Fig. 5.5. It adds two rules:

T-FunctorDef A rule to type a functor definition. This rule says that the *principal signature* of the body ($PS(\overline{d})$) must match with the signature specified in the ascription of the functor. Furthermore, each definition in the body must be correctly typed, assuming the well-typedness of all other definitions and the values defined in the argument structure.

T-FunctorApp A rule that types functor applications. This rule says that a functor application is well typed if the effective signature of the structure that was passed as an argument matches to the expected signature. Furthermore, it specifies that the structure is saved in the context. The output structure's identifier maps to the result signature of the functor and the body of the functor after substituting all references to the argument structure.

$$\frac{\Gamma[SigId].\Sigma \succeq PS(\overline{d}) \qquad \forall d_1 \in \overline{d} : PS(\overline{d} \setminus \{d_1\}) \cup \Gamma \vdash d_1}{\Gamma \vdash StrId : SigId = \overline{d} \rightarrow (StrId \mapsto \{ES(\overline{d}), \overline{d}\}), \Gamma} \quad \text{(T-FunctorDef)}$$

$$\frac{\Gamma[SigId_2].\Sigma \succeq PS(\overline{d}) \qquad \forall d_1 \in \overline{d} : PS(\overline{d} \setminus \{d_1\}) \cup \Gamma \vdash d_1}{\Gamma \vdash FunId(ArgStr : SigId_1) : SigId_2 = \overline{d} \rightarrow (FunId \mapsto \{ES(\overline{d}), \overline{d}, SigId_1\}), \Gamma} \quad \text{(T-FunctorApp)}$$

Figure 5.5: Updated typing rules for the Module language, extending Fig. 3.9.

5.1.3 Operational Semantics

The updated formalization of MiniML is concluded with a description of the updates to the operational semantics.

Fist, the concepts of values v is extended in Fig. 5.6 to reflect the addition of closures as a *first class value*. A closure is a pair, consisting of an expression and an environment in which this expression is evaluated. The environment consists of a set of bindings of identifiers to values, as shown in Fig. 5.7.

$$\begin{aligned} \text{Value } v ::= \ & \ldots \\ & \mid (e, \rho) \end{aligned}$$

Figure 5.6: Updated concept of 'values', extending Fig. 3.10.

Next, the concept of the module table T is extended, as shown in Fig. 5.8 so that it can keep track of the definitions of functors. The module table maps functor identifiers to their body of definitions.

$$\begin{aligned}\text{Environment } \rho ::= \; & \emptyset \\ & |(x \mapsto v), \rho\end{aligned}$$

Figure 5.7: Environments.

$$\begin{aligned}\text{Module Table } T \; ::= \; & \ldots \\ & | \; (\mathit{FunId}) \mapsto \overline{d}, T\end{aligned}$$

Figure 5.8: Updated concept of a 'module table', extending Fig. 3.10.

Closures

The addition of closures adds two evaluation rules, one concerning their definition and one for their application, shown in Fig. 5.9.

E-ClosureDef This rule describes how a λ-expression is evaluated. The λ-expression creates a closure value, consisting of the expression e and the environment. The environment contains bindings for all variables used in the defining expression e, but not defined within the expression. These are called the non-local of free variables of e, defined as $\mathit{FreeVar}(e)$.

E-ClosureApplication This rule describes how the evaluation of a closure happens. When the first expression of regular function application is evaluated to a closure value, it the function application is evaluated to the defining expression with all the arguments identifiers substituted with their values, and all mappings inside the environment applied.

$$\frac{\rho = \mathit{FreeVar}(e)}{T \vdash \lambda\overline{x}.e \to T \vdash (e, \rho)} \qquad \text{(E-ClosureDef)}$$

$$\frac{T \vdash e_1 \to T \vdash (e_2, \rho)}{T \vdash e_1\overline{v} \to [\overline{x} \mapsto \rho \cup \overline{v}]e_2} \qquad \text{(E-ClosureApplication)}$$

Figure 5.9: The evaluation rules concerning closures, extending on Fig. 3.11.

Functors

Adding functors to the language means there must be a way to evaluate functor definitions and functor applications. This means the additions of functors introduces two evaluation rules, as described in Fig. 5.10

E-FunctorDef When a functor definition is encountered, a mapping from the functor identifier to its definition body should be saved in the module table T.

E-FunctorApp When a structure is bound by a functor application, a mapping for the structure identifier is added in the module table T.

$$T \vdash \mathit{FunId}(\mathit{ArgStr} : \mathit{SigId}_1) = \overline{d}, P \rightarrow (\mathit{FunId} \mapsto \overline{d}), T \vdash P \qquad \text{(E-FunctorDef)}$$

$$T \vdash \mathit{StrId}_1 = \mathit{FunId}(\mathit{StrId}_2), P \rightarrow (\mathit{StrId}_1 \mapsto [\mathit{ArgStr} \mapsto \mathit{StrId}_2]\overline{d}), T \vdash P \qquad \text{(E-FunctorApp)}$$

Figure 5.10: The evaluation rules concerning functors, extending on Fig. 3.11 and Fig. 5.9.

5.2 Formalized Compiler

This section adapts the secure compiler formalized in Section 3.3.1 to the security measures and concerns introduced with the addition of closures and functors in Chapter 4. The context within which compilation happens is considered to be unchanged from that described in Section 3.3.

5.2.1 Formalization

A few additional base types are defined, to represent frames, meta-frames and closures. These types are the straightforward implementation of their representations as defined in Chapter 4. [1] The type `[0 x i8]*` is used to point to names and type `{%int, [0 x` *type*`]}*` is used for lists.

```
%frame = type {[0 x i8]*, %frame*, i1, {%int, [0 x %int]}*, {%int, [0
    x %int (%frame*, i8*)*]}*, {%int, [0 x %int]}*, %metaframe*}
%metaframe = type {{%int, [0 x %typedef]}*, {%int, [0 x %valuedef]}*}
%typedef = type {[0 x i8]*, %int, %int, %type*, i1}
%valuedef = type {[0 x i8]*, %type*}
%type = type {%int, %int, %int}
%closure = type {%int, {%int, [0 x %int]}*, i1}
```

The processing of module expressions *me* still occurs in alphabetical order of the identifiers.

$$[\![\overline{me}]\!] = [\![me_1, me_2, \ldots, me_n]\!] \quad \rightarrow \quad \text{sort}([\![me_1]\!], [\![me_2]\!], \ldots, [\![me_n]\!])$$

[1] A frame saves a little more information than mentioned in Chapter 4. Besides the name, target frame, trimming map and list of values, it also indicates whether the target frame is located in secure or insecure code, and has a map of the types that it defines to the integers τ_{int} that represent them in LLVM.

Static structure bindings

Processing structure bindings using the `struct` expression happens in much the same way as detailed in Section 3.3.1. The following changes are made:

- As explained in Section 4.2.5, the *SPM* no longer creates an entry point for every separate stub. Instead, all values are accessed through the structure value entry point, which calls the stubs. In this call, the stubs get an additional argument, a pointer to the frame corresponding with the stubs defining structure.
- Stubs get a pointer to the location where its arguments are saved. The stub can cast this pointer to the right type and copy all of the arguments to secure memory.

  This way, the LLVM IR type of any stub is exactly the same: `%int (%frame *, i8*)*`. Giving all the stubs the same type is beneficial because then their pointers can all be saved together in an LLVM array: `{%int, [0 x %int (% frame*, i8*)*]}*`.
- The code creating a representation of the structure as an LLVM value of type `%frame` and corresponding `%metaframe` is built while processing the structure.

  The target frame pointer is set to null in this value, well as the pointer to the trimming list, since these values are irrelevant for a staticly defined structure.

  This code is output by the compiler only later, inside the @initialize function.

The stubs still perform the masking and unmasking, as well as the stack switches or clearing the registers and flags.

Functor Definitions

Functors are processed much in the same way as static structure bindings.

An additional function is generated, with signature `void (%int, i1, i8*)`. This function is tasked with dynamically creating frames resulting from functor applications, as detailed in Section 4.2.4.

- The first integer represents the location of a frame. This frame must correspond to the argument structure.
- Variable i1 specifies whether the value of the first integer is an address of a frame in insecure memory, or whether it must interpreted as an index in the *f-list*.
- The rest of the arguments specify the trimming map.

$[\![FunId(ArgStr : ArgSig) : SigId = \overline{d}]\!] \rightarrow$

```
define void @FunId(%int %frame, i1 %ext, i8* %vargs){
   call void @initialize()
   ; The initialize function will initialize the f-list, if it isn't
    initialized already.
   br i1 %ext label %External, label %Secure
```

```
  External:
   ; The frame and meta-frame can be built.
   ; Read the trimming list
   ; No checking is necessary.
   ; Add frame to the f-list.
   ret void
  Secure:
   ; Get memory location of frame through f-list
   ; Use meta information associated with the argument frame to
   typecheck functor application
   ; build the frame. The meta-frame can be built statically, because
    it uses only information provided by the signature.
}
```

The body of definitions is translated just like static structure bindings, with a few notable exceptions:

- Stubs and internal functions that represent values x defined by a functor *FunId* take an extra argument: A frame location loc_{frame}. The frame f in this location represents the result of an application of functor *FunId*.

- The stubs read the frame f at location loc_{frame} and can use this to see what ints represent the types that were created by the functor application. This is important to type the other arguments.

  Indeed, every application of a functor to a structure creates new types. A stub statically knows that the type of an argument is supposed to be of the nth type that the structure defines. It does not know which integer is associated with this type, because this changes from application to application.

  However, from the frame that represents a functor application, it can perform a lookup, using the knowledge that it is the nth type the structure defines, gives back the associated integer value.

- The internal functions use the frame to look up the frame representing the argument structure. They need this to call values from argument structure.

  Frames contain a boolean with type `i1`, called the *security bit*. This signals whether the frame corresponds to an insecure structure or a secure structure. If the frame is an insecure structure, additional measures must be taken:

  - Mask all arguments.
  - Clear the registers and switch the stack pointer.
  - Push the return address to the return address stack.
  - Change the return address of the call to the returnback entry point.[2]
  - When execution returns, the return value is interpreted as a masked index.

[2] As the return address is not manipulatable using LLVM, this security measure must be implemented during the compilation from LLVM IR to assembly.

Functor Applications

Functor applications result only in code creating the LLVM frame representation being built by the compiler. Just as for static structure bindings, this frame creating code will be outputted inside the `@initialize` function.

λ-expressions

λ-expressions will be compiled to their own private function, with a compiler generated unique name *ClosureN*. This private function will take a variable amount of arguments.

$[\![\lambda\ \overline{x}.e : (\overline{\tau_1} \to \tau_2)]\!] \to$
`define private %int @`*ClosureN*`(i1 %sec, {%int, [0 x %int]}* %env, i8* %vargs){`
    $[\![e]\!]$
`}`

- The first of these arguments is a single bit, which indicates whether the function was called by the generic closure evaluation point or by other code inside the secure context. As this value is always set within the secure context, its value can be trusted.

  Depending on this value, the next arguments are interpreted as pointers cast to int, or as indices in the masking map. It will also determine whether the return value is a pointer cast to int, or is being masked.

- The other arguments correspond to the arguments $\overline{x}$ of the λ-expression itself and the pointer to the environment.

- The compilation of e will be adjusted so that it references variables that are neither the parameters, nor defined within expression e itself using the their offset in the environment.

The lambda expression will also lead to the creation of an LLVM object of type `%closure`. It is initialized with

- a reference to a list of pointers cast to int, representing the environment;
- a pointer to the function the λ-expression was translated to;
- and a single bit, called the security bit, to indicate whether the closure was secure or insecure.

Applying closures

When application is performed in secure code on an identifier that represents a closure, the closure value is loaded. The secure code checks whether the closure value represents a secure or an insecure closure. This is done using the security bit provided in the `%closure` type.

If the closure is secure, the secure context can simply convert the integer in the closure value to a function pointer and call the function. As arguments, it provides a value 1 to signal that the arguments are unmasked, followed by the arguments and the pointer to the environment.

If the closure is insecure, the secure context masks all arguments it wants to pass. It then must call the closure evaluation point provided by the *insecure context* with the masked arguments and the int representation of the closure value. In this call, the return address must be changed to the returnback entry point, and the real return address must be pushed on the return address stack. When execution returns, the return value must be interpreted as a masking index.

Returnback Entry Point

The returnback entry point, as specified by Agten et al. [ASJP12] is used to return from callbacks. Whenever a secure function calls a function in insecure memory, either directly or through the insecure closure evaluation point, the return address is changed to the returnback entry point and the real return address is pushed on the return address stack. When the returnback entry point is called, it will pop the first value off the return pointer stack, and jump to it.

The return address register however can normally not be manipulated from within LLVM. This is a consequence of LLVM's target independent nature. This leaves two options to implement this security measure, both out of the scope of this work:

- Add the security measure when the LLVM backend compiles the LLVM IRsource to assembly.
- Use the experimental `llvm.read_register` and `llvm.writeregister` intrinsics to perform platform dependent modifications.

Generic Closure Evaluation Entry Point

This is a generic entry point into the *SPM* provided to evaluate closure values created within the secure code and passed to the insecure context. It receives an int, which is the masked index of the closure value, and a pointer to a variable number of arguments.

```
define %int @GenericClosureEvaluation(%int %closure.mask, i8* %vargs){
    %closure.unmask = call %int @unmask(%int %closure.mask)
    %type = call %int @unmasktype(%int %closure.mask) ; Check that it
    really is a closure
    switch %int %type, label %Error [%int 5, label %Continue1]

    Continue1:
    %closure.valptr = inttoptr %int %closure.unmask to %closure*
    %closure = load %closure* %closure.valptr
    %closure.ptr = extractvalue %closure %closure, 0
    %closure.env = extractvalue %closure %closure, 1
    %closure.type = extractvalue %closure %closure, 2;Check that the
    closure is a secure closure.
    switch i1 %closure.type, label %Error [i1 1, label %Continue2]

    Continue2:
    %closure.fn = inttoptr %int %closure.ptr to %int (i1, {%int, [0 x
    %int]}*, i8*)*
```

```
    %ret = call %int %closure.fn(i1 0, {%int, [0 x %int]}* %closure.
    env, i8* %vargs)

    ret %int %ret

    Error:
    call void @exit(i32 -1)
    unreachable
}
```

Structure Value Entry Point

The addition of functors required that values were called by specifying the frame that corresponds to the structure that defines them, and the offset of the value within the frame. This Structure Value entry point had the following arguments:

- an index in the f-list;
- an offset for the value to be called;
- and any number of arguments meant for the value.

```
@.flist = private global {%int, [0 x %frame*]}* null

define %int @StructureEntryPoint(%int %findex, %int %index, i8* %vargs
    ){
    %flist.ptr = load {%int, [0 x %frame*]}** @.flist
    %flist = load {%int, [0 x %frame*]}* %flist.ptr
    %elems = extractvalue {%int, [0 x %frame*]} %flist, 0

    %check = icmp ult %int %findex, %elems
    br i1 %check, label %Continue, label %Error

    Continue:
    %frame.ptr2 = getelementptr {%int, [0 x %frame*]}* %flist.ptr, i32
     0, i32 1, %int %index
    %frame.ptr = load %frame** %frame.ptr2
    %frame = load %frame* %frame.ptr
    %valuelist.ptr = extractvalue %frame %frame, 4
    %valuelist = load {%int, [0 x %int (%frame*, i8*)*]}* %valuelist.
    ptr
    %list.ptr = getelementptr {%int, [0 x %int (%frame*, i8*)*]}* %
    valuelist.ptr, i32 0, i32 1
    %elems2 = extractvalue {%int, [0 x %int (%frame*, i8*)*]} %
    valuelist, 0
    %check2 = icmp ult %int %index, %elems2
    br i1 %check2, label %Continue2, label %Error
```

```
    Continue2:
    %val.ptr = getelementptr [0 x %int (%frame*, i8*)*]* %list.ptr,
    i32 0, %int %index
    %val = load %int (%frame*, i8*)** %val.ptr

    %ret = call %int %val(%frame* %frame.ptr,i8* %vargs)

    ret %int %ret

    Error:
    call void @exit(i32 -1)
    unreachable
}
```

5.2.2 Conclusion

The addition of closures and functors to MiniML demands some significant changes on the inner workings of a secure compilation scheme.

The addition of closures generates changes that are mostly *orthogonal* to the simpler compiler sketched in Section 3.3.1. An additional base type is created to represent a closure. The compiler can determine statically whether a function application in MiniML happens on a closure value or on a known function, but must perform a runtime check to know whether the closure is insecure or secure. How the closure application is then performed ofcourse depends on this runtime check.

Furthermore, the secure compilation of closures makes more demands about the calling convention between secure code and insecure context.

- The insecure context must provide a representation of an insecure closure as a single integer. It is up to the insecure context to choose whether this integer is simply a memory address, or an index in a map, or any other argument passing scheme.
- The insecure context must provide a way of evaluating any closure using a single function. This function should show the same functional behavior as the *generic closure evaluation entry point* to the *SPM*, described on Page 70.

The addition of functors makes more fundamental changes in the compiler of Section 3.3.1. Because changing the binding of a structure identifier from a static definition to a functor application preserves source-level contextual equivalence (see Section 4.2.5 on Page 58), the stubs that structure values generate are no longer entry points to the secure code.

Instead, a *structure value entry point* is created that can call any value using only a (masked) reference to a *frame*, and the offset of the value in the value list within that frame.

Chapter

6 Proving Full Abstraction

Some formal techniques of proving the correctness of a compilation scheme offering full abstraction exist and the idea of one of these proof techniques, based on trace semantics, is sketched in Section 6.1.

6.1 Formal Proof Techniques

Recalling from Section 1.1, *full abstraction* is a compiler property. It states that *contextual equivalence* for source-level objects is preserved by and reflected from their target-level translations.

$$O_1 \simeq O_2 \iff O_1^{\downarrow} \simeq O_2^{\downarrow}$$

Proving full abstraction of a compilation scheme, i.e. proving that it is secure, requires a proof for soundness and completeness.

Soundness Soundness expresses that the compilation of two source-level objects does not 'introduce' contextual equivalence in the target language. Instead, for the target-level objects to be contextually equivalent, the source-level objects have to contextually equivalent already.

$$O_1^{\downarrow} \simeq O_2^{\downarrow} \implies O_1 \simeq O_2$$

Soundness corresponds closely to the informal notion of compiler *correctness*. Indeed, formulating the logically equivalent contrapositive of the soundness property gives:

$$O_1 \not\simeq O_2 \implies O_1^{\downarrow} \not\simeq O_2^{\downarrow}$$

This expresses that two contextually *un*equivalent source-level objects result in contextually *un*equivalent translations. If a compilation scheme would *not* be sound, there would exist two contextually *un*equivalent source-level objects, whose translations would be contextually equivalent.

Clearly, such a compiler does not function 'correctly', as there is a context in which the source-level objects behave differently, but the translations do not. One of these translations does not accurately behave like the source-object it is derived from.

Completeness Completeness says that all contextually equivalent source-level objects are translated to contextually equivalent target-level objects. It expresses that the contextual equivalence of source-level objects, which provides certain security guarantees, are preserved when compiling.

$$O_1 \simeq O_2 \implies O_1^{\downarrow} \simeq O_2^{\downarrow}$$

As most compilers are expected to be 'correct' or sound, the most important part of the full abstraction proof is the proof of completeness. Proving completeness can be done using *trace semantics* and looking at the contrapositive of completeness.

$$O_1^{\downarrow} \not\simeq O_2^{\downarrow} \implies O_1 \not\simeq O_2$$

The following section will detail how trace semantics can be used to prove completeness of a compilation scheme.

6.1.1 Trace Semantics

Trace semantics [JR05, PC14] for the low-level language describe the behavior of a program $P^{\downarrow}$ within a context $O_C^{\downarrow}$ as all interactions that happen between the context $O_C^{\downarrow}$ and the program $P^{\downarrow}$. Only the interaction between context $O_C^{\downarrow}$ and program $P^{\downarrow}$ is described: this means that trace semantics do *not* capture any *internal* operation. The interactions described correspond with any exchange of data or information between context $O_C^{\downarrow}$ and program $P^{\downarrow}$.

Trace semantics produce a trace for a program $P^{\downarrow}$ executing within a context $O_C^{\downarrow}$. A trace is a sequence of *labels* that correspond with the executed instructions. In order to restrict the description to the *interaction* between program $P^{\downarrow}$ and context $O_C^{\downarrow}$, not all instructions are given a label. Instead, only the `call` and `ret` instruction are labeled. Trace semantics track whether the instruction was executed by program $P^{\downarrow}$ or by context $O_C^{\downarrow}$. A possible syntax [PC14] for traces could be the one given by Fig. 6.1.

$$\text{Trace } T \ ::= \ L \mid T \cdot L \qquad L \ ::= \ a \mid \tau \qquad a \ ::= \ g? \mid g! \qquad g \ := \ \texttt{call}\ p(\overline{v}) \mid \texttt{ret}\ v$$

Figure 6.1: Syntax for trace semantics.

In this syntax, labels are defined to be actions a or τ, where an action a is observable, and τ is not. It uses ! and ? to track whether the action was performed in program $P^{\downarrow}$ or context $O_C^{\downarrow}$, respectively. In a `call`, p represents the address that was called and $\overline{v}$ the values passed in registers. In a `ret`, v represents the value in the return register.

Full abstraction proofs based on trace semantics, for example the ones given by Patrignani et al. [PCP13, PC14] or Agten et al. [ASJP12], first need to show that the operational semantics of the target language are fully abstract to the proposed low-level trace semantics.

This is only the case if the proposed trace semantics in fact do capture *all* exchange of information and interactions between context $O_C^{\downarrow}$ and program $P^{\downarrow}$. For the trace semantics with syntax given by Fig. 6.1, this is only true if the operational semantics prohibit the exchange of information between context $O_C^{\downarrow}$ and program $P^{\downarrow}$ through memory or registers other than the return value register.

If the operational semantics do *not* prohibit the exchange of information through memory or registers other than the return register, then the syntax of labels must be changed to a syntax that captures these channels of communication, as stated by Curien [Cur07] and worked out by Patrignani et al. [PC14].

If the trace semantics are fully abstract w.r.t the operational semantics, then the proof of full abstraction of the compilation scheme becomes easier. As traces then fully capture all the interaction of context object $O_c^{\downarrow}$ with a target-level object $O_1^{\downarrow}$, two target-level objects $O_1^{\downarrow}$ and $O_2^{\downarrow}$ are indistinguishable with respect to a context if and only if their traces T_1 and T_2 are the same. It follows that the existence of a context $O_C^{\downarrow}$ that can distinguish between $O_1^{\downarrow}$ and $O_2^{\downarrow}$, implies that they produce different traces T_1 and T_2 for this context O_C.

As a consequence, the contrapositive statement of completeness can be rewritten formally as:

$$\text{Traces}_L(O_1^{\downarrow}) \neq \text{Traces}_L(O_2^{\downarrow}) \implies O_1 \not\approx O_2$$

Proving this statement is simpler than proving the unmodified contrapositive. A proof for this statement would for example consist of an algorithm that can create a high-level context O_C that is able to distinguish O_1 from O_2 using their differing low-level traces T_1 and T_2. This context O_C, called the witness [PCP13], shows that O_1 and O_2 aren't contextually equivalent on the high-level.

If an algorithm exists that for any such pair of traces can construe a high-level context O_C, then the existence of this algorithm proves full abstraction of the compilation scheme.

6.2 Conclusion

A proof of full abstraction of the compilation scheme can be achieved by proving a low-level trace semantics exists that is fully abstract with the operational semantics of the low-level language. The trace semantics must capture *all exchange of information* or *interactions* between the low-level program and its context. Curien [Cur07] stated that one can make sure the labels of trace semantics capture every interaction between program and context by either:

- making the labels more expressive, so that they can capture the additional channels of communication that can be used, besides the registers at calls and the return register at returns;
- modifying the operational semantics, so that they restrict the communication between context and program to those channels captured by the labels.

In this work, as well as that of Agten et al. [ASJP12] and Patrignani et al. [PCP13], the compiler modifies the operational semantics so that it indeed limits the exchange of information in this way.

Chapter

7 Conclusion

First, this chapter discusses related work in Section 7.1. It continues by discussing possible future work in section Section 7.2. A conclusion to the work is formulated as well in Section 7.3

7.1 Related Work

Extensive work trying to preserve the security of a source languages when compiling exists. The idea of using full abstraction to formalize secure compilation is introduced by Abadi [Aba99].

Different techniques to preserve security even in the low-level computing model were developed, for example the use of *Adress Space Layout Randomization* or ASLR to *prevent* reliable memory manipulation by an attacker. This technique randomly changes the arrangement of important program portions. For example, the location of the stack, heap and base of the executable might change location within a process's address space. This reduces the reliability of an attacker's view of the process's address space, making it harder if not impossible for attackers to reliably jump to certain exploitable parts of the code, or to reliably overwrite a certain value in memory.

The idea of ASLR catched on, and ASLR saw implementations in common operating systems such as Windows Vista, OS X Mountain Lion and some Linux distributions. The idea also raised scientific study, for example by Abadi and Plotkin [AP12] or Jagadeesan, et al. [JPRR11] and criticism [SPP+04, SYP+09].
Additionally, one can make use of stack canaries to try to *detect manipulations* by an attacker. This uses a random value, called a *stack canary*. This random value is placed in the memory, before any critical information, for example the *return address*. The return address is a critical piece of information that helps to organize control flow when calling and returning from functions. Even if an attacker can somehow overwrite the return address in memory, for example using a buffer overflow[EYP10], it is likely that this will change the value of the stack canary as well. Stack canaries are a detection technique, as the application can check the value of the stack canary against the known random value. If this value changed, then the attacker's manipulation is detected, and execution can be stopped.

Other techniques work by providing *isolation* of software components, and introducing security guarantees to memory access using access control. This way these isolated

software components cannot be compromised by attacking other software components that it interacts with. In other words, isolation of software components compartmentalizes these software components, in a way that compromising is only possible on an individual compartment basis. This is the goal of secure compilation.

An example of such research can be found in Agten et al. [ASJP12], who already provided a secure compilation scheme for an object based language, when access to memory is restricted based on the value of the program counter. This technique is called *Program Counter Based Access Control*, or *PCBAC* [iDRG]. Later work by Patrignani et al. [PCP13] introduced additional object oriented concepts to the fully abstract compilation scheme.

The restricted access of memory can be implemented in hardware [NAD+13, MAB+13] or using software [SP12, ASAP13]. A recent innovation in restricting access on a hardware level is Intel®Software Guard Extensions, or *SGX* [MAB+13].

The choice between using software or hardware to provide restricted access to the memory is very important. For example, this choice affects the size of the trusted computing base or *TCB*. The TCB of an application contains all components, whether they are hardware or software, on which the application can only place its trust to ensure correct execution. Even with fully abstract compilation, security issues in the TCB could lead to low-level attacks. Consequently, a good TCB is a small, and verifiable TCB.

7.2 Future Work

The MiniML language as described in Chapter 5 still does not provide the full feature set of ML. This section proposes some valuable extensions that could be made to the language.

- Currently, the MiniML language allows only a very restricted set of types to be communicated between the secure and insecure code. Arrays and pairs can be used within a structure or to implement an opaque type, but they cannot be the argument or return value of a publicly available function or the type of a public field. This is not a very limiting restriction: a structure can be created to implementing an abstract data type (ADT) that behaves like a list.

  However, it could still be a valuable expansion to the language to once again allow lists and pairs as a basic and first class type in the Module language. As mentioned in Section 4.1.2, the call-by-value [MTM97] semantics and declarative style of the ML source language do make this addition more difficult.

  For example, when passing a list from the insecure context to a function in the secure code, in source-level semantics this list is passed as a value, and its content is immutable.

  As the insecure memory is readable by secure code, the target language translation of the function can read the list directly from the memory, without passing execution to the insecure code. As the attack described in Section 4.1.2 shows, this capability can introduce security risks if the inherently mutable memory is changed during a callback to insecure code.

- The full-fledged ML language is aware of mutable memory locations, using the concept of a reference type `ref` τ. Future work could add this type to MiniML.
- MiniML functors are monadic, they only take a single argument structure. There are several ways of introducing polyadic functors into MiniML. One would work with the current version of MiniML: A functor taking multiple arguments could be decomposed in several functors that take a single argument, and output a structure for the next functor to be applied to. A drawback to this is that the current MiniML implementation would make each of the intermediate structures available.

  Possibly nicer ways of implementing this functionality are:

  - Implementing polyadic functors as real functors whose stubs take more than one frame argument. Additional difficulties would represent themselves in the way structures are represented by frames.
  - The addition of substructures to the module language.
- As LLVM IR is not architecture aware, not all necessary security precautions can be described in LLVM IR. Specifically, methods of manipulating the stack pointer or the return address are either experimental or nonexistent.

  These security precautions are mentioned here, and should be added when the LLVM IR is eventually compiled to machine code for a *Protected Module Architecture* [iDRG] by the LLVM backend.

7.3 Conclusion

The goal of this thesis was to describe a secure compilation scheme for a language that implements ML-style modules.

- Chapter 2 and Chapter 3 of this thesis describe and formalize a basic version of MiniML and its secure compilation scheme. Many of the concepts introduced in literature regarding the secure compilation other, object oriented languages can be reused with only small modifications.
- Chapter 4 expanded the MiniML language, with Chapter 5 providing a formalization of the additions. While ML is still a larger and more capable language than the MiniML language of Chapter 5, the final version of the language does include functors, one of the most important features of ML's powerful module language, as well as closures.

Section 5.2 formalized a compilation scheme for these additions, bringing secure compilation to an ML-style module language, thereby achieving the original goal of this thesis.

Appendix

A LLVM Code

A.1 @mask, @unmask & @unmasktype

This section contains the LLVM IR code for the `@mask`, `@unmask` and `@unmasktype` functions. These functions handle the masking process. They take an integer representing a pointer and return an integer representing the index of this pointer in the mask list, or vice versa.

Listing A.1: LLVM IR for the mask, unmask and unmasktype function.

```
1  %int = type i64
2
3  declare i8* @malloc(%int)
4  declare void @free(i8*)
5  declare void @exit(i32)
6
7  ;This is the type of elements in the masking list.
8  ;It represents the masking list as a linked list.
9  %masktype = type {%int, %masktype*, %int}
10
11 ;This is a pointer to the initial linked list element.
12 @vtable = private global %masktype* null
13
14 ;The mask() function.
15 ;Because LLVM registers are SSA, it uses a tailrecursive helper
       function.
16 define %int @mask(%int %val, %int %type){
17         %ret = tail call %int @mask_rec(%int %val, %int %type, %
       masktype** @vtable, %int 0)
18         ret %int %ret
19 }
20
21 define private %int @mask_rec(%int %val, %int %type, %masktype** %cptr
       , %int %index){
22         ;Load the current pointer to %masktype* & check if its null.
       If it is, add a new record to the linked list.
```

```
23          %current = load %masktype** %cptr
24          %check = icmp eq %masktype* %current, null
25          switch i1 %check, label %Valcheck [i1 1, label %Add]
26
27          ;If the current pointer is null, we're at the end of the
    linked list
28          ;Add a record to the linked list by allocating memory for an
    element, and storing the pointer to it in the current pointer
29          ;Store value and initialize pointer of the element to null.
30          Add:
31                  %ptr1 = call i8* @malloc(%int 24)
32                  %ptr = bitcast i8* %ptr1 to %masktype*
33                  store %masktype* %ptr, %masktype** %cptr
34                  %locval = getelementptr inbounds %masktype* %ptr, i32
    0, i32 0
35                  store %int %val, %int* %locval
36                  %locptr = getelementptr inbounds %masktype* %ptr, i32
    0, i32 1
37                  store %masktype* null, %masktype** %locptr
38                  %loctype = getelementptr inbounds %masktype* %ptr, i32
     0, i32 2
39                  store %int %type, %int* %loctype
40                  ret %int %index
41
42          ;If the current pointer is allocated, check the value.
43          ;if eq -> jump to the return
44          ;else -> change accumulator, rec jump
45          Valcheck:
46                  %locvalcheck = getelementptr inbounds %masktype* %
    current, i32 0, i32 0
47                  %valcheck = load %int* %locvalcheck
48                  %check2 = icmp eq %int %valcheck, %val
49                  switch i1 %check2, label %Return [i1 0, label %Loop]
50
51          Loop:
52                  ;get pointer to next
53                  %nextptr = getelementptr inbounds %masktype* %current,
     i32 0, i32 1
54                  %newindex = add %int %index, 1
55                  %tailindex = tail call %int @mask_rec(%int %val, %int
    %type, %masktype** %nextptr, %int %newindex)
56                  ret %int %tailindex
57
58          Return:
59                  ret %int %index
60  }
61
62  define %int @unmask(%int %index){
```

```
63          %ret = call %int @unmask_rec(%int %index, %masktype** @vtable)
64          ret %int %ret
65  }
66
67  define private %int @unmask_rec(%int %cindex, %masktype** %cpointer){
68          %current = load %masktype** %cpointer ;Get pointer to current
        masktype.
69          %check = icmp eq %masktype* %current, null
70          br i1 %check, label %Error, label %ZeroTest
71
72          Error:
73                  call void @exit(i32 -1)
74                  unreachable
75
76          ZeroTest:
77                  %check2 = icmp eq %int %cindex, 0
78                  br i1 %check2, label %RetVal, label %Loop
79
80          RetVal:
81                  %locval = getelementptr inbounds %masktype* %current,
        i32 0, i32 0 ; Get pointer to val pointer
82                  %val = load %int* %locval
83                  ret %int %val
84
85          Loop:
86                  %nextptr = getelementptr inbounds %masktype* %current,
         i32 0, i32 1 ; Get pointer to next ll-element pointer.
87                  %newindex = sub %int %cindex, 1
88                  %ret = call %int @unmask_rec(%int %newindex, %masktype
        ** %nextptr)
89                  ret %int %ret
90  }
91
92  define %int @unmasktype(%int %index){
93          %ret = call %int @unmasktype_rec(%int %index, %masktype**
        @vtable)
94          ret %int %ret
95  }
96
97  define private %int @unmasktype_rec(%int %cindex, %masktype** %
        cpointer){
98          %current = load %masktype** %cpointer ;Get pointer to current
        masktype.
99          %check = icmp eq %masktype* %current, null
100         br i1 %check, label %Error, label %ZeroTest
101
102         Error:
103                 call void @exit(i32 -1)
```

```
104                 unreachable
105
106         ZeroTest:
107                 %check2 = icmp eq %int %cindex, 0
108                 br i1 %check2, label %RetVal, label %Loop
109
110         RetVal:
111                 %loctype = getelementptr inbounds %masktype* %current,
      i32 0, i32 2  ; Get pointer to type pointer
112                 %type = load %int* %loctype
113                 ret %int %type
114
115         Loop:
116                 %nextptr = getelementptr inbounds %masktype* %current,
      i32 0, i32 1  ; Get pointer to next ll-element pointer.
117                 %newindex = sub %int %cindex, 1
118                 %ret = call %int @unmasktype_rec(%int %newindex, %
    masktype** %nextptr)
119                 ret %int %ret
120 }
```

A.2 @tyvarcheck

This section contains the LLVM IR code for the `@tyvarcheck` function. It is tasked with checking whether two `%tyvar` values are the same type.

Listing A.2: LLVM IR for the typevarcheck function.

```
1  @.ImplTable = private constant ; Completed in by compiler
2
3  define private i1 @tyvarcheck(%tyvar* %v1, %tyvar* %v2){
4          Start:
5          %v1.1 = load %tyvar* %v1
6          %v2.1 = load %tyvar* %v2
7          %t1 = extractvalue %tyvar %v1.1, 1
8          %t2 = extractvalue %tyvar %v2.1, 1
9          %equal = icmp eq %int %t1, %t2
10         br i1 %equal, label %Continue, label %Error
11
12         Continue:
13                 %basetype = phi %int [%t1, %Start], [%t.1, %Continue]
14                 %t = getelementptr [6 x %int]* @.ImplTable, i32 0, %
    int %basetype
15                 %t.1 = load %int* %t
16                 switch %int %t.1, label %Continue [%int 0, label %Ok
17
                       %int 1, label %Ok
```

```
18
                    %int 3, label %PairRec
19
                    %int 4, label %ArrayRec]
20
21          PairRec:
22                  %pairptrv1 = extractvalue %tyvar %v1.1, 0
23                  %pairptrv1.1 = inttoptr %int %pairptrv1 to %pair*
24                  %tyvarlptrv1 = getelementptr inbounds %pair*  %
    pairptrv1.1, i32 0, i32 0
25                  %tyvarlptrv1.1 = load %tyvar** %tyvarlptrv1
26                  %tyvarrptrv1 = getelementptr inbounds %pair*  %
    pairptrv1.1, i32 0, i32 0
27                  %tyvarrptrv1.1 = load %tyvar** %tyvarrptrv1
28                  %pairptrv2 = extractvalue %tyvar %v2.1, 0
29                  %pairptrv2.1 = inttoptr %int %pairptrv2 to %pair*
30                  %tyvarlptrv2 = getelementptr inbounds %pair*  %
    pairptrv2.1, i32 0, i32 0
31                  %tyvarlptrv2.1 = load %tyvar** %tyvarlptrv2
32                  %tyvarrptrv2 = getelementptr inbounds %pair*  %
    pairptrv2.1, i32 0, i32 0
33                  %tyvarrptrv2.1 = load %tyvar** %tyvarrptrv2
34                  call i1 @tyvarcheck(%tyvar* %tyvarlptrv1.1, %tyvar*  %
    tyvarlptrv2.1)
35                  %pairret = call i1 @tyvarcheck(%tyvar* %tyvarrptrv1.1,
     %tyvar*  %tyvarrptrv2.1)
36                  ret i1 %pairret
37
38          ArrayRec:
39                  %arrayptrv1 = extractvalue %tyvar %v1.1, 0
40                  %arrayptrv1.1 = inttoptr %int %pairptrv1 to %array*
41                  %v1length = getelementptr inbounds %array* %
    arrayptrv1.1, i32 0, i32 0
42                  %v1length.1 = load %int* %v1length
43                  %zerolv1 = icmp eq %int 0, %v1length.1
44                  %arrayptrv2 = extractvalue %tyvar %v2.1, 0
45                  %arrayptrv2.1 = inttoptr %int %pairptrv2 to %array*
46                  %v2length = getelementptr inbounds %array* %
    arrayptrv2.1, i32 0, i32 0
47                  %v2length.1 = load %int* %v2length
48                  %zerolv2 = icmp eq %int 0, %v2length.1
49                  br i1 %zerolv1, label %Ok, label %ArrayCont1
50
51          ArrayCont1:
52                  br i1 %zerolv2, label %Ok, label %ArrayCont2
53
54          ArrayCont2:
55                  %v1el1 = getelementptr inbounds %array* %arrayptrv1.1,
```

```
     i32 0, i32 1
56                 %v1el1.1 = getelementptr [0 x %tyvar*]* %v1el1, i32 0,
     i32 0
57                 %v1el1.2 = load %tyvar** %v1el1.1
58                 %v2el1 = getelementptr inbounds %array* %arrayptrv2.1,
     i32 0, i32 1
59                 %v2el1.1 = getelementptr [0 x %tyvar*]* %v2el1, i32 0,
     i32 0
60                 %v2el1.2 = load %tyvar** %v1el1.1
61                 %arrayret = call i1 @tyvarcheck(%tyvar* %v1el1.2, %
    tyvar* %v2el1.2)
62                 ret i1 %arrayret
63
64         Ok:
65                 ret i1 1
66
67         Error:
68                 call void @exit(i32 -1)
69                 unreachable
70 }
```

A.3 Polymorphic Example

In Listing A.4, the LLVM IRtranslation of Listing A.3 is shown, to illustrate how polymorphic functions are handled.

Listing A.3: The polymorphic Pair structure.

```
1  signature PAIRSIG =
2      sig
3          type t a
4          val createPair : a -> a -> t a
5          val getLeft : t a -> a
6      end
7
8  structure Pair :> PAIRSIG =
9      struct
10         type t a = (a,a)
11         fun createPair left right = (left,right)
12         fun getLeft pair = pair.#1
13     end
```

Listing A.4: Translation of the Pair structure.

```
1  %int = type i64 ; 1
2  %tyvar = type {%int, %int} ; 2
3  %pair = type {%tyvar*, %tyvar*} ; 3
4  %array = type {%int, [0 x %tyvar*]} ; 4
```

```
5  %Pair.t = type {%pair} ; 5
6
7  declare i8* @malloc(%int)
8  declare void @free(i8*)
9  declare void @exit(i32)
10
11 define private %Pair.t* @Pair.createPair_internal(%tyvar* %p1, %tyvar*
       %p2){
12         %ptr = call i8* @malloc(%int 16)
13         %ptr1 = bitcast i8* %ptr to %pair*
14         %fst = getelementptr %pair* %ptr1, i32 0, i32 0 ;%tyvar**
15         store %tyvar* %p1, %tyvar** %fst
16         %snd = getelementptr %pair* %ptr1, i32 0, i32 1 ;%tyvar**
17         store %tyvar* %p2, %tyvar** %snd
18
19         %ptr2 = bitcast %pair* %ptr1 to %Pair.t*
20         ret %Pair.t* %ptr2
21 }
22
23 define %int @Pair.createPair(%int %left, i2 %left.mask, %int %right,
       i2 %right.mask){
24         ;We need extra parameters, because we don't know the type that
       was passed. Is it an externally defined type, is it an int, or is
       it a mask?
25         %leftTyvar.ptr.1 = call i8* @malloc(%int 16)
26         %leftTyvar.ptr = bitcast i8* %leftTyvar.ptr.1 to %tyvar*
27         switch i2 %left.mask, label %Unmask1 [i2 0, label %External1
28
                                 i2 1, label %Int1]
29         Unmask1:
30         %left.ptr = call %int @unmask(%int %left)
31         %left.type = call %int @unmasktype(%int %left)
32         %leftAsTyvar.unmask.1 = insertvalue %tyvar undef, %int %
       left.ptr, 0
33         %leftAsTyvar.unmask.2 = insertvalue %tyvar %
       leftAsTyvar.unmask.1, %int %left.type, 1
34         br label %Create1
35
36         External1:
37         %leftAsTyvar.ext.1 = insertvalue %tyvar undef, %int %left, 0
38         %leftAsTyvar.ext.2 = insertvalue %tyvar %leftAsTyvar.ext.1, %
       int 0, 1
39         br label %Create1
40
41         Int1:
42         %leftAsTyvar.int.1 = insertvalue %tyvar undef, %int %left, 0
43         %leftAsTyvar.int.2 = insertvalue %tyvar %leftAsTyvar.int.1, %
       int 1, 1
```

```
44          br label %Create1
45
46          Create1:
47          %leftAsTyvar = phi %tyvar [%leftAsTyvar.unmask.2, %Unmask1],
        [%leftAsTyvar.ext.2, %External1], [%leftAsTyvar.int.2,%Int1]
48          store %tyvar %leftAsTyvar, %tyvar* %leftTyvar.ptr
49
50          %rightTyvar.ptr.1 = call i8* @malloc(%int 16)
51          %rightTyvar.ptr = bitcast i8* %rightTyvar.ptr.1 to %tyvar*
52          switch i2 %right.mask, label %Unmask2 [i2 0, label %External2
53
                        i2 1, label %Int2]
54
55          Unmask2:
56          %right.ptr = call %int @unmask(%int %right)
57          %right.type = call %int @unmasktype(%int %right)
58          %rightAsTyvar.unmask.1 = insertvalue %tyvar undef, %int %
        right.ptr, 0
59          %rightAsTyvar.unmask.2 = insertvalue %tyvar %
        rightAsTyvar.unmask.1, %int %right.type, 1
60          br label %Create2
61
62          External2:
63          %rightAsTyvar.ext.1 = insertvalue %tyvar undef, %int %left, 0
64          %rightAsTyvar.ext.2 = insertvalue %tyvar %rightAsTyvar.ext.1,
        %int 0, 1
65          br label %Create2
66
67          Int2:
68          %rightAsTyvar.int.1 = insertvalue %tyvar undef, %int %right, 0
69          %rightAsTyvar.int.2 = insertvalue %tyvar %rightAsTyvar.int.1,
        %int 1, 1
70          br label %Create2
71
72          Create2:
73          %rightAsTyvar = phi %tyvar [%rightAsTyvar.unmask.2, %Unmask2],
         [%rightAsTyvar.ext.2, %External2], [%rightAsTyvar.int.2,%Int2]
74          store %tyvar %rightAsTyvar, %tyvar* %rightTyvar.ptr
75
76          call i1 @tyvarcheck(%tyvar* %leftTyvar.ptr, %tyvar* %
        rightTyvar.ptr) ; type equation
77
78          %pair = call %Pair.t* @Pair.createPair_internal(%tyvar* %
        leftTyvar.ptr, %tyvar* %rightTyvar.ptr)
79
80          %pair.ptr = ptrtoint %Pair.t* %pair to %int
81          %pair.mask = call %int @mask(%int %pair.ptr, %int 4)
82
```

```
83          ret %int %pair.mask
84  }
85
86  define private %tyvar* @Pair.getLeft_internal(%Pair.t* %pair.ptr){
87          %pair.ptr.1 = bitcast %Pair.t* %pair.ptr to %pair*
88          %left.ptr = getelementptr %pair* %pair.ptr.1, i32 0, i32 0 ;%
        tyvar**
89          %left = load %tyvar** %left.ptr
90          ret %tyvar* %left
91  }
92
93  define %int @Pair.getLeft(%int %pair){
94          %pair.unmask = call %int @unmask(%int %pair)
95          %pair.type = call %int @unmasktype(%int %pair)
96          %pair.check = icmp eq %int %pair.type, 5
97          br i1 %pair.check, label %Continue, label %Error
98
99          Continue:
100         %pair.ptr = inttoptr %int %pair.unmask to %Pair.t*
101         %return = call %tyvar* @Pair.getLeft_internal(%Pair.t* %
        pair.ptr)
102         %left = load %tyvar* %return
103         %left.addr = extractvalue %tyvar %left, 0
104         %left.type = extractvalue %tyvar %left, 1
105         %retval = call %int @mask(%int %left.addr, %int %left.type)
106         ret %int %retval
107
108         Error:
109         call void @exit(i32 -1)
110         unreachable
111 }
112
113 @.ImplTable = private constant [6 x %int] [%int 0, %int 1, %int 2, %
        int 3, %int 4, %int 3]
114
115 define private i1 @tyvarcheck(%tyvar* %v1, %tyvar* %v2){
116         Start:
117         %v1.1 = load %tyvar* %v1
118         %v2.1 = load %tyvar* %v2
119         %t1 = extractvalue %tyvar %v1.1, 1
120         %t2 = extractvalue %tyvar %v2.1, 1
121         %equal = icmp eq %int %t1, %t2
122         br i1 %equal, label %Continue, label %Error
123
124         Continue:
125                 %basetype = phi %int [%t1, %Start], [%t.1, %Continue]
126                 %t = getelementptr [6 x %int]* @.ImplTable, i32 0, %
        int %basetype
```

```
127                 %t.1 = load %int* %t
128                 switch %int %t.1, label %Continue [%int 0, label %Ok
129 
                        %int 1, label %Ok
130 
                        %int 3, label %PairRec
131 
                        %int 4, label %ArrayRec]
132 
133 
134         PairRec:
135                 %pairptrv1 = extractvalue %tyvar %v1.1, 0
136                 %pairptrv1.1 = inttoptr %int %pairptrv1 to %pair*
137                 %tyvarlptrv1 = getelementptr inbounds %pair*  %
    pairptrv1.1, i32 0, i32 0
138                 %tyvarlptrv1.1 = load %tyvar** %tyvarlptrv1
139                 %tyvarrptrv1 = getelementptr inbounds %pair*  %
    pairptrv1.1, i32 0, i32 0
140                 %tyvarrptrv1.1 = load %tyvar** %tyvarrptrv1
141                 %pairptrv2 = extractvalue %tyvar %v2.1, 0
142                 %pairptrv2.1 = inttoptr %int %pairptrv2 to %pair*
143                 %tyvarlptrv2 = getelementptr inbounds %pair*  %
    pairptrv2.1, i32 0, i32 0
144                 %tyvarlptrv2.1 = load %tyvar** %tyvarlptrv2
145                 %tyvarrptrv2 = getelementptr inbounds %pair*  %
    pairptrv2.1, i32 0, i32 0
146                 %tyvarrptrv2.1 = load %tyvar** %tyvarrptrv2
147                 call i1 @tyvarcheck(%tyvar* %tyvarlptrv1.1, %tyvar*  %
    tyvarlptrv2.1)
148                 %pairret = call i1 @tyvarcheck(%tyvar* %tyvarrptrv1.1,
     %tyvar*  %tyvarrptrv2.1)
149                 ret i1 %pairret
150 
151         ArrayRec:
152                 %arrayptrv1 = extractvalue %tyvar %v1.1, 0
153                 %arrayptrv1.1 = inttoptr %int %arrayptrv1 to %array*
154                 %v1length = getelementptr inbounds %array* %
    arrayptrv1.1, i32 0, i32 0
155                 %v1length.1 = load %int* %v1length
156                 %zerolv1 = icmp eq %int 0, %v1length.1
157                 %arrayptrv2 = extractvalue %tyvar %v2.1, 0
158                 %arrayptrv2.1 = inttoptr %int %arrayptrv2 to %array*
159                 %v2length = getelementptr inbounds %array* %
    arrayptrv2.1, i32 0, i32 0
160                 %v2length.1 = load %int* %v2length
161                 %zerolv2 = icmp eq %int 0, %v2length.1
162                 br i1 %zerolv1, label %Ok, label %ArrayCont1
163 
```

```
164         ArrayCont1:
165                 br i1 %zerolv2, label %Ok, label %ArrayCont2
166
167         ArrayCont2:
168                 %v1el1 = getelementptr inbounds %array* %arrayptrv1.1,
        i32 0, i32 1
169                 %v1el1.1 = getelementptr [0 x %tyvar*]* %v1el1, i32 0,
        i32 0
170                 %v1el1.2 = load %tyvar** %v1el1.1
171                 %v2el1 = getelementptr inbounds %array* %arrayptrv2.1,
        i32 0, i32 1
172                 %v2el1.1 = getelementptr [0 x %tyvar*]* %v2el1, i32 0,
        i32 0
173                 %v2el1.2 = load %tyvar** %v2el1.1
174                 %arrayret = call i1 @tyvarcheck(%tyvar* %v1el1.2, %
       tyvar* %v2el1.2)
175                 ret i1 %arrayret
176
177         Ok:
178                 ret i1 1
179
180         Error:
181                 call void @exit(i32 -1)
182                 unreachable
183 }
```

Appendix

B

Paper

Secure Compilation and the ML Language

Matthias van der Hallen

***Abstract*—The development of software is a daunting task, even more so when security becomes an issue. Meanwhile, in today's technology driven world, computer security is not so much a luxury as it is a necessity. Nearly all development happens in a high-level programming language, mainly because it allows for easier reasoning about the program being written. This simplification of the thought process is a direct consequence of the abstract computing model that high-level languages usually offer to the programmer.**

The computing models offered by real-world architectures differ on many accounts from the one high-level languages offer. It is during the compilation process that many bugs are introduced to seemingly correct software, because the abstract properties of the high-level language are lost. These bugs can even pop up in software whose workings were formally verified using tools such as verifast [1]. The goal of secure compilation is strengthening the compilation process such that any security guarantees derived from the abstract computing model are preserved when compiling.

This work aims to bring secure compilation to a language with an ML style module system, which uses signature matching and functors to provide modularization, encapsulation and information hiding.

I. Introduction

The amount of software in today's world is growing at rapid rates. As software programs take on a larger and larger role in our lives, and security sensitive applications run side by side with software of the garden-variety, it is important that these applications behave as expected.

Most computer software is written using a high-level language, for example Java or ML. These high-level languages offer a computing model that abstracts away many subtleties of real architectures. These subtleties of the computing model however are reintroduced when the software is compiled from those high-level languages to a low-level language. Some of the subtleties that are abstracted away are:

- The existence of registers and memory.
- The fact that the software code and the values or objects it creates must all be stored in this same memory space.
- How the flow of control, i.e. the next command to be executed, is managed.
- Objects created by software must provide an implementation, they no longer are *algebraic data types* described only by their functionality.

Many of the abstractions allow the programmer to make certain assumptions about the security of their software. The confidentiality of certain values and their integrity, for example. Another abstraction is the atomicity of a function. A software programmer never assumes that a function could be executed only partially, taking no real note of the possible harm that could be done if an attacker would be able to bypass the part of a function where permissions are checked.

The reintroduction of these subtleties in the low level can cause these security assumptions to become void. Many attacks exist that abuse the way compilation reintroduces these subtleties, breaking the security guarantees assumed by the programmer or any formal verification software. For example, Y. Erlinsson et al. [2] list a number of ways that low-level attacks might manipulate control flow or values whose integrity was guaranteed in on source level.

These bugs can introduce severe problems, such as breaking into security sensitive parts of an application by exploiting its bindings with less secure parts of the application. Hence, a strengthening of the compilation process is required. The goal of *secure compilation* is to provide compilation that preserves all high-level security guarantees in the low-level output of the compiler.

II. Problem Setting

Informally, the goal of secure compilation was formulated as compilation that preserves all high-level security guarantees in the low-level. Contextual equivalence allows us to formalize what exactly are the security guarantees offered by the high-level language.

Contextual equivalence [3] does this by introducing an equivalence relation $\simeq$ on programs or their components. Two objects $O1$ and $O2$ are contextually equivalent if no third object O_C, called the *context*, is able to distinguish between the two components when it is run together with one of the objects as a programming. The two objects are thus perfectly substitutable, as their outward behaviour is the same: they function as one and the same blackbox.

$$\forall O_C : O_C[O_1] \rightarrow^* c \iff O_C[O_2] \rightarrow^* c$$

where $O_C[.]$ is a program where a certain component is unspecified. $O_C[O_1]$ is the program that results from linking O_C with O_1, where O_1 is used as the unspecified component.

Contextual equivalence captures security guarantees such as the public/private access modifier or the atomicity of function executions. For example, if two components O_1 and O_2, containing functions Listing 1 and Listing 2 respectively, are contextually equivalent then the atomicity of function execution of `f` is guaranteed. Being able to break function atomicity would result in the normally unreachable **`return`** in Listing 2 becoming reachable, which means substituting O_1 by O_2 would be observable from some context object O_C.

Listing 1: Simple return

```
public void f(){
    return 0;

}
```

Listing 2: Unreachable code

```
public void f(){
    return 0;
    return 1; //Unreachable
}
```

Now that ability of contextual equivalence to express the security guarantees of a language is established, secure compilation can be formalized by the notion of *full abstraction* [4], or the preservation and reflection of contextual equivalence throughout the compilation process. If O_1 is a high-level component and $O_1^{\downarrow}$ is its low-level result from compilation, full abstraction can be stated formally as:

$$\forall O_1, O_2 : O_1 \simeq O_2 \iff O_1^{\downarrow} \simeq O_2^{\downarrow}$$

This definition formulates that for any two programs that are contextually equivalent for any high-level context, their compiled representation should be contextually equivalent for any low-level context as well. This means compilation must preserve contextual equivalence, i.e. all security guarantees provided in the high-level language.

It also says that contextual equivalence should be reflected. This property is called soundness, and is closely linked to what is expected of a 'correct' compiler. Indeed, if the results of compilation are contextually equivalent but the high-level objects are not, there exists a high-level context O_C that can distinguish between O_1 and O_2, but without a low-level translation.

III. Contributions

This works introduces a secure compilation scheme for a language with an ML style module language. The ML language is a functional language that provides modularization of programs through its *module* language [5].

A. Structures

One of the primary concepts that the module language offers is that of a *structure*. A structure is simply a set of type and value definitions. Structures allow a programmer to divide a large program in sets of smaller units containing closely related type and value definitions. These units dependencies and connections are well-defined and explicit. An example of a structure is shown in Listing 3.

Listing 3: An example structure showing the definition of a dictionary in ML.

```
structure Dictionary =
    struct
        type dictionary = (string * string) list
        val emptyDictionary = []
        fun insert d, x, y = (x,y)::d
    end
```

B. Signatures

Another concept of the module language is a *signature*. A signature is a set of type declarations and value declarations. It can provide type declarations without giving a specific implementation. An example of such a signature is shown in Listing 4.

Listing 4: An example signature showing the declaration of a dictionary in ML.

```
signature DICTIONARYSIGNATURE =
    sig
        type dictionary
        val emptyDictionary : dictionary
        val insert: dictionary -> string ->
            string -> dictionary
    end
```

A signature can describe the interface of a structure. Any **`struct`** expression has a so called *principal signature*. When the **`struct`** expression is bound to a name using the **`structure`** keyword, it can be ascribed with a signature, either opaquely or transparent. This causes the interface of the structure to be type checked, and then altered, based on the type of ascription. This technique of structural typing is called *signature matching* [6].

The interface of *view* of a structure *Str* ascribed with a signature *Sig* can be computed as follows:

- Values defined in the **`struct`** expression but not declared in the ascribed signature are never available for code outside the **`struct`** expression. They do *not* become part of the interface of structure *Str*.
- Only types declared in the ascribed signature *Sig* are part of the interface of *Str*. Whether or not their *definition* is propagated depends on whether ascription was opaque or transparent.

C. Functors

The last concept of the module language is a *functor*. Functors behave as a function mapping an argument structure *StrIn* to output structures *StrOut* and are used to create *parametric* dependencies of one structure on another. An example of such a functor is shown in Listing 5.

Listing 5: The Dictionary as a functor. The structures and signatures that this example depends on are shown in Listing 6 in Appendix A.

```
functor DictionaryFn (KeyStruct:EQUAL) :>
    DICTIONARY where type key = KeyStruct.t =
    struct
        type key = KeyStruct.t
        type 'a dictionary = (key * 'a) list
        val emptyDictionary = []
        fun insert d x y = (x,y)::d
        fun lookup |[] x = error
                   |(key,value):ds x = if(
                       KeyStruct.equal key x)
                                   then value
                                   else (lookup
                                       ds x)
    end

structure StringDict = DictionaryFn(StringEqual)
    ;
```

1) Functor Definition: Functor definitions specify an identifier for an argument structure (`KeyStruct`) and define an output structure that can use values defined by the argument structure.

To limit the set of structures allowed as an argument, a functor also specifies a signature (`EQUAL`) with which the argument structure should match. This signature ensures the functor that the output structure it defines can trust the values it uses from the argument structure are really defined by the argument structure.

As functors themselves define an output structure as well, this output structure can be ascribed with a signature, in example Listing 5 this is would be the signature `DICTIONARY`.

2) Functor Application: On top of being defined, functors are also applied. The application of a functor is shown in Listing 5 on line 13. A functor application binds the structure that results as output of a functor to a name. Of course the functor application must be supplied a concrete structure as an argument, whose interface matches the expected argument signature.

3) Secure Compilation: The contribution of this work is a secure compilation scheme for a language that provides an ML style module language. In order to be able to preserve the security guarantees of ML when compiling to a low-level language, it is assumed that the low-level architecture offers certain access control semantics for the computer memory. The same access control semantics as specified by Agten et al. [3] are chosen (Table I).

This access control semantic assumes that all memory is split into a protected and an unprotected section. Protected memory has a further subdivision into a code section and a data section. A number of memory locations inside the protected code section are designated to be entry points. These are the only memory locations to which instructions in unprotected memory can jump. Furthermore, instructions in unprotected memory can only read from or write to memory locations located in unprotected memory.

From \To	Protected			Unprotected
	Entry Point	Code	Data	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

TABLE I: Program counter based access control semantics as specified in Agten et al. [3].

Agten et al. already specify a secure compilation scheme from a simple high-level language to this low-level architecture. Patrignani et al. [7] expand on this work, adding several concepts from *object oriented programming*.

When a collection of structures, signatures and functors is compiled with the secure compilation scheme, it is assumed that execution will happen by loading the securely compiled code to the protected code memory and linking it with an insecure context that resides in unprotected memory. Memory allocations in the protected code will allocate memory inside the protected data section.

In order to securely compile a language with an ML style module system, it is important to preserve the basic security precautions that were described by Agten et al. [3]. These are summarized here shortly:

- A single entry point is created for every funcion. The access control semantics of the low-level architecture make sure that control flow can only switch from the insecure code to the secure code by passing through such an entry point. This protects the 'atomicity' of function execution.
- A shadow stack should be used for all stack operations done in the secure code. When control flow switches from the insecure context to the secure module or back, the stack pointer and the shadow stack pointer should be switched.
- When control flow passes from the secure code to the insecure code, all flags and all registers not used to pass the return value should be cleared, with the exception of callee saved registers.
- A reordering of function definitions does not break contextual equivalence. Therefore, the order in

which functions are defined in the low-level language should not provide a way to distinguish between the compilation results of contextually equivalent modules.

The compilation scheme has to be modified to allow for an ML style module language.

Masking
: The first modification consists of adding the technique of masking, as introduced by Patrignani et al. [7]. This means that values created in the secure code do not leave the secure memory, as this might provide details over their implementation, and neither do direct memory pointers. ML allows for opaque types, meaning two structures that provide the same type but with a different implementation are contextually equivalent. As a result these implementation must be hidden from the insecure context. Instead of passing the values directly or passing a pointer to their memory location, every value created in secure code but passed to the insecure context is represented within insecure context as its index in a *masking* list.

Typing
: As a consequence of ML allowing for opaque types, simply having the correct structural implementation does not mean a function can operate on a certain value. For this to be allowed, the value must also be of the correct opaque type, which means the masking must keep track of type information.

Frames
: Because functors are not hidden on source-level from the insecure code, the insecure code can create new structures by applying functors. Security-wise, the resulting structures are still expected to be secured in the same way as if they were written in secure code itself. For example, applying the dictionary from Listing 5 should not result in anyone being able to tell whether the functor implements the dictionary type using a list of pairs, as in the case of the example, or a pair of lists.
  As a result, it is not adequate to statically compile away functor applications. This means that the compiled secure code must provide a way to *dynamically* create new structures that correspond to functor applications.
  This is achieved by compiling functors values using generic code that takes an additional parameter: the argument structure that the functor was applied to. For this, structures need to have a runtime representation containing a list with pointers to all their value definitions.
  These runtime representations are called *frames*, and are collected in a list called the *f-list*. The insecure code can refer to frames using their index in the *f-list*

Trimming Map
: Code that represents the functor can now address values of the argument structure using the list of pointers inside the argument frame, and an offset. Because the argument structure can define more values than the expected signature declares, the offset of a value needed by the functor within the argument structure might differ from the offset of this same value in the argument signature. To solve this, a mapping between these offsets must be established as well, called the *trimming map*. This is unique for every functor application.

Functor Applications
: Functor applications introduce new structures. That these structures are the result of functor applications is not a distinguishable feature in the ML source language. When a program defines a structure statically, this program is still contextually equivalent to a program that defines this structure as the result of a functor application. Note that this introduces two new problems:
  - These structures can be used as an argument to a functor again. This means that a frame must be created for these structures as well. This has to happen dynamically, checking the argument structure to make sure that it matches with the expected signature.
  - When calling a value of a structure resulting from functor application, this call is not allowed to look different from calls that acces a value defined in a static structure defintion. This means that all value calls must include a frame as an argument.
    This frame can only be the frame that represents the structure containing the value itself. Frames that represent the output structure of functor application must contain a reference to the frame that represents the argument of the functor application, so that the generic code representing the functor still gets a reference to the argument frame.

    As a result, all frames now look as follows, with an empty trimming map and pointer if the frame represents a statically defined structure.

Ordering of Structure Bindings
: The order in which structures are defined provides no way to distinguish two ML programs. As a result, the order of frames in the f-list has to be alphabetical for any frames that correspond to structures defined within the secure code.
Frames that represent structures resulting from *functor applications* within *insecure context* are appended in the order of their bindings.

A Single Entry Point
: Whether a structure was defined using functor application or statically is not a distinguishing feature in ML. This has an effect on how structure values should be called.
If the functions that represent these values were direct entry points to the module, two structures that both result from functor applications would share the same entry points. The inverse is true as well: sharing entry points is a proof of being defined using functor application.
This can be used in the low-level to distinguish between two programs, where one defines a structure using functor application and the other writes out the structure definition statically.
As a result, only a single entry point into the module is created. All values now share the same entry point. Specifying which value is called can be done by providing an index in the *f-list*, uniquely defining a structure, and the offset for the value within the list of values stored in the frame.

Implementing these checks and security precautions results in a secure compilation scheme for a language implementing an ML style module system.

IV. Related Work

Lots of work trying to preserve the security of a source languages when compiling exists. The idea of using full abstraction to formalize secure compilation is introduced by Abadi [4].

Different techniques to preserve security were developed, for example using *Adress Space Layout Randomisation* or ASLR.The idea of ASLR catched on, and ASLR saw implementations in common operating systems such as Windows Vista, OS X Mountain Lion and some Linux distributions. The idea also raised scientific study, for example by Abadi and Plotkin [8] or Jagadeesan, et al. [9] and criticism [10], [11].

Other techniques work by introducing security guarantees to memory access. For example, Agten et al. [3] already provide a secure compilation scheme for an object based language, when access to memory is restricted based on the value of the program counter. This technique is called *Program Counter Based Access Control*, or *PCBAC* [12]. Later work by Patrignani et al. [7] introduced additional object oriented concepts to the fully abstract compilation scheme.

The restricted access of memory can be implemented in hardware [13], [14] or using software [15], [16]. This choice affects the size of the trusted computing base or *TCB*. Even with fully abstract compilation, security issues in the TCB could lead to low-level attacks. A recent innovation in restricting access on a hardware level is Intel®Software Guard Extensions, or *SGX* [14].

V. Conclusion

This work aims to bring a secure compilation scheme to languages that implement an ML style module system. This work shows that such a secure compilation scheme is possible on a low-level architecture providing certain access control semantics as specified by Agten et al. [3]. This same architecture was already shown to allow for secure compilation of several advanced OOP concepts by Patrignani et al. [7].

Appendix A
Additional Code

Listing 6: The auxilary signatures and structures for the functor example of Listing 5 on Page 3.

```
signature DICTIONARY =
    sig
        type key
        type 'a dictionary
        val emptyDictionary : 'a dictionary
        val insert : 'a dictionary -> key -> 'a
            -> 'a dictionary
        val lookup : 'a dictionary -> key -> 'a
    end

signature EQUAL =
    sig
        type t
        val equal : t -> t -> bool
    end

structure StringEqual: EQUAL =
    struct
        type t = string
        fun equal t1 t2 = case String.compare(t1,
            t2)
                              of EQUAL => true
                              | _ => false
    end
```

References

[1] B. Jacobs and F. Piessens, "The verifast program verifier," 2008.

[2] Ã. Erlingsson, Y. Younan, and F. Piessens, "Low-level software security by example," in *Handbook of Information and Communication Security* (P. Stavroulakis and M. Stamp, eds.), pp. 633–658, Springer Berlin Heidelberg, 2010.

[3] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," in *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, (Washington, DC, USA), pp. 171–185, IEEE Computer Society, 2012.

[4] M. Abadi, "Protection in programming-language translations," in *Secure Internet Programming* (J. Vitek and C. Jensen, eds.), vol. 1603 of *Lecture Notes in Computer Science*, pp. 19–34, Springer Berlin Heidelberg, 1999.

[5] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[6] B. C. Pierce, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[7] M. Patrignani, D. Clarke, and F. Piessens, "Secure compilation of object-oriented components to protected module architectures," in *Programming Languages and Systems* (C.-c. Shan, ed.), vol. 8301 of *Lecture Notes in Computer Science*, pp. 176–191, Springer International Publishing, 2013.

[8] M. Abadi and G. D. Plotkin, "On protection by layout randomization," *ACM Trans. Inf. Syst. Secur.*, vol. 15, pp. 8:1–8:29, July 2012.

[9] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely, "Local memory via layout randomization," in *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pp. 161–174, June 2011.

[10] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, (New York, NY, USA), pp. 298–307, ACM, 2004.

[11] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, (New York, NY, USA), pp. 1–8, ACM, 2009.

[12] iMinds Distrinet Research Group, "Program counter based access control." https://distrinet.cs.kuleuven.be/software/pcbac/. accessed: 2013-10-02.

[13] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *22nd USENIX Security Symposium,*, USENIX Association, Aug. 2013.

[14] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, (New York, NY, USA), pp. 10:1–10:1, ACM, 2013.

[15] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, October 2012.

[16] N. Avonds, R. Strackx, P. Agten, and F. Piessens, "Salus: Non-hierarchical memory access rights to enforce the principle of least privilege," in *Security and Privacy in Communication Networks (SecureComm'13)*, 2013.

Appendix

C

Poster

Secure Compilation of ML Modules

KATHOLIEKE UNIVERSITEIT
LEUVEN

FACULTEIT
INGENIEURSWETENSCHAPPEN

Master
Computer-
wetenschappen

Masterproef
Matthias van der Hallen

Promotor
Prof. dr. ir. F. Piessens

Academiejaar
2013-2014

Situering

- Secure Compilation:
  Veiligheidsgaranties behouden over compilatiestap heen
  => Low-level aanvaller kan niet meer dan op high-level
- Contextuele equivalentie $\simeq$:
  - Formaliseert veiligheidsgaranties
  - Garantie op “blackbox” werking
    Interne aanpassingen niet waarneembaar

Doelstellingen

- Secure Compilation voor talen met een ML stijl Module systeem
- Basiscomponenten Module systeem:
  - Structures,
  - Signatures,
  - Functors.

```
functor DictionaryFn (KeyStruct:EQUAL) :> DICTIONARY where type
    key = KeyStruct.t =
   struct
       type key = KeyStruct.t
       type 'a dictionary = (key * 'a) list
       val emptyDictionary = []
       fun insert d x y = (x,y)::d
       fun lookup [] x = error
                | (key,value):ds x = if(KeyStruct.equal key x)
                                     then value
                                     else (lookup ds x)
   end

structure StringDict = DictionaryFn(StringEqual);
```

- Extra features:
  - Transparante en opaque typen
  - Hogere orde functies
  - Parametrisch polymorphisme
- LLVM als target van compilatie

Toepassingen

- Op een architecture met memory access control (PCBAC).

	protected			*unprotected*
	entry	code	data	
protected	***r x***	***r x***	***r w***	***r w x***
unprotected	***x***	**-**	**-**	***r w x***

- Implementaties:
  - software: Fides, Salus
  - hardware: Sancus, Intel SGX

Resultaten

- **Introductie van MiniML:**
  een simpele taal met ML stijl modules.
- **Formalisatie van MiniML:**
  - syntax
  - type system
  - operationele semantiek
- **Formalisatie Compiler**
- **Functors:**
  - Runtime representatie van structures: frames
  - Generische code
  - Alle oproepen via één entry point.
  - Members aangeduid door combinatie van:
    - Frame identifier
    - Member offset

Bibliography

[Aba99] Martín Abadi. Protection in programming-language translations. In Jan Vitek and ChristianD. Jensen, editors, *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 19–34. Springer Berlin Heidelberg, 1999.

[AP03] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[AP12] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.*, 15(2):8:1–8:29, July 2012.

[ASAP13] Niels Avonds, Raoul Strackx, Pieter Agten, and Frank Piessens. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In *Security and Privacy in Communication Networks (SecureComm'13)*, 2013.

[ASJP12] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 171–185, Washington, DC, USA, 2012. IEEE Computer Society.

[Cur07] Pierre-Louis Curien. Definability and full abstraction. *Electronic Notes in Theoretical Computer Science*, 172(0):301 – 310, 2007. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. *SIGPLAN Not.*, 10(6):114–121, April 1975.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[Els99] Martin Elsman. Static interpretation of modules. *SIGPLAN Not.*, 34(9):208–219, September 1999.

[EYP10] Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In Peter Stavroulakis and Mark Stamp, editors, *Handbook of Information and Communication Security*, pages 633–658. Springer Berlin Heidelberg, 2010.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:pp. 29–60, 1969.

[iDRG] iMinds Distrinet Research Group. Program counter based access control. `https://distrinet.cs.kuleuven.be/software/pcbac/`. accessed: 2013-10-02.

[JP08] Bart Jacobs and Frank Piessens. The verifast program verifier, 2008.

[JPRR11] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 161–174, June 2011.

[JR05] Alan Jeffrey and Julian Rathke. Java jr: Fully abstract trace semantics for a core java language. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer Berlin Heidelberg, 2005.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag.

[LA05] Chris Lattner and Vikram Adve. Llvm language reference manual. `http://llvm.org/docs/LangRef.html`, 2005. Accessed: 2013-10-25.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002.

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[NAD+13] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium,*. USENIX Association, August 2013.

[PC14] Marco Patrignani and Dave Clarke. Fully abstract trace semantics for low-level isolation mechanisms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1562–1569, New York, NY, USA, 2014. ACM.

[PCP13] Marco Patrignani, Dave Clarke, and Frank Piessens. Secure compilation of object-oriented components to protected module architectures. In Chung-chieh Shan, editor, *Programming Languages and Systems*, volume 8301 of *Lecture Notes in Computer Science*, pages 176–191. Springer International Publishing, 2013.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[SP12] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, October 2012.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[SYP+09] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.

[YJP12] Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Comput. Surv.*, 44(3):17:1–17:28, June 2012.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440, New York, NY, USA, 2012. ACM.

Fiche masterproef

Student: Matthias van der Hallen

Titel: Secure Compilation of ML Modules

Nederlandse titel: Veilige Compilatie van ML-stijl Modulesystemen

UDC: 681.3

Korte inhoud:
Malware infects many new computers each day. Some estimates suggest that up to 30% of all computers are in fact infected. This malware is transmitted by exploiting bugs in computer software. In many cases, these bugs abused by the exploit originate from the disparity between the computing model presented by high-level source language and the effective module used by the low-level target language.
These bugs are not prevented by analysis of the source language, for example using formal software verification tools, because they are effectively 'introduced' by the process of compilation from source language to target language. Instead, they must be prevented by strengthening the compilation process in a way that reduces the power of low-level attackers to that of high-level attackers. Compilers that achieve this provide 'secure compilation'.
This work uses the notions of full abstraction and contextual equivalence to formalize the requirements for a secure compilation scheme, and shows how secure compilation can be achieved for MiniML, a subset of the ML language. As a prerequisite however, the secure compilation scheme assumes that the result of compilation runs on an architecture that provides program counter based access control, called a protected module platform.
The source language for this secure compilation scheme, MiniML, is not object oriented. Instead it uses a module system with the powerful notion of a functor to provide modularization of code. It targets the LLVM Intermediate Representation as a target language. A formalization of this MiniML source language and the LLVM IR target language is presented, enabling a formalization of the secure compilation scheme to be given as well.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Artificiële intelligentie

Promotor: Prof. dr. ir. F. Piessens

Assessoren: Dr. ir. W. Meert
Dr. D. Devriese

Begeleiders: M. Patrignani
R. Strackx