VICTORIA UNIVERSITY OF WELLINGTON
*Te Whare Wānanga o te Ūpoko o te Ika a Māui*


School of Engineering and Computer Science
*Te Kura Mātai Pūkaha, Pūrorohiko*

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

# Compiling Whiley for Embedded Systems

Juan van den Anker

Supervisor: Dr David Piece

Submitted in partial fulfilment of the requirements for
Master of Computer Science.

**Abstract**

Whiley is a statically typed language that supports both object-oriented as well as functional programming language paradigms. Like many other programming languages, Whiley also supports variables, primitive types (e.g. int, bool), methods, control-flow statements (e.g. if, while) and many common expressions (e.g. +, -, ). Where Whiley importantly differs from other programming languages, is Whiley's support for expressing specifications, Union types, Unbounded Arithmetic and Extended Static checking. Microcontrollers like for example an Arduino, are small electronically based processors with extremely limited resources compared to processors found in computers. This projects goal is to design an develop a While Compiler plugin that translates Whiley Applications into microcontroller compatible C-code. Although this plugin supports only a subset of the Whiley Language Specification, it provides support for enough language features to successfully build, compile and execute an example application on a microcontroller. This document provides a background of this project, provides the design and implementation of the compiler plugin and also provides an example application and a hardware circuit to test the result of the compiler on an actual microcontroller. Finally, this project provides a solution to integrate and re-use a large set of testcases into our development cycle.

# Acknowledgments

First of all, I would like to thank my supervisor, Dr David Pearce, for his expert guidance and mentorship in the writing of this thesis and during my literature review. Without the support of Dr Karsten Lundqvist and Dr Peter Andreae during some of the more difficult times in recent months, I certainly would not have been able to complete my studies - so appreciated, thank you! Last but not least, I am deeply indebted to my wife Karin for her encouragement and patience during this master's program - without her, I would undoubtedly not have started uni, and I certainly would not have completed it today.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Whiley is a language designed by Dr David Pierce in response to Prof. Sir Tony Hoare's challenge to create a verifying compiler that: "uses mathematical and logical reasoning to check the correctness of the programs that it compiles".

Developing large bug-free software applications is a challenging task. Although many modern programming languages provide some compiler support to prevent the programmer from introducing those errors in the first place, Whiley takes this a step further. In Whiley, a developer can add pre- and post-conditions to functions thar are automatically verified by the Whiley compiler. In most cases, this makes it possible to automatically check (verify) if the actual implementation of a function satisfies the provided predicates.

Whiley is a flexible language that both supports object-oriented programming paradigms as well as functional programming paradigms. Currently, applications developed in Whiley can either run in a web browser (as JavaScript) or locally on a computer using the Whiley Development Kit.

This project goal is to design and implement a new compiler plugin, called WhileyToEmbeddedC, to translate a Whiley Application into Embedded C in such a way so that it can run on microcontrollers.

Microcontrollers are comparable in terms of functionality to processors found in ordinary computers. However, microcontrollers are far less powerful in comparison to those computer processors. They are also used for different applications. Where computer processors are used to run operating systems, games and applications, microcontrollers are used to control external hardware like displays, switches, motors, pumps. A microcontroller can be seen as the heart of any electrical circuit project.

For this project, we created a compiler capable of compiling Whiley applications for the microcontroller architecture. Additionally, we have developed an example Whiley application and integrated circuit to successfully run it on a microcontroller. We also have tested the quality of our compiler plugin by integrating an existing set of test cases into our development cycle.

## 1.1 Contributions

The project offers the following contributions:

1. The design and implementation of a Whiley to Embedded C compiler plugin. Using the existing Whiley compiler infrastructure, this plugin compiles Whiley applications into Embedded C applications that can be run on a microcontroller.

2. The design and implementation of an example Whiley application and microcontroller based execution platform. Using the created plugin, together with a custom build hardware project, we have demonstrated the ability to run Whiley application on a microcontroller.

3. The integration of existing JUnit tests into the Embedded C development toolchain. Using the existing set of Unit Tests, we have demonstrated the ability to execute those and verify the workings of the compilers generated output.

## 1.2 Organisation

The rest of this report is structured as follows:

- Chapter 2 provides background information in relation to Whiley and Microcontrollers in general
- Chapter 3 describes the architecture of the existing Whiley compiler and how our new compiler fits into this architecture
- Chapter 4 describes the architecture of the Embedded C compiler and the translation of Whiley specific language features
- Chapter 5 describes an example Whiley application and provides results of the available Unit Testing framework
- Chapter 6 describes the project conclusion and possible future work
- Appendix A includes the example Whiley application
- Appendix B includes the Arduino library used by the example application
- Appendix C includes the Embedded C version of the compiled Whiley application
- Appendix D includes the terminal output of installing the application onto the microprocessor
- Appendix E includes the Makefile generated by the compiler
- Appendix F includes the literature review compiled before this project started

# Chapter 2

# Background

This chapter provides a functional overview of Whiley and Microcontrollers in general. Using an example Whiley application, we will describe some important features of the Whiley language. We will give a high-level overview of microcontrollers in general provide more details on the Arduino microcontroller and the Arduino platform.

## 2.1 Literature Review

Typically, this project report should describe related work in this section. However, we completed the Literature Review report before we started working on this part of the project. Therefore, this review is now part of Appendix F.

## 2.2 Whiley

Whiley is a statically typed language that supports both object-oriented as well as functional programming language paradigms. Like many other programming languages, Whiley also supports variables, primitive types (e.g. int, bool), methods, control-flow statements (e.g. if, while) and many common expressions (e.g. +, -, ). Like many other popular programming languages, Whiley also supports *Lambda Expressions*.

Where Whiley importantly differs from other programming languages, is Whiley's support for *expressing specifications*, *Union types*, *Unbounded Arithmetic* and *Extended Static checking*. Whiley's support for creating safer applications is where it stands out from other programming languages.

*Static Checking* is used to prevent errors often made in other programming errors (e.g. *Array Out Of Bounds* errors, *Divide By Zero* or *Null Dereference*).

Specifications in Whiley are used to annotate functions that describe the expected behaviour of a function. Using clauses like *ensures* and *require* added to a functions definition, the developer can inform the compiler what to expect when the function returns a certain value or what to expect for given arguments to the function.

More importantly, the compiler uses this additional information to verify that the implementation of a function satisfies the supplied conditions (predicates) using an automated theorem prover. This is really a unique feature of the language that many other programming languages lack.

Last but not least, Whiley supports a more flexible type system compared to other programming languages, e.g. *Union Types*, where one variable can be of type $A$ or type $B$, depending on the program's current flow-of execution.

**Whiley Development Environment** Program in Whiley are written using either a local text editor in combination with the Whiley Development Kit (`https://github.com/Whiley/WhileyDevelopmentKit`), or using the online web IDE `https://whileylabs.com/`.

Using the Whiley Development Kit, you can compile any Whiley application to the Java Virtual Machine and use that to execute the application. The Online IDE compiles to JavaScript and runs directly in the same browser used to develop your application.

**Example Whiley Application** Whiley is a feature-rich language, so it would be impossible to give an example of all features. However, we will try to present some features of Whiley using some examples.

Figure 2.1 shows the usage of Whiley's function specification feature.

```
1          function positive(int x) -> (bool r)
2          requires x < 100 && x > -100
3          ensures r == true || r == false
4          ensures r == true && x > 0
5          ensures r == false && x <= 0
6              if x > 0:
7                  return true
8              else:
9                  return false
10
```

Figure 2.1: Example of a Whiley application that uses Specifications

This specific example uses *requires* and *ensures* clauses to provide the compiler of extra information related to this function. The *requires* clause can be used to define constraints on the values of parameter *x* and the *ensures* clause can be used to define constraints on the values of the returned values. *requires* are referred to as *preconditions* of a function. *ensures* are referred to as *postconditions* of a function.

In this example, the *requires* clause guarantees that the value of *x* will be less than 100 and greater than -100. No function can call this function with a value outside of that domain. The *ensures* clauses guarantee that all postconditions are satisfied. The verifier uses this information to guarantee (prove) that the implementation of this function satisfies the set of *ensures* clauses.

Now, this is a rather simple example, but it does demonstrate the power of Whiley in terms of developing safer and correct working applications.

**Features of Whiley** The previous example showed how *requires* and *ensures* are used, a more complex example using *Arrays*, *Records* and *Lambda Expressions* is shown in Figure 2.2.

```
1  // Type Point is a record
2  type Point is {int x, int y}
3
4  // Type of function which accepts and returns a Point record
5  type fun_t is function(Point) -> Point
6
7  // Apply the function 'fn' to every element in the list of points
8  function map(fun_t fn, Point[] points) -> Point[]:
9      int i = 0
10     while i < |xs|:
11         xs[i] = fn(xs[i])
12         i=i+1
13
14     return xs
15
16 // Translate all points based on dx and dy
17 function moveAll(Point[] points, int dx, int dy) -> Point[]:
18     fun_t fn = &(Point p -> { x: p.x + dx, y: p.y + dy)
19     return map(fn, points)
20
```

Figure 2.2: Example of a Whiley application that uses Specifications

In this example, the function *moveAll* is a function that moves all points in an array from $x$ to $x + dx$ (and the same for the y-coordinate). A *Point* is defined as a structure having two elements: $x$ and $y$.

The *map* function, expects an array of Points and a function to apply to every element of that array. It loops over every element of the array and applies the function for each element. Finally, it returns the new array. Because this function uses a function supplied as an argument to the function itself, it can easily be used to apply different transformations without having to modify the map function itself. This is a powerful feature of Whiley.

The actual transformation that is applied to the array, is actually defined in the function *moveAll*. A *Lambda* function is created at line 14 (*&(...)*) that takes a *Point* as input and returns a new record as its output. Finally, the map function is called with this lambda function.

Although this example is only 11 lines long (excluding comments), it already showcases a vast set of features of the Whiley Language: *Records*, *Function Types*, *Arrays*, *Array Length Expressions*, *Array Access Expressions*, *Function Calls*, *Record Initialisation Expressions* and *Lambda Expressions*.

**Future Versions** Whiley is still in active development, and today's version of Whiley is capable of detecting commonly made errors, for example, null pointer dereferences and array index out-of-bounds exceptions. Whiley's future versions might include more advanced features, for example, Termination Analysis or including User-supplied proofs as part of a Whiley application.

## 2.3 Arduino and AVR Microcontrollers

An essential part of an ordinary computer or laptop is the processor (for example Intel Core i9). The processor allows modern operating systems — like Linux, OSX or Microsoft — to run a sheer amount of different applications on that system.

A microcontroller is similar to a processor, except that it is far less powerful than an ordinary processor. Microcontrollers are more often used to control hardware (LEDs, displays, motors, etc.), run in low-powered environments (often powered by batteries or solar cells), have a limited instruction set, and have limited memory.

The ATmega328p microcontroller is in the family of AVR microcontrollers made by Atmel (`https://ww1.microchip.com/`). An overview of this specific microcontrollers specifications is given in Figure 2.1. It also shows how resource limited this microcontroller actually is. Any application written for this microcontroller can be no larger than 32 kilobytes and has access to a working memory of 2 kilobytes.

| **ATmega328p Features** |
|---|
| 8-bit microcontroller |
| 32Kbytes programmable memory (FLASH) |
| 2Kbytes working memory (SRAM) |
| RISC architecture |
| 32 registers |
| 23 input/output ports |

Table 2.1: Overview of Arduino compatible functions.

This particular microcontroller is also used by the popular Arduino family of microcontroller boards. The Arduino platform is a set of boards that all contain a specific version of a microcontroller, has a power adapter, USB interface and easy to access ports. All placed onto a small PCB board.

The Arduino van be programmed using the Integrated Development Environment (IDE) supported by the Arduino Team. This IDE hides most of the complexities involved in compiling and installing (uploading) that application onto the microcontroller. However, for our project, we use *avr-gcc* to compile our application and *avrdude* to upload it to the microcontroller.

As said, the Arduino IDE hides a lot of complexities while developing applications for a microcontroller. That is probably also one of the reasons why the Arduino has been such a success. Although applications for a microcontroller are technically written in C, applications in Arduino are called *Sketches* and does not *feel* like you are developing in C.

Every sketch has two methods: *setup* and *loop*. The setup function acts as the starting point of the application and contains code (as the name implies) to *setup* the hardware. Ports on an Arduino can be used as input or output ports to either read from or write to. How a specific port is used, has to be defined at startup, and this is exactly where the setup function is for.

The *loop* function will contain all user-defined logic (for example, applying certain behaviour based on the state of ports). Whenever the loop function terminates, the Arduino framework will guarantee that it will be called upon the next cycle.

The *Hello World* equivalent for the Arduino is the *Blink* sketch. Every Arduino board contains a LED wired to port 13. The sketch in Figure 2.3 shows an Arduino example that flashes that LED on and off every second.

```
int LED = 13;

void setup() {
    pinMode(LED, OUTPUT);
}

void loop() {
    digitalWrite(LED, HIGH);
    delay(1000);

    digitalWrite(LED, LOW);
    delay(1000);
}

```

Figure 2.3: Example of an Arduino Blink Sketch

For our project, we tried to mimic the structure of Arduino sketches in our Whiley application by also defining a *setup* and *loop* method. The same Blink example can be created in Whiley, as shown in Figure 2.4.

```
import * from Arduino

int ledPort = 13

method setup():
    pinMode(ledPort, OUTPUT)

method loop():
    digitalWrite(ledPort, HIGH)
    delay(1000)

    digitalWrite(ledPort, LOW)
    delay(1000)

```

Figure 2.4: Example of a Whiley Blink Sketch

# Chapter 3

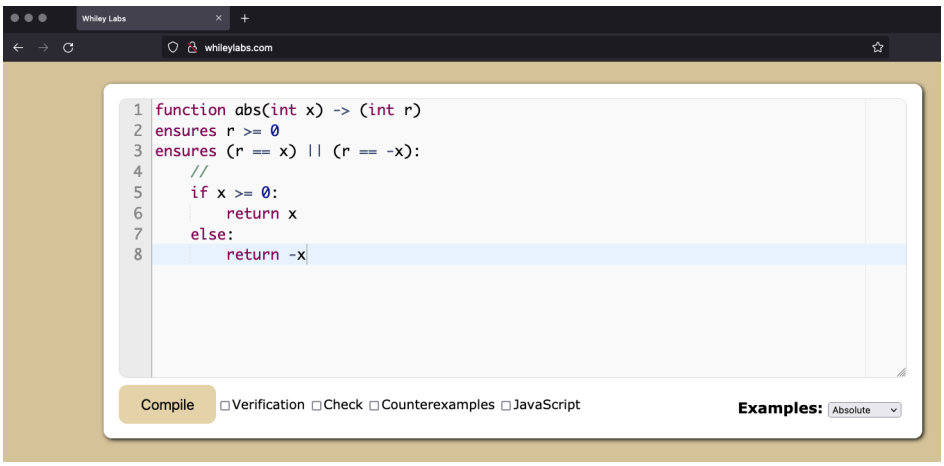
# Design

In this chapter, we will describe the architecture of the new compiler that translates the Whiley code into Embedded C code. First, we will shortly describe how a Whiley program is translated into an intermediate form and next we will describe the general process of translating that intermediate version into a domain model. Then, we will provide examples of Whiley language features and how they are translated into Embedded C code. This project aims to compile Whiley applications to Embedded C so that they can be executed on microcontrollers.

## 3.1 Compiling a Whiley Program

A Whiley program can be compiled by two methods. The first is by using the available web-application that allows you to enter code and run that inside of the browser (see Figure 3.1). This online IDE is available at `http://whileylabs.com/`.

Figure 3.1: Example of the online Whiley IDE




The second option is to install the Whiley Development Kit and use the command line or any other Java IDE to compile any code available on your local computer. The Getting Started With Whiley web-page (`http://whiley.org/about/getting-started/`) describes the steps involved into installing the Whiley Development Kit (WDK) onto your machine.

After the WDK has been installed, a Whiley application can be compiled and run as shown in Figure 3.2.

```
1          wy build
2          wy run test
3
```

Figure 3.2: Example of building and running a Whiley application

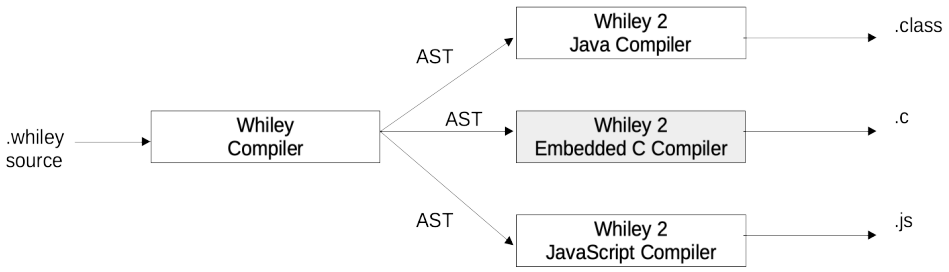
## 3.2 Whiley Compilation Pipeline

Figure 3.3 shows how the existing pipeline of the Whiley Development Kit compiles a Whiley file to a JavaScript file.

First, the Whiley Compiler parses the Whiley File into an Abstract Syntax Tree (AST). That syntax tree is subsequently forwarded to a compiler back-end, in this case a Whiley to Javascript compiler. That compiler processes the AST and generates a Javascript file.

Our project will use a similar approach. We have build a compiler plugin, similar to the Javascript plugin, but only to generate Embedded C code instead of Javascript code.

Figure 3.3: Overview of the Whiley Compiler Pipeline




In the next sections, we will focus on the Whiley Development Kit to compile Whiley applications. We also will describe how this new plugin is incorporated into the existing pipeline.

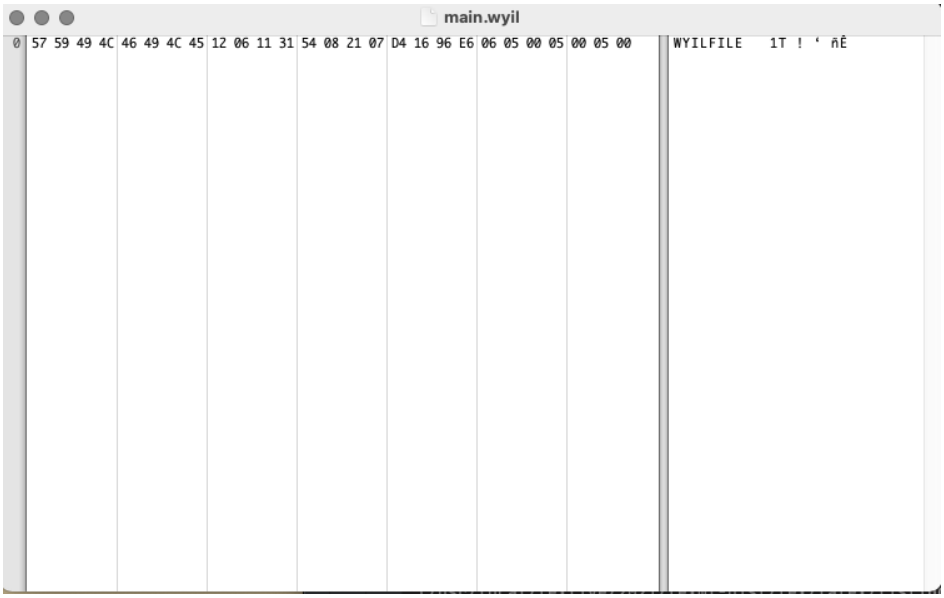
### 3.2.1 Whiley Intermediate Language

The Whiley Compiler is responsible for the translation of a Whiley file into a intermediate language (Whiley Intermediate Language or WyIL). A WyIL file, is a binary representation of the original Whiley source code, similar to a Java bytecode file. Binary files are easier to process by the Whiley Runtime environment. This WyIL-file is not directly executable by the operating system, so the Whiley Development Kit is used to actually execute that file.

Any statement is translated into an opcode with optional operands. A *while-statement* has the opcode (165) and three operands (condition, invariants and statements). Figure 3.2.1 shows an example of a Whiley assume-statement together with its binary representation in Figure 3.4.

```
1  function main() -> void
2      assume true
3
```

Figure 3.4: Example of the Whiley Intermediate Language




## 3.3 Whiley compiler plugins

The Whiley Development Kit can be extended by the means of compiler plugins. A plugin will be part of the compilation process once added to its configuration file (*wy.toml*). Figure 3.3 shows an example of a plugin (EmbeddedC) that is added to the compilation chain.

```
1 [package]
2 name="main"
3
4 [build]
5 platforms=["whiley","EmbeddedC"]
6
```

A Whiley compiler plugin is an ordinary Java application with some specific classes that allows the WDK to recognise the application as a valid compiler plugin. Any plugin needs to have at least an *Activator* class that implements the provided *Module.Activator* interface provided by the Whiley Command Line Interface.

One a plugin is created, it will be part of the compilation chain. During compilation, the plugin is provided the intermediate version of the Whiley application which it can use to generate code for the target environment — in this case Embedded C.

The Activator class is responsible for registering additional commands which will be part of the compilation chain. In this project, we have created a plugin to transform Whiley code into Embedded C code which is described in the next section.

### 3.3.1 Embedded C Plugin

Next to the previously describe *Activator* class that is run during compilation, the *Activator* adds a *Task* that actually transforms the intermediate representation into meaningful code.

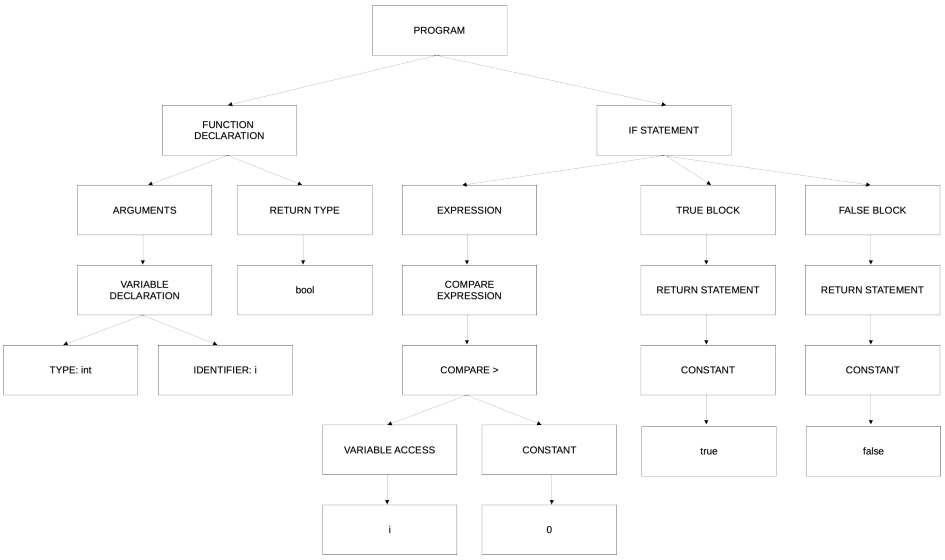
The compilation task is responsible for translating the supplied intermediate representation into Embedded C code. When the task executes, it creates domain model of the Whiley application which in turn is used to be translated into Embedded C code.

The plugin created for this project is based on the already available Whiley To Javascript plugin (`https://github.com/Whiley/Whiley2JavaScript`).

The WDK provides a class AbstractTranslator which can be used to create a new language translator. The *AbstractTranslator* uses a *Visitor*-pattern to process each opcode present in the intermediate language and generates calls to *Construct-** methods, for example *constructWhile*. Those methods can be overridden by a subclass to implement specific behaviour. In this project, we build a domain model (Abstract Syntax Tree) that represents the original Whiley application. Subsequently, that domain model is then translated into Embedded C code. An example of a Whiley application is shown in Figure 3.3.1 and its corresponding Abstract Syntax Tree is shown in Figure 3.5.

```
1 function test(int i) -> bool:
2     if i > 0:
3         return true
4     else:
5         return false
6
```

Figure 3.5: Example of an Abstract Syntax Tree




## 3.4 Compiling for a microcontroller

After the Embedded C code is generated, a *Makefile* is created so that the C-code can be compiled into Embedded C code. For this project we chose to compile for the popular *Arduino* platform. To compile for that platform (Atmel), we used the *avrdude* project available at `https://www.nongnu.org/avrdude/`.

An example of a Makefile is provided in Appendix E.

Once the code is translated into Embedded C and a Makefile is present, the code can then be uploaded to a microcontroller by invoking *make*. The code is than compiled by avr-gcc and uploaded to a connected Arduino by *avrobjcopy* and *avrdude*. An example of the output generated while installing the application onto a microcontroller, is provided in Appendix C.

# Chapter 4

# Implementation

In this chapter, we will describe how specific Language Features of Whiley will be translated into Embedded C code. We start by describing Whiley's primitive types (int, byte), then we will describe some issues related to supporting dynamically allocated memory. Finally we will present the design of the other language features we will support (e.g. Records, Arrays, Control-Flow, Union Types).

## 4.1 Primitive Types and values

Whiley is a statically typed language, and every function, variable or expression has a predefined type. The type-system in Whiley has more features than the type-system in the language C. In some cases, this means additional code must be generated to support those types that are not present in Embedded C. In other cases, this means the type itself cannot be supported at all.

Table˜4.1 describes the available Whiley primitive types and corresponding Embedded-C types.

| **Whiley** | **Embedded-C** |
|---|---|
| Void | void |
| Null | (void *) set to NULL |
| Bool | bool |
| Byte | uint_8 |
| Int | int |
| Real | float |

Table 4.1: Overview of Whiley primitive types and corresponding Embedded-C types.

Whiley's *Null* type is translated to a *void* * set to NULL because NULL can only be assigned to pointers and not to primitives. An example of how specific Whiley primitive types is translated to Embedded C is shown in Figure 4.1.

```
function test() -> bool:
    bool x = false
    byte b = 1
    int i = 2
    real r = 3.1
    return x
```

```
bool test(void)
{
    bool x = false;
    uint_8 b = 1;
    int i = 2;
    float r = 3.1;
    return x;
}
```

Figure 4.1: Example of primitive types

Whiley supports unbound integer values which are not supported by either the translator or Embedded C. As a result, overflows while running the application on a microcontroller, will result in undefined behaviour can occur if this feature is relied upon. The reason is that this project explores the ability to run a Whiley application on a microcontroller — using only a subset of the available Whiley Language Features and not to provide a complete implementation of all available Whiley features. Future versions of Whiley could support fixed-sized integer types that could benefit this compiler (see `https://github.com/Whiley/RFCs/blob/master/text/0003-statically-sized.md`)

## 4.2 Memory Safety

Many of Whiley's language features result in safer code. For example, Whiley prevents writing code that result in Array Index out of Bounds exceptions or Null-pointer de-references.

While many of these mistakes can be prevented by static code analysis, one area of errors is more difficult to prevent: memory errors. Given the limit amount of memory available on the microcontroller (2 KBytes), extra care must be taken to prevent *out of memory* errors.

How an application behaves in such a scenario is undetermined. Because random parts of memory will be overwritten, the application might crash or produce unexpected behaviour. Our example application (described in Chapter 5) won't cause many real world issues, but imagine the impact it could cause when using a microcontroller to control an elevator.

An application can run out of memory because of many reasons. Some of them are;

- trying to allocate too many variables. Every instance of a variable takes up some amount of memory. Allocating more variables than the available amount of memory results in a out of memory error.
- using recursive functions. Every variable defined inside of a function is allocated onto a *Stack*. A function that calls itself is called a recursive function. On each invocation of that function, if more and more stack space is allocated it will result in an out of memory if the function is executed too many times.
- dynamic memory allocation. Applications can allocate memory for variables using either the stack or dynamic memory. If dynamically allocated memory is not freed in a precise order, or not at all, an out of memory situation might occur.

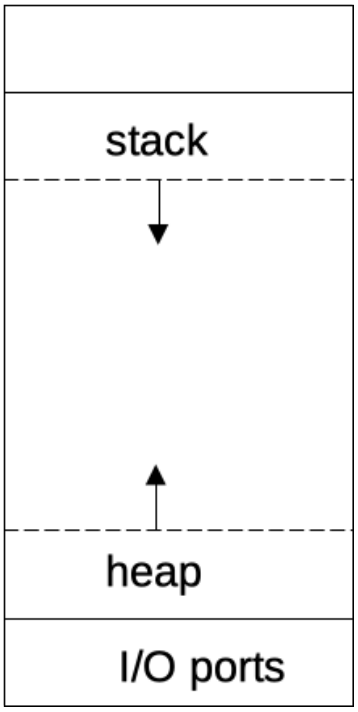
**Memory Allocation** Two main areas of memory are available to allocate variables:

1. Stack Allocation

2. Dynamic Memory Allocation / Heap Allocation

Figure 4.2 shows how the memory of a microcontroller is organised. The bottom part of memory (starting at memory location zero) will the controllers input and output registers. The rest of the remaining memory is available to be used by variables. The heap starting after the reserved IO memory and grows upwards, and the stack, starting from the upper memory and grows downwards.

Figure 4.2: Memory layout




**Issues with Dynamic Memory Allocation** An example of both stack based allocation and dynamic memory allocation is shown in Figure 4.3.

```
1 void test()
2 {
3     int i = 0                          // allocated onto the stack
4     int *j = malloc(sizeof(int))       // allocated onto the heap
5     free(j);                           // free allocated memory
6 }
7
```

Figure 4.3: Example of stack and heap allocation

In this example, two variables are allocated *i* and *j*. The first is allocated onto the stack, whereas the second is allocated onto the heap. One important task related to the allocation of objects (variables), is to release the allocated memory after use. Stack based allocated objects are automatically freed by the compiler when that variable no longer is in scope (when a function exits). However, dynamically allocated objects have to be manually freed by the developer.

The advantage of using dynamically allocated memory over stack based allocation, is that dynamically allocated objects can *live* outside of the context where it has been allocated, as shown in Figure 4.4.

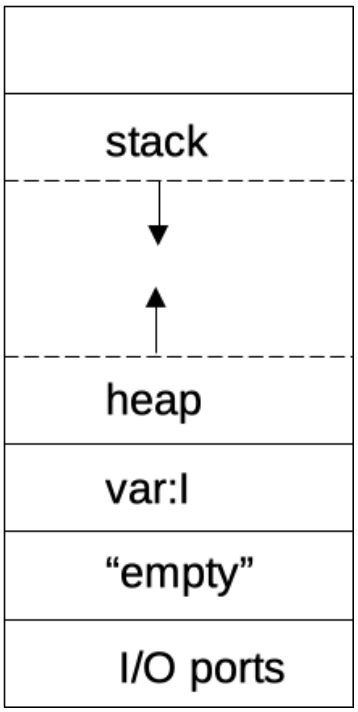
```
1  int *test()
2  {
3      int i = 0                          // allocated onto the stack
4      int *j = malloc(sizeof(int))       // allocated onto the heap
5      return j;                          // 'i' is no longer usable
6  }
7
8  void test2()
9  {
10     int *x = test();
11     printf("%d", *x);                  // variable 'j' allocated in 'test'
12                                        // is still available
13     free(j);                           // free allocated memory
14 }
15
```

Figure 4.4: Example of variable scope

The function *test* allocates two variables *i* on the stack, and *j* in the heap. The variable *i* is no longer available after the function returns, but the variable *j* will still be available after the function returns. So, after invoking the function *test* from within *test2*, that function can still use the variable defined in function *test*.

In this example, the variable is freed once it is no longer needed. However, if many variables are allocated onto the heap, and they are not freed in the reverse order of how they were created or not freed at all, heap fragmentation will occur (as shown in Figure 4.5). If three variables are (dynamically) created in order *X, Y, Z*, then they have to be freed in the order *Z, Y, X* to prevent gaps in the heap area.

Figure 4.5: Memory layout




In this example, a *gap* of memory is created below variable *i*. A side-effect of this is that if the application tries to allocate some memory in the future, it might fail to do so. It could be

possible that there isn't a large enough amount of continuous memory available to store the required variable. Even when the total amount of memory required by all declared variables doesn't exceed the total amount of available memory.

Applications that use dynamically allocated memory are prone to unspecified behaviour, one time the application might work without any issues. While another time the same application runs, the application might run out of memory.

**Dynamic Memory Allocation in Whiley** Many objects declared in Whiley require dynamic memory allocation. Especially *Arrays*, that can be created of flexible sizes, or be initialised with some set of values, only to be reset to a different set of values.

An example of this flexibility in Whiley is shown in Figure 4.6.

```
1 function test(int x) -> int[]:
2     int[] a = [1,2,3]
3     a = [1,2]
4
5     return [0; x]
6
```

Figure 4.6: Example of Whiley's array reassignment

In this example, the variable *a* is initially assigned an array containing three elements and re-assigned an array with two elements thereafter. Whilst technically this can be implemented using stack based allocation, often those dynamically sized variables are allocated onto the heap.

The example also showcases an example of an array generator (*return [0;x]*) which does require Dynamic Memory allocation in order to work. Such an object can only be allocated onto the heap, since the number of elements is only known at runtime and not at compile-time.

**Restricting Dynamic Memory allocation** Because of the previously mentioned issues with dynamic memory allocation, we have chosen not to support dynamically allocated objects in our compiler and to only support stack based allocated objects.

This creates one major restriction in terms of supporting specific Whiley language features:

- Array Generator Expressions are not supported

Only fixed sized arrays, known at compile time, are supported by this compiler. In the next sections we will show how specific Whiley Language Features are translated to Embedded C, while keeping the previously mentioned restriction in mind.

## 4.3 Records

Where Primitive types can only store one specific value, a Record is a set of values. A Record can be used to have multiple fields grouped into a structure that specifically describes some object or state, for example: a set of coordinates, or some identifying information about a person (name, age, etc).

Similarly to the translation of *Arrays* and *Records* are translated into Embedded C structures. Records can be initialised by a Record Initialisation expression. Those expressions are captured during translation and converted into declarations of temporary variables in order to support declarations in calls to functions or other expressions.

An example of a Record declaration and its corresponding Embedded C code is shown in Figure 4.7.

```
type Record is (int x, int y)
function test() -> Record:
    Record r = (1, 2)
    return r

```

```
struct Record_II {
    int x;
    int y;
};

struct Record_II test()
{
    struct Record_II temp_0 =
    {1, 2};
    struct Record_II r = temp_0;
    return r;
}

```

Figure 4.7: Example of a Record declaration and its corresponding Embedded C code

**Record Comparisons** Similar to comparing arrays, record comparisons are implemented by generating a comparison function for every record type as shown in Figure 4.8.

```
type Record is (int x, int y)
function test() -> bool:
    Record r = (1, 2)
    if (1,2) = (1,2):
        return true
    else:
        return false

```

```
struct Record_II {
    int x;
    int y;
};

bool compare_II(struct Record_II
    a, struct Record_II b)
{
    if (a.x != b.x) return false;
    if (a.y != b.y) return false;
    return true;
}

struct Record_II test()
{
    struct Record_II temp_0 =
    {1, 2};
    struct Record_II r1 = temp_0;

    struct Record_II temp_1 =
    {1, 2};
    struct Record_II r2 = temp_1;

    if (compare_II(r1, r2)) {
        return true;
    } else {
        return false;
    }
}

```

Figure 4.8: Example of Records Comparison Functions

**Unsupported Features** Whiley includes the concept of *Open Records* and *Functions/Methods* inside record definitions which both are not supported by this compiler. Not because those can't be implemented in Embedded C, but merely because this compiler/translator provides a proof of concept and is not a fully featured implementation of a Whiley to Embedded C compiler.

## 4.4 Union Types

Union types are types that can be of more than one of the other types (primitives, records, arrays). If a variable is declared as a union type, then that variables value can be one of the types in the union type definition. For example, given the following variable declaration `int|bool var`, the variable *var* can be set to either an integer or a boolean.

Union types are translated to union structures in Embedded C. Figure 4.9 shows an example of such translation.

```
function test():
    int|bool var = 1

```

```
struct Union_IB {
   int tag;
   union Data_Union_intint__ {
    int  m0;
    bool m1;
   } data;
};

int test()
{
    struct Union_IB var;
    var.tag = 0;
    var.data.m0 = 1;
}

```

Figure 4.9: Example of union types

While initialising a union type, first a variable is declared of that specific union type. Next the tag is set to the specific type it will contain. Then its value is set using the inner-defined union. This basically translates to Tagged Unions used by for example the Cyclone language.

The value of tag is set to the type that will be used to set the value to and maps to value zero (0) for the first member of the inner union, one (1) for the second member, and so on.

**Type Test Expressions** Using this tagged union structure, support for Type Test Expressions in Whiley is rather easy to transform. Figure 4.10 shows an example of a type test in combination with union types.

```
function test(int|bool var):
    if var is int:
        var = false
    else:
        var = 2

```

```
struct Union_IB {
   int tag;
   union Data_Union_intint__ {
    int  m0;
    bool m1;
   } data;
};

int test(struct Union_IB var)
{
    if (var.tag == 0) {
        var.tag = 1;
        var.data.m1 = false;
    } else {
        var.tag = 0;
        var.data.m0 = 2;
    }
}

```

Figure 4.10: Example of type testing and Tagged Unions

In this example, if an integer is stored in the union type, then it will be swapped for a boolean value. First the proper tag is set, next the value. If a boolean value is set (by inspecting the tag), then it will be swapped for an integer value.

Figure 4.11 shows another example of a Type Test Expression and *Flow Typing* where the supplied argument *var* is either an integer or null. If it is an integer, then that value is returned. Otherwise, zero (0) is returned by the function.

```
function test(int|null var) ->
    int:
    if var is null:
        return 0
    else:
        return x
```

```
struct Union_IB {
   int tag;
   union Data_Union_intint__ {
    int  m0;
    void *m1;
   } data;
};

int test(struct Union_IB var)
{
    // one (1), because NULL is the second
    element of the union type
    if (var.tag == 1) {
        return 0;
    } else {
        return var.data.m1;
    }
}
```

Figure 4.11: Example of Tagged Unions and Flow Typing

## 4.5 Arrays

An Array in Whiley is a type used to store a set of values of a specific type. Whiley supports the definition of an Array, accessing elements inside of an array, inspecting the number of elements an Array contains and initialising arrays.

The declaration of an Array in Whiley does not have to include how many elements will be stored in the Array. Simply defining the type of the Array together with an Array initialiser is enough to create a dynamically sized Array.

Array initialisers are also expressions, which means that they can be used in (for example) If-expressions.

Finally, arrays can be compared against equality.

Supporting all of the previously mentioned features in Embedded C, proved to be quite challenging. In the next section we will provide Whiley examples of all of the previously mentioned features. After that, we will describe how we have implemented these Array features into Embedded C together with the issues we have identified.

### 4.5.1 Arrays in Whiley

As previously mentioned, Whiley supports the following Array expressions:

- Array Length
- Array Access
- Array Generators
- Array Initialisers
- Array Initialisers as an expression
- Array Comparisons

The Whiley example in Figure 4.12, shows all of there features in action.

```
function test():
    int[] a = [1, 2, 3] // array declaration + array initialisation
    int element = a[0]  // array access
    int length = |a|    // array length
    int[] b = [1; 3]    // array generator
    if a = [1, 2, 3]:   // array comparisons + array initialiser expression
        // equal
    else:
        // not equal

```

Figure 4.12: Example of array functions

Arrays can be passed as arguments (by value) to a function and functions can return arrays.

### 4.5.2 Variable Scopes and Function Return values

Supporting Whiley arrays in Embedded C proved to be quite a challenge. Especially because one of the design goals was to rely on stack allocation of variables and to not support dynamic memory allocation. Because of this, all variables declared in function will be allocated onto the stack which causes certain issues while translating Whiley expressions into Embedded C.

To better understand the issues while implementing Arrays in Embedded C, we first have to explain Variable Scopes. After that, we will explain how a function is able to return large (non primitive) object. Finally, we will describe Whiley's Array Language Features and how we translate those into Embedded C.

**Variable Scopes** Variables defined in a function are bound to its local scope and will be freed when that scope exits. A scope in Embedded C, is started by an open accolade () and is terminated by a close accolade (). All variables defined in a scope, are available only inside of that scope. Scopes do link to its parent scope and as a result, all variables in a parent

scope are also available in the current scope.

An example of scope and the usability of a variable is shown in Figure 4.13.

```
1  void print(int t)
2  {
3      printf("total = %d\n", t);
4  }
5
6  void main(void)
7  {                                // starts a new scope
8      int total = 0;               // with a variable 'total' in the current scope
9
10     for (int i=0; i<10; i++) {   // this starts a new scope with
11                                  // one variable 'i' present in that scope
12
13       total += i                 // variable 'i' is accessible in this scope
14                                  // variable 'total' is also accessible (parent scope)
15     }                            // end the scope started by the 'for' statement
16
17     // at this point variable 'i' is not available because
18     // the 'scope' started by the for-statement has ended at this point
19
20     print(total);                // the function print is called with the total (by value)
21 }                                // ends the scope 'main'
22
```

Figure 4.13: Example of variable scope

**Returning large objects** Although the C Programming language specification describes *how* memory is laid out when using arrays, it does not specify how compilers should implement this. For example, it does describe that a function can return an array defined in that function, but it does not describe how a compiler should implement the code to return that array to its caller.

Returning primitives from a function to its caller is often achieved by using a dedicated processor register, but how a function is able to return large data structures, is determined by a specific implementation of the compiler. Using a different compiler can produce different results exactly because this feature is not defined in the specification.

An example of returning an array is shown in Figure 4.14.

```
1  struct Array {
2      int length;
3      int elements[10];
4  };
5
6  struct Array create()
7  {
8      struct Array arr = {.length=2, .elements={1,2}};
9      return arr;
10 }
11
12 void main(void)
13 {
14     struct Array arr = create();
15     printf("%d", arr.length);
16 }
17
```

Figure 4.14: Example of returning an array

The function 'main' calls function 'create'. Function 'create', creates a new variable, initialises a structure and an array, and returns that array to its caller. Finally, the length of the array is printed onto the screen.

Obviously, the data inside the structure does not fit inside any processor register. Also, the variable created in the function 'create' is a stack allocated variable which is unavailable as soon as that function ends.

So, how are these larger stack allocated structures returned to its caller?

As mentioned before, how a specific compiler implements this feature is defined by that compiler itself and every compiler can do this differently.

There are basically two methods available to return large objects:

- pre-allocate some memory before calling a function that returns a large object and modify the functions signature to include a pointer to that pre-allocated memory. The called function uses that extra argument to store the returned values

- allocate space inside the callee continue with its normal execution path. Copy back the data to the callers stack frame when the function returns

An example of the most commonly used latter option is shown in Figure 4.15 that shows how the code on the left could be translated by the compiler shown on the right.

```
struct Array {
    int length;
    int elements[10];
};

struct Array create()
{
    struct Array temp =
    {.length=2, .elements={1,2}};
    return temp;
}

void main(void)
{
    struct Array arr = create();
    printf("%d", arr.length);
}
```

```
struct Array {
    int length;
    int elements[2];
};

// modified function signature with
    additional argument
void create(struct Array *temp)
{
    // write to parent scope
    *temp = {.length=2,
    .elements={1,2}};
}

void main(void)
{
    // pre-allocate space in this stack-frame
    struct Array arr;

    // call function with address of locally
    defined variable
    create(&temp);

    printf("%d", arr.length);
}
```

Figure 4.15: Example of using pre allocation to return large objects

### 4.5.3 Whiley Arrays in Embedded C

In this section, we will describe how we have implemented Whiley's Array language features in Embedded C starting with an overview of the design decisions made to support arrays.

**Design** In Whiley it is possible to inspect the length of an array by using the $|array|$ expression. In Embedded C, the length of an array isn't part of the array declaration (*int arr[] = 1, 2, 3;*). Although it is possible to determine the length of any array using the $sizeof()$ function, but because Whiley supports *Union-types* (`int|null`), this method cannot be used. Primarily, because the actual type (and therefore its size) of an array element cannot be determined at that point in time at compile time.

So, in order to be able to support (for example) the Length expression, we have to store additional information along any array declaration. In general, every array declared will look like the (pseudo) code in Figure 4.16.

```
struct Array {
    int length;
    <TYPE> elements[<LENGTH>]; // for example: int elements[3];
};

int test()
{
    struct Array arr = {
        .length=<LENGTH>;
        .elements={<ELEMENTS>}
    };
}

```

Figure 4.16: Example of an array structure

Every Array will transform into a structure that contain the number of elements and the elements themself. The type of the elements member has to be part of the array definition, which causes a unique structure to be defined for every array using different types. Although it is possible to use a generic type *void∗* in Embedded C, this option cannot be used here because in that case, the elements will be stack allocated and won't be available when a function terminates.

Also, the number of elements has to be part of the array definition because initialisation of flexible array members is not allowed according to the C-standard. A flexible array member is a declaration of an array without specifying the number of elements it eventually will contain, for example: *int array[]*. This is disallowed in C because the size of every structure has to be known at compile-time.

In the next sections, we will describe the Whiley Array features, starting with an array declaration.

**Array Declaration** While building the Abstract Syntax Tree during the Transformation phase described earlier, all array declarations found in the Whiley application to compile, are stored in a list. This list is then used to create the structures that will define the arrays in Embedded C.

An example of two array declarations is given in Figure 4.17.

```
function test():
    int[] arr1 = [1,2,3]
    bool[] arr2 = [true, false]

```

```
struct Array_I {
    int length;
    int elements[3];
};

struct Array_B {
    int length;
    bool elements[2];
};

int test()
{
    struct Array_I arr1 = {
        .length=3,
        .elements={1,2,3}
    };

    struct Array_B arr2 = {
        .length=2,
        .elements={TRUE, FALSE}
    };
}

```

Figure 4.17: Example of an array declaration

**Array Length Expression** In order to support the Length expression, we transform a Whiley array to a Embedded C structure that, besides its elements, it also stores the number of elements present in the array, as shown in Figure 4.18. Although this information can be deduced from the array definition itself, we have chosen not to implement that to reduce complexity of the *Length* function implementation.

```
function test() -> int:
    int[] arr = [1,2,3]
    return |arr|

```

```
struct Array {
    int length;
    int elements[3];
};

int test()
{
    struct Array arr = {
        .length=3,
        .elements={1,2,3}
    };
    return arr.length;
}

```

Figure 4.18: Example of using the array length expression

**Array Initialiser Expression** Arrays in Whiley can be initialised by two methods:

- by type and an array initialiser
- by type and an array generator

An example of both constructs is shown in Figure 4.19. The first example, creates an array of three (3) elements, where the first element is set to 1, the second to 2 and the last element to 3. The second example, creates an array of three (3) elements all where each element is set to the value 1.

```
function test():
    int[] a = [1, 2, 3]
    int[] b = [1, 3]

```

Figure 4.19: Example of array initialisers

Because an Array Generator Expression can contain an expression by itself for both arguments (*count, value*), using an Array Generator will result in an unknown length of the array at compile time. Therefore, this feature is currently not supported by the translator.

The right part of a variable declaration ($[1,2,3]$) is also an expression itself (the Array Initialiser Expression), which is captured during translation. Every Array Initialiser expression is first translated to an array declaration, and the resulting variable is then substituted as the declarations right side part as shown in Figure 4.20. Capturing and replacing Array Initialisation expressions makes it possible to use these expressions in calls to functions and other Whiley expressions (for example *if* statements).

```
function test():
    int[] arr = [1,2,3]
    function_x([1,2])

```

```
struct Array {
    int length;
    int elements[3];
};

int test()
{
    struct Array temp_0 = {
        .length=3,
        .elements={1,2,3}
    };
    struct Array arr = temp_0;

    struct Array temp_1 = {
        .length=2,
        .elements={1,2}
    };

    function_x(temp_1);
}

```

Figure 4.20: Example of an array initialiser

**Array Access Expression** Accessing elements of an array isn't a straight-forward translation of simply accessing the *.elements* member of the structure. For all primitive array types,

it can be as simple as accessing the *.elements* member, but especially when $Union - types$ are used, the translation is more difficult, as shown in Figure 4.21

```
function test():
    int|bool[] arr = [1,2]
    int value = arr[0]

```

```
struct Array_I {
    int length;
    int elements[2];
};

struct Array_B {
    int length;
    bool elements[2];
};

struct Union_IB {
   int tag;
   union Data_Union_II {
    struct Array_I m0;
    struct Array_B m1;
   } data;
};

int test()
{
    struct Array_I arr = {
        .length=3,
        .elements={1, 2}
    };

    struct Union_IB temp_0 = {
        .tag = 0,
        .data.m0 = arr
    };

    int value =
    temp_0.data.m0.elements[0];
}

```

Figure 4.21: Example of array access

**Array Comparison Expressions** In Whiley two arrays can be compared for equality. Because the C language does not support equality comparisons of structs, translation of the equality `[1,2] == [3,4]` requires that we generate an auxiliary method (compare_0 in this case) to implement the comparison.

For every array comparison expression found while compiling a Whiley application, a comparison function is generated. However, if a comparison expression uses the same data type as a previously handled expression, then the already generated comparison function will be used and a new one won't be generated.

An example of such a compare function is shown in Figure 4.22.

```
function test() -> bool:
    return [1,2] == [3,4]

```

```
int compare_0(struct Array_II a1, struct Array_II a2) {
    for (int i=0; i<a1.length; i++) {
        int  v1 = a1.elements[i];
        int  v2 = a2.elements[i];
        if (v1 != v2) return -1;
    }
    return 0;
}

bool test()
{
    struct Array_II temp_0 = {
        .length=2,
        .elements={1,2}
    };

    struct Array_II temp_1 = {
        .length=2,
        .elements={3,4}
    };

    return compare_0(temp_0, temp_1);
}

```

Figure 4.22: Example of an array compare function

## 4.6 Control-Flow

A program is a list of instructions (statements and expressions) where when a program starts, every statement of that list is executed one after the other. A `Control Flow` statement is a statement that interferes with the *normal* program execution flow. For example, a *for* control flow statement will execute every statement in its list of instructions repeated *X* times (depending on its definition). Translating control flow statements to Embedded C is rather straightforward, because almost every Whiley control flow statement is natively supported by Embedded C. One exception is the *fail* statement, which is translated to a never ending loop.

Table 4.2 shows how every control flow statement present in Whiley is translated to Embedded C.

An example of a specific control-flow statement that uses a *do-while* loop is shown in Figure 4.23.

| Whiley | Embedded-C |
|---|---|
| Break | break |
| Continue | continue |
| Do/While | do … while(expr) |
| Fail | for(;;); |
| If | if (expr) … else … |
| Return | return expr; |
| Switch | switch (expr) case … |
| While | while(expr) … |

Table 4.2: Translating Whiley Control Flow Statements

```
function test() -> int:
    int i = 0
    do:
        i = i + 1
    while i < 10
    return i
```

```
int test()
{
    int i = 0;
    do {
        i = i + 1;
    } while (i < 10);
    return i;
}
```

Figure 4.23: Example of a do-while loop

# Chapter 5

# Evaluation

In this chapter, we will describe an example Whiley application and translate it into Embedded C code. We then describe the process of installing that application onto the Atmel micro-controller. Finally, we will discuss the results concerning the transformation of a Whiley application to Embedded C.

## 5.1 Example Application

For this case study, we have created a Whiley application and electrical unit (Figure 5.1) that displays the numbers starting from zero (0) to nine (9), over an over again. Might not be very useful, but even this simple application allows us to explore the possibilities and challenges of compiling Whiley applications for a microcontroller.

In contrast to a Web Application — where the user of that application uses a browser to access the application —, or a Desktop Application — where the user would access the application that is installed on his/her computer —, we have created an Embedded Program that runs on a microcontroller.

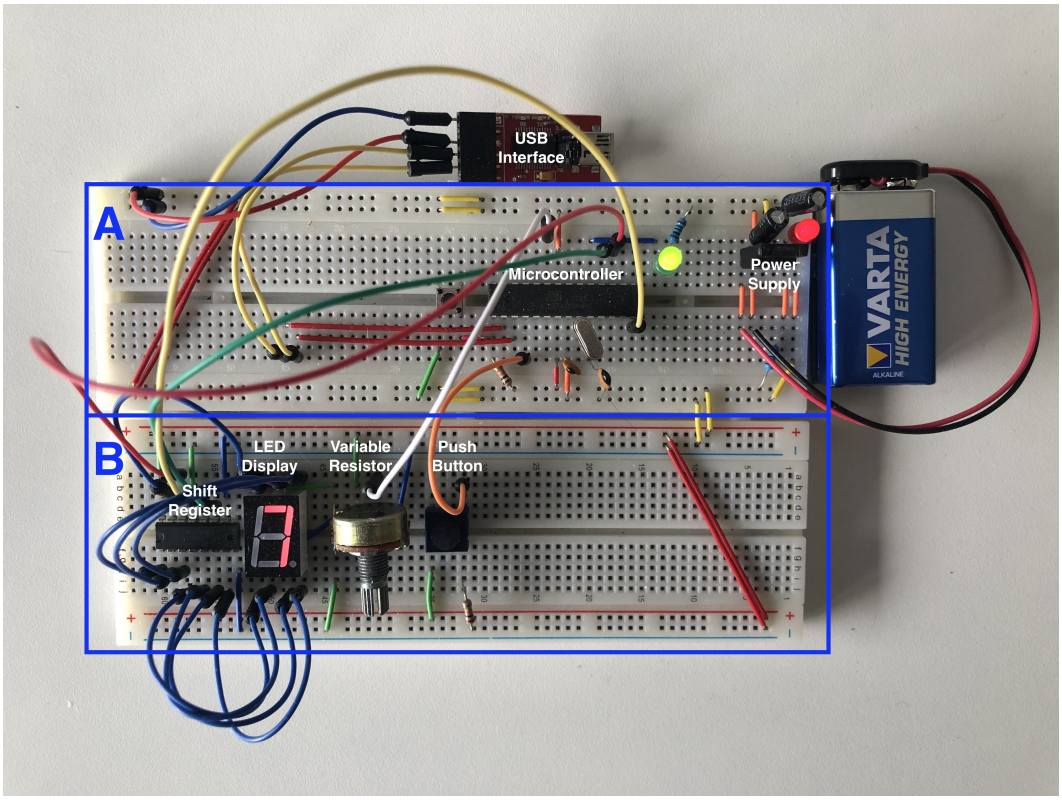
Since this project is concerned with the execution of Whiley on embedded devices, we developed a simple embedded application and tested it on an actual microcontroller.

When our application starts on the microcontroller, it first executes a *test* function to show that the seven segment display is working properly. This test sequence flashes all segments 5 times before it continues with the rest of the application. After this initial test sequence, it will start counting up from zero to nine separated by a small delay before the next number is shown. After showing the numbers from zero to nine, it simply starts over from zero again.

The time the application should wait in between each digit is adjustable by turning the potentiometer that is present on the breadboard. A potentiometer can be compared to the volume-control *dial* present on many stereo-sets. To increase the delay, the potentiometer should be turned to the left. To decrease the delay, the potentiometer should be turned to the right.

To reset the application, and to start from the initial state (test) again, the user can press the button that is present on the breadboard.

Figure 5.1: Image of the final hardware configuration




### 5.1.1 Hardware components

An overview of which components we have used to build this application is shown in Table 5.1.

| File | Description |
|---|---|
| ATmega328p | Microcontroller |
| 5161AS | 7-segment LED display |
| Potentiometer | $10K\Omega$ variable resistor |
| Button | Make contact button |
| SN74HC595N | 8-bit shift register |

Table 5.1: Overview of hardware components.

The upper part of the breadboard (Part A in Figure 5.1) is a breadboard version of a standard Arduino UNO and — besides components as LEDs and resisters — also contains the Atmel microcontroller (ATmega328p).

The components present in the lower part of the diagram (Part B in Figure 5.1), are used to support the example application. It contains the shift register, 7-segment LED display, the potentiometer and a button to reset the application.

**Breadboard** We have used a so called 'breadboard' to mount the hardware components. A breadboard is often used to create a prototype of an electronic circuit, because components can quickly be removed, re-positioned and wired without much trouble. Once the prototype

works, often a more durable and permanent version can be created using for example Print Circuit Boards (PCB).

**7-segment LED display** A 7-segment LED display is basically a set of 8 LEDs (7 segments and one dot) where each LED is positioned in such a way that it is able to display numeric digits. Because there are 8 individually controllable LEDs, in order to be able to control all LEDs at once, you would have to use 8 unique connections (and thus ports) of the microcontroller.

Because we haven't used all ports of the microcontroller, we just could wire-up 7 connections to the microcontroller , but to reduce the number of connections — and to make this example application a little bit more complex — we have used an 8-bit shift register.

This particular shift register that we used, allows you to control 8 individual output ports while using only three input ports — and therefore only using three physical connections (ports) of the microcontroller.

**Bit Shift Register** A shift register works by controlling three ports, a *clock*, a *latch*, and a *data* port. On each rise of the clock, the value present in *data* (HIGH or LOW) is shifted into an 8 bits buffer present on the shift register.

Because of this buffer, the output ports of the shift register aren't directly manipulated when shifting in bits. Initially, while shifting bits, the *latch* port should be set to LOW. Once the *latch* port is set to HIGH, the value present in the buffer is written to the output ports of the shift register. This way, intermediate results — while shifting bits — won't directly interfere with the output ports.

**Potentiometer** The potentiometer is used to control the delay while displaying the numbers. A potentiometer is basically a adjustable resistor where the actual value of its resistance is determined by the position of the knob. If you turn the meter left, the resistance will decrease but the delay will increase. If you turn it right, the resistance will increase but the delay will decrease. The potentiometer is connected to an analog port of the microcontroller. In contrast to digital ports, which can only have a value that is either HIGH or LOW, analog ports can have a range of values. The number of values the port can support is determined by its resolution. In this case, an Atmel analog port has a resolution of 10 bits.

A 4 bit resolution means there are only $2^4$ possible values (0-15), so a 10 bit resolution can contain $2^{10}$ values (0-1023).

The example application uses the value of the analog input to calculate the delay between advancing to display the next number.

**Button** The button is used to reset the application and revert to its initial state. When the button is pressed, a hardware interrupt is generated which is handled by a method defined in the example Whiley application.

The reason for creating an interrupt, is to show that it is possible to interrupt a running application, even when it is currently executing a delay statement which normally prevents any other method from being executed.

**Arduino on a Breadboard** While we were able to use an available Arduino UNO, we have decided to build an *Arduino on a Breadboard* for this project. An Arduino on a Breadboard is simply put — all components present on a standard Arduino placed onto a breadboard.

We briefly describe the arduino on a breadboard components. The board consists of three parts:

1. Power module - to convert 9–15 volts to 5 volts
2. Microcontroller - supported by a oscillator, reset button and a standard LED. This is the component that will run the Whiley application
3. Serial adapter (FT232RL) - to upload an application from the computer to the microcontroller over an USB connection

### 5.1.2 Software implementation

The software part of this application consists of two main components, the Whiley application and a supporting Arduino library partly written in Whiley and supported by a Embedded C backend.

#### Arduino Library

To make developing applications for the ATMEL platform easier, we have created an Arduino-like library that contains functions similarly present in existing Arduino IDEs (*pinMode, delay, etc*).

The Arduino library contains the following subset of popular Arduino functions:

| Function | Description |
|---|---|
| *pinMode(int port, int mode)* | Sets the specified port as either INPUT (read) or OUTPUT (write) |
| *delay(int ms)* | Delays the application for *ms* milliseconds |
| *digitalWrite(int port, bool value)* | Writes a boolean (HIGH-true/LOW-false) value to the specified port |
| *digitalRead(int port)* $\rightarrow$ *bool* | Reads a boolean value (HIGH-true/LOW-false) from a port |
| *analogRead(int port)* $\rightarrow$ *int* | Reads a ten-bit value from the specified analog port |
| *attachInterrupt(int pin, handler h)* | Binds a Whiley method to an external interrupt pint (INT0/INT1) |

Table 5.2: Overview of Arduino compatible functions.

All of the methods mentioned in Table 5.2 are physically implemented in C. The method definitions in the Whiley library are only external declarations (using Whiley's native modifier, see Figure 5.2) which have to actually be implemented in Embedded C. This is because from a Whiley runtime point of view, a Whiley application cannot directly interface with the microcontroller hardware. All methods that provide hardware interactions are controlled by a the Embedded C-library backend.

```
1 native export method digitalWrite(int port, bool value)
2
```

Figure 5.2: Example of a externally implemented method in Whiley

The library also contains two methods to modify a global boolean value (*setGlobal / getGlobal*). These two methods were necessary, because Whiley does not support globally defined variables and the example Whiley application needed one for the user-reset routine (triggered by the interrupt).

An example of a program flow while writing to a port is shown below;

1. A user application (Whiley) calls the method *digitalWrite(13, TRUE)*

2. That method is defined in the *Arduino.whileylibrary*, *butbecauseitisdefinedasnative*, it is actually defined in the Embedded C-library
3. The Embedded C-library then writes to PORTB5 of the microcontroller (port 13 on an Arduino board)

All of the previously mentioned methods are implemented in C. However, there is one method available in this library that is fully implemented in Whiley.
The method *shiftOut(int latchPin, int dataPin, int clockPin, byte val):* controls the shift-register by using the three control pins (*latch, data and clock*) to set the specified byte value.

### Whiley Application

In this section we will describe the example application from a functional point of view and include short extracts of the actual application. The full source code of the example application can be found in Appendix A.

The Whiley application starts by executing the *main* method (Figure 5.3). This method has two responsibilities, first is to configure the hardware ports as either INPUT or OUTPUT. Second is to start a continuous loop whose only goal is to execute the *loop* method over and over again.

On the Arduino platform, it is common to have a single *loop* method that contains code which runs (as the name implies) in a loop. The Arduino platform guarantees that whenever that function finishes executing, the loop is called again. We have implemented a similar pattern to closely match Arduino's design. In our code, the *main* function will call the *loop* method over and over again and the *main* function never terminates.

```
1   // main method, simply loops forever
2   method main() -> int:
3       setup()
4
5   while true:
6       setGlobal(true)
7       loop()
8
9   return 0
10
```

Figure 5.3: Example of main function

As mentioned, the *setup* method is responsible for configuring certain ports on the microcontroller for either INPUT or OUTPUT mode. The three pins that drive the shift-register are all set to OUTPUT. The button and the potentiometer pins are both set to INPUT. Further more, because the button is connected to an external interrupt pin, a method handler is specified so that when an interrupt occurs, that method handler is invoked.

The *loop* method contains the actual application logic. It contains a *for-loop* that counts from zero to nine. In each iteration, the current value is displayed on the segment display and a short delay is executed before the loop continues. In this for loop, the state of the global variable is retrieved, and if it is *false*, the for loop terminates. This global variable is *true* by default, but is set to *false* by the Interrupt Handler (described below). Once the for-loop exits, the *loop* method terminates and the application resumes the *main* method. Which will call *loop* again, but again starting counting from zero. The application has two methods used in the *loop* method, *showDigit* and *pause* as shown in Figure 5.4.

```
1  // show the expected number on the 7-segment display
2  method showDigit(int number):
3      byte[] digits = [
4          0b01111110, // 0
5          0b01001000, // 1
6          0b00111101, // 2
7          0b01101101, // 3
8          0b01001011, // 4
9          0b01100111, // 5
10         0b01110111, // 6
11         0b01001100, // 7
12         0b01111111, // 8
13         0b01101111  // 9
14     ]
15
16     shiftOut(latchPin, dataPin, clockPin, digits[number])
17
18 // delay for a number of milliseconds based on the position
19 // of the potentiometer
20 method pause():
21     int val = analogRead(potPin)
22     val = map(val, 0, 1023, 0, 9)
23     delay(map(val, 0, 9, 100, 1000))
24
```

Figure 5.4: Example of show digit method

The *showDigit* method declares an array where each entry represents the bit value of a number to display. For example, if we want to display a 2 on the seven-segment display, then the value *0b00111101* must be written to the bitshift register.

The *pause* method reads the current value from the analog input pin connected to the potentiometer. Once the value is read, its domain (0-1023) is converted to a value ranging from 0–9. After that, the value is once more mapped to a range between 100 and 1000 milliseconds which is used by the call to *delay*.

Finally, there is one more function in this example application. The function *buttonHandler* shown in Figure 5.5, is the Whiley implementation of a callback method that is executed whenever the external interrupt is raised.

```
1  // This method is invoked by triggering INT0 on the Atmega
2  method buttonHandler():
3      // set global variable to 'false' so that the loop terminates
4      // after finishing this int-handler routine
5      setGlobal(false)
6      blink()
```

Figure 5.5: Example of attaching an Interrupt Handler

Once the interrupt is triggered (by pressing the button), current execution of the application is stopped and control is given to the attached interrupt-handler. Once the handler executes, it sets the global variable to *false* so that once the interrupted application continues, the counter is reset and starts from zero again. Secondly, just before this method finishes, it

displays the test-routine again (blinking all LEDs on and off five times).

Although the Embedded C compiler does not fully supports function references or lambda functions, it does demonstrate the ability to use function references in Whiley and the translated code.

### 5.1.3 Compilation and Installation

After the application is compiled into an Embedded C file, it is ready to be uploaded to the microcontroller. Normally, you would have to compile the C code into a binary before it can be uploaded. To make this process easier, we also generate files presented in Table 5.3.

| Component | Description |
|---|---|
| *arduino.c* | The C-implementation of the Arduino library |
| *arduino.h* | Standard C-Header file for the Arduino library |
| *main.c* | Embedded C version of the Whiley application |
| *Makefile* | GNU Makefile to compile the Embedded C application and upload it to the microcontroller |

Table 5.3: Overview of additionally generated files.

To compile the Embedded C application and upload it to the microcontroller, a terminal window should be opened and from the location of the generated Embedded C files, and the command *make* can be executed (see Appendix D). After compiling the C file and converting it to a microcontroller compatible format, the program is automatically uploaded to the microcontroller.

## 5.2 Unit Test Framework

While developing an application, instead of writing one big monolith that is hard to maintain and easy to get wrong (bugs), an application is often build using small functions that — when combined together — provide the intended functionality.

Breaking up the functionality of a large application into smaller parts will makes it easier to write correct code and is most likely easier to maintain.

For example, an *Online Banking Application* could be broken up into the following modules:

- Authentication Module (login, logout)
- Transactions Module (show transaction, create transaction)
- Reporting Module (create report, print report)
- and many more ...

Breaking up an application into modules, modules into classes, all the way down to breaking up classes into functions gives us the opportunity to write test-cases for those functions. Writing a test-case for a function is a way to prove that the function under test, behaves like it is intended to.

Many programming languages support writing tests for functions which is often provided by a *Unit Test Framework*. Such a framework allows the developer to write tests in a uniform way and it provides functions to check expected results (assertions).

For example, in Whiley the function shown in Figure 5.6 allows the user of that function to sum two integers and to return the result to the caller.

```
1 function sum(int a, int b) -> int:
2     return a + b
3
```

Figure 5.6: Example of a Sum function

Now, this is a rather simple function, and it is easy to see that the function indeed does what it is intended to do. But, most applications will have functions that are much larger than the one shown in the example and it will be much harder to — just by looking at the function — confidently say that the function behaves like it is meant to be.

This is where Unit Testing plays an important part. The previous example could have been unit tested by a test-case shown in Figure 5.7.

```
1 function sum(int a, int b) -> int:
2     return a + b
3
4 public export method test():
5     assume sum(2, 3) == 5
6
```

Figure 5.7: Example of a Sum function and test case

Notice the added method *test*. This method will be executed by the compiler and the *assume* statement is executed to guarantee that the result of *sum(2,3)* equals *5*. In the case that it isn't equal to *5*, the compiler will inform the user of this error.

One side effect of writing unit tests, is that it makes it easier to *re-factor* larger applications. One could modify the implementation of the *sum* function and be sure that modification doesn't break other parts of the application, by simply executing an existing set of unit tests written for that function. If those tests succeed, then the modification (probably) won't affect other parts of the application. Now, it is important that functions are as small as possible, do nothing other than what the function is created for and have no side effects. Functions with side affects — for example the function calculates the sum of two integers, **and** write that result to disk — are much harder to test exactly because of those side affects.

### 5.2.1 Whiley's Unit Tests

Our compiler is based on the the existing Whiley2JavaScript (`https://github.com/Whiley/Whiley2JavaScript`) compiler. That compiler already contain a suite of 734 test cases that test a wide area of the JavaScript compilers output.

Integrating these tests into our compiler introduced a few issues;

1. The JavaScript tests are meant to be run in a Browser, whereas our tests should ideally be run on a microcontroller.

2. The JavaScript test suite uses many Whiley language features that haven't been implemented in our compiler

**Regarding the first issue;** while it would be possible to execute a test on the microcontroller, doing so doesn't really make sense. *What should happen if a test fails? How do we inform the developer that the test failed? Should it make a sound or flash a LED?*

So, to give the developer useful feedback regarding the result of a test, we will execute testcases on the computer itself and not on the microcontroller. However, the generated C code will not run on the computer by default. For example, it does not have a *main* method and can't interact with the microcontrollers hardware (e.g. *digitalWrite*).

To overcome these issues, we implemented the following steps while executing the Unit Tests;

1. Every Whiley testcase is compiled into an Embedded C variant
2. For every Embedded C variant, an additional *main* method is generated
3. Instead of compiling that variant with *avr-gcc*, *gcc* is used instead
4. While executing that version, a *stub* Arduino library is used. One that does contain the standard functions, but have an *empty* implementation.

**Regarding the second issue;** The JavaScript compiler implemented a complete set of Whiley Language Features whereas our compiler implemented only a small set of features. As a result, the JavaScript test suite contains tests that haven't been implemented in our compiler, which when executed using our compile, definitely will fail.

One example of a feature we haven't implemented — because our target environment does not support this — is Whiley's Array Generator expression.

An Array Generator expression is an method to build a variable sized array as shown in Figure 5.8.

```
1  function copy(int[] a) -> (int[] b):
2  int n = |a|
3  return [0; n]
4
5  public export method test():
6  assume copy([]) == []
7  assume copy([1]) == [0]
8  assume copy([1,2]) == [0,0]
9  assume copy([1,2,3]) == [0,0,0]
10
```

Figure 5.8: Example of an Array Generator and test case

In this example, the *copy* function creates an array (*[0; n]*) based on the length of the supplied array with every element set to zero (0).

We could have removed all tests that are guaranteed to fail, but decided no to do this. Even if our compiler does not support a certain feature *now*, it might do so in the future. If so, an available unit test would already be available and does not have to be created again.

Given the limitations described above, we were able to re-use the large set of existing tests and therefore, we were able to validate that parts of our compiler were correctly implemented.

### 5.2.2 Unit Test results

The test suite contains 734 tests that test specific language features, for example: tests that test *Arrays*, *For-Loops*, *While-loops*, *Lambdas* etc.

Out of those tests, only 178 tests passed (see Table 5.4).

| **Total** | **734 (100%)** |
|---|---|
| Passed | 178 (24%) |
| Failed | 509 (70%) |
| Skipped | 47 (6%) |

Table 5.4: Unit tests results

However, many language features *were* implemented by our compiler and test-cases for those did pass, but many test-cases that were designed to test a specific language feature, used other language features in that test that weren't implemented by our compiler.

For example, the test shown in Figure 5.9 to test arrays and unit types passed but the test in Figure 5.10 failed because it used an array generator.

```
1  type arr is ((int|null)[] n)
2
3  function read(arr x, int i) -> (int|null r)
4  requires i >= 0 && i < |x|:
5  //
6  return x[i]
7
8  function write(arr x, int i, int n) -> (arr r)
9  requires i >= 0 && i < |x|:
10 //
11 x[i] = n
12 //
13 return x
14
15 public export method test():
16 //
17 arr a = [1,null,3]
18 assume read(a,0) == 1
19 assume read(a,1) == null
20 //
21 assume write(a,1,2) == [1,2,3]
22
```

Figure 5.9: Example of an arrays test case

```
function copy(int[] a) -> (int[] b):
int n = |a|
return [0; n]

public export method test():
assume copy([]) == []
assume copy([1]) == [0]
assume copy([1,2]) == [0,0]
assume copy([1,2,3]) == [0,0,0]

```

Figure 5.10: Example of an array generator test case

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

For this project, we designed and implemented a While to Embedded C plugin (*Whiley2EmbeddedC*) that translates a Whiley application to EmbeddedC and can be run on a microcontroller (Arduino). We also demonstrated that we could re-use the large test suite which was already provided by the Whiley 2 Javascript project. Although, unlike the Javascript back-end compiler which supports all of Whiley's Language Features, we only support a subset of the Whiley Language Feature.

We support all of the control-loop statement like for example *for*, *while* and *if*. We provide partial support for many of the other expressions like for example *arrays*, *records* , *Union Types* and *Type Flowing*. Our project supports a subset of Whiley's type system, but fully supports *Union Types* by means of using *Tagged Union Types*. Arrays are partly supported (no support for Array Generator expressions) because we restricted ourselves to disallow dynamic memory allocation.

By means of creating an example Whiley application and a working electronic circuit, we demonstrated that we support enough of Whiley's statements, expressions and type system in our compiler to successfully compile and run an example application on the microcontroller.

## 6.2 Future Work

Although this compile is capable of producing working Embedded C code, it can still be improved by future research in many ways:

- Improving Functionality. This project created a compiler plugin that translates only a subset of the available Whiley language features present. Future work could implement more of the Whiley language features currently not supported by our compiler. Also, language features that cannot be (fully) supported on a microcontroller (e.g. lambda expressions) could be disallowed by the compiler by informing the user of doing so.

- Improving usability. The plugin translates Whiley code into Embedded C code and generates the infrastructure (Makefile, libraries) to install the application onto a microcontroller. But the actual installation of the application is a manual step. An extra step could be added to the compiler plugin that automatically installs the application onto the microcontroller.

- Improvements for safety critical environments. Whiley provides excellent language features (e.g. requires, ensures, where clauses) that allows it to run on safety critical systems. However, non of these features are currently used by our compiler. Future work might provide an implementation for these features available in Whiley.
- Runtime safety. Given the limited amount of memory available to microcontroller applications, future work could suggest changes to the Whiley language to better handle those resource limited systems. For example, the compiler can be modified using techniques such as those developed elsewhere for embedded systems ( [3], [2] or [1])
- Extends Whiley's Array Syntax. Our project relied on fixed-sized arrays and stack allocated objects only. Whiley supports dynamically sized arrays (allocated onto the heap) and does not provide support for fixed-sized arrays. Future work might include extending Whiley's syntax to support such fixed-sized structures (e.g. *int[3]* for array declarations)

# Bibliography

[1] CHATTERJEE, K., MA, D., MAJUMDAR, R., ZHAO, T., HENZINGER, T. A., AND PALSBERG, J. Stack size analysis for interrupt-driven programs. In *Proceedings of the 10th International Conference on Static Analysis* (Berlin, Heidelberg, 2003), SAS'03, Springer-Verlag, pp. 109–126.

[2] KÄSTNER, D., AND FERDINAND, C. Proving the absence of stack overflows. In *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 202–213.

[3] REGEHR, J., REID, A., AND WEBB, K. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embed. Comput. Syst. 4*, 4 (Nov. 2005), 751–778.

# Appendix A

# Example Whiley Application

```
/*
   -a-
  f   b
   -g-
  e   c
   -d-  dot

0000.0001 g
0000.0010 f
0000.0100 a
0000.1000 b

0001.0000 e
0010.0000 d
0100.0000 c
1000.0000 dot
*/

import * from Arduino

// these pin-numbers are compatible with an Arduino and are
// translated to Atmega pin-numbers
int latchPin = 8
int clockPin = 12
int dataPin = 11
int potPin = 14 // A0
int buttonPin = 2 // INT0


// enable en disable all segments 5 times
method blink():
    for i in 0..5:
        shiftOut(latchPin, dataPin, clockPin, 0b11111111)
        delay(100)

        shiftOut(latchPin, dataPin, clockPin, 0b0)
        delay(100)
```

```
// show the expected number on the 7-segment display
method showDigit(int number):
    byte[] digits = [
        0b01111110, // 0
        0b01001000, // 1
        0b00111101, // 2
        0b01101101, // 3
        0b01001011, // 4
        0b01100111, // 5
        0b01110111, // 6
        0b01001100, // 7
        0b01111111, // 8
        0b01101111  // 9
    ]

    shiftOut(latchPin, dataPin, clockPin, digits[number])


// This method is invoked by triggering INT0 on the Atmega
method buttonHandler():
    // set global variable to 'false' so that the loop terminates
    // after finishing this int-handler routine
    setGlobal(false)
    blink()


// delay for a number of milliseconds based on the position
// of the potentiometer
method pause():
    int val = analogRead(potPin)
    val = map(val, 0, 1023, 0, 9)
    delay(map(val, 0, 9, 100, 1000))


// configure pins as OUTPUT or INPUT
// and attach the external interrupt to a whiley method
method setup():
    pinMode(latchPin, OUTPUT)
    pinMode(clockPin, OUTPUT)
    pinMode(dataPin, OUTPUT)

    pinMode(buttonPin, INPUT)
    pinMode(potPin, INPUT)

    // attach INT0 to the interrupt handler method
    attachInterrupt(buttonPin, &buttonHandler)

    blink()
```

```
// display the number 0 - 9 on the segment
method loop():
    for i in 0..10:
        // exit this loop once the global variable has been set to 'false'
        if getGlobal() == false:
            return

        showDigit(i)
        pause()


// main method, simply loops forever
method main() -> int:
    setup()

    while true:
        setGlobal(true)
        loop()

    return 0
```

# Appendix B

# Arduino Library in Whiley

```
int LED_BUILTIN = 13

bool HIGH = true
bool LOW = false

int INPUT = 0
int OUTPUT = 1

type handler is method() -> null

native export method pinMode(int port, int mode)
native export method delay(int ms)
native export method digitalWrite(int port, bool value)
native export method digitalRead(int port) -> bool
native export method analogRead(int port) -> int
native export method micros() -> int
native export method attachInterrupt(int pin, handler h)

native export method setGlobal(bool value)
native export method getGlobal() -> bool

method shiftOut(int latchPin, int dataPin, int clockPin, byte val):
    digitalWrite(latchPin, LOW)

    for i in 0..8:
        if (val & 0b10000000) != 0b0:
            digitalWrite(dataPin, HIGH)
        else:
            digitalWrite(dataPin, LOW)

        val = val << 1

        digitalWrite(clockPin, HIGH)
        digitalWrite(clockPin, LOW)

    digitalWrite(latchPin, HIGH)
```

```
method map(int val, int from_a, int from_b, int to_a, int to_b) -> int:
    int from_diff = from_b - from_a
    int to_diff = to_b - to_a

    int r = (val * to_diff) / from_diff
    return to_a + r
```

# Appendix C

# Translated Embedded C Code

```
#include <assert.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <avr/io.h>
#include <util/delay.h>
#include "arduino.h"

struct Array {
    int length;
    void* elements;
};

#define handler void *(f)(void)

int Arduino_LED_BUILTIN_static = 13;
bool Arduino_HIGH_static = true;
bool Arduino_LOW_static = false;
int Arduino_INPUT_static = 0;
int Arduino_OUTPUT_static = 1;

void Arduino_shiftOut_IIIU_(int latchPin, int dataPin, int clockPin, byte val) {
    digitalWrite(latchPin, Arduino_LOW_static);
    for(int i = 0;i < 8;i = i + 1) {
        if((val & 0b10000000) != 0b0)  {
            digitalWrite(dataPin, Arduino_HIGH_static);
        } else  {
            digitalWrite(dataPin, Arduino_LOW_static);
        }
        {
            byte ___0 = val << 1;
            val = ___0;
        }
        digitalWrite(clockPin, Arduino_HIGH_static);
        digitalWrite(clockPin, Arduino_LOW_static);
    }
    digitalWrite(latchPin, Arduino_HIGH_static);
```

```
}

int Arduino_map_IIIII_I(int val, int from_a, int from_b, int to_a, int to_b) {
   int from_diff = from_b - from_a;
   int to_diff = to_b - to_a;
   int r = (val * to_diff) / from_diff;
   return to_a + r;
}

int main_latchPin_static = 8;
int main_clockPin_static = 12;
int main_dataPin_static = 11;
int main_potPin_static = 14;
int main_buttonPin_static = 2;

void main_blink__(void) {
   for(int i = 0;i < 5;i = i + 1) {
      Arduino_shiftOut_IIIU_(
        main_latchPin_static,
        main_dataPin_static,
        main_clockPin_static,
        0b11111111);
      delay(100);
      Arduino_shiftOut_IIIU_(
        main_latchPin_static,
        main_dataPin_static,
        main_clockPin_static,
        0b0);
      delay(100);
   }
}

void main_showDigit_I_(int number) {
   byte arr_1[] = {
        0b1111110, 0b1001000, 0b111101, 0b1101101,
        0b1001011, 0b1100111, 0b1110111, 0b1001100,
        0b1111111, 0b1101111};
   struct Array arr_2 = {10, arr_1};
   struct Array digits = arr_2;
   Arduino_shiftOut_IIIU_(
        main_latchPin_static,
        main_dataPin_static,
        main_clockPin_static,
        ((byte*)digits.elements)[number]);
}

void main_buttonHandler__(void) {
   setGlobal(false);
   main_blink__();
}
```

```
void main_pause__(void) {
   int val = analogRead(main_potPin_static);
    {
      int ___1 = Arduino_map_IIIII_I(val, 0, 1023, 0, 9);
      val = ___1;
   }
   delay(3000);
}

void main_setup__(void) {
   pinMode(main_latchPin_static, Arduino_OUTPUT_static);
   pinMode(main_clockPin_static, Arduino_OUTPUT_static);
   pinMode(main_dataPin_static, Arduino_OUTPUT_static);
   pinMode(main_buttonPin_static, Arduino_INPUT_static);
   pinMode(main_potPin_static, Arduino_INPUT_static);
   attachInterrupt(main_buttonPin_static, main_buttonHandler__);
   main_blink__();
}

void main_loop__(void) {
   for(int i = 0;i < 10;i = i + 1) {
      if(getGlobal() == false)  {
         return;
      }
      main_showDigit_I_(i);
      main_pause__();
   }
}

int main(void) {
   main_setup__();
   while(true)  {
      setGlobal(true);
      main_loop__();
   }
   return 0;
}
```

# Appendix D

# Output of make

```
avr-gcc -c \
    -o main.o main.c \
    -save-temps \
    -Os \
    -DF_CPU=16000000 \
    -mmcu=atmega328p \
    -Wall \
    -Wstrict-prototypes \
    -std=gnu99

avr-gcc -o main main.o arduino.o -save-temps -Os -DF_CPU=16000000 -mmcu=atmega328p -Wall \
    -Wstrict-prototypes -std=gnu99

avr-objcopy -O ihex -R .eeprom main main.hex

avrdude -v -c arduino -p ATMEGA328P -P /dev/cu.usbserial-AI03T16N \
    -b 115200 -U flash:w:main.hex

avrdude: Version 6.3, compiled on Nov 16 2020 at 16:16:00
         Copyright (c) 2000-2005 Brian Dean, http://www.bdmicro.com/
         Copyright (c) 2007-2014 Joerg Wunsch

     System wide configuration file is "avrdude.conf"
     User configuration file does not exist or is not a regular file, skipping

     Using Port                    : /dev/cu.usbserial-AI03T16N
     Using Programmer              : arduino
     Overriding Baud Rate          : 115200
     AVR Part                      : ATmega328P
     Chip Erase delay              : 9000 us
     PAGEL                         : PD7
     BS2                           : PC2
     RESET disposition             : dedicated
     RETRY pulse                   : SCK
     serial program mode           : yes
     parallel program mode         : yes
     Timeout                       : 200
```

```
     StabDelay                      : 100
     CmdexeDelay                    : 25
     SyncLoops                      : 32
     ByteDelay                      : 0
     PollIndex                      : 3
     PollValue                      : 0x53
     Memory Detail                  :

                              Block Poll               Page                       Polled
       Memory Type Mode Delay Size  Indx Paged  Size   Size #Pages MinW  MaxW   ReadBack
       ----------- ---- ----- ----- ---- ------ ------ ---- ------ ----- ----- ---------
       eeprom        65    20     4    0 no       1024    4      0  3600  3600 0xff 0xff
       flash         65     6   128    0 yes     32768  128    256  4500  4500 0xff 0xff
       lfuse          0     0     0    0 no          1    0      0  4500  4500 0x00 0x00
       hfuse          0     0     0    0 no          1    0      0  4500  4500 0x00 0x00
       efuse          0     0     0    0 no          1    0      0  4500  4500 0x00 0x00
       lock           0     0     0    0 no          1    0      0  4500  4500 0x00 0x00
       calibration    0     0     0    0 no          1    0      0     0     0 0x00 0x00
       signature      0     0     0    0 no          3    0      0     0     0 0x00 0x00

     Programmer Type : Arduino
     Description     : Arduino
     Hardware Version: 3
     Firmware Version: 4.4
     Vtarget         : 0.3 V
     Varef           : 0.3 V
     Oscillator      : 28.800 kHz
     SCK period      : 3.3 us

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: safemode: hfuse reads as 0
avrdude: safemode: efuse reads as 0
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "main.hex"
avrdude: input file main.hex auto detected as Intel Hex
avrdude: writing flash (2802 bytes):

Writing | ################################################## | 100% 1.05s

avrdude: 2802 bytes of flash written
avrdude: verifying flash memory against main.hex:
avrdude: load data flash data from input file main.hex:
avrdude: input file main.hex auto detected as Intel Hex
avrdude: input file main.hex contains 2802 bytes
```

```
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.94s

avrdude: verifying ...
avrdude: 2802 bytes of flash verified

avrdude: safemode: hfuse reads as 0
avrdude: safemode: efuse reads as 0
avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done.  Thank you.
```

# Appendix E

# Example Makefile

```
BIN=main
OBJS=main.o arduino.o
DEPS=arduino.h

PORT = /dev/cu.usbmodem14201
TOOLCHAIN = avr8-gnu-toolchain-darwin_x86_64/bin
OBJCOPY = ${TOOLCHAIN}/avr-objcopy
CC = ${TOOLCHAIN}/avr-gcc
AVRDUDE = /usr/local/bin/avrdude
BUILD_MCU = atmega328p
BUILD_F_CPU = 16000000
CDEFS = -DF_CPU=$(BUILD_F_CPU) -mmcu=atmega328p
OPT = s
CSTANDARD = -std=gnu99
CDEBUG = -g$(DEBUG)
CWARN = -Wall -Wstrict-prototypes
CFLAGS = $(CDEBUG) -O$(OPT) $(CDEFS) $(CWARN) $(CSTANDARD)

.PHONY: all compile install clean

all: ${BIN} install

${BIN}.hex: main
${OBJCOPY} -O ihex -R .eeprom $< $@

#${BIN}.elf: %.c ${DEPS}
%.o: %.c ${DEPS}
${CC} -c -o $@ $< ${CFLAGS}

main: ${OBJS}
${CC} -o $@ $^ ${CFLAGS}

install: ${BIN}.hex
${AVRDUDE} -F -V -c arduino -p ATMEGA328P -P ${PORT} -b 115200 -U flash:w:$<

clean:
rm -f ${BIN}.elf ${BIN}.hex ${OBJS}
```

# Appendix F

# Literature Review

VICTORIA UNIVERSITY OF WELLINGTON
*Te Whare Wānanga o te Ūpoko o te Ika a Māui*


School of Engineering and Computer Science
*Te Kura Mātai Pūkaha, Pūrorohiko*

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

# Methods to Create Memory Safe Applications

Juan van den Anker

Supervisor: Dr David Pierce

Submitted in partial fulfilment of the requirements for
Master of Computer Science.

**Abstract**

C is an example of an unsafe programming language which is used by many developers world-wide to create low-level applications. Unfortunately, because of the inherently unsafe characteristics of C, many safety related bugs have been created. In this document we will describe a set of potential security related issues and, based on the studied literature, we will examine alternative methods and solutions to these problems.

# Contents

# Figures

# Chapter 1

# Introduction

The programming language C has been created in the early 70s and is still used today. Because C is a low-level programming environment, it is mostly used to design device-drivers, operating systems and programs that run on embedded hardware.

Unfortunately, because C is a rather simple language which leaves a lot of responsibility in terms of safety to the user of the language, many safety related issues have been identified over the years. A large number of these are collected and documented by the Open Web Application Security Project (OWASP) [7]. They maintain an annually updated list of issues, together with an explanation and guidance on how to correct those errors. Although OWASP focuses on Web Application Security, many of the found solutions common to problems in web applications, can be applied to non-web applications.

In this document we will describe the most common types of errors found while using the programming language C and based on the studied literature, we will describe possible methods to detect, prevent or eliminate those errors. We focus on the programming language C, because the research project that we will explor in the next phase of this thesis, is related to supporting the programming language Whiley on Embedded Systems and C is the de-facto language for those platforms.

This paper is organised as follows. In Chapter 2 we provide an overview of potential programming related security issues based on the literature reviewed in this document. In Chapter 3 we will describe different solutions to these problems. In Chapter 4 we will annotate these documents in the form of a discussion. Finally, in Chapter 5 we will give our conclusions.

# Chapter 2

# Background

The C programming language was developed at Bell Labs between 1972 and 1973 and is still used by many developers worldwide to write low-level applications.

Because the programming language C is inherently unsafe, developing reliable and safe applications is a challenge. C is a powerful language that enables the user to perform low-level data manipulations and pointer arithmetic. Still, misunderstanding or misusing these features can result in many errors that surface at run-time.

Applications can be run in several different ways. Source code can directly be executed by an interpreter without the need for a compilation step. Another way is to transform source code into an intermediate form which can be executed upon a *Virtual Machine*. This *Virtual Machine* reads the intermediate code (*Byte Code* in Java for example), and provides a safe environment for the application to run. For example, the *Virtual Machine* detects buffer overflow errors and will terminate the application if such a condition arises. The final method is where an application is compiled into native machine code. In this case, the executable runs on the operating system itself. If the compiler does not generate instructions that safe-guard against buffer overflow errors and if the programmer does not include those checks manually, then the application might run in unpredictable ways with potentially severe consequences like the *Heartbleed* bug [27] that caused the exposure of sensitive information through web applications. The programming language C is an example of a programming language that compiles applications down to machine code to run on the *bare metal* (the operating system itself). So extra care must be taken by the application developer to prevent against errors (such as those described in this chapter).

Out of all of the possible failures, a *Buffer Overflow* error is relatively common amongst applications written in C and is still described on the OWASP's list of Application Security Vulnerabilities [7] and by the company *Mitre* [6] — that maintains a list of vulnerabilities — as one of the more common security vulnerabilities caused by invalid programming constructs. Because these errors can have significant security implications, much research has been invested into identifying and mitigating these.

Often, errors or security-related issues mentioned in this chapter are caused by programmers who write invalid code or ignore best practices. Also, many of these errors can be detected using so-called *static analysis tools*, which are tools that can be used during development to check the source code for these kinds of errors. The language or compiler itself can also employ other methods to prevent these errors. For example, *Bounds Checking*

code can be generated by the compiler to safeguard access to specific data structures, so if the programmer tries to access memory beyond the length of the underlying structure, the compiler generates an error or warning for the programmer. However, adding code by the compiler to check these conditions at run-time adds a certain amount of overhead. One study found that adding bounds-check code to an application reduced the performance of that application by up to 87 per cent [23].

## 2.1 Common programming errors

This chapter will describe the most common programming errors and related attacks one by one and explain through code examples what it means and the implications. In the next chapter, we will discuss various changes in the C language and tools to mitigate these errors. All code examples are written in C.

### 2.1.1 Buffer Overflow

Buffer Overflow errors are caused by writing more data to an assigned or reserved set of memory than there is actually reserved [10]. Buffer Overflow errors can corrupt the memory or stack of a process and cause it to crash (segmentation fault) or produce unexpected results. Specially crafted attacks can be formed to gain improper access to a system.

```
int main(void) {
 char temp[10];
 temp[10] = 'a';
}
```

Figure 2.1: Buffer Overflow code example

In this specific example (see Figure 2.1), there are 10 bytes allocated for storage referenced by variable *temp*. Arrays in C are indexed starting from zero (0), so in this example *slots* are available in the range from zero (0) to nine (9). Here a slot with index ten (10) is written to, which causes a buffer overflow.

If the size of a particular variable is known at compile-time, the compiler might prevent these kinds of errors (Bounds Checking). If the size isn't known at compile-time, specific languages might add run-time checks to guard users against these errors. In the programming language *Java* [14], any access to an array is always checked by the *Virtual Machine* for valid bounds. If the application tries to access an invalid location, the Virtual Machine will generate an *ArrayOutOfBounds* exception. Other languages simply might not check these kind of boundary violations and as a result simply override some other part of memory currently reserved for the application, or — in case memory is accessed which does not belong to the currently running process — potentially result in being terminated by the operating system.

### 2.1.2 Buffer Overread

Similar to *Buffer Overflow* are *Buffer Overread* errors where an application tries to read more data from a variable or buffer than the available length reserved for that variable or buffer. Buffer overreads can result in serious bugs such as the Heartbleed bug [5] where anyone

connected to the Internet is able to read the memory of systems that happen to use a vulnerable version of OpenSSL [1].

Applications experiencing from Buffer Overread errors can suffer from the same behaviour as mentioned previously. An application (or process) can crash, produce unexpected behaviour and/or sensitive information might be exposed.

```
int main(void) {
 char temp[10];
 return temp[10];
}
```

Figure 2.2: Buffer Overread code example

As shown in the example, Buffer Overread errors are similar to Buffer Overflow errors, and the same techniques are available to prevent these (Bounds Checking).

### 2.1.3 Use After Free

Use After Free attacks are caused by improper handling of references to data structures. When an application needs to allocate some amount of memory, the programmer can decide to either use stack-allocated storage or heap-allocated storage. Because stack storage is limited, large data structures are often allocated using heap storage. When an object is allocated onto the heap, a pointer is returned, pointing to a reserved amount of allocated memory. When that object is no longer needed, the previously reserved amount of memory can be returned to the object pool. This is called *freeing* or *releasing* the pointer. The previously reserved memory is then again available for other allocations.

The problem of Use After Free arises when some pointer is released, and other objects are allocated into the previously reserved memory, but the original pointer still references the *old* location. This is called a *dangling pointer*. Using this dangling pointer, new data contained at the same location can now be accessed, which can cause data corruption, application crashes or access to privileged information.

```
int main(void) {
 char *x = malloc(10);
 strcpy(x, "12345");
 free(x);
 printf("%s", x);
}
```

Figure 2.3: Use After Free code example

In the example shown in Figure 2.3, 10 bytes are reserved for variable *x*. Next the string *12345* is copied into that variable and finally the reserved memory is freed by the call to

[1] www.openssl.org

*free(x)*. However, even after the memory has been released by the call to *free*, the variable *x* still points to the original location in memory just before it was released and the call to *printf* will try print the contents of that location. In the best case, it would still contain the original string (12345), in the worst case, the memory would have been overwritten by different data — not necessarily a null-terminated string — and cause unexpected behaviour.

This also shows that these errors can be difficult to detect, because in some cases — if the program does print the original content — the programmer might not even notice this programming mistake.

### 2.1.4 Double Free

Double Free errors are caused by applications that allocate dynamic memory (heap allocation) and free that same amount of memory more than once. When dynamic memory is freed, the list that maintains all references (pointers) to dynamically allocated memory is updated to reflect this change. If that memory (or, more specific the pointer pointing to that amount of dynamically allocated memory) is freed once again using the same pointer, that list might become corrupted.

This corruption can result in a crash of the application, but more often is used by attackers to execute specially crafted code which is used to gain access to the system.

```
int main(void) {
 char *x = malloc(10);
 free(x);
 free(x);
}
```

Figure 2.4: Double Free code example

Figure 2.4 demonstrates freeing variable *x* more than once. However, this specific example does not demonstrate possible attacks.

### 2.1.5 Null Dereference

Null Dereference problems — or more commonly known as *NullPointerExceptions* — are caused by incorrectly using a pointer that is set to *NULL* as if it is pointing to some actual data. When a pointer (variable) is allocated but not actually pointing to some valid amount of data on the heap, the value of that variable is set to *NULL*. Null pointers — or null references — were not available before the invention by C.A.R. Hoare [3] as part of the language ALGOL W [2]. However, at a conference in 2009, Hoare stated that this invention was a "Billion Dollar Mistake" [17].

When a program tries to use some memory pointed by that variable (*dereferencing*), the operating system will terminate that application with a *segmentation fault* or raise an exception that can be handled by the running application. This is caused because the operating system — with the help of a hardware Memory Management Unit — does not allow a process to access arbitrary locations in memory. Only that amount of memory allocated by the operating system for a process is allowed to be read or written to by that process.

Typically, Null Dereferences are caused by improper handling of pointers and result in crashing the application. But, if an attacker is able to generate these exceptions on demand, and if the application outputs debugging information after raising this issue, that information is often used by attackers to plan further attacks.

```
int main(void) {
 char *x = NULL;
 printf("%s", x);
}
```

Figure 2.5: Null Dereference code example

In Figure 2.5, a variable is allocated, pointing to nothing (*NULL*). The next line tries to print whatever the variable is pointing to, which is *NULL* and will result in a crash of the running application.

### 2.1.6 Format String Attack

Format String attacks are caused when an application does not correctly validate user input and can result in an attacker executing arbitrary code [21]. Crashing the application and reading sensitive data are possibilities of this attack.

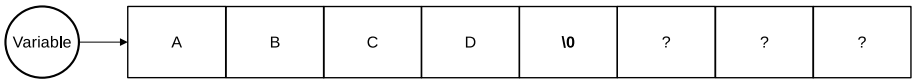



Figure 2.6: Null terminated strings in C

In Figure 2.6, a variable is pointing to a buffer (an allocated set of memory) that contains the string *ABCD*. Strings in the programming language C are stored as a sequence of characters followed by a null character (*\0*). In this example, there are 8 bytes allocated for the buffer, but only 4 bytes are used to describe the string, the remaining bytes are often set to zero, but could also contain data left-over from previously allocated structures. Other languages, like Pascal, use a different approach. In Pascal, the length of the string is followed by the individual characters of the string. Attacks like the Format String attack described here are possible exactly because C does not store the length of the string as part of the string itself. For example, the function *printf* is available to print the contents of a variable onto the screen. If you want to display the contents of a *string*, you could use the following code: `printf("%s", buffer)`, where *buffer* is a pointer to a string. Because the *printf* function does not know in advance how many characters to output, the function will output every character available in *buffer* until it reaches the null character. Now consider a situation where the buffer is not null terminated. In this case the *printf* function will not terminate and undesired behaviour can be expected.

In the programming language C, the function *sprintf* is used to convert a number of arguments into a string in a format specified by its second argument to the function (the format string). The *sprintf* function requires at least two arguments, one is a pointer to the destination to write the formatted output to, the second is either a format string or a value.

The number of additional arguments depend on the contents of the supplied second argument — the format string. If the format string contains special formatting commands (*for example: %s or %d*), then as many additional arguments have to be supplied to the function as there are formatting commands specified in the formatting string.

For example: `sprintf(buffer, "number = %d", 3);` has one placeholder (*%d*) which is supplied as an extra argument (*3*) to the sprintf function. After executing this function, the value `number = 3` is written into the destination specified by the first argument to the function call (*buffer*).

```
#include <stdio.h>
void main(int argc, char **argv)
{
char* buffer = malloc(100);
sprintf(buffer, argv[1]);
}
```

Figure 2.7: Format String Attack code example

In this example shown in Figure 2.7, the first argument supplied to the application (*argv*) is used in the call to the *sprintf* function to store the value of the argument into the allocated buffer. Normally, this code is run as `main "value"`, which copies the text *value* into the buffer. However, if the code is run as `main "%s"`, then — by design of the *sprintf* function — the function expects a third argument pointing to the data to store in the buffer. In this case, because only one argument is supplied, the *sprintf* function will use a random set of data available in the memory, which can leak sensitive data.

Besides the *sprintf* example, other functions in the C programming language are also vulnerable to these kinds of attacks. It is the responsibility of the application developer to use the functions the right way and validate user supplied input when neccesary.

### 2.1.7 Code Injections

Code Injection is a method of attack where arbitrary code is injected into the run-time of the application, which is then executed by that application. This attack is most often the result of poor handling of user data. User data, any data that is generated by users or any data that is generated *outside* of the application in question, should be considered unsafe and must be adequately validated. For instance, numeric data supplied by the user which is consumed by the application should be checked so that it actually is numeric data by the application.

```
int start(char* command) {
 char* argv[] = { NULL };
 char* envp[] = { NULL };
 execve(command, argv, envp);
}
```

Figure 2.8: Code Injection code example

In this example, the function *start* is expecting a string that resembles the name of an external application to start by the function *execve*. Normally, a command is supplied by the user of this Application Programming Interface (API) and should be restricted to a limited set of allowed commands to prevent security related issues. Because the contents of the command is not validated before the call to *execve*, **any** command can be supplied by the user. As a result, any external application can be executed using the same credentials as the application that provides this API function. Depending on how broad those permissions are set up, the consequence can be enormous.

### SQL Injection

Another more common example of code injection is *SQL Injection* where specially crafted arguments are supplied in a call to a database function. Structured Query Language (*SQL*) is a specific language used to retrieve and modify the contents of relational database systems (Progress, Ingress, Microsoft SQL Server, etc.). Many programming languages supply libraries to access a database using the SQL language. Similar to the previously mentioned code injection example where arguments should be validated, SQL queries suffer from the same issue. Any query that expects an argument whose value is supplied by a user should be validated before executing the query. Normally, the database library provides two kind of functions to execute queries, one where the query — including its arguments — is simply supplied as a string (Figure 2.9) and one where arguments to the query are supplied individual from the query itself as so called *Prepared Statements* (Figure 2.10).

```
 // supplied by the user of the application,
 // for example using a web-interface
String name = "john";

// function to check access to a system
String query = "SELECT id, name FROM users WHERE name = '" + name + "'";
PreparedStatement ps = connection.prepareStatement(query);
ResultSet results = ps.execute();
```

Figure 2.9: Simple SQL statement with arguments passed as string in Java

```
 // supplied by the user of the application,
 // for example using a web-interface
String name = "john";

// function to check access to a system
String query = "SELECT id, name FROM users WHERE name = ?";
PreparedStatement ps = connection.prepareStatement(query);
ps.setString(1, name);
ResultSet results = ps.execute();
```

Figure 2.10: Prepared SQL statement with arguments escaped

Comparing both examples, the first example (Figure 2.9) supplies the value of *name* to

the query by concatenating the value to the query string without validating and escaping the value beforehand. The second example uses prepared statements to safely supply the value of *name* as an argument to the query (specified by the question mark ?).

Now, lets imagine this piece of code is used to determine if a user is allowed to access a specific function of the application — if a record is returned by the query then the user is present in the table *users* and access is allowed, otherwise denied. An attack can then be launched if data is allowed to enter the system as shown in Figure 2.11.

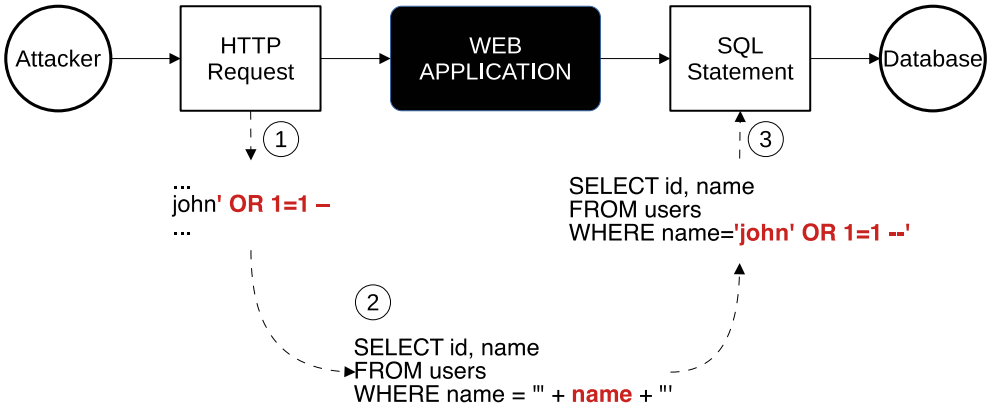



Figure 2.11: Example of SQL Injection attack

In this case, because the input data (1) is not validated before constructing the query (2), the database will execute the query as `SELECT id, name FROM users WHERE name='john' OR 1=1 --'` (3), which basically translates to `SELECT id, name FROM users` without any conditions. Most likely that statement will return some data such that access is granted where it should not based on the supplied value for *user*.

The second example as shown in Figure 2.10 can prevent this attack because the user supplied value of name is properly escaped before executing the query. In this case, the query is executed as `SELECT id, name FROM users WHERE name='john \' OR 1=1 --'` (note the escaped quote character \'), which most likely does not return any data and access is denied as a result.

Similar to Code Injection attacks, SQL attacks can result in exposing sensitive data from the database, being able to modify database contents and even execute administrative functions on the database itself (for example shutting down the database).

### 2.1.8 Denial Of Service attack

The primary goal of a Denial of Service (DoS) attack is to render an application or server unavailable. Given the complexity of any significant application architecture (servers, application layers, services), many attack vectors are available. A web server, for example, might be overloaded with requests that make it unavailable for other users of the same server.

More often than not, a Denial of Service attack focuses on rendering the resource unavailable by overloading a resource. These kinds of attacks can result in increased service response

times, service disruptions or an impact on the availability of a platform.

Similar to Code Injections, DoS attacks can be caused by not validating user supplied data. Consider the example shown in Figure 2.12. Because the user supplied value — through an HTTP POST request — of *names* is not validated before being used by the for-loop, the user has full control over the loop. That isn't an issue in this simplified example, but it might be the case when the processing part takes some time to complete and an attacker that posts thousands of *name* entries to the application.

```
public class MyServlet extends ActionServlet {
  public void doPost(HttpServletRequest req, HttpServletResponse res) {
    String[] names = req.getParameterValues("names");
    for (int i=0; i<names.length; i++) {
      // ...some code to process each supplied name...
    }
  }
}
```

Figure 2.12: Example of DoS vulnerability in Java

# Chapter 3

# Existing Solutions

In the previous chapter, we discussed security and stability related errors that can occur in applications caused by using potentially unsafe language constructs. In this chapter, we will describe a selection of research papers that address these issues in greater detail. Even though most articles provide a solution to more than one issue simultaneously, in this paper we will focus specifically on issues caused by memory safety-related problems.

The research papers addressed here all address the *Buffer Overflow* and *Buffer Overread* errors but took different approaches to provide a solution. We categorise these papers into three different approaches.

1. Human Oriented approach
2. Tool Oriented approach
3. Language Oriented approach

The *Human Oriented* approach does not change the used underlying technology — for instance, the type of programming language used — but puts an emphasis on a set of guidelines and best practices. The *Tool Oriented Approach* uses separate applications together with the source-code to check for programming errors or compile-time and run-time algorithms to identify or prevent programming errors. These tools or applications are used separate from a development environment or can be part of a development environment itself. The *Language Oriented* approach, it is either solving these errors by using a different and more safe programming language than the one already in use or modifying existing features and adding new language features to an existing inherently unsafe language in order to make it safer.

In the following sections, we will describe each approach given the studied literature. We will explain the contributions or novelties to address these kinds of issues of the papers in detail.

## 3.1 Human Oriented Approach

When using the *Human Centered* approach to tackle memory safety-related issues, the focus is not so much on changing the used programming language features or using (automated) checking tools, but more on changing the behaviour of the programmers themselves. Since its invention, the programming language C has been used to build many different applications. Simply switching to a different — and more safe — language by rewriting all of the already written C-applications is naive.

It is fairly common for a development team within a large organisation to have a set of guidelines or coding standards created, which — if followed correctly and consequently — should be able to reduce the number of safety errors introduced. Because, in practice, even the most experienced application developers introduce errors [15]. One study [15] revealed that after analysing their existing code-base, that 50 per cent of the issues found were caused by misusage of features provided by the language (C). Another study [9] suggests using guidelines to "completely stop tomorrow's attacks".

These previously mentioned guidelines should contain a set of agreements to which the whole development team agrees to. Two examples of these are "limiting the use of *Pointer Arithmetic*" or "Producing Buildable Code (without compile errors) at the end of each day". The actual type of agreements in this guideline document are often based on the team's development experience. However, unlike existing House Building Standards used for building houses which are the result of years of experience and mistakes while building homes, there is quite surprisingly "often little consensus on what a good coding standard is" [18].

Unlike Building Standards, which every building company should adhere to, coding standards (or guidelines) are most often not shared across companies. Even though there are commonalities across the guidelines, if you could compare standards between two different companies, no guideline will look the same. As stated by [18], the number of rules is also important. If the number of rules exceeds a certain threshold, it is more likely some or none of the rules will be followed by the developers [18].

### 3.1.1 Limited set of rules

In *The Power of 10: Rules for Developing Safety-Critical Code* [18], the author suggests limiting the number of rules to 10 so that developers are more encouraged to actually follow the set of rules. In the following sections, we will shortly describe some of these rules specifically related to memory safety issues.

#### Give all loops a fixed upper bound

Although this rule was intended to prevent *Runaway* code, applying this rule would also address the *Buffer Overrun* issue described earlier. This rule states that every loop used in a program must contain an upper bound. If an upper bound is present for every iteration, then static checking tools are able to prove that the code will not loop forever or that bounds are exceeded.

In [9], instead of simply adding bounds checking guards, the author suggests using a technique called *sane bounds checking*. In that case, Bounds Overreads are prevented by automatically returning zero, whereas Bounds Overwrites are prevented by dynamically extending the length of the array.

**Restrict the use of pointers**

To make applications more readable and easier to understand, the author [18] suggests limiting the use of pointers. Especially to only allow one level of dereferencing and to disallow pointers to functions. Restricting the level of dereferencing makes it easier to analyse the flow of data in the application. Also, not allowing pointers to functions makes it possible for automated static tool checkers to prove that an application does not use recursion, preventing stack overflow-related errors.

## 3.2 Tool Oriented Approach

If a new application were to be developed, one could choose to do this in a safer language that provides better memory safety features. However, it isn't always possible to use a new language if much legacy code exists that still has to be maintained and updated to a safer standard. For these cases, an alternative to the Human Oriented approach, is the Tool Oriented approach which focusses on using tools to statically or dynamically check existing source code. Another area of focus are specific algorithms to prevent or detect certain errors. A final method are so called Formal Methods, where an application or a function is formally described using mathematics.

In this section we will explore all these methods together with some examples.

### 3.2.1 Static Source code checking

There are tools that don't change the syntax or functionality of C, but statically check C source code for security vulnerabilities and programming errors. One study "Tools to make C programs safe a deeper study" [19] compares a set of different static source checking tools and source-level instrumentation tools. These tools differ from other options in such a way that errors are still able to be created by the programmer at design time, but these errors can be detected and corrected before compile-time. However, it is still the responsibility of the developer to correct those mistakes and the compiler will not reject compiling code that contain security errors.

Tools like Flawfinder (David, A., 2003) or SpLint (Larochelle and Evans, 2001) [19] are able to inspect — by statically parsing — C code to detect safety violations and common programming mistakes.

While static analysis tools are able to detect many security related errors, they are not able to detect every possible error using only static analysis [26]. An alternative to static analysis — where source code is used to detect errors at design-time — is dynamic analysis. Dynamic analysis operates at two different levels, one where source code is annotated by a compiler (or pre-compiler) to add run-time checks (SAFECode, BaggyBounds) or at a binary level, where the compiled application is run by a program that validates the application during execution (Valgrind, Purify) [26].

### 3.2.2 Formal Methods

An alternative to statically inspecting source code, is to formally specify the desired behaviour of a program through the use of predicates and let the compiler or rule checker enforce those rules. One such system is described by the study in "Integer Range Analysis

for Whiley on Embedded Systems" [24] which focuses on memory-constrained systems. To prevent bounds violation exceptions, the compiler must know the range of a particular type or data structure in advance. In Whiley it is possible to add this information using *Type Invariants*. A type invariant allows programmers to define custom types that can be restricted in their domain.

```
type month is (int m) where m >= 1 && m <=12
```

Figure 3.1: Type variant in Whiley

In the example shown in Figure 3.1, a new type *month* is defined, of which its domain is restricted from 1 to 12 (including). Another option are *Preconditions* and *Postconditions*. Where type invariants restrict a certain type, pre- and post-conditions either restrict the domain of function arguments or ensure that the return-value satisfies the specified conditions. The latter makes it possible to automatically verify the correct implementation of the function.

```
function increase(int x) -> (int y)
requires 0 <= x && x < 10
ensures y > x:
 return x + 1
```

Figure 3.2: Predicates in Whiley

In the example shown in Figure 3.2, one *requires* condition is specified and one *ensures* condition. The required condition is used to validate incoming arguments to the function. Any call to the function *increase* with an invalid range of x — less than 0 or larger than 9 — will be rejected by the compiler. Finally, the *ensures* condition is used to validate the return statement. It makes sure (*ensures*) that — in this case — the return value is larger than the supplied argument value.

The previous example displayed two possible conditions. Another possible condition are *loop invariants*. Loop invariants are available in Whiley to describe the behaviour of program loop code (for / while). However, according to the author [11], writing loop invariants is "challenging for both novices and experts alike". In their study "Finding Bugs with Specification-Based Testing is Easy!" [11], they propose a system that automatically identifies violations of loop invariants by using a tool named *QuickCheck*. The main idea of this research is to test applications leveraging specified pre- and post-conditions in combination with a variable input domain. Using the information specified by the pre-conditions, a valid set of input values is defined and executed against the function. The result is then compared against the post-conditions. Any violations are reported as counterexamples, and these can then be used to correct the functionality of the implementation or to further refine loop-invariants.

Being able to verify an application for Functional Correctness is a good system for preventing memory-related security problems in applications. Similar systems like Whiley are Dafny [25] and SPARK/Ada [12] and are all capable of verifying applications at design-time.

One system that is not bound to the language used, is TLA+ [22]. TLA+ allows you to specify the same restrictions as in the previously mentioned languages but does not incorporate those specifications into the language. It is designed to make the specifications of the system in a custom-designed editor where the design can be verified before the actual development starts. One difference is that TLA+ is able to check the *whole set of possible application states* [22] compared to checking the requirements of a single function.

## 3.3 Language Oriented Approach

An alternative to the Human Oriented approach and Tool Oriented approach, is the Language Oriented approach. In this approach, instead of changing the development processes or adding tools to the tool-chain, we focus on selecting a alternative programming language that provides better security and memory safety by design.

Whether you choose to use a different programming language or different language dialects, both options introduce restrictions compared to the original language in order to make the final application safer than without those restrictions. The specific set of restrictions present in an alternative programming language (*Java, Rust, Cyclone, Whiley*), are determined by the design goals of that alternative language — *does it prevent unchecked casts?, does it prevent pointer arithmetic?*

Alternatively to choosing a completely different programming language, you can also restrict the usage of unsafe features or programming constructs of the existing language [20]. By doing so, you do not have to invest into learning a completely new programming language, and much of the existing code-base can be reused. Modifying an existing language results in a dialect of that programming language.

Before we will describe potential solutions to the problems described in section 2.1, we first will give a brief overview in the next section of two alternative programming languages: Rust and Whiley.

### 3.3.1 Two alternative languages, Rust and Whiley

#### Language features of Rust

According to a survey by StackOverflow in 2020 [1], only 5% of the developers were using Rust for their day-to-day development in 2020. But, 86% of the respondents rated it their "*most loved language*", far more than the 33% for the programming language C. Rust was invented by *Graydon Hoare* and is currently maintained by *Mozilla* and the *Rust Foundation*.

Like many other languages, Rust also supports basic language features like *Functions* where arguments can be passed by value or by reference. The real difference compared to a language like C, are the concepts of *Mutability* and *Ownership*. Variables are immutable by default in Rust and the compiler enforces that only one part of code can mutate that variable at a time. This concept prevents *Race Conditions* in multi-threaded applications.

```
fn main() {
    let a = 1;
    println!("a = {}", a);
    a = 2;
    println!("a = {}", a);
}
```

Figure 3.3: Example of Immutable Variables

The example shown in Figure 3.3 allocates a variable *a*, prints the value on screen and reassigns value 2 to it. However, the variable is immutable by default, so the compiler will re-

port an error "`cannot assign twice to immutable variable`". Variables can be declared *mutable*, by using the *mut* keyword (`let mut a = 1`).

A unique feature to Rust is the concept of *Ownership* where every object (data) has a variable that is called the current *Owner* of that data. Different to C, where multiple variables can point to the same object, Rust only allows one owner at a time. Related to ownership are *Lifetimes*, Rust keeps track of objects allocated in a specific lifetime and automatically de-allocates that object once it goes out of scope.

```
fn main() {
  let a = 1;
  { // start of second lifetime
    let b = 2;
    println!("b = {}", b);
  } // end of second lifetime
}
```

Figure 3.4: Example of Lifetimes

In Figure 3.4, the variable *b* is automatically de-allocated once the lifetime ends. Keeping track of lifetimes and automatically de-allocating variables makes it possible for Rust to provide a garbage collector enforced by the compiler (instead of a separate background process like in Java).

```
fn main() {
  let a = 1;
  do_something(a);
  println!("a = {}", a);
}
```

Figure 3.5: Example of Ownership

Figure 3.5 demonstrates an illegal usage of the variable *a* after ownership has been given to a different function (*do_something*). The key concept is that once you pass a variable to a different function, that function will become the owner of the variable. In this example, the variable *a* is passed to *do_something* so printing the value after transferring ownership is not possible. If transfer of ownership wasn't the intention, then Rust allows you to *borrow* some variable by prefixing it with an ampersand symbol (&). By keeping track of ownership, Rust is able to determine the lifetime of an object in order to automatically de-allocate it at the right time.

**Language features of Whiley**

Similar to Rust — which is an imperative programming language — Whiley is a safe object-oriented and functional programming language. Whiley, designed by Dr D. Pierce, is the result of a challenge to create a programming language that is able to "*mathematically check the correctness of the program that is compiles*" [16]. The result of this challenge is the programming language Whiley. Whiley uses static source code verification techniques to prevent

security related programming errors at compile time.

Unlike Rust, Whiley has two concepts of functions, functions with side effects (called methods) and functions without side effects (pure functions). Pure functions in Whiley, can be annotated with pre- and post-conditions — similar to SPARK/ADA [12] — that the compiler uses to enforce correctness if possible, if the compiler can't determine its correctness then a runtime check will be generated instead. Using these annotations, Whiley is able to protect against issues described in Section 2.1 — for example Bound Errors and Null Dereference errors.

```
function max(int a, int b) -> (int r)
ensures r == a || r == b:
if a > b:
  return a
else
  return b
```

Figure 3.6: Example of a Whiley function

The function shown in Figure 3.6 is an example of a pure function that is annotated with a post-condition requirement. Using this condition, the Whiley compiler is able to verify at compile time that the specific implementation of this function meets the requirements. An error will be generated if the implementation does not meet the requirement.

Whiley applications are compiled into Java Byte Code, which makes it easy to integrate it into existing development chains.

### 3.3.2 Language restrictions imposed by alternative languages

Safer alternatives to C are available, such as *Rust* [8], *Java* [14], Whiley [4] or Cyclone [20]. Cyclone is a language designed around the C-syntax and provides all low-level functionality like C does, but it rejects complex pointer arithmetic and includes automatic bounds checking — either at compile-time or at run-time [20]. Cyclone was designed to support existing applications written in C in order to make the transition to Cyclone as easy as possible. Cyclone support all of the standard C lexical constructs and is almost POSIX compliant. To protect against *Buffer Overflow* attacks, Cyclone introduces a concept called *Fat Pointers*. Fat Pointers are like ordinary pointers in C, except that Cyclone does not permit pointer arithmetic and stores additional information for each pointer. Next to the address in memory, the pointer is pointed at; it also records information about the length of the data structure it is pointing at. Fat Pointers do introduce extra overhead in terms of memory usage, which is addressed in a different study in *Low Fat Pointers* [13] where they propose an alternative encoding to store the length of the object inside of the pointer itself. In their study, a set number of bits of a pointer is reserved to store the length of the allocated structure. Different algorithms can be used to store this size information. The author suggest to create a set of variable length blocks (regions) of size 16bytes, 32bytes, 64bytes, .... The used region is then encoded into the newly allocated pointer.

Many applications rely on functionality provided by other applications (application libraries, operating system libraries), which expect pointer arguments to be in the *'normal'* C-format.

But because Fat Pointers implement a different data structure for pointers, those applications are not Binary Compatible with existing libraries. One option is to rewrite those libraries using Cyclone, but the authors of *Stack Bounds Protection with Low Fat Pointers* [13] claim that source code is not always available for those libraries. They introduce the concept of *Low Fat Pointers* which does guarantee binary compatibility and also reduces the memory overhead caused by Fat Pointers.

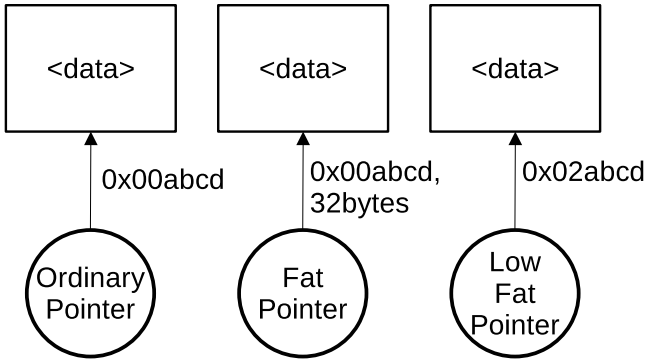



Figure 3.7: Ordinary pointers, fat pointers and low fat pointers

Unlike *Fat Pointers*, applications using *Low Fat Pointers* are binary compatible because the structure of the pointer itself does not change. As mentioned before, the length of the data structure that a Low Fat Pointer refers to, is encoded inside the pointer itself. A set number of bits of the pointer is reserved to indicate which region (the size of the object) a pointer is referring to. Figure 3.7 displays three pointers, an ordinary pointer returned by a call to *malloc*, a fat pointer (which also records the size of the object) and the mentioned low fat pointer.

Similar to *Lifetimes* in Rust, Cyclone prevents dereference of dangling pointers by *Region Analysis* [20]. As a matter of fact, Rust is inspired by Region Analysis of Cyclone [8]. Cyclone tracks variable definitions and the associated lifetime of these in each region (block of code), and automatically de-allocates those variables when that region ends. Any variable used after that region is considered to be illegal (non-live variable access), and the compiler will reject that program.

A common error as a result of using C is *Use After Free* or *Double Free* (recall Section 2.1.3). Similar to Cyclone, Rust prevents this by automatically de-allocating objects when the lifetime of that object expires. Unlike C, in Rust variables with references to heap-allocated data are automatically garbage collected at compile time (without the overhead of a Garbage Collector at run-time like in Java) by "statically tracking the lifetime of the object and de-allocating it" [8]. This means that once an object is no longer valid, the Rust compiler automatically generates code to de-allocate the object. An advantage over a compile-time garbage collector over a run-time garbage collector is that the latter introduces run-time overhead whereas the former does not.

# Chapter 4

# Discussion

Out of all of the issues and possible solutions described, adding specific runtime code to the application (such as bounds checking at runtime) to prevent memory related safety issues, will potentially decrease the runtime performance. However, tools or programming languages that are able to address these issues at compile time, do not suffer from these performance issues. Whiley and Rust are able — to a certain extend — to validate boundaries and null dereferences at compile time. Where they are not able to check these at compile time, they will generate run-time code instead.

Preventing dangling pointers is well studied in the literature. One solution is static pointer analysis, where the compiler restricts unsafe access to dangling pointers based on the lifetime of a pointer, another is to completely restrict or remove pointer arithmetic in the (dialect) language (like Cyclone). To prevent bound exceptions, one possible solution is to add extra runtime information to pointers. The biggest down-side of the latter is that binary compatibility with existing libraries can be broken and those libraries have to be recompiled. Binary compatibility is a major concern that prevents a broad adaption of these solutions. Supporting existing code-bases, without having to rewrite everything into a new language is also important in terms of adaptation. Whiley compiles to Java Byte Code, and because of this, writing critical parts of an application in Whiley seems like a reasonable solution.

Systems like TLA+ and SPARK/ADA provide a good method for formally specifying required functionality and behaviour of a system (design by contract). The downside is that is used to *design* a system and it does not generate code that adheres to that system. That is, in my opinion, the biggest risk for using those methods. It might give you the false impression that your code is *safe*. However, languages like Whiley and Daphy do support some form of formal specification and combines these with the source code into one code-base. This makes the code more robust, but implementing those predicates — especially loop invariants — seem to be quite a difficult task for unexperienced developers.

Also, tools that statically check existing code (either source code or binary code), provide a lot of help in terms of reducing the number of bugs. But, again, there is a gap between a memory safety related issue that can be detected by those tools and verifying that the code behaves like intended.
Overall, I find this to be an interesting research area where even after some many years, the desired state hasn't been reached.

# Chapter 5

# Conclusion

Reducing the amount of code helps in reducing the number of potential bugs [9]. But in situations where cutting down on the number of lines is not possible, applying Formal Methods to the remaining set of code and functions is a suitable option to precisely describe — using mathematical expressions — how a program or system should behave [22]. However, it does requires extra expert knowledge which can be challenging to most developers who are less experienced with these systems [11].

Where Formal Methods sounds like the *Holy Grail* to preventing errors, using a safe programming language sounds like a reasonable alternative. While developing an application using Formal Methods, you could prove that every single function of the application and every possible application state conforms to the (formal) design — assuming that the formal design describes the complete functionality of the application. But, writing formal specifications is not an easy task for most developers [11].

Languages like Rust and Whiley do provide a programming environment that is competitive to C, but they neglect the enormous amount of legacy code that has been created of the past 40-50 years [19]. For new projects though, using languages like Rust, Whiley or Java seems a viable route to take.

Instead of just rewriting existing code, changing, adapting or restricting the existing C language is the next best option. This acknowledges the enormous investment that has already been placed into creating that code, and at the same time can detect and prevent safety related issues.

Overall there doesn't seem to be one good answer to prevent and reduce the number of bugs in C applications. And there (still) isn't a single programming language that combines all of the best options into one. However, there does seem to be a roadmap from adapting unsafe and unreliable applications to a more safe and reliable one. One might suggest to start with using (static checking) tools, then use a dialect of C, next apply formal methods to the codebase only to end with actually abandoning all of those band-aids and start using a real safe language, ideally in combination with a language that supports formal methods as an integral part of the language.

# Bibliography


[1] 2020 Developers Survey. `https://insights.stackoverflow.com/survey/2020#technology`, *Last accessed: 14 June 2021*.

[2] Algol W. `https://en.wikipedia.org/wiki/ALGOL_W`, *Last accessed: 14 June 2021*.

[3] Tony Hoare. `https://en.wikipedia.org/wiki/Tony_Hoare`, *Last accessed: 14 June 2021*.

[4] Whiley - A Programming Language with Extended Static Checking. `http://whiley.org/about/overview/`.

[5] Heartbleed bug causes major security headache. *The Idaho business review* (2014).

[6] Cve Mitre Database, June 2021. `https://cve.mitre.org/`, *Last accessed: 14 June 2021*.

[7] Open Web Application Security Project (OWASP). `https://owasp.org/`, June 2021.

[8] BALASUBRAMANIAN, A., BARANOWSKI, M. S., BURTSEV, A., PANDA, A., RAKAMARI, Z., AND RYZHYK, L. System Programming in Rust: Beyond Safety. *Operating systems review 51*, 1 (2017), 94–99.

[9] BERNSTEIN, D. Some thoughts on security after ten years of qmail 1.0. pp. 1–10.

[10] BIERBAUMER, B., KIRSCH, J., KITTEL, T., FRANCILLON, A., AND ZARRAS, A. Ifip advances in information and communication technology, 2018.

[11] CHIN, J., AND PEARCE, D. Finding Bugs with Specification-Based Testing is Easy. *The Art, Science, and Engineering of Programming, 2021, Vol. 5, Issue 3.* (2021).

[12] CREUSE, L., HUGUET, J., GARION, C., AND HUGUES, J. Spark by Example: an introduction to formal verification through the standard c++ library. *ACM SIGAda Ada Letters 38*, 2 (2019), 89–96.

[13] DUCK, G., YAP, R., AND CAVALLARO, L. Stack Bounds Protection with Low Fat Pointers. In *NDSS* (2017).

[14] GOSLING, J., JOY, B., STEELE, G., BRACHA, G., AND BUCKLEY, A. *The Java®Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 2014.

[15] HAXX, D. Half of CURL's vulnerabilites are C mistakes, March 2021.

[16] HOARE, T. The Verifying Compiler: A Grand Challenge for Computing Research. *J. ACM 50*, 1 (Jan. 2003), 63–69.

[17] HOARE, T. Null References The Billion Dollar Mistake, 03 2009.

[18] HOLZMANN, G. J. The power of 10: rules for developing safety-critical code. *Computer (Long Beach, Calif.) 39*, 6 (2006), 95–99.

[19] JI-MIN, W., LING-DI, P., XUE-ZENG, P., HAI-BIN, S., AND XIAO-LANG, Y. Tools to make C programs safe. A deeper study. *Journal of Zhejiang University. A. Science 6*, 1 (2005), 63–70.

[20] JIM, T., MORRISETT, J., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of c. In *Cyclone: A safe dialect of C* (01 2002), pp. 275–288.

[21] KILIC, F., KITTEL, T., AND ECKERT, C. Blind Format String Attacks. In *International Conference on Security and Privacy in Communication Networks* (Cham, 2015), J. Tian, J. Jing, and M. Srivatsa, Eds., Springer International Publishing, pp. 301–314.

[22] LAMPORT, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[23] NECULA, G., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy software. *ACM transactions on programming languages and systems 27*, 3 (2005), 477–526.

[24] PEARCE, D. J. Integer Range Analysis for Whiley on Embedded Systems. *2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops* (2015), 26–33.

[25] RESEARCH, M. Dafny: A Language and Program Verifier for Functional Correctness, December 2008.

[26] SAEED, A., AHMADINIA, A., AND JUST, M. Tag-protector: An Effective and Dynamic Detection of Illegal Memory Accesses through Compile Time Code Instrumentation. *Advances in Software Engineering 2016* (04 2016).

[27] THAKKAR, A. Heartbleed: A Formal Methods Perspective. *CIS 673: Computer-Aided Verification* (2020).