Degree Project in Computer Science and Engineering
Second cycle, 30 credits

Decompiling Go
Using metadata to improve decompilation readability

MATTIAS GRENFELDT
Decompiling Go

Using metadata to improve decompilation readability

MATTIAS GRENFELDT

Degree Programme in Computer Science and Engineering
Date: December 1, 2023

Supervisors: Linnea Stjerna, Simon Lindholm, David Broman
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Swedish title: Dekompilering av Go
Swedish subtitle: Att använda metadata för att förbättra dekompileringens läsbarhet
Abstract

Malware written in Go is on the rise, and yet, tools for investigating Go programs, such as decompilers, are limited. A decompiler takes a compiled binary and tries to recover its source code. Go is a high-level language that requires runtime metadata to implement many of its features, such as garbage collection and polymorphism. While decompilers have to some degree used this metadata to benefit manual reverse engineering, there is more that can be done.

To remedy this, we extend the decompiler Ghidra with improvements that increase the readability of the decompilation of Go binaries by using runtime metadata. We make progress towards enabling Ghidra to represent Go's assembly conventions. We implement multiple analyses: some which reduce noise for the reverse engineer to filter through, some which enhance the decompilation by adding types, etc. The analyses are a mix of reimplementations of previous work and novel improvements. The analyses use metadata known beforehand but in new ways: applying data types at polymorphic function call sites, and using function names to import signatures from source code. We also discover previously unused metadata, which points to promising future work.

Our experimental evaluation compares our extension against previously existing extensions for decompilers using multiple readability metrics. Our extension improves on metrics measuring the amount of code, such as lines of code. It also decreases the number of casts. However, the extension performs worse on other metrics, producing more variables and glue functions. In conclusion, our extension produces more compact code while also increasing its informativeness for the reverse engineer.

Keywords

Decompilation, Go, Runtime Metadata, Reverse Engineering
Sammanfattning

Datorvirus skrivna i Go ökar, men verktyg för att undersöka Go-program, såsom dekompilatorer, är begränsade. En dekompilator tar en kompilerad binär och försöker återskapa dess ursprungliga källkod. Go är ett högnivåspråk som kräver metadata under körtid för att implementera många av dess funktionaliteter, såsom automatisk minneshantering och polymorfism. Medan dekompilatorer i någon mån har använt denna metadata för att gynna manuell reverse engineering, så finns det mer som kan göras.

För att åtgärda detta bygger vi en utökning till dekompilatorn Ghidra som förbättrar dekompileringens läsbarhet för Go-binärer genom att använda körtidsmetadata. Vi gör framsteg mot att få Ghidra att kunna representera Gos assemblerkonventioner. Vi implementerar flera analyser: några som minskar bruset för undersökaren att filtrera bort, några som förbättrar dekompileringen genom att lägga till datatyper, etc. Vissa analyser är återimplementationer av tidigare arbeten, och vissa är originella. Analyserna använder tidigare känd metadata, men på nya sätt: de applicerar datatyper vid anrop till polymorfiska funktioner, och använder funktionsnamn för att importera funktionssignaturer från källkod. Vi upptäcker även tidigare okänd metadata, som är lovande att undersöka i framtida studier.


Nyckelord

Dekompilering, Go, Körtidsmetadata, Reverse Engineering
Acknowledgments

I would like to thank my supervisors Linnea Stjerna, Simon Lindholm, and David Broman for a lot of support, discussions, and feedback. Without you, I would not have been able to finish this project.

Thanks to Calle Svensson for his help during the project and for inspiring me to become interested in computer security, to begin with.

Thanks to Josh Grunzweig for giving permission to reproduce Figure 1.1 in this thesis.

Thanks to Johan Adamsson and Wiktor Dobrosierdow for the company in the lab during the project. Many productive (and unproductive) conversations helped keep the spirit up.

Lastly, I thank my partner Asta for all the support and collaboration during these years at KTH.

Stockholm, December 2023
Mattias Grenfeldt
### B Implementation examples
- **B.1 Source code**
- **B.2 Binaries**

### C List of non-returning functions

### D x86-64 registers

### E Polymorphic runtime functions analyzed

### F Evaluation binaries

### G Results
- **G.1 Comparing phases**
- **G.2 Comparing candidates**
- **G.3 Tables**
Chapter 1

Introduction

A decompiler [6] takes the output of a normal compilation procedure and tries to reverse it. That is, it tries to reconstruct the high-level source code from the resulting binary executable. Decompilation has many uses: improving the speed of manual reverse engineering, enabling static analysis tools that only work on source code, and recovering source code for lost projects. Within the reverse engineering aspect there are many applications as well: reverse engineering for interoperability, malware analysis, and vulnerability discovery. The focus of most decompilers which are considered state-of-the-art [10, 24, 26] (IDA¹ and Ghidra²) is on decompiling C programs.

Several threat intelligence companies have observed that Go is on the rise as a malware development language³. Figure 1.1 shows how the amount of Go malware rose between 2016 and 2019. Go is attractive to malware developers because of its ease of development, statically compiled binaries (no dependency problems), and simple cross-compilation from a single code base to multiple operating systems and architectures. Antivirus software has also struggled with detecting Go malware because of its large file sizes and because the binary structure and assembly conventions are different from classic C/C++ malware.

There has been some work adapting decompilers to the Go language. Some have started supporting Go’s custom application binary interface (ABI) and calling conventions, but the support is still limited. Go binaries are known

¹IDA: https://hex-rays.com/
²https://ghidra-sre.org/
Figure 1.1: Timeline of Go Malware samples based on first seen dates. Source: https://unit42.paloaltonetworks.com/the-gopher-in-the-room-analysis-of-golang-malware-in-the-wild/. Permission was granted by the author of the article for reproduction of the graph.
to contain a wealth of metadata [11]. This metadata is used to support high-level language features. However, most decompilers only make little use of this information to enhance their output.

With the increasing need to reverse engineer malware written in Go, it becomes desirable to improve the readability of the decompilation as much as possible. Malware is often analyzed manually by a human, for which readability is key to understanding the program [10, 16, 43]. The goal of this thesis is to use as much metadata present in the binaries as possible to enhance readability for the reverse engineer by extending and modifying a decompiler.

1.1 Research question

Here follows the research question:

• **How much can the readability of the decompilation of Go programs be improved, for manual analysis, by taking advantage of the inherent information present in the binaries?**

This can further be specified in the following sub-questions:

1. What kind of information present in Go binaries can aid in decompilation?

2. How can that information be used to improve readability?

3. How much do these improvements increase readability compared to previous work?

1.2 Research methodology

To answer the research question, we implement an extension for the decompiler and reverse engineering suite Ghidra. Our extension implements many improvements that utilize metadata. Chapter 3 describes our extension in detail.

We compare our extension against several pre-existing decompilers and extended decompilers. As test cases, we use several malware binaries. They are selected at random from different versions of Go to make the set representative. A large open-source program is also included in the test suite.

Code readability is a subjective concept that does not have an objective definition. A qualitative user study could be used to measure the readability of
the decompilations. Instead, we choose to measure readability quantitatively using several metrics. The metrics are taken from previous work. Chapter 4 describes our evaluation and results.

We focus our study on improving the readability of the decompiled code, but another interesting aspect is the code's correctness. Correctness here means whether the decompiled code accurately represents the semantics of the underlying assembly code. We do not perform any dedicated correctness evaluation of the implemented transformations. For manual analysis, it is still valuable to improve readability, even if the decompilation is not fully correct.

1.3 Delimitations

The focus of the project is to show how using metadata is useful for improving decompilation, not to make a fully complete system that supports all cases. Therefore, several delimitations are made to save time.

The project only considers binaries produced by the standard Go compiler gc which is developed by the Go team. This is what gets invoked when running go build. There are other compiler implementations such as gccgo\(^1\), gollvm\(^2\), and TinyGo\(^3\). The main reason for this delimitation is that gc is the most commonly used implementation.

We pick primarily version 1.19.5 of Go to work on since this is the most recent version of Go as of starting this project. The results are likely applicable to adjacent versions, but some features will not work on older or future versions.

Another assumption is that the binaries are not post-processed in any way after compilation. It is common in malware development to obfuscate and pack code to make reverse engineering more difficult. Counteracting these measures is outside the scope of this project. We do allow binaries to be compiled with the following flags: go build -ldflags="-s -w". These flags strip debug symbols from the binary.

The project restricts itself to x86-64 binaries. This is because most malware targets Windows systems, which almost all use x86-64.

In version 1.18, Go introduced generics to the language. We do not try to decompile generics since it is such a new feature of the language.

\(^1\)https://gcc.gnu.org/onlinedocs/gccgo/
\(^2\)https://go.googlesource.com/gollvm/
\(^3\)https://tinygo.org/
1.4 Contributions

The contributions of this thesis are:

- A systematic overview of the available metadata in Go binaries.
- A review of previous Go reverse engineering research.
- An approach to utilizing Go's metadata to improve decompilation.
- An implementation of the approach as a Ghidra extension.
- An evaluation of our extension and several existing solutions based on multiple readability metrics.

1.5 Structure of the thesis

Chapter 2 presents background information and related work. Chapter 3 goes through the implemented extension in detail, describing all improvements. Chapter 4 describes the evaluation and the results. Chapter 5 discusses and reflects on the results and the project as a whole. Finally, Chapter 6 suggests future work and concludes the thesis.
6 | Introduction
Chapter 2

Background

This chapter starts by giving a short introduction to the Go programming language (Section 2.1). It then describes the useful metadata that can be found embedded in Go binaries (Section 2.2). Afterward, a description of decompilers in general is given, and a more detailed description specifically for Ghidra’s decompiler (Section 2.3). Finally, the related work is detailed (Section 2.4).

2.1 The Go programming language

Go\(^1\) is a general-purpose, imperative programming language created in 2007 at Google. The language is sometimes referred to as golang, however its proper name is Go. The language features static typing and a garbage collector. The language compiles directly to native code and a developer’s code is bundled with a runtime. All binaries produced are statically linked. This means that all library code is included in the binary, as opposed to dynamic linking where library code resides in separate files. This results in large binaries, but no dependency problems. Because of the static linking, there is more freedom to change the ABI between language versions. Since, in general, code does not interact with older or newer compiled libraries. Go does not follow C’s calling conventions internally.

Go has a strong focus on concurrent programming and implements so-called goroutines as primitives in the language. A goroutine is a green thread that can simply be invoked by using the keyword `go` in front of a function call. A green thread, as opposed to a normal operating system (OS) thread, is not

\(^1\)https://go.dev/
scheduled by the OS scheduler but by a scheduler that is part of a runtime in user space. Green threads are more lightweight than OS threads. Go's runtime contains a scheduler that controls the distribution of the goroutines across several OS threads.

There are several built-in data structures in the language: slices (dynamically sized lists), maps, and channels. Since they are part of the language and not the standard library, there is some syntactic sugar for using them. Channels are used to communicate in a FIFO manner across goroutines.

Go code is split into packages. A package is essentially a directory with .go source code files. Projects typically use code from many packages. The packages can be divided into three categories: standard library, third-party, and user-written code. Go's standard library is feature-rich, containing for example packages for building HTTP(S) servers and clients, cryptography, archiving, etc. With this large amount of useful features, most malware is sure to make extensive use of the standard library. There is a plethora of open-source third-party Go packages on sites like GitHub. As a reverse engineer, the most interesting part of a Go binary is the packages written by the author of the program itself.

2.1.1 Go's ABI

Until version 1.17, Go used an ABI with a stack-based calling convention, called ABI0. In version 1.17 ABIInternal\(^1\) was introduced, containing a register-based calling convention. The new ABI is unstable and can change between Go versions.

\(^1\)Detailed design document: https://go.dev/s/regabi

ABIInternal primarily passes arguments and return values in registers. Note that Go allows multiple return values from functions. When the registers are too small to pass the given value, it is instead placed on the stack. The caller also allocates spill space on the stack for all arguments passed in registers. All registers are caller-saved, that is, the callee can overwrite any registers however they want.

2.1.2 Interfaces

The language features interfaces as its main form of subtype polymorphism. An interface specifies a set of functions that the implementer must implement to fulfill the interface. At runtime, calling these virtual functions is implemented using dynamic dispatch. With dynamic dispatch, the function to
Listing 2.1: Examples of functions, structs, interfaces, methods, and receivers.

type formatter interface {
    number(int64) string
}

type radix struct {
    base int
}

// Method with a value receiver: (r radix)
func (r radix) number(n int64) string {
    return strconv.FormatInt(n, r.base)
}

// Normal function
func print100(f formatter) {
    // Dynamic dispatch of f.number
    fmt.Println(f.number(100))
}

func main() {
    print100(radix{base: 16})
}

call is determined based on a runtime value. An example of interfaces can be seen in Listing 2.1. Note the difference between methods and normal functions in the example. Methods are functions with receivers. A receiver, for example (r radix), is an extra argument to the function. The method is attached to the type of the receiver.

The language implements type assertions and type-based switch cases. These can be used at runtime to determine the concrete type of an interface. The language also implements reflection at runtime through the standard library reflect package. The package can be used to introspect types and create new ones. All of these features are for example used in the standard library json package when encoding and decoding JSON data.
2.2 Information in Go binaries

The module data (moduledata)\(^1\) data structure contains runtime metadata that is used for various purposes. The structure is embedded in every binary upon compilation and there is no compiler flag for excluding it. Here follows an explanation of the data it contains that might be of use for decompilation. The intended use by the runtime of the information is also described. A summary of the tables presented in this section can be seen in Figure 2.1.

2.2.1 Basic tables

There are some tables of metadata that are used in subsequent more advanced tables. They are:

- The function name table (funcnametab) contains the package and function name of every function.

\(^1\)https://github.com/golang/go/blob/go1.19.5/src/runtime/symtab.go#L410-L457
• The file table (filetab) contains the name of every source code file that was part of the compilation.

2.2.2 PCDATA table

The PCDATA table (pcdata)\(^{1}\) is a structure which can map values of the program counter (PC) to metadata. The table holds several sub-tables containing mappings from PC to stack pointer (SP) values, file names (through filetab), line numbers, etc.

2.2.3 Function table

The function table (ftab) contains information about every function in the binary. It contains the entry point address of the function, its name (using funcnametab), the size of its arguments, and various offsets into other tables. Using these offsets, it is possible to look up in pcdata which source code line a certain assembly instruction was compiled from. This is used in stack traces.

Being able to look up the name of a function based on the value of the PC is used by the runtime scheduler when considering whether to preempt a goroutine. The runtime can preempt goroutines in several ways, one of which is called asynchronous preemption. Asynchronous preemption can happen when a goroutine is at an asynchronous safe point, by the runtime sending a signal to the goroutine and taking over execution. However, functions that are part of the runtime cannot themselves be asynchronously preempted, though they can be preempted in other ways. Therefore, the scheduler must first check whether the target goroutine is currently executing a runtime function before it decides to asynchronously preempt it.

2.2.4 FUNCDATA table

For every function entry in ftab, there exists an associated FUNCDATA table (funcdata), which contains extra metadata about the function. Just like pcdata, funcdata consists of several sub-tables containing different metadata. Here follows a description of the different kinds of metadata that are available using a combination of sub-tables in pcdata and funcdata for a certain function.

\(^{1}\)Design document: https://go.dev/s/go12symtab
Using StackMapIndex in pcdata, and ArgsPointerMaps and LocalsPointerMaps in funcdata, it is possible to look up which values on the stack are live pointers, for any value of PC. This is used by the garbage collection (GC) as roots during its mark and sweep algorithm.

The StackObjects sub-table in funcdata contains information about the offset and size of objects on the stack: arguments, return values, and local variables. It also contains a bitmap of which words in the objects are pointers and which are not. This is used when the stack of the current goroutine needs to be resized or moved by copying. During the copying, all pointers targeting the old stack must be adjusted to point to the same offset in the new stack. Note that the bitmap contains all pointers, not just the ones that need to be adjusted.

Using InlTreeIndex in pcdata, and InlTree in funcdata it is possible to know where exactly functions have been inlined, and which function it was. This information is represented as a tree structure, so it even contains information about multi-level inlining. This information is used when the scheduler considers whether to asynchronously preempt a goroutine, as described in Section 2.2.3. That is, if a goroutine is currently executing an inlined runtime function, this can be detected and taken into consideration. The inlining information is also used when printing stack traces during crashes.

The ArgInfo sub-table in funcdata contains information about the structure of function arguments. This is used when printing stack traces during crashes, to look up argument values on the stack. The table details the offset and size of each field in the arguments, and which fields are part of structures. The table does not however give any indication of which type each field or struct has. An example can be seen in Listing 2.2. The information is also limited: it only details up to 10 fields and 5 levels deep of structure nesting.

### 2.2.5 Type links

The type links (typelinks) structure is a list of type descriptor structures (_types). Each _type contains a full description of the type it represents: structures contain names, types, and tags for fields, anonymous functions contain their full signatures, types with methods defined on them have full descriptions of all methods, etc. These descriptors are used for several purposes. For example as arguments to functions that allocate data, such as runtime.newobject. As arguments to functions that create instances of polymorphic data structures: runtime.makechan, runtime.makeslice, and runtime.makemap. They are also used as
Listing 2.2: Example of the information contained in the ArgInfo sub-table of funcdata.

```go
package main

import "fmt"

func main() {
    type Bar struct {
        x int64
    }

    type Foo struct {
        x int16
        y Bar
    }

    // ArgInfo for Zoo will contain the following.
    // Each field is described by (offset, size).
    // {(0, 2), {(2, 8)}}, (10, 4)
    func Zoo(f Foo, n int32) {
        // ...
    }
}
```

part of interfaces so that the underlying type can be asserted.

Another table that is strongly related to the typelinks is the Interface table (itab), which describes which types implement which interfaces. When some method is called on an interface, an entry in itab is used to find the concrete function to call.

## 2.3 Decompilers

Decompilers are in general structured similarly to compilers, with a frontend, middle end, and backend. There have been many decompilers designed over the years. The boundaries between the front, middle, and back end are not definite and often differ between decompilers. Their general pattern is described below.

A decompiler starts with disassembling the binary and lifting the assembly code to an intermediate representation (IR). Even the first step, disassembly, is not entirely straightforward [33]. To make further analysis simpler and instruction set architecture (ISA) independent, the assembly is lifted to an IR [17, 25]. IRs are often similar to reduced instruction set computer (RISC) architectures, having few, simple instructions.

To be able to utilize classic compiler algorithms, it is required to convert the linear IR into a control flow graph (CFG). Recovering the CFG and identifying function boundaries can also be tricky [2, 29, 33]. Some
problematic concepts are non-returning functions and indirect control flow.

This often marks the end of the frontend and the start of the middle end. The middle end consists of several passes, often iterated in a loop until the result is satisfactory. Just as in compilers, decompilers utilize static single assignment (SSA) to make many of the passes more efficient [41]. Types of variables are inferred using hints and constraints from how the data is used [22, 23, 32, 40, 47]. An important step is control flow structuring: the process of converting the CFG to structured code, that is, code with if-statements and loops [15, 39, 44]. Data flow analysis is utilized to recover expressions. It is also used to recover the signatures of functions based on the calling convention.

In the backend, there is usually an extra pass of control flow structuring. Finally, code in the chosen high-level language, usually C, is constructed and emitted. Decompilers in reverse engineering suites are often constructed as iterative tools, rather than one-shot tools. This means that a reverse engineer often continually enhances the decompilation with extra types, function signatures, and names and lets the decompiler rerun all of its analyses.

2.3.1 Use cases and requirements

Decompilers can be designed for different purposes, thus having different requirements for their output. If the goal of the decompilation is to be fed into other automatic analyzers, it might be more desirable to optimize for correctness of the decompilation than for readability [16, 26, 39, 42]. On the other hand, if manual analysis is desired, then readability becomes more important than correctness [10, 15, 43, 44]. Previous research has even shown that presenting control flow in a way that is not an exact representation of the underlying assembly can improve the effectiveness of manual reverse engineering [10].

2.3.2 Ghidra

Ghidra is a reverse engineering suite developed by the National Security Agency (NSA). It was released as open-source software after a presentation at the RSA Conference in 20191. Ghidra features a graphical user interface (GUI) application built in Java and a decompiler written in C++. The two components communicate using XML-RPC.

1https://www.youtube.com/watch?v=r3N13ig8H7s (Accessed: 2023-02-22)
Figure 2.2: Ghidra's decompilation flow, as described in its documentation.

How exactly Ghidra's decompiler works has not been described in any academic paper by the authors of the tool. However, since it is open source it is available for anyone to investigate. An overview of Ghidra's decompilation pipeline, according to its documentation, can be seen in Figure 2.2. Ghidra disassembles and lifts the assembly into Low P-code. After the decompilation, the result is in High P-code, which is rewritten into C and is presented to the user.

Ghidra's decompiler can incorporate user annotations on the decompiled code to improve the decompilation. These annotations consist of assigning types to variables, setting function signatures, *etc*.

## 2.4 Related work

This section presents the related work for this thesis. Section 2.4.1 discusses the academic related work and Section 2.4.2 discusses the non-academic related work.

### 2.4.1 Academic

Engelke [11] describes the metadata structures present in Go 1.8 binaries, lifts binaries to a custom IR, and performs constraint-based type inference on the IR. Even though the stack pointer bitmap and stack object layout metadata (see Section 2.2.4) is mentioned in their thesis, they do not use it to aid the type
Eklind [8] created a modular decompiler with the goal of components being interchangeable. Components communicate using language-agnostic interfaces, enabling the components to be implemented in different languages. The decompiler's high-level output is Go code. However, the goal of the project was not to improve the decompilation of Go binaries but rather to investigate the feasibility of modular decompilers.

Decompilers can be divided into those working on native ISAs and those working on bytecode. Some examples of bytecode languages that have been decompiled include Java Virtual Machine (JVM) bytecode [16, 31], Python bytecode [1], and Dalvik bytecode (for Android) [28]. Overall, decompiling bytecode is simpler than decompiling native machine code [9, 12, 31]. Decompilation of certain bytecode languages is even considered by some to be "a solved problem" [28]. While the general techniques are similar, bytecode is often simpler, having fewer instructions that are purpose-made to represent the higher-level language. These bytecode formats also contain more type information. The programming languages compiling to bytecode often implement reflection, requiring extra metadata to be present at runtime.

Even though Go is a high-level, garbage-collected language akin to those that compile to bytecode, it instead compiles to native machine code. To support Go's features, its binaries do however contain a wealth of runtime metadata (see Section 2.2). Similarly, Swift compiles to native machine code and its binaries also contain an abundance of metadata [19]. The metadata is used to implement reflection, and generics, and to assist in debugging. However, to the best of our knowledge, this metadata has not been used to improve Swift decompilation.

C++ binaries sometimes contain metadata called runtime type information (RTTI). It contains information about types, classes, and polymorphism. RTTI has been used to improve C++ decompilation [9, 13, 46]. A more language-agnostic form of metadata is debug information in the form of DWARF data or PDB files. This debug information has been used to improve decompilation [20, 21]. Go's compiler generates DWARF data by default, though this can easily be turned off by setting certain compilation flags: go build -ldflags="-s -w".

Debug information and RTTI are valuable for decompilation, but the information is often stripped or not generated at all during compilation. On the other hand, much of the metadata in Go binaries is essential for the runtime to function, making it a more reliable source of information (see Section 2.2).
2.4.2 Non-academic

There has been a lot of non-academic previous work when it comes to utilizing metadata to simplify reverse engineering of Go binaries. The previous work is most often in the form of an extension to a reverse engineering suite. An overview of previous works and their features can be seen in Table 2.1. The table is constructed from reading the source code of the extensions. Since we only have partial access to IDA during this project, the features are guessed from public information and a few test decompilations, so the accuracy may vary. Almost all extensions restore the names of functions based on the ftab.

Many extensions employ different heuristics for detecting and recovering strings. Read-only strings in Go binaries are not stored null-terminated, but rather in one large blob. Pointers to this blob are combined with a size to define the string. We do not implement any string-finding heuristics in our extension.

Many extensions parse the typelinks data structure, recreate all _type structures, and label them with descriptive names. Then, when any _type is used as an argument, the label will show which type is used. Even though many parse the typelinks, few use this information to automatically recover the discovered structures. None of the extensions investigated then apply these recovered structures in the decompilation to improve the output.

Some extensions have added varying support for Go's calling conventions. These extensions all target Ghidra. Ghidra has flexible features for introducing new calling conventions such as its compiler specification (.cspec) files and a feature called Custom Storage.

Go's compiler generates two utility functions called runtime.duffzero and runtime.duffcopy which consist of long sequences of repeating assembly code used to zero out or copy large chunks of memory. Because of this, the functions need to be annotated with a custom calling convention. Only a few extensions recognize the special functions.

It should be noted that IDA has in recent versions (starting with 7.6, released on 22 March 2021) significantly increased their built-in support for Go reverse engineering. They support most of the features mentioned in Table 2.1, including support for both calling conventions. This might be a reason why IDA plugins have not focused on trying to add the calling conventions. IDA also recovers the signatures of all methods, using information from typelinks.

Two previous works, Ghidra 10.3 and Monoidic, had their release during or after we finished implementing our extension. Both implement features that, before their release, only our extension implemented. Ghidra 10.3 handles
| Table 2.1: Non-academic previous work for simplifying Go reverse engineering. For links to the projects, see Appendix A. |
|---|---|---|---|---|---|---|---|---|---|
| **jeep-go-lang-antlr** | x | x | x | x | x | x | x | x | x |
| **go-rex** | x | x | x | x | x | x | x | x | x |
| **IDZ-go-helper** | x | x | x | x | x | x | x | x | x |
| **R2-go-helper** | x | x | x | x | x | x | x | x | x |
| **Gostrings** | x | x | x | x | x | x | x | x | x |
| **binary-go-lang-symbol-restore** | x | x | x | x | x | x | x | x | x |
| **GoString** | x | x | x | x | x | x | x | x | x |
| **Go(No)Return** | x | x | x | x | x | x | x | x | x |
| **Ghidra Built-in, from 10.3** | x | x | x | x | x | x | x | x | x |
| **Monoid** | x | x | x | x | x | x | x | x | x |
| **Cujo AI** | x | x | x | x | x | x | x | x | x |
| **ugo-ghidra** | x | x | x | x | x | x | x | x | x |
| **Radare2 gostringsr2** | x | x | x | x | x | x | x | x | x |
| **R2-gohelper** | x | x | x | x | x | x | x | x | x |
| **R2-go-helpers** | x | x | x | x | x | x | x | x | x |
| **Ghidra Built-in** | x | x | x | x | x | x | x | x | x |
| **JEB jeb-golang-analyzer** | x | x | x | x | x | x | x | x | x |
| **Table** | 2.1: |
| **Non-academic previous work for simplifying Go reverse engineering.** |
| **For links to the projects, see Appendix A.**

**CC** stands for calling convention.
non-returning functions correctly, see Section 3.3 for more details. Monoidic restores the signatures of library functions by parsing library source code, see Section 3.7 for more details. Both of these improvements depend on that function names have been restored.

Two of the projects, GoReSym and the Go Reverse Engineering Tool Kit (go-re.tk), are not designed as extensions to specific reverse engineering suites, but rather as standalone frameworks. Their goals are to be more stable across Go versions than normal extensions and provide a basis for other tools to build on.

While many extensions use the ftab for recovering names, line numbers and file names, and use the typelinks for recovering type information, it seems like these sources of information are not used to their fullest potential. None of the extensions, nor IDA, make use of the information in funcdata.

Our extension implements all features mentioned in Table 2.1, except for string finding heuristics and stack-based calling convention support. The most basic features are reimplementations of previous work: restoring function names, labeling _type structures, and recovering structures. A few reimplementations also improve on the previous work: the register-based calling convention support, and the handling of Duff's devices. As mentioned earlier in this section, some features we implemented were novel, but later became effectively reimplementations when more related work was published during our implementation phase: marking of non-returning functions, and importing signatures from libraries. Finally, we also add some novel features not present in the table: type recovery at polymorphic function calls, and patching out a stack growth prologue.
Background
Chapter 3

Implementation

Our extension of Ghidra consists of several parts. Figure 3.1 shows an overview of our implementation, its different components, and how they interact. The components are numbered (1)-(11). We implement the blue components. Sections 3.1 to 3.9 describe the implementation of our components in detail. We extend version 10.2.3 of Ghidra since it was the newest when we started the project. The full implementation is open-sourced on GitHub\(^1\).

Part (1), goretk_util, is a standalone command line application written in Go whose purpose is to gather information used by other components. It uses two libraries not written by us: (2) the go-re.tk library\(^2\) for extracting metadata from binaries, and (3) go/parser\(^3\), the Go parser available in the standard library, for parsing library source code. Given a binary or a directory with source code, goretk_util will extract information and transform it into a common JSON format. This information is fed to and used by components (8)-(10).

Components (5)-(11) are all individual Ghidra analyzers implemented in Java. The analyzers can interact with the representation of the binary in Ghidra, applying extra types, function signatures, etc. Together with the .cspec file (4), they form a Ghidra extension.

We add a .cspec file (4) to Ghidra to describe the basics of Go's ABI. Ghidra can handle binaries for multiple processors by using its SLEIGH disassembler. SLEIGH is a processor specification language that was originally based on SLED [35]. Using SLEIGH, Ghidra can be extended

\(^{1}\text{https://grenfeldt.dev/projects/masters-thesis/}\)
\(^{2}\text{https://go-re.tk/}\)
\(^{3}\text{https://pkg.go.dev/go/parser}\)
to handle new ISAs as well as new compilers for pre-existing ISAs. The compilers are specified in so-called .cspec files. A .cspec file is an XML file that specifies the alignment and sizes of basic integer data types as well as one or more calling conventions. By adding our .cspec file, Ghidra can use the new calling convention in its automatic analyses. Sections 3.1 and 3.8 go more into detail about representing Go's calling convention.

None of the components (5) and (6) use any information provided by goretk_util. They both however improve the decompilation in some way. Component (5), the Duff function annotator, finds and annotates two compiler-generated functions for copying and zeroing memory, called runtime.duffcopy and runtime.duffzero. These functions use a different calling convention from the standard one. Section 3.2 discusses the details.

Both components (6) and (7) reduce noise in the decompilation. Component (6) removes a function prologue present in all Go functions that use the stack. This removes a few lines from every function and sometimes helps Ghidra's automatic analyses to better track variable usage. Section 3.4 gives more details. Component (7) gives Ghidra's automatic analyses more information by marking which functions are non-returning. Without this, the decompilation of functions calling non-returning functions contains lots of
incorrect code. Section 3.3 shows examples before and after marking.

Components (8) and (9) both use the metadata extracted from the binary to recover functions and types, respectively. Sections 3.5 and 3.6 describe the two components. The recovered function names and data types are used by components (7), (10), and (11).

The library signature importer (10) applies the signatures from Go source code to functions based on their name. This can be done since the original names of functions are restored, thanks to the function name recovery component (8). For more details, see Section 3.7.

Finally, the polymorphic signature annotator (11) looks at function calls taking _type descriptor structures as arguments and enhances their signature based on it. This relies on the data type recovery component (9) having already annotated which data type each _type descriptor refers to. This component is described in Section 3.9.

All examples that include source code in the following sections are compiled with Go 1.19.5 using the following command: go build -ldflags="-s -w" code.go.

3.1 Calling convention

To model Go's calling convention as closely as possible, we use a combination of a .cspec file, Custom Storage, return structures, and register assumptions. Two challenges are encountered when modeling the calling convention with only the .cspec file, both of which can be addressed by using Ghidra's Custom Storage feature.

The first challenge is sending large structures in registers by value. As described in Section 2.1.1, Go uses a register-based calling convention as part of its ABIInternal ABI. Go often passes structures by value as arguments or return values. The fields of the structures are then split into different registers. Modelling which registers are used for argument and result passing is done by adding so-called pentry tags to the .cspec file. An example of this can be seen in Listing 3.1. To model that multiple registers can be used to send structures, special "join"-entries are created which specify several registers. Before Ghidra 10.3 however, the "join"-entries are limited to joining together only 4 registers. Our extension is built for version 10.2.3. The pieceN attributes seen in Listing 3.1 only go up to piece4. In ABIInternal for x86-64, up to 9 registers can be used in tandem to pass structures.
Listing 3.1: Entries in the .cspec file that specify which registers are used to pass arguments and return values. Special "join"-entries are created to model that multiple registers may be used together to send structures.

```plaintext
<!- 8 byte arguments -->
<pentry minsize="1" maxsize="8">
  <register name="RAX"/>
</pentry>
<pentry minsize="1" maxsize="8">
  <register name="RBX"/>
</pentry>
<!- etc... -->

<!- 16 byte arguments -->
<pentry minsize="9" maxsize="16">
  <addr space="join" piece1="RBX" piece2="RAX"/>
</pentry>
<!- etc... -->

<!- Join pentries for 24 and 32 byte arguments/results -->
```

The second challenge is about the order in which values are passed in registers or on the stack. Using ABIInternal, values are passed primarily in registers but also on the stack. Once the 9 registers are occupied, the remaining values are placed on the stack. This can be modeled in the .cspec file. However, ABIInternal also specifies that individual values should either be placed entirely in registers or on the stack, not both. If there are unoccupied registers left after some value has been placed on the stack, the next value is not forced to be placed on the stack, but can instead occupy the registers. This makes it possible for a function call taking three arguments to be stored in any of the following arrangements, depending on the arguments' sizes: [register][register][register], [register][register][stack], [stack][register][stack], [register][stack][register], etc. This is not modelable in the .cspec file since it requires different storage orders for different functions.

To address these two limitations of the .cspec file, it is possible to specify the argument and result passing storage individually per function using Ghidra's Custom Storage feature. Using Custom Storage, it is possible to override the storage pattern specified by the .cspec file. There is no limitation on the number of registers, and the order of values placed on the stack or in registers can be mixed in any way. Custom Storage is used extensively by the Library Signature Importer described in Section 3.7.
However, even using Custom Storage, there are still issues with modeling how structures are passed as values. Section 3.8 discusses the problems.

Being designed to decompile C code, Ghidra can only model functions with a single return value. However, as mentioned previously, functions in Go can return multiple values (see Section 2.1.1). This is solved by creating an extra structure bundling together all return values for a given function. This approach is also taken by the previous work gotools, see Section 2.4.2. When there is only one return value, no return structure is created. We use this approach whenever the signature of a function is assigned or when a function pointer type with an associated signature is created.

ABIInternal also specifies that the register XMM15 always has the value zero. This is to quickly be able to zero out large memory areas. We therefore set a register value assumption in Ghidra to always assume that XMM15 has the value zero. This helps the decompiler in functions where XMM15 is used without first being explicitly set.

### 3.2 Duff functions

As described in Section 2.4.2, runtime.duffcopy and runtime.duffzero are compiler-generated functions for quickly copying and zeroing memory. Each function's assembly code is hardcoded for each supported architecture\(^1\).

We therefore locate the functions using a linear search for the specific byte sequence corresponding to the machine code over the code segment of the binary. The approach is architecture-specific, but can easily be expanded to all of Go's supported architectures.

An example of the code of runtime.duffcopy can be seen in Listing 3.2. It consists of 64 repeating segments of copying code. Depending on how much is to be copied, functions call into the runtime.duffcopy function at different offsets. Therefore, once the Duff function is located, we define one function at each offset that a caller might use. Each function will zero out or copy a constant amount of bytes, which is included in the function name (example: duffcopy.960).

Both runtime.duffcopy and runtime.duffzero only use a few specific registers and take arguments in different registers than what is demanded by ABIInternal. Because of this, we add two extra calling conventions to the .cspec file, one for each function. When functions are

\(^1\)https://github.com/golang/go/blob/go1.19.5/src/runtime/mkduff.go
Listing 3.2: Assembly code of runtime.duffcopy on x86-64. Callers will enter at different offset depending on how much is to be copied.

```assembly
1; runtime.duffcopy
2  MOVUPS  XMM0, xmmword  ptr  [RSI] ; Potential entry point
3  ADD     RSI, 0x10
4  MOVUPS  xmmword  ptr  [RDI], XMM0
5  ADD     RDI, 0x10
6 ; ...  Repeats 64 times. Many potential entry points.
7  MOVUPS  XMM0, xmmword  ptr  [RSI] ; Potential entry point
8  ADD     RSI, 0x10
9  MOVUPS  xmmword  ptr  [RDI], XMM0
10 ADD     RDI, 0x10
11 RET
```

created, they are annotated with the correct calling convention. Figure 3.2 shows decompilations before and after the analysis is run.

The functions are similar to Duff’s devices [7], hence their names. A Duff’s device [7] is a coding pattern in C, whereby combining a while loop and a switch case, manual loop unrolling can be achieved.

### 3.3 Mark non-returning functions

Functions that terminate the program never return to their caller. These include functions that panic or simply exit. For Ghidra to be able to correctly model the behavior of the caller, it must know which functions are non-returning and which are not. Figure 3.3 shows the same piece of code, before and after runtime.gopanic is marked as non-returning. In the before example, it can be seen that the stack pointer RSP is assumed to be a variable called register0x00000020. Just before each function call, the return address is manually assigned to an offset from the register variable. This effect is baked into the CALL instruction and usually does not show up in the decompilation. After the panic function has been marked as non-returning, the noise from the assumed register variable is cleaned up.

While Ghidra does perform analysis to determine if a function is non-returning or not, this does not always work. Therefore, since function names have been restored by the function renamer (see Section 3.5), we mark a set of known non-returning functions as such. The list of functions is found in Appendix C.
Go source code

```go
//go:noinline
func makebytes(b byte) *[128]byte {  
    var buf [128]byte  
    for i := 0; i < 128; i++ {  
        buf[i] = b  
    }  
    return &buf  
}
func main() {  
    ptr := makebytes('A')  
    buf := *ptr // Makes a copy  
    os.Stdout.Write(buf[:])
}
```

Before

```c
void main.main() {  
    undefined local_88 [128];  
    main.makebytes(0x41);  
    FUN_0045a770();  
    os.(File).Write(DAT_004da530,local_88,0x80,0x80);  
    return;
}
```

After

```c
void main.main() {  
    void *src;  
    undefined local_88 [128];  
    src = (void *)main.makebytes(0x41);  
    duffcopy.128(local_88,src);  
    os.(File).Write(DAT_004da530,local_88,0x80,0x80);  
    return;
}
```

Figure 3.2: Example of decompilation before and after the Duff function annotator is run. Note how FUN_0045a770 has been replaced with duffcopy.128 and the surrounding types have changed.
Before

```c
void main.main(runtime._type *g) {
  undefined8 uVar1;
  undefined8 uVar2;
  undefined *puVar3;
  undefined *unaff_RBP;
  do {
    puVar3 = (undefined *)register0x00000020;
    if (*(undefined **)&g->hash <= register0x00000020 && *(undefined *)-register0x00000020 != *(undefined **)&g->hash) {
      puVar3 = (undefined *)((long)register0x00000020 + -0x20);
      *(undefined **)((long)register0x00000020 + -8) = unaff_RBP;
      unaff_RBP = (undefined *)((long)register0x00000020 + -8);
      if (1 < _DAT_004da8d8) {
        uVar1 = *(undefined8 *)(DAT_004da8d0 + 0x10);
        uVar2 = *(undefined8 *)(DAT_004da8d0 + 0x18);
        *(undefined8 *)(long)register0x00000020 + -0x28) = 0x461959;
      }
    }
    *(undefined8 *)(long)register0x00000020 + -0x28) = 0x461976;
  } while (true);
  runtime.morestack_noctxt();
  register0x00000020 = (BADSPACEBASE *)puVar3;
}
```

After

```c
while (&stack0x00000000 < *(undefined **)&g->hash || &stack0x00000000 == *(undefined **)&g->hash) {
  runtime.morestack_noctxt();
}
if (1 < _DAT_004da8d8) {
  os.(*File).WriteString(g,DAT_004da530,*(undefined8 *)(DAT_004da8d0 + 0x10), *(undefined8 *)(DAT_004da8d0 + 0x18));
  return;
}
/* WARNING: Subroutine does not return */
runtime.gopanic(g,&Lstring,&PTR_DAT_0048bcb8);
```

Figure 3.3: The same code snippet, before and after marking `runtime.gopanic` as a non-returning function. The corresponding source code is found in Appendix B.1.
Listing 3.3: Example of stack growth prologue present in most Go functions.

```assembly
; Entrypoint of function foo
foo:
    CMP   RSP, qword ptr [R14 + 0x10]
    JBE   foo_allocate_more_stack
; Function body
; ...
foo_allocate_more_stack:
    ; Spill register variables to stack
    CALL  runtime.morestack_noctxt
    ; Restore spilled variables to registers
    JMP   foo
```

### 3.4 Patch out stack growth prologue

At the start of every function that uses the stack, a check is made to see if the stack needs to be expanded. This is needed because all goroutine’s stacks live as memory allocations outside the normal stack. Even the main thread in a Go executable is also a goroutine and performs these checks. The assembly code of the check can be seen in Listing 3.3. The check is performed by comparing the stackguard0 field of the current goroutine structure, stored at offset 0x10 from R14, with the current stack pointer value (RSP). If more stack space is needed, runtime.morestack_noctxt is called and then the function starts over.

This prologue is code that was generated by the compiler and was not typed by the programmer, therefore it can safely be removed to reduce noise for the reverse engineer to filter out. We remove the prologue by replacing the initial CMP and JBE instructions with NOP instructions. This replacement does not affect the executable file on disk, only the representation of the program that is seen by Ghidra.

Apart from removing the usually three line long while-loop at the start of every function (see After in Figure 3.3 for an example of the loop), removing the prologue can also simplify the tracking of variables through the function. Listings 3.4 and 3.5 show the same code, before and after the prologue has been removed. In the before example, param2 is split into the variables uVar{1,3–5} at the start of the functions. These variables are spilled to the stack and later restored around the call to runtime.morestack_noctxt. Finally, when calling image.NewRGBA, the variables are concatenated together to recreate the original param2. In the after example, no bookkeeping is needed around the call to runtime.morestack_noctxt.
and param2 is simply passed directly to image.NewRGBA.

### 3.5 Rename and create functions

The go-re.tk library extracts the names and entry points of all functions present in the program from metadata in the binary. This is fed to our analyzer in Ghidra which defines any missing functions and renames functions already present. The fact that functions are correctly named is used by other analyses (see Sections 3.3, 3.7 and 3.9).

### 3.6 Creating types in Ghidra

Recovering all types described by _type structures (see Section 2.2.5) in the binary is done in three steps. First, the go-re.tk library extracts the types pointed to by typelinks. Second, the data is transformed by goretk_util from Go-specific types to C-like structures. Third, the C types are transferred to a Ghidra analyzer which defines the structures in Ghidra's type library using their concrete memory layout.

Figure 3.4 shows the conversion done by goretk_util from the information that is extractable from _type structures to C structures. Simple primitive types such as integers and floating point numbers are converted straightforwardly. Strings in Go, unlike in C, consist of both a pointer and a length and are therefore represented with a structure.

Polymorphic types, such as slices, maps, and channels, are implemented in Go as general structures containing untyped pointers (unsafe.Pointer or uintptr) to keys and elements. Each _type structure describing such a polymorphic type also describes its keys and elements. Therefore, we generate concrete versions of the general structures for each encountered type parameter. For example, upon seeing that the Join method of JoinSplit in Figure 3.4 takes a slice ([]T) with the element type string, the string_slice structure is generated. Instead of its first field being a generic void pointer, it is a gostring pointer.

Interfaces (see Section 2.1.2) are at runtime divided into two subtypes: ifaces and efaces. An iface is used when the underlying concrete type must implement a set of methods. Therefore, an iface contains a pointer to a virtual method table: an array of function pointers referring to the corresponding methods, called an interface table or an itab in Go's terminology. The iface also contains a pointer to the concrete
Listing 3.4: Before removing the stack growth prologue. The binary can be found in Appendix B.2.

```c
undefined [24] main.CreateImage(runtime.g *g,image.←
    Rectangle param_2) {
    undefined8 uVar1;
    image.RGBA *piVar2;
    undefined8 uVar3;
    undefined8 uVar4;
    undefined8 uVar5;
    undefined8 in_stack_00000008;
    undefined8 in_stack_00000010;
    undefined8 in_stack_00000018;
    undefined8 in_stack_00000020;
    error_iface local_30;
    undefined local_20 [16];
    code **local_10;
    uVar5 = SUB328((undefined [32])param_2,0x18);
    uVar3 = SUB328((undefined [32])param_2,0x10);
    uVar4 = SUB328((undefined [32])param_2,8);
    uVar1 = SUB328((undefined [32])param_2,0);
    while (&stack0x00000000 < (undefined *)g->stackguard0 ||
        &stack0x00000000 == (undefined *)g->stackguard0) {
        in_stack_00000008 = uVar1;
        in_stack_00000010 = uVar4;
        in_stack_00000018 = uVar3;
        in_stack_00000020 = uVar5;
        runtime.morestack_noctxt(g);
        uVar1 = in_stack_00000008;
        uVar3 = in_stack_00000018;
        uVar4 = in_stack_00000010;
        uVar5 = in_stack_00000020;
    }
    local_10 = (code **)0x0;
    local_30 = (error_iface)0x0;
    local_30 = errors.New(g,(gostring)CONCAT88(0x18,0x5c07dd)
        ←
        CONCAT88(uVar5,uVar3),CONCAT88(uVar4,uVar1)))
    local_20 = CONCAT88(&local_30,0x56a040);
    local_10 = (code **)local_20;
    piVar2 = image.NewRGBA(g,(image.Rectangle)CONCAT1616(
        CONCAT88(uVar5,uVar3),CONCAT88(uVar4,uVar1))
    ;
    (**local_10)();
    return CONCAT168((undefined [16])local_30,piVar2);
    }
```

Go type information from _type structures

```go
type DataHolder struct {
    // int is 32 or 64 bit
    // depending on architecture.
    A int
    B int32
    C float64
    D string
}
type JoinSplit interface {
    Join([]string) string
    Split(string) []string
}
```

Corresponding C types

```
typedef struct {
    char *str;
    long len;
} gostring;
typedef struct {
    long A;
    int B;
    double C;
    gostring D;
} DataHolder;
typedef struct {
    gostring *data;
    long len;
    long cap;
} string_slice;
typedef struct {
    struct runtime_interfacetype *inter;
    struct runtime__type *_type;
    unsigned int hash;
    unsigned char _[4]; // Padding
    gostring (*Join)(string_slice);
    string_slice (*Split)(gostring);
} JoinSplit_itab;
typedef struct {
    JoinSplit_itab *tab;
    void *data;
} JoinSplit_iface;
```

Figure 3.4: Example of the conversion from Go-specific types to C-like structures performed by goretk_util. The data on the left is extractable from _type structures and the data on the right is what Ghidra needs to define its types.
Listing 3.5: After removing the stack growth prologue. The binary can be found in Appendix B.2.

```go
undefined [24] main.CreateImage(runtime.g *g, image.←
                  Rectangle param_2) {
    image.RGBA *piVar1;
    error_iface local_30;
    undefined local_20 [16];
    code **local_10;
    local_10 = (code **)0x0;
    local_30 = (error_iface)0x0;
    local_30 = errors.New(g, (gostring)CONCAT88(0x18,0x5c07dd)←
                      );
    local_20 = CONCAT88(&local_30,0x56a040);
    local_10 = (code **)local_20;
    piVar1 = image.NewRGBA(g, param_2);
    (**local_10)();
    return CONCAT168((undefined [16])local_30,piVar1);
}
```

value which implements the methods. Ghidra's type system can represent function pointers with specified parameters and result types. Therefore, when a _type structure describing an interface is encountered, a virtual method table is generated containing function pointers with correct signatures. Figure 3.4 shows an example of converting the interface JoinSplit to its corresponding iface and itab.

An iface is on the other hand used to represent the empty interface type (interface{}) which can be fulfilled by any type without any requirements on method implementation. An iface consists of a pointer to a _type structure as well as a pointer to the concrete value. We represent the iface as a single general structure with a void pointer for the concrete value.

We use recursion to convert from Go-specific types to C structures in goretk_util. Because both structures and interfaces can be self-referential, care is taken to avoid infinite recursion. Listing 3.6 shows an example of self-referential types.

Since Ghidra uses the concrete memory layout of structures, a field is defined as starting at a particular memory offset from the start of its structure. To calculate the offset of a field, the size of all previous fields must be known. Therefore, the size of a structure might be needed before it is fully defined. An example can be seen in Listing 3.7. If the recursive algorithm starts by defining Node, it will end up trying to define Tree which embeds a field
Listing 3.6: Example of self-referential structures and interfaces.

```go
// Linked list as a struct
type LinkedStruct struct {
    Next  *LinkedStruct
    Value int
}

// Linked list as an interface
type LinkedInterface interface {
    Next() LinkedInterface
    Value() int
}
```

Listing 3.7: Example showing that structures' sizes need to be known before defining their fields.

```go
// Recursively define all structures, starting with Node.
type Node struct {
    Tree   *Tree
    Parent *Node
    Children []*Node
    Value   int
}

type Tree struct {
    // To embed Root in Tree, the size of Node needs to be known.
    Root    Node
    TreeMetric int
}
```

called Root of type Node. Without knowing the size of the Node structure, it is not known at which offset in Tree that TreeMetric should be placed. We solve this using a separate recursion which calculates the size of a structure by recursively summing the size of its fields. That recursion can never be infinite since structures cannot embed themselves, even transitively.

Recovering data types and defining them in Ghidra does not in itself improve the decompilation since they are never used. This does however make the types available to be applied, either manually by a reverse engineer, or automatically. Section 3.9 describes an analysis that automatically applies the recovered types.
3.7 Library signature importer

Given that the names of all functions and which packages they belong to have been restored (see Section 3.5), it is simple to identify which functions belong to the standard library, third-party packages, and user-written code. It is then possible to automatically restore signatures of library functions present in the code. We do this by parsing the corresponding Go source code, defining any missing data types in Ghidra (similarly as described in Section 3.6), and applying the signatures. A Go parser is not implemented from scratch, instead, we use the one present in the standard library\(^1\).

The version of the parsed source code must be as close as possible to the code that was compiled. The further apart the versions drift, the more incorrect the applied signatures become. The version of the standard library is the same as the version of the language, which is also embedded as a string in the binary. By default, there is also version information present for the third-party packages used as part of file paths. If the information is obfuscated or removed, versions can be inferred from several sources. For example, using which set of functions exist in the binary. Currently, the extension can automatically parse, import, and apply signatures from a user-chosen local directory. Automatically identifying and acquiring source code to parse is left as future work.

Figure 3.5 shows an example before and after the library signature importer is run. Notice in the after example how the signatures of `OpenFile`, `Stat`, and `ReadAt` have been enhanced. Many local variables also have more specific types.

3.8 Structure representation

This section covers the issue of representing structures in Ghidra in a way that is compatible with both of `ABIInternal`'s requirements for alignment and the way that values should be passed in registers. First, we describe the two conflicting requirements and then we suggest two approaches for structure representation, each with its drawbacks.

First, `ABIInternal` requires that fields in structures be aligned. Meaning that their position must be a multiple of their alignment value. For all built-in types, except for `complex128`, the type's alignment is the same as its size. An `int32` must therefore be positioned at any offset \(4n\) and an `int64`\(^1\)

\(^1\)https://pkg.go.dev/go/parser
Before

```c
undefined main.checkinfection(undefined8 param_2) {
    long lVar1;
    long *plVar2;
    undefined auVar3 [16];
    undefined auVar4 [16];
    // ...
    auVar3 = os.OpenFile();
    // ...
    auVar4 = os.(*File).Stat();
    lVar1 = (**(code **) (SUB168(auVar4, 0) + 0x38)) (SUB168(auVar4 >> 0x40, 0));
    plVar2 = (long*)runtime.makeslice(&Lbyte, 0xc, 0xc);
    os.(*File).ReadAt(
        SUB168(auVar3, 0),
        plVar2,
        0xc,
        0xc,
        lVar1 + -0xc);
    // ...
}
```

After

```c
undefined main.checkinfection(undefined8 param_2) {
    os.File *f;
    long lVar1;
    long *plVar2;
    undefined8 in_RBX;
    os.OpenFile_ret oVar3;
    os.(*File).Stat_ret oVar4;
    // ...
    oVar3 = os.OpenFile(
        (gostring) CONCAT88(in_RBX, param_2),
        0,
        (undefined *) 0x0);
    f = SUB248((undefined [24]) oVar3, 0);
    // ...
    oVar4 = os.(*File).Stat(f);
    lVar1 = (**(code **) (SUB248((undefined [24]) oVar4, 0) + -0x38)) (SUB248((undefined [24]) oVar4, 8));
    plVar2 = (long*) runtime.makeslice(&Lbyte, 0xc, 0xc);
    os.(*File).ReadAt(
        f,
        (byte_slice) CONCAT816(0xc, CONCAT88(0xc, plVar2)),
        lVar1 + -0xc);
    // ...
}
```

Figure 3.5: Before and after running the library signature importer. The binary can be found in Appendix B.2.
Since the type `int64` needs to be aligned to an 8-byte boundary, 4 padding bytes are placed between `Small2` and `Large`.

```go
type Counters struct {
    Small1 int16
    Small2 int16
    // 4 bytes alignment padding
    Large int64
}
```

**Approach 1:** with padding

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small1</td>
<td>0</td>
<td>2</td>
<td>AX</td>
</tr>
<tr>
<td>Small2</td>
<td>2</td>
<td>2</td>
<td>BX</td>
</tr>
<tr>
<td>Alignment padding</td>
<td>4</td>
<td>4</td>
<td>???</td>
</tr>
<tr>
<td>Large</td>
<td>8</td>
<td>8</td>
<td>RCX</td>
</tr>
</tbody>
</table>

**Approach 2:** no padding

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small1</td>
<td>0</td>
<td>2</td>
<td>AX</td>
</tr>
<tr>
<td>Small2</td>
<td>2</td>
<td>2</td>
<td>BX</td>
</tr>
<tr>
<td>Large</td>
<td>6</td>
<td>8</td>
<td>RCX</td>
</tr>
</tbody>
</table>

Figure 3.6: Example of alignment padding in structures and our two approaches for structure representation.

at any offset $8n$. When the next free position in a structure is not a multiple of the alignment required to place the next field, unused padding bytes are added until the position is aligned. In Figure 3.6, the structure `Counters` needs alignment padding between fields `Small2` and `Large`.

Second, when structures are passed by value, their fields are placed in registers, according to ABIInternal. In particular, integer and boolean types that fit in a register are given that entire register, even if the type requires less space. The register assignment of `Counters`, defined in Figure 3.6, is the following: field `Small1` in `RAX`, field `Small2` in `RBX`, and field `Large` in `RCX`. For an explanation of x86-64's registers, see Appendix D. When structures are passed in registers, any alignment padding needed when placing the structure in memory is ignored.

We suggest two approaches for representing structures that require alignment. In the first approach, the alignment padding is part of the structure. In the second approach, the padding is not present in the structure. Figure 3.6 shows the memory layout of `Counters` using the two approaches.

Consider attempting to assign how `Counters` should be stored in registers when passed as an argument under Approach 1. This requires using the more flexible Custom Storage feature of Ghidra instead of its `.cspec` files, see Section 3.1 for more details. When assigning storage, the total size of the storage must equal the size of the variable. First, field `Small1` must be assigned storage. Assigning it to `AX`, the lowest 2 bytes of `RAX`, will give it the right size and put it in the right register as required by ABIInternal. Similarly, field `Small2` could be assigned to `BX`, and field `Large` to `RCX`. 
Go source code

type S struct {
    A uint8
    // 7 bytes of alignment padding
    B uint64
}

func f() *S {
    return &S{A: 3, B: 5}
}

Approach 1: with alignment padding

main.S * main.f() {
    pmVar1 = runtime.newobject(&main.S);
    pmVar1->A = 3;
    pmVar1->B = 5;
    return pmVar1;
}

Approach 2: no alignment padding

main.S * main.f() {
    pmVar1 = runtime.newobject(&main.S);
    pmVar1->A = 3;
    *(undefined8 *)(((long)&pmVar1->B + 7)) = 5;
    return pmVar1;
}

Figure 3.7: Example of how accessing fields of structures by pointer differs between Approach 1 and Approach 2. The left listing shows the original source code containing a structure S which requires alignment padding between A and B. The two right listings show decompiled versions of the program, using the two different approaches for representing structures.

The padding bytes must also be assigned storage. However, in practice, the padding is not stored explicitly in any register or other location during the call, since its content is semantically irrelevant. It would therefore be desirable to use some sort of "void" or "ignore" storage location for the padding. Unfortunately, no such location exists. Trying to use some other concrete existing storage location could make variables have data flow dependencies on each other that do not exist.

Now consider what happens when using Approach 2. There are no problems passing the structure in registers since the padding does not exist. Problems do arise however when the reading or writing from the structure when placed in memory. Since the offsets of all fields after the missing padding are wrong, accesses will look like they are referring to the wrong field or a nonsense field. An example of this is shown in Figure 3.7.

Our extension uses Approach 2 since the negatives of Approach 1 are deemed as more severe. It should be noted that the issues of both approaches only occur when alignment padding is needed. When it is not, both approaches...
3.9 Polymorphic functions

Go's runtime contains several functions which can operate on multiple different types. These are often used for allocation, conversion between different types, and interaction with polymorphic data types, such as maps. These functions often take a pointer to a _type structure as an argument to be able to determine the types to operate on. The signatures of some example functions are explained in Listing 3.8.

Knowing the names of all functions (see Section 3.5), and having parsed and defined types for all _type structures (see Section 3.6), it is possible to determine the concrete data types that the polymorphic functions operate on, at any given call site. Most often, the _type pointer argument is constant and refers to a globally allocated _type structure present in the typelinks. Our analysis therefore looks at all the call sites for a specific polymorphic function, looks up the corresponding type based on the constant _type pointer argument, and then adds a call signature override with the

Listing 3.8: Examples of polymorphic functions in the runtime package.

```
// newobject allocates an object on the heap. The returned
// pointer points to a typ.
func newobject(typ *_type) unsafe.Pointer

// convT allocates a new copy of v on the heap. Both the
// return value and v point to a typ.
func convT(t *_type, v unsafe.Pointer) unsafe.Pointer

// growslice increases the capacity of a slice to the
// requested cap. The element type of both old and the
// returned value is et.
func growslice(et *_type, old slice, cap int) slice

// mapaccess1 fetches the value associated with key in map
// h. The type of h, key and the return value is determined
// by t. Note that maphype embeds a _type.
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) <-
    unsafe.Pointer
```

are equivalent. We have reported the issues described in this section to Ghidra's issue tracker\(^1\).

\(^1\)https://github.com/NationalSecurityAgency/ghidra/issues/51
new argument and return types. Call signature overrides let the user change a function's signature, just for a specific call site. In the few cases where the _type pointer is not constant, the analysis does nothing. Figure 3.8 shows an example of running the analysis.

Performing the analysis, more specific types can be inferred by the decompiler, improving decompilation. This analysis is particularly strong because it allows automatically applying user-created types. A list of all analyzed runtime functions can be seen in Appendix E. Since the new enhanced signature can depend on the pointer argument in complex ways (see Listing 3.8), adding a new function to the analysis requires adding a new source code file and recompiling the extension.
Go source code

```go
//go:noinline
func createMap() map[string]Point {
    return make(map[string]Point)
}
func main() {
    m := createMap()
    m["somewhere"] = Point{X: 5, Y: 7}
}
```

Before

```c
void main.main() {
    runtime.hmap *h;
    undefined8 *puVar1;
    h = (runtime.hmap *)main.createMap();
    puVar1 = (undefined8 *)runtime.mapassign_faststr(
        (runtime.maptype *)&Lhmap.string_to_main.Point,
        h,
        (gostring)CONCAT88(9,0x466281));
    *puVar1 = 5;
    puVar1[1] = 7;
    return;
}
```

After

```c
void main.main() {
    hmap.string_to_main.Point *h;
    main.Point *pmVar1;
    h = (hmap.string_to_main.Point*)main.createMap();
    pmVar1 = runtime.mapassign_faststr(
        &Lhmap.string_to_main.Point,
        h,
        (gostring)CONCAT88(9,0x466281));
    pmVar1->X = 5;
    pmVar1->Y = 7;
    return;
}
```

Figure 3.8: Decompiled code before and after the polymorphic function annotator is run. Note how the arguments and return value of `mapassign_faststr` change.
Chapter 4

Evaluation and Results

The goal of the thesis is to figure out how much the readability of the decompilation of Go programs can be improved (see Section 1.1). This Chapter quantitatively evaluates the readability of the produced decompilations of our implemented extension in comparison with various previous work.

Figure 4.1 shows the evaluation plan. Our extended version of Ghidra 10.2.3 is evaluated against several other candidates: two unextended versions of Ghidra (10.2.3 and 10.3), Ghidra 10.2.3 extended with 3 different previously existing plugins, as well as an unextended version of IDA. For more reasoning about why these test candidates are chosen and more information about their setups, see Section 4.1.

Each evaluation candidate decompiles several test case binaries. The binaries are a mix of malware samples and open-source programs. Section 4.2 describes the selection process for the test cases.

The resulting decompilations are compared using multiple metrics such as number of lines, number of casts, etc. The metrics are measured by first parsing the decompilations using a C parser and then calculating the metrics on the Abstract Syntax Tree (AST). Section 4.3 details the reasoning for picking the metrics and Section 4.4 describes how and why the C parser is used. Finally, Section 4.5 shows the results of the measurements.

4.1 Evaluation candidates

Out of the previously existing extensions mentioned in Table 2.1, our extension is compared against:

- Mooncat
These extensions are chosen for comparison because they all, except for vanilla Ghidra 10.2.3, attempt to represent \texttt{ABIInternal}. The other extensions mentioned in Table 2.1 do help with making reverse engineering Go binaries easier by for example restoring function names, finding and annotating strings using heuristics, and recovering structures. They however do not make any improvements that could directly simplify the decompilation. Therefore they are excluded from the evaluation.

Our extension, Mooncat, Monoidic, and Cyberkaida all extend version 10.2.3 of Ghidra. Therefore, we also compare against an unextended Ghidra 10.2.3, to see the difference the extensions make. Since an unextended Ghidra 10.2.3 does not recover function names, we run our function renamer after its initial analysis.

Our library signature importer (Section 3.7) and Monoidic essentially perform the same analysis, both relying on being provided the source code of the external library to apply in Ghidra. While both extensions can be used to
import any libraries' signatures, here we only import the standard library. Due to time limitations, we always import version 1.19.5 of the standard library instead of picking the version based on the investigated binary.

Monoidic requires that function names are present before its analysis can run and does not implement function name recovery itself. Therefore we first run our function renamer before their extension.

Cyberkaida consists of three components: a .csp file, a string analyzer, and a function analyzer. The .csp file adds support for ABIInternal. The string analyzer searches for references from the code to memory that look like Go's string structures and if found, defines them. Unfortunately, it also clears the defined assembly listing of many functions, making the decompiler unable to decompile them. Therefore, the string analyzer is disabled. The function analyzer looks for several symbols and uses them to find the metadata that contains every function's entry point. Once the metadata is found, it defines all functions (without names) in Ghidra. Utilizing symbols that are assumed to exist it violates our assumptions about the level of obfuscation (see Section 4.2). Because of these issues, Cyberkaida is only tested with its .csp file enabled, as well as our function renamer.

4.2 Test cases

A mix of malware samples and one open-source project is used for evaluation. All binaries are x86-64 programs since our extension only has support for that architecture. Malware is used since manual malware analysis is the main use case envisioned for the extension. The malware samples are chosen using the following process.

1. A YARA\(^1\) rule looking for runtime.duffcopy's assembly code (see Section 3.2) is matched against all malware samples uploaded to MalwareBazaar\(^2\) using Yara Scan Service\(^3\). The rule can be seen in Listing 4.1 and the scan results are available here: https://riskmitigation.ch/yara-scan/results/73203bb334c67b16fa187ddaf599daf33443c809d8ae4c1458144b660746b41/ The scan was performed on 2023-04-11. This yields a list of 751 samples which are all x86-64 Go binaries.

---
\(^1\)https://virustotal.github.io/yara/
\(^2\)https://bazaar.abuse.ch/
\(^3\)https://riskmitigation.ch/yara-scan/
Listing 4.1: YARA rule matching against the assembly code of *runtime.duffcopy* on x86-64.

```python
rule golang_duffcopy_amd64 {
    strings:
        $duffcopy_amd64 = { 0f 10 06 48 83 c6 10 0f 11 07 48 83 
            c7 10 0f 10 06 48 83 c6 10 0f 11 07 48 83 c7 10 0f 10 
            06 48 83 c6 10 0f 11 07 48 83 c7 10 0f 10 06 48 83 c6 
            10 0f 11 07 48 83 c7 10 0f 10 06 48 83 c6 10 0f 11 07 
            48 83 c7 10 c3 }
    condition:
        $duffcopy_amd64
}
```

2. All binaries with symbols still present are filtered out. Symbols can easily be stripped by malware authors using compiler flags: `go build -ldflags="-s -w"`. Our extension is built on the assumption that symbols have been stripped. This leaves 500 binaries.

3. All binaries with versions earlier than 1.17 are removed since that is when `ABIInternal` was introduced, leaving 192 binaries. The version detection is performed by the go-re.tk library.

4. All programs that are not Portable Executable (PE) files are removed, leaving 159 binaries. Most malware targets Windows and PE is the executable file format for Windows binaries. During development, the extension has also been tested against Executable and Linkable Format (ELF) files, the executable file format on Linux.

5. The remaining binaries are split into groups based on their major version: 44 from 1.17, 66 from 1.18, 43 from 1.19, and 6 from 1.20. One binary is randomly selected from each group. This gives a spread across versions.

   The binary selected from version 1.18 turned out to be strange. None of the extensions can correctly handle the binary. Our extension defined functions at wrong addresses and vanilla Ghidra 10.3 simply threw an exception when attempting to parse the binary. Therefore another 1.18 binary is drawn from the collection as a replacement during evaluation. The originally picked 1.18 binary is listed among the others in Appendix F.

   Most malware samples are small, containing relatively little user-written code. To see how our extension handles larger programs an open source project
called Hugo\(^1\) is also used during evaluation.

Hugo is picked since it is large (over 100k lines of code), has many dependencies, and has many GitHub stars (over 69k). Its large amount of source code yields a binary with a size of 55 MB. Version v0.111.3, which is compiled with Go version 1.20.1, is evaluated.

The full list of evaluation binaries including their SHA256 hashes and download links can be found in Appendix F.

### 4.3 Metrics

Several potential metrics can be used to evaluate the readability of the produced decompilation. Here follows a discussion of which metrics are suitable for the project.

The simplest metric employed is to compare the line counts of the decompiled programs [38, 44]. The line count acts as a proxy for how concise and readable the decompilation is. The idea is that if there are fewer lines to read, manual reverse engineering will be faster. However, if those fewer lines are denser, the improvements might be counterproductive. This metric also does not capture improvements that are local to one line.

Schulte et al. [38] evaluate their decompiler by counting the occurrences of language constructs that reduce readability. Such as casts, *gos, extra variables, scopes, macros, etc.* These extra constructs often appear as glue when a decompiler cannot fully represent the semantics of the disassembly. This metric is more direct than just counting the number of lines. The tricky thing is to pick which constructs should be counted. There is a risk of picking just the constructs that make the metrics look good for the experiment at hand.

During the development of our extension, we observe that many "glue functions" are generated by Ghidra in the decompilation. These are exactly the types of constructs mentioned in the above paragraph. Therefore "glue functions" are included as a metric. The functions generally combine or split up values in unconventional ways. For example, \( \text{SUB42}(x, c) \), which truncates the four byte large value of \( x \), to a two byte value, first removing the \( c \) least significant bytes. Example: \( \text{SUB42}(0xaabbccdd,1) = 0xbbcc \).

Another approach is to count the number of high-level constructs that were recovered. This approach is taken by Fokin et al. [14] when trying to recover class inheritance hierarchies and exception handling in C++ binaries. If the goal of an improvement is to recover a specific type of high-level construct,
then this is a straightforward metric to use. This approach can only be used on programs where the source code is available and therefore cannot be used on malware. This metric is not used since the resulting decompilation is not recompilable.

If the decompilation is good enough that parts of it can be recompiled, then the accuracy of the decompilation can be evaluated by calculating the difference between the recompiled assembly code and the original binary [4]. This approach requires knowing the exact compiler version and the flags used during compilation. This information might be difficult to come by when it comes to malware.

Naeem, Batchelder, and Hendren [30] introduce metrics for measuring the effectiveness of decompilers and obfuscators when it comes to readability. They introduce three metrics: code size, as measured by AST node count, control flow complexity, measuring the number of if-statements and abrupt control flow (break, continue, and goto), and conditional complexity, measuring the complexity of branching conditions. One advantage of measuring code size using AST node count instead of line count is that any code formatting differences are ignored.

Considering the above discussion, we pick the following metrics. For all the metrics, less is better. Figure 4.2 shows a function together with its counted metrics.

- Line count
- AST node count
- Variable count
- Cast count
- Glue function count (SUB, CONCAT, ZEXT, etc.)

4.4 Parsing

One of the desired metrics is AST node count. Both Ghidra and IDA have internal representations of the decompiled C code they generate. One approach is to calculate metrics based on this AST. The problem is that the two ASTs' structures are likely different, which would yield incomparable node counts. Therefore we opt to use an external Python-based C parser called pycparser\(^1\).

\(^1\)https://github.com/eliben/pycparser/
undefined [24] main.CreateImage(runtime.g *unaff_R14,
    undefined8 param_2,undefined8 param_3,undefined8 param_4→,
    undefined8 param_5)
{
    image.RGBA *piVar1;
    error_iface local_30;
    undefined local_20 [16];
    code **local_10;

    local_10 = (code **)0x0;
    local_30 = (error_iface)0x0;
    local_30 = errors.New(unaff_R14, (gostring)CONCAT88(0x18, 0→
      x5c07dd));
    local_20 = CONCAT88(&local_30, 0x56a040);
    local_10 = (code **)local_20;
    piVar1 = image.NewRGBA(unaff_R14, (image.Rectangle)←
      CONCAT1616(CONCAT88(param_5, param_4), CONCAT88(param_3, ←
        param_2)));

    (**local_10)();
    return CONCAT168((undefined [16])local_30, piVar1);  
}

<table>
<thead>
<tr>
<th>Metric</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>16</td>
</tr>
<tr>
<td>AST nodes</td>
<td>127</td>
</tr>
<tr>
<td>Variables</td>
<td>4</td>
</tr>
<tr>
<td>Casts</td>
<td>6</td>
</tr>
<tr>
<td>Glue functions</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 4.2: Example showing a function and how the metrics are counted. The table shows the counted metrics. The binary containing the function can be found in Appendix B.2.
Another reason for using a parser is because using simple string matching or regex to match complex constructs such as casts or variable declarations is error-prone. It is likely to miss certain syntactic variations when writing the matching rules, yielding incorrect measurements.

Before parsing, all C code and type definitions are exported from Ghidra and IDA. The type definitions are needed to distinguish identifiers from being variable or function names versus type names. To be able to correctly parse the exported C decompilations, a few modifications are made to Mooncat and Monoidic. For both extensions, when C keywords are used as identifiers, for example as field or parameter names, they are made safe for parsing by prepending an underscore. For Mooncat, the string type is renamed from `string` to `gostring` to not collide with Ghidra's built-in string type and function names are deduplicated.

The exported code cannot be parsed directly since it contains several errors, therefore a preprocessing phase is first run. Some types and macros are missing and need to be defined. IDA exports code written in a C++ fashion. Therefore extra typedefs must be generated so that structs can be referred to using only their names and not requiring the `struct` keyword as a prefix. Many more small syntactic and semantic issues in both Ghidra's and IDA's decompilations are cleaned up before parsing.

### 4.5 Results

All metrics are measured per function. Therefore the results show how the decompilations change on a per-function basis. Table 4.1 shows how many functions were decompiled without errors by each candidate on all test cases. It also shows how many of the functions were in common across candidates based on the entry points of the functions.

Notice how IDA manages to decompile much fewer functions than the candidates based on Ghidra. Listing 4.2 shows how errors occur when IDA decompiles functions from the `go1.17.8.exe` binary. In total, IDA seems to have detected 5840 functions, out of which only 2479 are queued for decompilation, and 1568 fail to decompile, leaving the 911 successfully decompiled functions mentioned in Table 4.1. The two different errors that occur over all test case decompilations are "call analysis failed" and "stack frame is too big".

Because of the issues with IDA's decompilation, IDA is excluded from the following measurements and results. Keeping IDA would mean only comparing the few functions that IDA manages to decompile. Since IDA...
Table 4.1: The table shows how many functions were decompiled without errors by all evaluation candidates on each test case. It also shows how many functions were in common among the candidates, based on the entry points of the functions. Since IDA failed to decompile many functions the, in-common total is presented including and excluding IDA.

Listing 4.2: Example showing when IDA fails to decompile functions, taken from the decompilation of go1.17.8.exe. The top error is a function which failed decompilation and the bottom error is the summary at the end of the decompilation.

```plaintext
// [More above...]
// ---- (00000000006903C0) --------------------------
#error "690420: call analysis failed (funcsize=25)"

// nfuncs=5840 queued=2479 decompiled=2479 lumina nreq=0
// worse=0 better=0
#error "There were 1568 decompilation failure(s) on 2479 <-> function(s)"
```
fails at decompiling the majority of functions, the subset that it succeeds with decompiling is likely not representative of all functions.

4.5.1 Graphs

Figure 4.3 shows our two different types of graphs. The first graph compares the evaluation candidates and the second graph compares the different phases of our extension. These two specific graphs look at the AST node count on the test binary `go1.17.8.exe`. Both graph types show one box plot per candidate or phase describing the distribution of the metric, in this case, node count, per function, over all in-common functions. For the `go1.17.8.exe` test binary, those are the 2965 (first graph) and 3047 (second graph) functions mentioned in Table 4.1.

The graphs show $Q_2$ (the second quartile / median) as the orange center line of the box. The first and third quartiles, $Q_1$ and $Q_3$, are the left and right edges of the box. The right whisker ends at the largest value within $[Q_3, Q_3 + 1.5IQR]$ where $IQR = Q_3 - Q_1$. Similarly, the left whisker ends at the smallest value in the left interval. In the graphs comparing different candidates, the box plots are sorted by the tuple $(Q_3, Q_2, Q_1)$. In the phase comparison graphs, the phases are sorted in the order they are applied.

For a few functions, the measured values are extreme. These would be plotted using so-called outliers in the box plot: markings outside the whiskers. By including the outliers, the graph is zoomed out to the extent that it is not possible to get an overview of the bulk of the distribution. Therefore, we choose to exclude the outliers from the graphs. The maximum and average values of each metric can however be viewed in the tables in Appendix G.3.

4.5.2 Phases of extension

The phases of our extension are generally self-consistent, behaving in the same way across the evaluation binaries. Figures 4.4 and 4.5 show how the different phases perform when measuring casts in `go1.17.8.exe` and variables in `hugo.exe`. These two graphs are representative of the graphs with the most difference between phases and the least difference, respectively. Appendix G.1 contains graphs comparing the phases across all metrics and test binaries. Below, we discuss how the different phases affect the metrics. The phase called "initial" is measured after the binary has been imported and Ghidra's auto-analysis has been run.
Figure 4.3: Example of two different graph types. The first compares evaluation candidates and the second compares phases of our extension.
Function renamer

In general, the function renamer does not affect any metrics since its only purpose is to define and name all functions retrieved from the metadata. Sometimes it might affect the metrics a little bit since it is also responsible for adding the register assumption that XMM15 is always zero (Section 3.1). This can be seen in for example Figure G.19. When the renamer is used for comparability as part of other candidates (see Section 4.1), the register assumption is turned off.

Non-returning function annotator

As shown by the examples in Section 3.3, when a non-returning function is not marked as such, its callers will contain a lot of nonsense code. Therefore, the non-returning function annotator has the effect of reducing all metrics.

The annotator seems to affect the malware samples a lot more than the hugo binary. This can be seen by for example comparing Figures G.1 to G.4 which looks at malware samples with Figure G.5 which looks at hugo. The malware binaries are much smaller than the hugo binary, making the runtime
Figure 4.5: Compares phases of our extension when measuring the number of variables in the test binary hugo.exe.

and any standard library functions used make up a large share of the total code. Among these functions, there are many more calls to non-returning functions than in user-written code or dependencies. This explains why the effect on the malware binaries is much greater.

**Stack growth prologue**

The stack growth prologue analyzer removes at least three lines from every function using the stack, and often more. This has the effect of reducing all metrics. Figures G.1, G.6, G.11 and G.16 show the effect.

**Duff's device annotator**

The Duff function analyzer does not do much that is visible in the metrics. Even though Duff functions might be called often, their footprint in the decompilation is usually very small: a single line with a function call. Adding this analysis offers more correctness by allowing variable flows to be correct and it gives a hint to the reverse engineer of how many bytes are copied or zeroed. Casts increase slightly after this analysis, which can be seen in
Figures G.11 to G.15. This is probably due to incompatible types between the arguments passed to the functions and their newly assigned signatures.

**Data type recovery**

The data type recovery phase does not affect the metrics. This is expected since it only recovers type definitions from the binary metadata and creates the types in Ghidra, but does not apply them anywhere in the decompilation.

**Library signature importer**

The library signature importer phase adds a lot of information to the decompilation. All standard library and runtime functions get assigned their correct signatures with types. This type information propagates from return values and parameters to local variables inside functions.

Structures returned from functions are placed in registers. For some reason Ghidra displays field accesses to these variables using the `SUB` glue function and a cast. Similarly, when passing arguments to a function expecting a structure, it is common to see Ghidra use the `CONCAT` function to glue together several values, cast the result to the parameter type, and pass it to the function. Therefore, this phase adds many casts and calls to glue functions. This can be seen in Figures G.12 and G.22.

Another reason for the increase in casts is that many new types are introduced to the decompilation, while the old basic types are also present to a large extent. This results in casts appearing everywhere that the variables with new and old types interact. As a consequence of the added glue functions and casts, the number of AST nodes increases a lot and lines increase a little. Figures G.4 and G.9 exemplify this.

The variables metric is unaffected or increases just slightly (see Figures G.16 to G.20). This is unexpected since adding more type information should allow coalescing many smaller variables into fewer large ones. However, this does not seem to happen in practice. Perhaps Ghidra is already able to determine the general size of variables based on their use and therefore the added type information does not cause variables to merge. Or perhaps it is again due to the poor handling of structures in registers, leading to the use of glue functions, which might stop variable merging.

Just like with the non-returning function annotator, this analyzer has a larger effect on the malware binaries than on hugo since they consist mainly of runtime and standard library code and not so much user-written code. Hugo is not as affected since it contains a majority of user-written and third-party
library code. However, the signature importer could easily be used to import third-party library code (see Section 3.7), which would likely have a similarly large effect on the metrics.

**Polymorphic analyzer**

The polymorphic function analyzer does not do much that is visible in the metrics. This is probably because so few polymorphic functions are analyzed, due to time limitations. Appendix E lists the functions we analyze.

### 4.5.3 Comparing candidates

Appendix G.2 contains graphs comparing the evaluation candidates, on all metrics, over all test binaries. This section discusses these results.

For line, node, and cast count, our extension performs well on the malware binaries in comparison with the other candidates. Figure 4.6 shows an example of this. For more graphs that exemplify this, see Figures G.26 to G.29, G.31 to G.34 and G.36 to G.39. On all malware binaries, our extension produces the least or the second to least amount of the measured metric. The other contenders for first place are Ghidra 10.2.3 and sometimes Ghidra 10.3. On hugo.exe, our extension performs significantly worse, coming in at fourth place, for all three of the metrics. See Figure 4.7 for an example. Figures G.30, G.35 and G.40 show this in the appendix.

Both line and node count generally measure the "amount of code" present. As mentioned in Section 4.5.2, our marking of non-returning functions decreases the amount of code, but is a lot more effective on binaries with a large share of runtime and standard library code, as is the case with the malware.

Cyberkaida consistently performs the worst on the metrics line, node, and cast count, over all test binaries.

Figure 4.8 shows a graph of the variable count metric. Ghidra 10.2.3 produces the fewest variables, over all binaries. Ghidra 10.3, on the other hand, produces the most variables on all binaries, except for go1.18.3.exe. Our extension performs poorly, coming in at 4th and 5th place of fewest variables produced, only placing 2nd on go1.20.2.exe. As mentioned when discussing the library signature importer phase in Section 4.5.2, the variables metric was not affected as expected. Our increase of information does not help Ghidra coalesce variables. See Section 5.7.1 for a discussion on the suitability of the variable count metric for measuring readability.
Figure 4.6: Compares candidates when measuring lines in go1.17.8.exe.

Figure 4.7: Compares candidates when measuring lines in hugo.exe.
Figure 4.8: Compares candidates when measuring variables in `go1.19.3.exe`.

The graphs of the glue function metric show that only Monoidic and our extension give rise to glue functions, see Figure 4.9. Looking at the different phases of our extension, it becomes apparent that the glue functions appear because of the library signature importer analysis (see Section 4.5.2). Monoidic also performs a similar analysis and therefore also gives rise to many glue functions. However, in the binary `hugo.exe`, only our extension generates many glue functions while Monoidic does not. It is unclear why this is the case. Section 5.7.3 shows examples of why only these two candidates give rise to glue functions and discusses the readability aspects of these artifacts.
Figure 4.9: Compares candidates when measuring glue functions in go1.20.2.exe.
Chapter 5

Discussion

This chapter discusses various aspects of the evaluation, the entire thesis project as well as the results. Section 5.1 details why IDA is excluded from the evaluation. Threats to validity are discussed in Section 5.2 and one specific threat is reflected upon in more detail in Section 5.3. Section 5.4 discusses the correctness of our extension's transformations and how they can be tested. Section 5.5 discusses the different features implemented and reimplemented. Section 5.6 talks about the research question and the results of our project. Readability as a concept is reflected upon in Section 5.7. The chapter ends by discussing the societal impacts of improving reverse engineering in Section 5.8.

5.1 IDA exclusion

As mentioned in Section 4.5, IDA fails to decompile many functions. It also seems that IDA does not queue all functions for decompilation, only some. During the evaluation, we only had partial access to IDA and could therefore not investigate these issues in detail. Therefore we choose to exclude IDA from the evaluation results.

5.2 Threats to validity

The malware test cases contain relatively little user-written code, making the runtime and any used parts of the standard library make up a large share of the binaries' code. While the malware binaries are compiled with different versions of Go, the difference in the runtime and standard library code between
versions is likely not that large. Together, this makes the code diversity among the malware test cases low.

While the malware binaries are representable of the target use cases for the extensions, the readability perceived by the reverse engineer comes from the decompiled user-written code. A reverse engineer generally does not dig into the decompilation of the runtime or standard library since they are standard.

Therefore, to measure the readability of the decompilation as accurately as possible, the non-user-written code could be excluded from the measurements. This ensures that the same code is not re-evaluated several times. This was not done due to time limitations.

As mentioned in Section 4.4, the output of the decompilers cannot be directly fed into a C parser. The exported code contains too many syntactic and semantic errors, for example, missing type definitions. This requires us to preprocess the exported decompilation before parsing. The preprocessing is error-prone and ad-hoc. While it attempts to fix issues, it might introduce others.

To rank the evaluation candidate box plots we sort by the tuple \((Q_3, Q_2, Q_1)\). This is a very ad-hoc way of comparing distributions. It is very possible for two distributions \(Q\) and \(P\) that \(Q_3 > P_3\) while \(Q_2 < P_2\) and \(Q_1 < P_1\). We note, however, that for most of the graphs, it holds that if \(Q_3 > P_3\), then \(Q_2 > P_2\) and \(Q_1 > P_1\). Our way of ranking also ignores the functions with the 25% highest metric values. There are certainly more principled ways of comparing distributions, but due to a lack of time, this way of ranking is used.

5.3 Iterative vs. one-shot

Reverse engineering suites are iterative tools by design. The intended workflow is for the user to examine a piece of disassembled or decompiled code, add some new annotations that enhance the clarity of the code, let the software react to the annotations, and then repeat. The annotations could be naming variables, creating or assigning types, assigning function signatures, etc.

The enhancements made, specifically for Go, in some of the evaluation candidates, also keep this workflow in mind. For example: Ghidra 10.3 ships with a script for annotating functions with correct custom storage settings after they have been manually annotated with the right number and type of parameters. The script is only useful after a reverse engineer has done some manual work.
Our evaluation uses a one-shot approach instead of an iterative one. The reason for this is that it is much simpler to evaluate a one-shot tool than an iterative one. The problem is that the results are not representative of the performance of the tools when used in their intended fashion. The script mentioned above is for example not used at all.

One way of evaluating in an iterative manner would be to perform a user study. The users would be allowed to naturally interact with the tool over time. We choose not to perform a user study since it is not within our area of expertise to conduct such a study and it is outside the scope of this project.

5.4 Correctness of decompilation

This project focuses on improving the readability of the produced decompilation. As mentioned in Section 2.3.1, decompilers can also be focused on providing correct output. Correctness here means that the outputted code exactly represents the semantics of the underlying assembly, or that when recompiled, produces a binary with the same semantics as the original.

While our extension does not focus on correctness, it is still interesting to ask whether the transformations implemented are correct. Even if our decompilation is readable, it might be incorrect, misleading the reverse engineer and wasting time. There are many ways of testing for correctness. One way of testing for correctness is to spot-check some decompiled functions for defects.

Liu and Wang [24] use another, more sophisticated approach. They compare the correctness of several C decompilers using Equivalence Modulo Inputs (EMI) testing. They generate a random C program using Csmith [45], mutate the program in an EMI fashion, compile the mutation, decompile it, recompile it, and finally compare the output of the recompiled binary with a compiled version of the original source code. This way, they can detect many types of errors: decompilation failures, recompilation failures, and correctness errors when comparing the output of the recompiled binary with the original.

During development, any bugs found were of course fixed. However, no dedicated correctness testing was performed as it is outside the scope of this project.

It is important to note that improving readability is still valuable even if the decompiled code is not fully correct. For manual analysis, the fact that most of the code is readable is still important even if a line sometimes is incorrect. When there is doubt about the correctness of the decompilation,
the disassembly can always be consulted. But when something is consistently wrong across most lines of the code, the concept of readability loses meaning.

### 5.5 Features

When implementing our extension, we face the issue of lacking a foundation to build upon, requiring us to reimplement many pre-existing analyses. For extracting useful metadata from Go binaries, there are two excellent foundations to build upon, go-re.tk and GoReSym, of which we use go-re.tk. However, when it comes to analyses such as recreating all structures based on metadata or annotating Duff functions, either the analysis is implemented for another reversing suite such as IDA, or the extension available for Ghidra is monolithic, making it difficult to extend. An implementation of the function renamer could probably have been taken from a previous extension.

Two evaluation candidates were released while working on or after we finished implementing our extension: Monoidic and Ghidra 10.3. These two releases implement some of the things we decided to implement: library signature importer (Section 3.7), support for Go’s calling convention using Custom Storage (Section 3.8), marking non-returning functions (Section 3.3), etc. If they had been released before starting our implementation, we would have been able to build upon the previous work instead of reimplementing it. Unlike other previous extensions, both of these are easy to build upon. Monoidic is modular and the improvements made in Ghidra 10.3 are part of the Ghidra platform itself.

The large amount of reimplementation caused a lack of time for implementing more new analyses. Unfortunately, our improvements only use metadata that was previously known and used, albeit in new ways. Utilizing the previously unused metadata, such as the funcdata table (see Section 2.2.4), is left as promising future work (see Section 6.2).

### 5.6 Results

The goal of the thesis is to answer the research question:

- **How much can the readability of the decompilation of Go programs be improved, for manual analysis, by taking advantage of the inherent information present in the binaries?**

Which is further specified in the following sub-questions:
1. What kind of information present in Go binaries can aid in decompilation?

2. How can that information be used to improve readability?

3. How much do these improvements increase readability compared to previous work?

In Section 2.2, we present the metadata that is available in Go binaries. We describe previously known metadata, such as the entry point to function name mapping and the typelinks list of all types. However, we also described the funcdata table, metadata that has not been used by any previous work which might prove useful for enhancing decompilation.

Chapter 3 then shows how we use the metadata in our extension to improve the readability of the decompilation. Through eight implemented analyzers our extension performs three major tasks: making Ghidra understand Go's ABIInternal, reducing noise in the decompilation, and enabling Ghidra to represent the code better by adding more type information.

Finally, in Chapter 4, we evaluate how much the implemented improvements increase readability by comparing against several previous extensions. Code readability is a subjective concept without an objective definition. In this thesis, we evaluate the decompiled code's readability using quantitative metrics (see Section 4.3). These metrics act as a (hopefully) accurate proxy for the "true" readability of the code. Section 5.7 goes more into depth about the concept of readability, if the chosen metrics were suitable, and in what way our extension improved readability.

In comparison with previous work, our extension improves on some metrics and not so much on other metrics. Our extension is good at reducing noise for the reverse engineer to filter through. This is thanks to the non-returning function annotator and the stack growth prologue remover. At the same time, our extension introduces new information in the form of types and signatures, making the decompiled code more information-dense. All in all, our extension puts some effort towards increasing readability while also attempting to increase how informative the code is for the reverse engineer. These two goals do not always align, but they are both desirable aspects for manual reverse engineering.
5.7 Readability

This section reflects on the concept of readability in relation to the chosen metrics used in the evaluation to represent readability.

5.7.1 Variable count

After some reflection, we do not think that the variable count metric is an accurate proxy for readability. There is no clear interpretation of why readable decompilations contain fewer variables than less readable decompilations. The previous work that uses the variable count metric also does not have any motivation for how it represents readability [38].

An example of when fewer variables lead to more readable code is that of stack structures. Sometimes it happens that stack structures are recovered by decompilers as separate variables instead of fields of one structure. By applying the correct type to the stack location, the decompilation can display field accesses to the single structure instead of accesses to separate variables.

Yet, it is also common that "splitting" one variable into multiple temporally separated ones yields more readable code. Because of register allocation during compilation, multiple variables can be assigned to the same register in the generated code, as long as their definition and usage intervals do not overlap. Ghidra sometimes decompiles this into a single variable. However, Ghidra can be instructed to "split" the variable according to its definition and usage intervals into multiple variables with separate types and names.

5.7.2 Line and AST node count

Line and AST node count, both metrics that measure the "amount of code" present, are generally accurate proxies for readability. You can argue that if the code becomes too short it instead becomes more difficult to read. Since the goal of decompilers is not to intentionally produce obfuscated code, it is rarely a problem that decompilations are hard to understand because of their short length.

A larger risk for the inaccuracy of "amount of code" metrics is that code is missing because of some error and therefore the metrics are incorrect. Since we only evaluate based on the readability metrics, it is possible for some evaluation candidates to simply remove large parts of the code and therefore perform better on the metrics. To some extent, that is what our extension does when patching away the stack growth prologue (see Section 3.4). However,
we motivate why it is reasonable to do so.

Looking at Listing 5.1 showing the output of the Mooncat candidate, we can observe that all functions have been set with zero return values. This behavior makes Mooncat generate less code since the assignment of the return value is never made, thus performing better on the metrics. However, the generated code is wildly incorrect, since the return values are never tracked. Instead, variables like `extraout_RAX` show up without any apparent origin, containing what Ghidra assumes is the return value of earlier calls. There are also other argument and return value tracking issues in the code snippet, caused by Mooncat only using Custom Storage and not a `.cspec` file for specifying the calling convention. These issues are also not evaluated by the metrics.

Listing 5.2 shows the output of our extension on the same function as Mooncat. Since our extension does not set every function to have no return value and because we use a `.cspec` file (see Section 3.1), the return values in the code flow correctly to their destination.

### 5.7.3 Glue functions and casts

Because of how Ghidra displays passing structures stored in registers as arguments, it is more readable according to the metrics to not automatically apply function signatures. Ghidra displays these structures using a call to the `CONCAT` glue function and a cast. Listing 5.2 shows the pattern, as produced by our extension. As mentioned in Section 4.5.3, this only appears in the output of Monoidic and our extension.

The other candidates do not automatically apply function signatures and therefore do not produce any glue functions. However, if a compound type would manually be added to a function signature when using the other candidates, the pattern would again appear since it is caused by Ghidra itself.

Listing 5.3 shows how the Ghidra 10.3 candidate displays the function calls when the signatures are not automatically applied. The functions look like they take more arguments than they do. This happens because Go generates the same assembly code for a function taking two integer arguments as it would do for a function taking a single structure with two integer fields. Based on the assembly alone, Ghidra cannot do any better.

The question is which way of displaying calls is more readable. Listing 5.4 compares these two ways of displaying the function calls with a shortened version of `concatstring2`. The code with the signature applied is more correct and more informative since it shows the correct amount of arguments,
Listing 5.1: Example output from the Mooncat candidate on test case go1.18.3.exe.

```c
void main.LeaveInstructions(longlong param_1, undefined8 ← param_2) {
    // ...
    while (&stack0xfffffffffffffff8 < *(undefined **) (unaff_R14 + 0x10) || &
        stack0xfffffffffffffff8 == *(undefined **) (ulonglong ←
        *) (unaff_R14 + 0x10)) {
        runtime.morestack_noctxt();
    }
    runtime.concatstring2(0, &UNK_0052816e, 0xe5, &
        ← UNK_00527da6, 0x4f);
    runtime.concatstring2(0, extraout_RAX, &UNK_0052816e, &
        ← UNK_00527833, 0x3a);
    runtime.concatstring2(0, extraout_RAX_00, extraout_RAX, &
        ← UNK_00527df5, 0x50);
    runtime.concatstring2(0, extraout_RAX_01, extraout_RAX_00←
        , &UNK_00526562, 0x2d);
    runtime.concatstring4(local_38, extraout_RAX_02, ←
        extraout_RAX_01, &UNK_0051df8c, 9,
        ← goss_wU1VC46OGC_5eeaa0, DAT_005eeab8, &UNK_0051cb7f, ←
        4);
    runtime.concatstring2(0, extraout_RAX_03, extraout_RAX_02←
        , &UNK_00527ff5, 0xb1);
    runtime.concatstring2(0, extraout_RAX_04, extraout_RAX_03←
        , &UNK_00528253, 0x127);
    local_18 = extraout_RAX_05;
    runtime.concatstring3(0, param_1, param_2, "\\", 1, ←
        goss_Look_at_this_instruction.txt_5eeab0, DAT_005eeab8←
        );
    strings.Replace(extraout_RAX_06, param_1, &UNK_0051b8c2, ←
        2, "\\", 1, 0xfffffffffffffff);
    os.OpenFile(extraout_RAX_07, param_1, 0x41, 0x1a4);
    // ...
}
```
Listing 5.2: Example output from our extension on test case

gol.18.3.exe.

1 os.(*File).Write_ret main.LeaveInstructions(undef
2     """
3     _
4     _
5     _
6     _
7     _
8     _
9     _
10    _
11    _
12    _
13    _
14    _
15    _
16    _
17    

// ...

Listing 5.3: Example output from the Ghidra 10.3 candidate on test case go1.18.3.exe.

```c
void main.LeaveInstructions(undefined8 param_1, undefined8 →
                                  param_2) {
  // ...
  param_10 = param_1;
  param_11 = param_2;
  while (&stack0xfffffffffffffff8 <= CURRENT_G.stackguard0)←
  {
    runtime.morestack_noctxt();
  }
  auVar2 = runtime.concatstring2(0, &DAT_0052816e, 0xe5, &←
                                  DAT_00527da6, 0x4f);
  auVar2 = runtime.concatstring2(0, auVar2._0_8_, auVar2.←
                                  _8_8_, &DAT_00527833, 0x3a);
  auVar2 = runtime.concatstring2(0, auVar2._0_8_, auVar2.←
                                  _8_8_, &DAT_00527df5, 0x50);
  auVar2 = runtime.concatstring2(0, auVar2._0_8_, auVar2.←
                                  _8_8_, &DAT_00526562, 0x2d);
  auVar2 = runtime.concatstring4(local_38, auVar2._0_8_, ←
                                  auVar2._8_8_, &DAT_00527ff5, 0xb1);
  auVar2 = runtime.concatstring2(0, auVar2._0_8_, auVar2.←
                                  _8_8_, &DAT_00526f2e, &DAT_0051f8e8, DAT_005eeab8);
  local_18 = auVar2._0_8_;
Listing 5.4: Example showing the difference in output between candidates that automatically apply function signatures and those who do not. The "CONCAT and cast" pattern appears when passing structures in registers as arguments.

```plaintext
// No automatic signatures applied: Ghidra 10.3, etc.
s = concatstring2(&DAT_0052816e, 0xe5, &DAT_00527da6, 0x4f)
concatstring2(s._0_8_, s._8_8_, &DAT_00527833, 0x3a)

// Automatic library signatures: Our extension and Monoidic
s = concatstring2{
  (gostring)CONCAT88(0xe5, 0x52816e),
  (gostring)CONCAT88(0x4f, 0x527da6)
}
concatstring2(s, (gostring)CONCAT88(0x3a, 0x527833))
```

as well as their types. However, the code without the signature applied is more readable according to the metrics and is more concise.

Irrespective of which extensions are used, a reverse engineer manually analysing a binary is always going to add correct types to function signatures. This lets the types propagate around call sites and inside the functions, raising the abstraction level of the entire decompilation back towards the original source code. Since this is desirable and can be automated, both we and Monoidic decided to automate it. The fact that the Ghidra platform displays this artifact poorly should not reflect negatively on the improvement.

### 5.7.4 Reflection

There is an interesting balance between the concepts of readability, correctness, informativeness, and understandability when looking at decompiled code. When the overall correctness is too low, caused by for example systematic decompilation errors, then the other aspects lose meaning. But if incorrectness is local, for example to a single statement or branch, it is still desirable to improve the other aspects in the rest of the code. An increase in correctness can lead to both an increase or a decrease in readability, as shown in Sections 5.7.2 and 5.7.3.

Adding more information to a decompilation might make the code less "clean" or clear to read, but a large part of readability is also how understandable the code is. Decompilers are used for manual reverse engineering since understanding the behavior of the program by only reading the disassembly is too inefficient. One interpretation of readability is therefore that the most readable representation of a binary would be the original source
code it was compiled from. Then any step that brings the decompilation closer to the original source code increases readability. With this interpretation in mind, it is clear that each of our improvements enhances readability.

5.8 Societal impact

The legality of reverse engineering software has been a highly debated topic over the years [3, 5, 37]. There are both positive and negative uses of reverse engineering and decompilation. Malware is often reverse engineered to gain information on its operation. Reverse engineering could be used for industrial espionage and constructing software clones. However, it could also help in detecting clones [34].

Discussing how malicious users could obfuscate their programs brings this information to light, possibly informing other malicious users to employ the same techniques. The hope is that highlighting these techniques gives defenders and tool makers the ability to understand and counteract the threats.

Vulnerability researchers often use reverse engineering to discover software vulnerabilities. Whether this is positive or negative depends on the goals of the researcher. If their goal is to responsibly disclose any found issues, it should not be considered illegal or immoral. [27]

Clearly, decompilation is a dual-use application [36]. Over the years, there has been a lot of discussion regarding the ethics of cyber-security research [18]. Overall, the positive aspects of improving decompilation outweigh the negative ones.
Chapter 6

Conclusions

This chapter summarizes the conclusions of the thesis and discusses the potential future work.

6.1 Conclusions

The goal of the thesis is to answer the question:

• *How much can the readability of the decompilation of Go programs be improved, for manual analysis, by taking advantage of the inherent information present in the binaries?*

We answer the research question by building an extension for the reverse engineering suite Ghidra. Several different analyses offering improvements to readability are implemented. Some are reimplementations of previous work and some are novel. Only metadata that was known about beforehand is used, albeit in new ways. The readability of our extension's decompilation is compared against previous extensions in the Go reverse engineering space. Our extension improves on some metrics in comparison with previous work, while being worse at others. All in all, our extension produces more compact code while also enhancing its informativeness for the reverse engineer.

6.2 Future work

Here are ideas for future work that can further improve the reverse engineering experience when working on Go binaries:
• Resolve the structure representation issue discussed in Section 3.8 to better represent ABIInternal.

• Reduce noise by patching away calls to gcWriteBarrierXX.

• Closure function analysis. Detect all functions called X_funcN or which have live-in RDX registers. See which offsets relative to RDX are used to automatically build up the closure context structure.

• Annotate inlined functions with comments using the InlineTree from the funcdata table. This has previously been done with line numbers and file name references, but this is an additional way.

• Use StackObjects and ArgInfo from the funcdata table to approximate the shape and count of arguments.

• Beautifying decompilation passes, to make things look like Go. For example, lift runtime function calls to original language syntax in the output: starting goroutines, using maps and channels, defers, string concatenation, etc.
References


Appendix A

Go reverse engineering projects

Table A.1 contains the links to the Go reverse engineering projects.
### Name - URL

<table>
<thead>
<tr>
<th>Name</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoReSym</td>
<td><a href="https://github.com/mandiant/GoReSym">https://github.com/mandiant/GoReSym</a></td>
</tr>
<tr>
<td>goretk</td>
<td><a href="https://github.com/goretk">https://github.com/goretk</a></td>
</tr>
<tr>
<td>alphagolang</td>
<td><a href="https://github.com/SentineLabs/AlphaGolang">https://github.com/SentineLabs/AlphaGolang</a></td>
</tr>
<tr>
<td>Go parser</td>
<td><a href="https://github.com/0xjiayu/go_parser">https://github.com/0xjiayu/go_parser</a></td>
</tr>
<tr>
<td>IDAGolangHelper</td>
<td><a href="https://github.com/sibears/IDAGolangHelper">https://github.com/sibears/IDAGolangHelper</a></td>
</tr>
<tr>
<td>golang_loader_assist</td>
<td><a href="https://github.com/strazzere/golang_loader_assist">https://github.com/strazzere/golang_loader_assist</a></td>
</tr>
<tr>
<td>goutil</td>
<td><a href="https://gitlab.com/zaytsevgu/goutil">https://gitlab.com/zaytsevgu/goutil</a></td>
</tr>
<tr>
<td>Ghidra 10.3</td>
<td><a href="https://ghidra-sre.org/">https://ghidra-sre.org/</a></td>
</tr>
<tr>
<td>Monoidic</td>
<td><a href="https://github.com/moondic/go-api-parser">https://github.com/moondic/go-api-parser</a></td>
</tr>
<tr>
<td>Cyberkaida</td>
<td><a href="https://github.com/cyberkaida/golang-ghidra">https://github.com/cyberkaida/golang-ghidra</a></td>
</tr>
<tr>
<td>gostrings</td>
<td><a href="https://github.com/nccgroup/gostrings">https://github.com/nccgroup/gostrings</a></td>
</tr>
<tr>
<td>de-strip.py</td>
<td><a href="https://gist.github.com/rolandshoemaker/1ac463a589f24830940e89ab2333280c">https://gist.github.com/rolandshoemaker/1ac463a589f24830940e89ab2333280c</a></td>
</tr>
<tr>
<td>Mooncat</td>
<td><a href="https://github.com/mooncat-greenpy/Ghidra_GolangAnalyzerExtension">https://github.com/mooncat-greenpy/Ghidra_GolangAnalyzerExtension</a></td>
</tr>
<tr>
<td>Cujo AI</td>
<td><a href="https://github.com/getCUJO/ThreatIntel/tree/master/Scripts/Ghidra">https://github.com/getCUJO/ThreatIntel/tree/master/Scripts/Ghidra</a></td>
</tr>
<tr>
<td>ugo-ghidra</td>
<td><a href="https://github.com/osirislab/ugo-ghidra">https://github.com/osirislab/ugo-ghidra</a></td>
</tr>
<tr>
<td>gotools</td>
<td><a href="https://github.com/felberj/gotools">https://github.com/felberj/gotools</a></td>
</tr>
<tr>
<td>golang_renamer.py</td>
<td><a href="https://github.com/ghidraninja/ghidra_scripts/blob/master/golang_renamer.py">https://github.com/ghidraninja/ghidra_scripts/blob/master/golang_renamer.py</a></td>
</tr>
<tr>
<td>golang_pclntab_parser</td>
<td><a href="https://github.com/dipusone/golang_pclntab_parser">https://github.com/dipusone/golang_pclntab_parser</a></td>
</tr>
<tr>
<td>golang_1_18_restore_names</td>
<td><a href="https://github.com/scmerrill/golang_1_18_restore_names">https://github.com/scmerrill/golang_1_18_restore_names</a></td>
</tr>
<tr>
<td>binja-golang-symbol-restore</td>
<td><a href="https://github.com/d-we/binja-golang-symbol-restore">https://github.com/d-we/binja-golang-symbol-restore</a></td>
</tr>
<tr>
<td>bn-goloader</td>
<td><a href="https://github.com/f0rki/bn-goloader">https://github.com/f0rki/bn-goloader</a></td>
</tr>
<tr>
<td>gostringsr2</td>
<td><a href="https://github.com/CarveSystems/gostringsr2">https://github.com/CarveSystems/gostringsr2</a></td>
</tr>
<tr>
<td>r2-gohelper</td>
<td><a href="https://github.com/JacobPimental/r2-gohelper">https://github.com/JacobPimental/r2-gohelper</a></td>
</tr>
<tr>
<td>r2-go-helpers</td>
<td><a href="https://github.com/f0rki/r2-go-helpers">https://github.com/f0rki/r2-go-helpers</a></td>
</tr>
<tr>
<td>go-helpers</td>
<td><a href="https://github.com/zlowram/radare2-scripts/tree/master/go_helpers">https://github.com/zlowram/radare2-scripts/tree/master/go_helpers</a></td>
</tr>
<tr>
<td>jeb-golang-analyzer</td>
<td><a href="https://github.com/pnfsoftware/jeb-golang-analyzer">https://github.com/pnfsoftware/jeb-golang-analyzer</a></td>
</tr>
</tbody>
</table>

Table A.1: Project links for Go reverse engineering extensions. For more details, see Table 2.1.
Appendix B

Implementation examples

This appendix contains source code and links to binaries for examples used in Chapter 3.

B.1 Source code

All examples that include source code are compiled with Go 1.19.5 using the following command: `go build -ldflags="-s -w" code.go.

Source code for example of non-returning functions in Figure 3.3.

```go
func main() {
    if len(os.Args) < 2 {
        panic("usage: ./non_returning <message>"
    }
    os.Stdout.WriteString(os.Args[1])
}
```

B.2 Binaries

Binaries used in examples are listed below. Note that for some binaries, the SHA256 hash of the binary is listed in the URL.
Example reference
URL and SHA256
Figure 4.2 and Listings 3.4 and 3.5
https://bazaar.abuse.ch/sample/58e19ae5b80eaf4e0d071146b05c118143b432c2e4d117141dd71bc82cbf34c3/
Figure 3.5
SHA256: cc795abb661bfeeaf0cc23de92a3a3ccf558bae3398b8ec3af271b52ff88729
Appendix C

List of non-returning functions

This is the list of all functions marked as non-returning. For more details, see Section 3.3.

runtime.abort.abi0
runtime.exit.abi0
runtime.dieFromSignal
runtime.exitThread
runtime.fatal
runtime.fatalthrow
runtime.fatalpanic
runtime.gopanic
runtime.panicdivide
runtime.throw
runtime.goPanicIndex
runtime.goPanicIndexU
runtime.go PanicSliceAlen
runtime.goPanicSliceAlenU
runtime.goPanicSliceAcap
runtime.goPanicSliceAcapU
runtime.goPanicSliceB
runtime.goPanicSliceBU
runtime.goPanicSlice3Alen
runtime.goPanicSlice3AlenU
runtime.goPanicSlice3Acap
runtime.goPanicSlice3AcapU
runtime.goPanicSlice3B
runtime.goPanicSlice3BU
runtime.goPanicSlice3C
runtime.goPanicSlice3CU
runtime.goPanicSliceConvert
runtime.panicIndex
runtime.panicIndexU
runtime.panicSliceAlen
runtime.panicSliceAlenU
runtime.panicSliceAcap
runtime.panicSliceAcapU
runtime.panicSliceB
runtime.panicSliceBU
runtime.panicSlice3Alen
runtime.panicSlice3AlenU
runtime.panicSlice3Acap
runtime.panicSlice3AcapU
runtime.panicSlice3B
runtime.panicSlice3BU
runtime.panicSlice3C
runtime.panicSlice3CU
runtime.panicSliceConvert
runtime.panicdottypeE
runtime.panicdottypeI
runtime.panicnildottype
runtime.panicoverflow
runtime.panicfloat
runtime.panicmem
runtime.panicmemAddr
runtime.panicshift
runtime.goexit0
runtime.goexit0.abi0
runtime.goexit1
runtime.goexit.abi0
runtime.Goexit
runtime.sigpanic
runtime.sigpanic0.abi0
os.Exit
Appendix D

x86–64 registers

Figure D.1 shows the 16 general purpose registers in x86–64. Each register exists in 4 different versions depending on which bytes of the register that are referred to. For example: EAX refers to the lowest 4 bytes of RAX.
Figure D.1: Diagram showing general purpose registers in x86–64. Note that the registers exist in several versions, depending on which bytes of the register are referred to.
Appendix E

Polymorphic runtime functions analyzed

Below follows a list of all polymorphic runtime functions analyzed by the polymorphic function analyzer described in Section 3.9:

- convI2I
- convT
- makeslice
- growslice
- mapaccess1_faststr
- mapassign_faststr
- mapiterinit
- newobject
Appendix F

Evaluation binaries

The binaries used for evaluation are listed below. Note that for some binaries, the SHA256 hash of the binary is listed in the URL.
<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>go1.17.8.exe</td>
<td>6.0 MB</td>
<td><a href="https://bazaar.abuse.ch/sample/f2a09b611b6fca3e82b8c3098abc35929779685a9e3f851a6acf4040be002f41">https://bazaar.abuse.ch/sample/f2a09b611b6fca3e82b8c3098abc35929779685a9e3f851a6acf4040be002f41</a></td>
</tr>
<tr>
<td>go1.18.3.exe</td>
<td>2.0 MB</td>
<td><a href="https://bazaar.abuse.ch/sample/46d340eaf6b78207e24b6011422f1a5b4a566e493d72365c6a1cace11c36b28b">https://bazaar.abuse.ch/sample/46d340eaf6b78207e24b6011422f1a5b4a566e493d72365c6a1cace11c36b28b</a></td>
</tr>
<tr>
<td>go1.19.3.exe</td>
<td>4.8 MB</td>
<td><a href="https://bazaar.abuse.ch/sample/b3d5e4d38a475f509d0ad29beb88753d09509ac3100bdddcf0a42773b2d14a90">https://bazaar.abuse.ch/sample/b3d5e4d38a475f509d0ad29beb88753d09509ac3100bdddcf0a42773b2d14a90</a></td>
</tr>
<tr>
<td>go1.20.2.exe</td>
<td>1.8 MB</td>
<td><a href="https://bazaar.abuse.ch/sample/9a22c8cc9928574868022d5b47738b8fc85027d0cecc46dd2f91f885d19ad2f18">https://bazaar.abuse.ch/sample/9a22c8cc9928574868022d5b47738b8fc85027d0cecc46dd2f91f885d19ad2f18</a></td>
</tr>
<tr>
<td>hugo.exe</td>
<td>55 MB</td>
<td><a href="https://github.com/gohugoio/hugo/releases/download/v0.111.3/hugo_extended_0.111.3_windows-amd64.zip">https://github.com/gohugoio/hugo/releases/download/v0.111.3/hugo_extended_0.111.3_windows-amd64.zip</a></td>
</tr>
<tr>
<td>SHA256: 6138181688936354abde022523ebc09e97cc72183f813626b02982e50270b0bb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strange.go1.18.3.exe</td>
<td>14 MB</td>
<td><a href="https://bazaar.abuse.ch/sample/03e0491d1e10eb09d68e7420f3ebd8a3da94f3162fa9ac170123773b178d7aff">https://bazaar.abuse.ch/sample/03e0491d1e10eb09d68e7420f3ebd8a3da94f3162fa9ac170123773b178d7aff</a></td>
</tr>
</tbody>
</table>
Appendix G

Results

G.1 Comparing phases

Figures G.1 to G.25 show how the phases of our extension compare against each other on all metrics and test binaries.
Figure G.1: Compares phases when measuring lines in go1.17.8.exe.

Figure G.2: Compares phases when measuring lines in go1.18.3.exe.
Figure G.3: Compares phases when measuring lines in `go1.19.3.exe`.

Figure G.4: Compares phases when measuring lines in `go1.20.2.exe`.
Figure G.5: Compares phases when measuring lines in *hugo.exe*.

Figure G.6: Compares phases when measuring nodes in *go1.17.8.exe*. 
Figure G.7: Compares phases when measuring nodes in go1.18.3.exe.

Figure G.8: Compares phases when measuring nodes in go1.19.3.exe.
Figure G.9: Compares phases when measuring nodes in `go1.20.2.exe`.

Figure G.10: Compares phases when measuring nodes in `hugo.exe`. 
Figure G.11: Compares phases when measuring casts in \texttt{go1.17.8.exe}.

Figure G.12: Compares phases when measuring casts in \texttt{go1.18.3.exe}.
Figure G.13: Compares phases when measuring casts in `go1.19.3.exe`.

Figure G.14: Compares phases when measuring casts in `go1.20.2.exe`.
Figure G.15: Compares phases when measuring casts in `hugo.exe`.

Figure G.16: Compares phases when measuring variables in `go1.17.8.exe`.
Figure G.17: Compares phases when measuring variables in go1.18.3.exe.

Figure G.18: Compares phases when measuring variables in go1.19.3.exe.
Figure G.19: Compares phases when measuring variables in `go1.20.2.exe`.

Figure G.20: Compares phases when measuring variables in `hugo.exe`. 
Figure G.21: Compares phases when measuring glue functions in `go1.17.8.exe`.

Figure G.22: Compares phases when measuring glue functions in `go1.18.3.exe`.
Figure G.23: Compares phases when measuring glue functions in go1.19.3.exe.

Figure G.24: Compares phases when measuring glue functions in go1.20.2.exe.
Figure G.25: Compares phases when measuring glue functions in `hugo.exe`. 
Figure G.26: Compares candidates when measuring lines in go1.17.8.exe.

G.2 Comparing candidates

Figures G.26 to G.50 compare the different evaluation candidates on all metrics and test binaries.
Appendix G: Results | 107

Figure G.27: Compares candidates when measuring lines in `go1.18.3.exe`.

Figure G.28: Compares candidates when measuring lines in `go1.19.3.exe`.
Figure G.29: Compares candidates when measuring lines in `go1.20.2.exe`.

Figure G.30: Compares candidates when measuring lines in `hugo.exe`.
Figure G.31: Compares candidates when measuring nodes in go1.17.8.exe.

Figure G.32: Compares candidates when measuring nodes in go1.18.3.exe.
Figure G.33: Compares candidates when measuring nodes in go1.19.3.exe.

Figure G.34: Compares candidates when measuring nodes in go1.20.2.exe.
Figure G.35: Compares candidates when measuring nodes in `hugo.exe`.

Figure G.36: Compares candidates when measuring casts in `go1.17.8.exe`.
Figure G.37: Compares candidates when measuring casts in `go1.18.3.exe`.

Figure G.38: Compares candidates when measuring casts in `go1.19.3.exe`.
Figure G.39: Compares candidates when measuring casts in `go1.20.2.exe`.

Figure G.40: Compares candidates when measuring casts in `hugo.exe`.
Figure G.41: Compares candidates when measuring variables in `go1.17.8.exe`.

Figure G.42: Compares candidates when measuring variables in `go1.18.3.exe`. 
Figure G.43: Compares candidates when measuring variables in go1.19.3.exe.

Figure G.44: Compares candidates when measuring variables in go1.20.2.exe.
Figure G.45: Compares candidates when measuring variables in `hugo.exe`.

Figure G.46: Compares candidates when measuring glue functions in `go1.17.8.exe`.
Figure G.47: Compares candidates when measuring glue functions in `go1.18.3.exe`.

Figure G.48: Compares candidates when measuring glue functions in `go1.19.3.exe`.
Figure G.49: Compares candidates when measuring glue functions in `go1.20.2.exe`.

Figure G.50: Compares candidates when measuring glue functions in `hugo.exe`.
## G.3 Tables

Tables G.1 to G.5 show the median, average, and maximum values for each metric, over all test binaries.

### G.1 Lines

<table>
<thead>
<tr>
<th>Metric</th>
<th>go1.17.8.exe</th>
<th>go1.18.3.exe</th>
<th>go1.19.3.exe</th>
<th>go1.20.2.exe</th>
<th>hugo.exe</th>
</tr>
</thead>
</table>

### G.2 Nodes

<table>
<thead>
<tr>
<th>Metric</th>
<th>go1.17.8.exe</th>
<th>go1.18.3.exe</th>
<th>go1.19.3.exe</th>
<th>go1.20.2.exe</th>
<th>hugo.exe</th>
</tr>
</thead>
</table>

### G.3 Casts

<table>
<thead>
<tr>
<th>Metric</th>
<th>go1.17.8.exe</th>
<th>go1.18.3.exe</th>
<th>go1.19.3.exe</th>
<th>go1.20.2.exe</th>
<th>hugo.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go1.17.8.exe</td>
<td>go1.18.3.exe</td>
<td>go1.19.3.exe</td>
<td>go1.20.2.exe</td>
<td>hugo.exe</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
<td>---------</td>
</tr>
<tr>
<td>Ghidra 10.2.3</td>
<td>9</td>
<td>12</td>
<td>459</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>Mooncat</td>
<td>12</td>
<td>15</td>
<td>459</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>Our extension</td>
<td>12</td>
<td>16</td>
<td>565</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>Ghidra 10.3</td>
<td>13</td>
<td>19</td>
<td>499</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>Monoidic</td>
<td>11</td>
<td>15</td>
<td>459</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>Cyberkaida</td>
<td>11</td>
<td>13</td>
<td>167</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Table G.4: Shows the median, average, and maximum number of Variables, over all test binaries.

<table>
<thead>
<tr>
<th></th>
<th>go1.17.8.exe</th>
<th>go1.18.3.exe</th>
<th>go1.19.3.exe</th>
<th>go1.20.2.exe</th>
<th>hugo.exe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ghidra 10.2.3</td>
<td>0</td>
<td>11</td>
<td>559</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Mooncat</td>
<td>0</td>
<td>11</td>
<td>559</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Our extension</td>
<td>4</td>
<td>10</td>
<td>563</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>Ghidra 10.3</td>
<td>0</td>
<td>11</td>
<td>243</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Monoidic</td>
<td>4</td>
<td>13</td>
<td>932</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>Cyberkaida</td>
<td>0</td>
<td>11</td>
<td>646</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

Table G.5: Shows the median, average, and maximum number of Glue functions, over all test binaries.
Malware written in Go is on the rise, and yet, tools for investigating Go programs, such as decompilers, are limited. A decompiler takes a compiled binary and tries to recover its source code. Go is a high-level language that requires runtime metadata to implement many of its features, such as garbage collection and polymorphism. While decompilers have to some degree used this metadata to benefit manual reverse engineering, there is more that can be done.

To remedy this, we extend the decompiler Ghidra with improvements that increase the readability of the decompilation of Go binaries by using runtime metadata. We make progress towards enabling Ghidra to represent Go’s assembly conventions. We implement multiple analyses: some which reduce noise for the reverse engineer to filter through, some which enhance the decompilation by adding types, etc. The analyses are a mix of reimplementations of previous
work and novel improvements. The analyses use metadata known beforehand but in new ways: applying data types at polymorphic function call sites, and using function names to import signatures from source code. We also discover previously unused metadata, which points to promising future work.

Our experimental evaluation compares our extension against previously existing extensions for decompilers using multiple readability metrics. Our extension improves on metrics measuring the amount of code, such as lines of code. It also decreases the number of casts. However, the extension performs worse on other metrics, producing more variables and glue functions. In conclusion, our extension produces more compact code while also increasing its informativeness for the reverse engineer.

"Keywords[eng]": Decomposition, Go, Runtime Metadata, Reverse Engineering


"Keywords[swa]": Dekomprimering, Go, Körtidsmetadata, Reverse Engineering