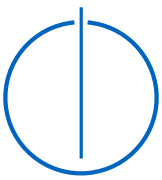
Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Reconstructing Program Semantics from Go Binaries

Alexis Engelke



Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Rekonstruktion der Programmsemantik anhand von Go-Binärdateien

Reconstructing Program Semantics from Go Binaries

Author:	Alexis Engelke
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Julian Kirsch, M.Sc.
Submission Date:	15. September 2017

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum

Alexis Engelke

Acknowledgments

First, I would like to thank Claudia Eckert for giving me the opportunity to write this thesis. I thank Julian Kirsch for his permanent encouragement, for reading through an earlier version of this thesis and providing lots of feedback, and for listening to my ideas. I also thank Charlie Groh for reading this thesis and giving additional feedback. Finally, I want to thank my mother for her extraordinary support, without her I would not be able to write a thesis at this place and time.

Abstract

Go is a new programming language with the goal to combine simplicity, safety and efficiency. As such, the official compiler does not involve standard compiler infrastructures but uses a custom code generation procedure. With the increasing popularity, however, the language also gains traction among malware developers, motivating the research of binary analysis procedures on Go binaries. Existing tools for automated analysis have problems in handling peculiarities of the Go compiler and the available tooling for extracting metadata has a very limited scope. Moreover, the internal data structures generated by the compiler are poorly understood and not documented publicly.

In this thesis, we will provide a documentation of metadata structures and specifics of the code generation procedure relevant for reverse engineering along with a strategy to extract this information from stripped binaries compiled by Go 1.6–1.8. Furthermore, we present an assembly-based code representation named *Higher-level Go Assembly* which models specifics of the compiler appropriately and concisely and therefore facilitates automated and manual analysis. Based on this, we propose a constraint-based type analysis strategy to determine the basic type of function parameters and return values, but encounter modeling limitations for specific language features and find that an explicit analysis appears beneficial.

Contents

1 Introduction

The Go programming language is a newer language developed at Google aiming to combine simplicity, safety and efficiency [31]. An important goal is to provide memory safety, preventing many kinds of vulnerabilities like buffer overflows found in widespread languages like C or C++. For efficiency, Go is compiled to machine code and has built-in support for multi-threading and parallel computations. Another fundamental property is efficiency of compilation.

To achieve a performant compilation, the official Go compiler (*Gc*) is not based on existing compiler infrastructures like LLVM [22, 31] but includes a custom optimization process, code generation procedure and assembler and also uses a different calling convention for functions. A front-end for the GCC compiler infrastructure exists (*Gccgo*), which performs more expensive optimizations at the cost of compile-time and uses the standard code generation procedure of GCC [30]. Beside the different code generation strategy, there exists another major difference to compilers of conventional languages: Go binaries include a huge amount of metadata to allow specific language features such as garbage collection, reflection or stacktrace generation. In addition to being linked statically by default, this metadata significantly contributes to the size of the binaries.

Even though the language was just published in 2009 and had its first stable release in 2012 [31], Go has rapidly grown in popularity and is ranked in the Top 10 of the 2017 IEEE Spectrum ranking of programming languages [12]. A survey conducted by the Go developers reveals that a common use case of Go are web and other network services as well as infrastructure-related tasks [16] and in fact the standard library especially focuses on topics related to web programming, e.g. the HTTP protocol, cryptographic functions or support for HTML templates[1].

In addition, also first samples of malware being written in Go have been observed. For example the *Encriyoko* trojan[2] discovered in 2012 attempts to encrypt files in several formats on affected computers; and in 2016 the trojan *Rex*[3] was found, which tries to fetch credentials from infected computers and allows to launch a remotely controlled DDoS attack.

However, existing analysis tools have difficulties in handling Go binaries. For example, the Hex-Rays Decompiler [17] has significant problems in handling the unique calling convention as well as specific aspects of the code generation strategy and the result of the decompilation is overly complex in most cases. In general, the included metadata about functions and data types is ignored completely in most cases. Few attempts have been made to develop plug-ins for the IDA disassembler [18], but these have very special

[1] `https://golang.org/pkg/`, accessed 2017-08-25

[2] `https://www.symantec.com/connect/blogs/malware-uses-google-go-language`, accessed 2017-08-25

[3] `https://vms.drweb.com/virus/?_is=1&i=8436299&lng=en`, accessed 2017-08-25

assumptions or a limited scope, or both. Moreover, the documentation of the internals of the Go compiler and runtime is poor[4] and existing blog post are partially outdated.

Focusing on the x86-64 architecture, we will provide a documentation of important metadata structures along with a strategy to extract this information from Go binaries of different versions. In addition to this, we will describe the employed calling convention and crucial aspects of the code generation strategy. Using this information, we provide a very low-level intermediate representation streamlining some peculiarities for Go functions to ease analysis. Based on this, we will explore the possibilities of the analysis of function types using constraints in the context of Go binaries.

Outline

The remainder of this thesis is structured as follows. In Chapter 2, we will describe existing work related to the analysis of Go binaries and basic foundations for our type analysis. Chapter 3 revisits some uncommon language constructs of Go along with a description of the fundamental data types, the metadata included in the binaries and the calling convention as well as other specifics of the code generation procedure. In this chapter, we will also describe the requirements for memory safety and briefly summarize ways to break this. In Chapter 4 we present our strategy to extract the metadata from a given binary and describe a more idiomatic low-level intermediate language for Go code as well as a lifting procedure. Based on this, we describe our constraint-based type analysis for compiled Go functions. We will evaluate and discuss the proposed strategies in Chapter 5. Finally, we will summarize our findings and give an outlook to future work and possibilities for further extensions.

Contributions

Key contributions of this thesis include:

- A documentation of several Go internals on the x86-64 architecture as of version Go 1.8, including the layout of core data types, the calling convention, critical metadata structures and specifics of the code generated by the Go compiler.
- A strategy and tool to identify the Go version and to extract information about functions and types from stripped Go binaries.
- An assembly-based low-level intermediate representation which allows the elimination of specific compiler generated code idioms, simplifying the assembly code and easing analysis; combined with a lifting procedure for functions compiled from Go code.
- A constraint-based analysis of the type of the functions making use of available type information.

[4] `https://github.com/golang/go/issues/16199`, accessed 2017-08-25

2 Background

In this section, we will describe existing work in the field of analyzing Go binaries and lay the foundations for our type analysis of Go functions employing type constraints.

2.1 Related Work

To the best of our knowledge, little work on reverse engineering Go binaries has been done so far. We know a set of scripts named *goutils*[1] with the goal to ease analysis of stripped, Go-compiled ELF binaries using IDA [18]. These scripts provide (1) a simple heuristic to get the size of the argument region when calling functions; (2) a heuristic to find strings in the binary; (3) a simple heuristic to find and extract function names from the symbol information table (later referred to as runtime symbol information or the *pclntab*); and (4) a parser for the included type information where the Go version has to be specified manually. These scripts, however, do not support PE files (Windows) and also ignore the fact that much more metadata (e.g. the size of the argument region) *is* readily available from the binary.

We also know a blog post by Tim Strazzere[2] about reverse engineering with Go binaries using IDA. The developed plugin for IDA attempts to find functions by tracing calls to a specific routine found at the beginning of most functions[3]. Based on the *goutils* described above, it similarly extracts the function names from the symbol information table. Additionally, the script applies some heuristics to identify strings as well as their length in the binary. However, this script also ignores large parts from the available metadata and the employed heuristics appear to be rather simple, and are likely to fail for complex binaries and functions. For example, this script will not identify simple functions or closures and when storing constant strings in memory, the address is expected in one of only four registers although more registers are available and generally also used.

Additionally, the Go standard library includes a package (`debug/gosym`) which parses a given symbol information table as it is included in Go binaries.[4] However, this package only parses small parts of the symbol information and requires an additional metadata table, which is no longer present in binaries produced by recent versions of the compiler.

In the context of parsing meta information from C++ binaries, MARX [26] is a framework to extract information about class hierarchies from striped binaries using static analysis. Starting with a heuristic to find virtual function tables (*vtables*), the

[1] `https://gitlab.com/zaytsevgu/goutils`, accessed 2017-08-22

[2] `https://rednaga.io/2016/09/21/reversing_go_binaries_like_a_pro/`, accessed 2017-08-21

[3] The function is `runtime.morestack_noctxt` and is conditionally called to enlarge the stack if necessary.

[4] See `https://golang.org/pkg/debug/gosym/`, accessed 2017-08-22

identification of the class hierarchy is based on an analysis of entries in the function table combined with a data flow analysis. There are also approaches to reconstruct this meta information using the Runtime Type Information (RTTI) included in many C++ binaries [14].

2.2 SMT Solvers

The Satisfiable Modulo Theories (SMT) problem is a decision problem for formulas, extending the Boolean Satisfiability (SAT) problem by arithmetic, arrays, uninterpreted functions and other theories [11]. The task of an SMT solver is to decide the satisfiability of given set of formulas or constraints. There are two main approaches for the implementation of SMT solvers, referred to as *eager* and *lazy* approaches [4]. The idea of the *eager* approach is to convert the SMT input problem into a SAT problem, making implicit consequences of the used theories explicit. This allows for the use of existing SAT solvers on the transformed formula. However, depending on the input the number of constraints might grow large. In contrast, solvers following the *lazy* approach integrate SAT solvers with specialized solvers for different theories. A common variant of lazy SMT solvers are DPLL[5] solvers [2, 4]. In its simplest form, the input is abstracted into a SAT problem, where the underlying SAT solver either proves unsatisfiability, implying that also the input is unsatisfiable, or provides a satisfying assignment, which is given to the theory solver. If the assignment is consistent with the theory, the input is satisfiable, otherwise an additional constraint regarding the inconsistency is added to the abstracted SAT problem.

If a set of SMT constraints is found to be unsatisfiable, the *unsatisfiability core* is an unsatisfiable subset of constraints [4]. One approach (*assumption-based* approach) to receive an unsatisfiability core is to add a new boolean selector variable S_i for each constraint C_i, to replace each constraint C_i with $S_i \Rightarrow C_i$ and to force $S_i = true$. In case of a conflict while solving the constraints, the constraints whose selector variables S_i are part of the conflict are returned as unsatisfiability core.

Z3 [11] is a SMT solver developed at Microsoft Research following the lazy approach. In addition to deciding satisfiability, it is also capable of providing a model satisfying the constraints and an unsatisfiability core using an assumption-based approach. The constraints can be formulated using the SMT-LIB 2.6 [3] language or a dedicated Application Programming Interface (API).

[5] Abbr. Davis-Putnam-Logemann-Loveland

3 The Go Language

The Go programming language and compiler have some major differences compared to other languages like C or C++. After describing uncommon language concepts, the internal representation of data types, the calling convention as well as the meta information included in compiled binaries will be described. Then, the aspects of memory safety in Go programs will be analyzed briefly. Finally, some peculiarities of the code generated by the Go compiler will be explained.

This description focuses on the x86-64 architecture. All references to the source of the Go runtime and compiler refer to Go 1.8.1, as released on April 7, 2017.[1]

3.1 Core Language Concepts

The Go language has some rather uncommon language concepts. In this section, some of these concepts will be described briefly.

The idea of *goroutines* is to run multiple, parallel executing functions on a set of threads provided by the operating system [31]. When a goroutine blocks, e.g. by waiting for a mutex or issuing a system call, the runtime transparently switches between the goroutines on the same thread. Therefore, compared to an operating system thread, a goroutine is more lightweight: a goroutine has a dynamically sized stack (starting at 2kiB) and a small control data structure, allowing to create and execute many goroutines efficiently.

In terms of structures within the runtime, a goroutine is represented by an instance of the type `g`, which is also accessible at address `fs:[-0x8]` in the Thread-local Storage (TLS). An operating system thread is represented by the type `m`. Each operating system thread additionally has a separate signal handling goroutine and a scheduling goroutine, which use the stack provided by the operating system.

Communication between goroutines can either happen via a shared variable and appropriate locking or via *channels*. Channels allow to pass values across different goroutines and can also be used for synchronization. A special `select` statement allows a goroutine to wait for multiple communication operations in parallel [32].

A Go function can *defer* a function call by pushing it to a list, where the actual function call is performed after the function returned [32]. This is commonly used to call clean-up functions (e.g. closing a file, releasing a lock) independently of the following control flow. An example of a simple defer statement is shown in Listing 3.1. The arguments of a deferred function call are evaluated when the defer statement is evaluated and stored with the function to call.

[1] `https://github.com/golang/go/tree/go1.8.1`, accessed 2017-05-10, tag `go1.8.1`, commit `a4c18f0`

```
func foo(name string) error {
    file, err := os.Open(name)
    if err != nil {
        return err
    }
    defer file.Close()

    ...
}
```

Listing 3.1: *Example of a Defer Statement. The file is closed independently of the following control flow.*

```
func foo(name string) error {
    defer func() {
        if r := recover(); r != nil {
            ...
        }
    }()
    ...
    panic(...)
}
```

Listing 3.2: *Example of a Defer Statement with recover. The panic is not propagated outside of the function `foo` but catched inside the deferred function.*

Instead of exceptions, Go employs two different approaches of propagating errors through the program: one approach is to add an additional error return value, which the caller has to check. The other approach is *panicking*. A panic can be issued by the `panic` function or by the runtime, e.g. when a `nil` pointer is dereferenced. In case of a panic, the ordinary control flow is stopped and the goroutine enters the panicking mode. The panic propagates through the call stack, calling the deferred functions of the panicking function and its callers. When the panic reaches the top-most function, a stack trace is printed and the program is aborted.

However, it is possible to *recover* from a panic. When a deferred function calls the `recover` function while the goroutine is panicking, the panic is stopped and the normal control flow continues at this point. The state of the functions between the panic and the recovering is discarded.

An *interface* is a type which specifies a set of methods [32]. All data types which implement all required methods can be implicitly converted to the corresponding interface type. The interface type which has no methods (the *empty interface*) is implemented by all types and can be compared to the `Object` base class in Java. The actual type of the value stored in an interface variable at run time is referred to as *dynamic type*.

It is also possible to convert an interface back to a fixed type or another interface type using *type assertions*, which are similar to class casts in object-oriented programming languages. A simple example for a type assertion shown in Listing 3.3. If the dynamic type of the interface value does not have the required type or does not implement the required interface type, the invalid assertion must be caught or a panic occurs. In contrast

```
func foo(v interface{}) int {
    if i, ok := v.(int); ok {
        return i
    }
    return 0
}
```

Listing 3.3: *Example of a type assertion. If the argument does not have an integer type at runtime, a constant is returned.*

```
func foo(v interface{}) int {
    switch i := v.(type) {
    case int: return i
    case uint: return int(i)
    }
}
```

Listing 3.4: *Example of a type switch. Depending on the actual type of the argument a different code path is executed.*

to many other languages, it is also possible to use a switch statement over the dynamic type of the interface, as shown in Listing 3.4.

3.2 Data Type Representation

The Go programming language provides a range of simple and structured types. In this section, the provided types as well as their internal representation in the x86-64 architecture will be described briefly. An overview is given in Table 3.1. Each data type in Go has an alignment requirement, which must always be satisfied.

The *numeric types* of Go contain a range of types for integers, floating-point numbers and complex numbers. They include signed and unsigned integers with sizes of 8-, 16-, 32- and 64-bits as well as floating-point numbers with sizes of 32-bits and 64-bits. Furthermore, complex numbers with a size of 64- or 128-bits consisting of two appropriately sized floats are supported. The `uintptr` type has the size of a pointer, which is 64-bit. Likewise, the `int` and `uint` types have a native size of 64-bit. The boolean type (`bool`) is stored as byte with the values 0 or 1.

A *string* in Go is a tuple of (1) a pointer to the actual string and (2) the length of the string. In contrast to C, the string data is not zero-terminated but just an array of bytes [27]. As strings are immutable, slicing operations on a string are implemented by modifying the pointer and the length of the substring. The actual string data is placed in the binary or on the heap.

An *array* is a constant-length sequence of values of the same type with padding between the elements to satisfy alignment requirements. A *slice* provides a view on an array and is stored as a tuple of (1) a pointer to the underlying array, (2) the length of the slice and (3) the capacity of the underlying array [15, 32]. If the capacity and length of the slice are zero, the data pointer is allowed to be `nil` and the slice is referred to as `nil`-slice. *Maps* and *channels* are pointers to the corresponding runtime structures. The length of the map or channel is stored as integer in the first field and the capacity of a channel is stored in the second field of the runtime structure. These fields are known to the compiler and generated code can access the length and capacity directly.[2]

A *data pointer* is a simple pointer to the data, or `nil`. A *function pointer*, however, is not a pointer to the function itself but to a structure containing the function pointer and additional data. This data is used for closures to store captured variables, e.g. variables which are preserved across calls. For further details on the implementation of closures refer to the design document about function calls [9].

An *interface* is stored as a tuple of (1) a pointer to the interface metadata and (2) the actual data [7]. The metadata (`itab`) includes the type of the interface, the type of the data and a function table for the required functions. Although for some combinations of interfaces and data types the corresponding structure is already allocated in the binary, the metadata structure is filled at runtime. For the empty interface (the interface without functions) the type of the data is stored directly instead of a pointer to the metadata structure. As opposed to the description by Cox [7], it turns out that the data field of

[2] Source: `cmd/compile/internal/gc/ssa.go:3890`

Table 3.1: *Overview of the Go type system and the internal representations.*[†] *The size and alignment requirements relate to the x86-64 architecture. The internal name refers to the name of the corresponding structure in the runtime package.*[‡]

Type	Size	Align	Representation	Internal Name
Basic Types				
`bool/int8/uint8/byte`	1	1		
`int16/uint16`	2	2		
`int32/uint32/rune`	4	4		
`float32`	4	4		
`complex64`	$2 \cdot 4$	4	`[2]float32`	
`int64/uint64`	8	8		
`int/uint/uintptr`	8	8		
`float64`	8	8		
`complex128`	$2 \cdot 8$	8	`[2]float64`	
`string`	$2 \cdot 8$	8	`{ptr,len}`	`stringStruct`
`unsafe.Pointer`	8	8		
Array (`[n]T`)	–[§]	–[§]		
Slice (`[]T`)	$3 \cdot 8$	8	`{ptr,len,cap}`	`slice`
Map (`map[K]V`)	8	8	`*{len,...}`	`*hmap`
Channel (`chan T`)	8	8	`*{len,cap,...}`	`*hchan`
Pointer (`*T`)	8	8		
Function Pointer (`func(...)`)	8	8	`*{fn,...}`	`*funcval`
Empty Interface (`interface{}`)	$2 \cdot 8$	8	`{type,ptr}`	`eface`
Interface (`interface{...}`)	$2 \cdot 8$	8	`{itab,ptr}`	`iface`
Structures (`struct{...}`)	–[§]	–[§]		

[†] Sources: `src/go/types/sizes.go`, `src/cmd/compile/internal/gc/type.go`
[‡] Source: `https://golang.org/pkg/runtime/?m=all`, accessed 2017-08-18
[§] Size and alignment of structures and array types depends on the field or element types

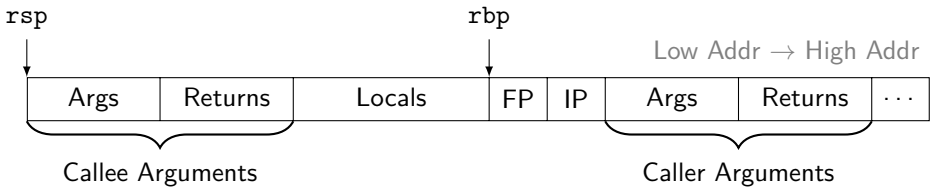


Figure 3.1: *Stackframe of a Go function. Arguments and return values are passed on the stack. The frame pointer is not used for leaf functions without local variables.*

an interface always stores a pointer to memory in contrast to storing raw data (e.g. an integer) directly, most probably to simplify garbage collection.[3]

The members of a *structure* are laid out sequentially in memory. To satisfy alignment requirements of the elements, the padding is included between the members when necessary.

3.3 Calling Convention

Contrary to other programming languages and compilers, the Go compiler does not employ the standard System-V [23] calling convention.[4] In general, all registers except `rbp` and `rsp` are considered as scratch registers. The register `rbp` is used as frame pointer to ease stack unwinding for functions, except for leaf functions without local variables.[5] Arguments and return values are passed on the stack as depicted in Figure 3.1, where the argument and return regions are both 8-byte aligned, independently of the alignment required by the individual argument or return types. The receiver of a function (i.e. the value on which a method is called) is passed internally as the first argument. The arguments do not need to be preserved, the callee can use the arguments as normal variables. Return values are initialized with zero by the callee. However, any padding between the individual arguments or return values is not necessarily initialized.

For variadic functions, where the last parameter has the type `...T`, the additional arguments are copied into a slice of type `[]T` by the caller and passed as single argument. If no additional arguments are passed, the `nil`-slice is used [32]. If a slice is passed instead of variadic arguments (via `slice...`), the slice is used as argument without copying.

As described in Section 3.2, a function pointer includes metadata in addition to the address of the function. When a function is called indirectly (closure), the address of the structure containing address and metadata (the *context* containing captured variables, `runtime.funcval` structure) is passed in the register `rdx` in addition to the arguments.

However, some runtime functions have a different calling convention: The functions to add a deferred procedure (`runtime.deferproc`), which gets automatically executed at the end of the function, and to start a new go-routine (`runtime.newproc`) have a variably-sized argument region. The size of the argument region is given as first parameter. Additionally, the function `runtime.deferproc` returns a value in the register `rax`, indicating whether

[3] Source: `cmd/compile/internal/gc/subr.go:2183`

[4] `https://github.com/golang/go/issues/16922`, accessed 2017-05-10

[5] There is an option to disable the use of the frame pointer entirely.

the function returned normally (value 0) or whether the deferred function recovered from a panic (value 1).

The special Duff-functions for zeroing (`runtime.duffzero`) or copying memory (`runtime.duffcopy`) are an unrolled loop to process up to 1024 bytes of memory.[6] The amount of memory to be processed is determined by the compiler, which emits the jump to an appropriate position *in the middle of the function*. The arguments are passed in registers: for the Duffzero function, `rdi` contains the start address and `xmm0` must be zero. For the Duffcopy function, `rsi` contains the source address and `rdi` contains the destination address. As these functions cannot set the frame pointer themselves, which is required for proper trace generation, the caller will perform the necessary stack frame setup.[7]

3.4 Static Metadata

Beside the actual code and program data, the Go compiler additionally includes a significant amount of metadata into the compiled program, mainly used for the Garbage Collector (GC) and stack trace generation.

3.4.1 Module Data

The *module data* structure is included in each executable or shared object file and contains meta-information about the executable. This includes the limits of specific sections (e.g. the code section (`text`) or the `type` section), information for the GC about the types of global variables and a pointer to the *Runtime Symbol Information* (`pclntable`). Furthermore, pointers to types and interface metadata are included.

3.4.2 Runtime Symbol Information

To (1) ease stack unwinding, (2) for the implementation of garbage collection and (3) the mapping of Program Counter (PC) value to additional meta information a Go binary contains a separate section with function metadata used by the language runtime [10].

The section begins with a constant 32-bit magic number (-5), which is used to identify the table version and the endianness of the binary. After two zero bytes follows one byte indicating the *instruction size quantum* (i.e. the required PC alignment, 1 byte on x86-64) and one byte indicating the size of a pointer (8 bytes on x86-64). This header is followed by the function table.

The table begins with a 64-bit value containing the number of entries in the table, followed by a list of function entry addresses and the offsets to the function metadata in the section, sorted by function addresses in ascending order. The remainder of the section is filled with data referred to by the offsets in an unspecified order.

The structure of the function metadata contains the name of the function (offset relative to section) and the size of the arguments. It also includes a variable-length table of per-function data, which is currently used for a list of bitmaps (`stackmap`) indicating

[6] Sources: `runtime/mkduff.go`, `runtime/duff_amd64.s`
[7] Source: `cmd/internal/obj/x86/asm6.go:3727`

which parts of the stack frame and argument region are pointers. This information can be different at different points in the function as regions on the stack can have different types. However, this information is only valid at function calls as this is the only point where the garbage collector can get triggered.

Additionally, the function metadata includes several PC-value tables, the encoding is described in the corresponding design document [10]. This currently includes mappings from the PC (1) to the stack frame size (`pcsp`), (2) the line number (`pcln`) and (3) the file name (`pcfile`), as well as (4) to the stackmap index from the function metadata.

3.4.3 Runtime Type Information

To allow for interfaces, reflection and garbage collection, the Go compiler includes type information for many types in the binary.

Beside the actual description of the type, also links to functions responsible for checking equality and computing a hash are included. The type information also includes a bitmask indicating which parts of the type in memory are pointers.[8] This is used for garbage collection. Immediately after the common type information follows type-specific information, e.g. the element type and length of an array or the signature of a function. All named or structure types include additional information (the *uncommon type*): for these the type information is followed by an additional structure, containing the methods of the type sorted by name, which are used to create the method table when casting a type to an interface, and the package path which is used to check type equality.

A method, however, has two function pointers: one for the function which has the type itself as receiver type and one for the function which has the type as it is stored in an interface structure (usually a pointer) as receiver. If the type itself fits into the 8-byte field of the interface structure (e.g. a pointer), both pointers point to the same function. Otherwise (e.g. for a structure), the compiler emits an additional wrapper function.

3.5 Runtime Memory Layout

The runtime memory layout of a Go program is designed to be deterministic. Go binaries are usually statically linked against the runtime and are linked to a fixed address. The Go runtime does not use the standard libc heap implementation but instead contains a different heap implementation at a deterministic position. This implies that Go programs in general do not benefit from Address Space Layout Randomization (ASLR), except for the stack provided by the operating system.[9]

Beside standard heap allocations, the heap also includes the stacks for the goroutines. Contrary to stacks provided by the operating system, the Go stacks have a dynamic size and can grow or shrink on demand [24]. Each function includes a prologue which checks whether the stack is large enough and eventually calls a stack enlargement function

[8] A type can also have the *GCProg* flag set in which case the bitvector is stored in a compact representation.

[9] See: `https://groups.google.com/d/msg/golang-nuts/Jd9tlNc6jUE/fJodJUqZV-YJ`, accessed 2017-08-31

(`morestack`), which switches the stack.[10] Shrinking the stack does not require a stack switch as it is a simple free operation.

The stack switching is realized by copying the stack contents to the newly allocated memory region and fixing pointers to the stack afterwards. By definition, the only place where pointers to the stack can reside at the beginning of the function is the stack itself. Therefore, the fixup function iterates through the stack and adjusts the pointers. To ensure correctness, only slots in the stack frame of the function which contain pointers are updated, this information is stored in the function metadata (see Section 3.4.2).

3.6 Safety

One of the design goals of Go is safety, which influences the language design as well as the machine code generation [31]. At language level, Go is statically typed and does in general not allow type casts between types of a different memory layout. Type assertions and type switches ensure that the value of the interface has the required type and otherwise cause a panic unless the invalid cast is caught.

During compilation, bounds checks for array and slice accesses (and slice operations) are inserted, resulting in a panic on an out-of-bounds access. When a pointer is dereferenced with an offset larger than `0x1000` (e.g. a field of a large structure), an additional `nil`-check which might cause a segmentation fault is inserted. This implies that a (safe) Go program can cause a segmentation fault. These are caught by a signal handler which also causes a panic. However, the address of an illegal memory access should be within the address range of `0x0`–`0x1000`.[11]

This `nil`-checking strategy implies that the Go compiler assumes that the page at address `0x0` is never mapped. While this is usually the case, all compiler-generated `nil`-checks will succeed and no panic occurs when this page *is* mapped, significantly increasing the likelihood of vulnerabilities.

Following Cox [8], there are two mechanisms to break the safety of the language: the unsafe package and data races. While the Go language does not provide explicit pointer arithmetic [31], pointer arithmetic and casts are possible using the *unsafe* package. Although this can improve performance and might even be necessary for some applications (e.g. to perform a bitwise cast of a floating-point value to an integer), the unsafe package can also be used to access arbitrary memory with the corresponding consequences. We add that also the *syscall* package can be used to break memory safety, a proof-of-concept for this can be found in Appendix E.

As a consequence of non-atomic multi-word structures, e.g. interfaces and slices, a *data race* can occur when one goroutine writes to the structure while another goroutine reads the structure. In this case, the reading goroutine might read the result of a partial update, breaking type safety (for interfaces) or bounds checks (for slices). These data

[10] Earlier versions of Go used *stack splitting* instead.

[11] Source: `runtime/signal_unix.go:271`, `runtime/signal_windows.go:160`

races are shown to be fully exploitable.[12] This problem can be circumvented by using synchronization [8]. To aid development, the Go tool chain includes a race detector, which instruments code and emits a warning when a data race is detected and can be activated via a compiler flag [36].

When a Go program is vulnerable, e.g. by linking against a vulnerable (C) library, exploitation is eased by a significant amount of code and gadgets combined with deterministic addresses and writable function pointers in interface tables.

3.7 Compiler

The Go compiler for the x86-64 architecture (internally referred to as `6g`) generates machine code using a strongly-typed Single-Static Assignment (SSA) intermediate stage. After parsing and verifying the input files the code for a module is transformed into the SSA form. In this stage, different light-weight optimization passes (e.g. dead code elimination and null check elimination) and lowering passes (e.g. instruction selection, register allocation) are applied. In the remainder of this section, the structure of the resulting machine code as a consequence of the applied passes will be described.[13] The results are mostly inferred from the compiler itself and analysis of compiled code.

3.7.1 Register Usage

As the calling convention does mandate the usage of almost all registers (see Section 3.3), the Go compiler is able to use almost all available registers.

General Purpose Registers For closures, the closure pointer is passed in the register `rdx`. If this pointer is no longer used or stored elsewhere, the register will be used as normal register. A pointer to the metadata structure of the goroutine (`g`) is loaded in the register `rcx` at the beginning of a function and is only used to check whether a stack enlargement is required.[14] After this check, the register will be used freely.

The compiler never uses the *high-byte* registers (e.g. `ah`) of the architecture explicitly.[15] The only case where the register `ah` is used is the result of an 8-bit multiplication. However, such an instruction will always be followed by a `mov dl,ah` instruction.[16]

Furthermore, the compiler will never use the fact that operations on the 8-bit or 16-bit registers do not clear the untouched part of the register. In fact, instructions like `movzx` are used to explicitly zero the upper part. However, the Go compiler *will* access shorter parts of a register (e.g. `ax` of `rax`) for truncations.

[12] `http://blog.stalkr.net/2015/04/golang-data-races-to-break-memory-safety.html`, accessed 2017-05-10

[13] A Go program may also contain non-compiler generated assembly code, e.g. in form of hand-written assembly, which might not follow these rules and observations.

[14] Source: `cmd/internal/obj/x86/obj6.go:981`

[15] Source: `cmd/compile/internal/sss/gen/AMD64Ops.go:17`

[16] Source: `cmd/compile/internal/amd64/ssa.go:298`

Vector Registers The Streaming SIMD Extension (SSE) registers have three functions: first, they are used for floating-point arithmetic on `float32` or `float64` values, in which case the lowest 32-bit or 64-bit are used. Second, they are sometimes used to zero small ($\leq$ 64 bytes) parts of memory, in which case a `xorps xmmn,xmmn` instruction precedes one or more 128-bit stores (`movups`).[17] Third, they are used to move small ($\leq$ 16 bytes) structures within memory, in which case a 128-bit load is followed by a 128-bit store.[18]

3.7.2 Instruction Selection

In general, the Go compiler will only emit basic arithmetic (integer and floating-point), move and control flow instructions. (A full list of instructions can be found in Appendix C.) Most importantly, memory operations are in general only performed with moves (including sign- and zero-extending variants) or atomic operations as well as string instructions (e.g. `rep movsq`).[19] Additionally, loads of 8-bit and 16-bit values are always sign-extending or zero-extending to the whole register width.[20] It is also notable that common instructions for stack operations, e.g. `push`, `pop` and `leave`, are not used. The x86-64 address operands used by memory access instructions and the source operand of a `lea` instruction is either relative to the Instruction Pointer (RIP) or composed of additions and shifts during optimizations.[21] It turned out that the stack pointer (`rsp`) is commonly used in a memory operand (including the `lea` instruction) and not as register operand. As a consequence of this, the Go compiler will also emit `lea rdi,[rsp]` instead of `mov rdi,rsp`. Likewise, even if the binary is linked to a static address, the compiler will only use RIP-relative addressing.

For many integer arithmetic operations on 8-bit and 16-bit operands the 32-bit sized instruction is used. For example, the addition of two 8-bit operands in `al` and `dl` is lowered to `add eax, edx`. For floating-point arithmetic, only the basic arithmetic and conversion instructions from the SSE instruction set are emitted. This also implies that no vector instructions are emitted. As described in Section 3.7.1, the SSE registers are also used for small memory moves and zeroing of memory.

In contrast to other compilers like GCC, the Go compiler does not emit jump tables, even for numeric switches. Instead, all kinds of switches are lowered as a binary tree of comparisons where possible. Furthermore, the Go compiler does not perform tail call optimizations for ordinary functions, implying that each function is terminated with a return instruction.

3.7.3 Status Flags

The handling of status flags in the Go compiler partially differs from the handling of other compilers: it is simply treated as additional register in the internal SSA form, with the difference that it cannot be stored in memory and must be recomputed if another

[17] Source: `cmd/compile/internal/amd64/ggen.go:129`
[18] Source: `cmd/compile/internal/ssa/gen/AMD64.rules:320`
[19] Earlier compiler versions also used memory operands also in other instructions.
[20] Source: `cmd/compile/internal/amd64/ssa.go:41`
[21] Source: `cmd/compile/internal/ssa/gen/AMD64.rules`

instruction clobbers the flags. As a consequence, only flags set by instructions without any other effects (`cmp`, `test`, `ucomiss`, `ucomisd`) are used.

The status flags set by one instruction can be used multiple times: for example, a single floating-point comparison can be used to test for "less-than" followed by a test for "Not-a-Number". Other instructions which do not set flags (e.g. a store) can occur between two flag usages and flags are also maintained over basic block boundaries. Moreover, an instruction which uses flags can depend on multiple instructions which actually set the flags.

3.7.4 Call Types

Within the compiler, there are five different types of function calls (cf. Section 3.3): first, there are *static calls* to a function within the module, the address is resolved at compile time. Second, there are *closure calls*, which are indirect calls with the closure context passed in the register `rdx`. Then, there are *go calls* and *defer calls*, which are referred to as non-normal calls[22] and correspond to the respective language constructs. Finally, there are *interface calls* (internally also referred to as *intercalls*), where the address of the function is loaded from the interface metadata.

3.7.5 Compiler-generated Checks

A compiler-generated `nil`-check is lowered to the `test al,[rax]` instruction.[23] If the pointer in `rax` points to invalid memory, this instruction will cause a segmentation fault, which will be caught using a signal handler and propagated as a panic.

Bounds checks for indexing[24] and slicing[25] of arrays are lowered to conditional branches to corresponding internal panic functions in the runtime. Type assertions are lowered to conditional branches to panic functions or to runtime functions for interface type assertions, which also have variants returning whether the type assertion was successful instead of panicking.[26] A type switch generates a binary search over the hash of the type.[27]

3.7.6 Runtime Support

Beside compiler-generated checks, many language features heavily rely on runtime support.[28] This includes the allocation of objects and slices as well as conversions of interfaces, but also most functions and language constructs operating on string, hash maps and channels. Most notably, many of these functions not only take a pointer to the actual object as argument but also require an explicit pointer to the type of the object. For

[22] Source: `cmd/compile/internal/gc/ssa.go:2546`
[23] Source: `cmd/compile/internal/amd64/ssa.go:865`
[24] Source: `cmd/compile/internal/gc/ssa.go:3288`
[25] Source: `cmd/compile/internal/gc/ssa.go:3301`
[26] Source: `cmd/compile/internal/gc/ssa.go:4055`
[27] Source: `cmd/compile/internal/gc/swt.go:677`
[28] Source: `cmd/compile/internal/gc/builtin/runtime.go`

example, almost all operations on maps are runtime functions, which take the type of the map as first argument.

To allow concurrent garbage collection, the compiler emits write barriers for writes of pointer types to the heap. These write barriers are necessary in order to give the garbage collector a consistent view of memory. To improve performance, the write barriers are controlled using a global variable: if a global variable (`writebarrier.enabled`) is set, the memory operation, i.e. a pointer write, memory move or memory zero, is performed using runtime functions instead of performing the operation using normal instructions.[29] The runtime functions for memory moves and zeros take the type of the object to move as additional parameter.[30]

[29] Source: `cmd/compile/internal/gc/writebarrier.go:29`
[30] Source: `cmd/compile/internal/gc/builtin/runtime.go`

4 Design

Binaries produced by the Go compiler include additional metadata compared to other compilers and languages. Therefore, we will first describe our strategy to identify the version of the Go compiler and extract important metadata even from stripped binaries in detail. Then, we will describe our intermediate representation for code, an extended assembly language abstracting some specifics of the Go compiler, as well as a procedure to lift Go code of different versions into this representation. Based on this, we will finally present our approach for the analysis of function types using type constraints.

4.1 Extracting Metadata

An important step in analyzing Go binaries consists in finding and extracting the included metadata described in Section 3.4. While the metadata is mostly used to enable garbage collection and dynamic typing via interfaces, it also provides information about used datatypes and functions of the runtime or other packages.

It is entirely possible that the included metadata is not correct and that the included information to be modified by another program after compilation. While the range of potential modifications is limited to maintain the correctness of the program, it is possible to modify the names of the functions, the mapping of the PC to lines and files as well as the names and package paths of defined datatypes and the corresponding methods.

In the following, we will assume that the metadata in the binary is produced by the compiler and contained in the binary in unmodified form. We will particularly use the name of the functions to identify special runtime functions. We note that these runtime functions can also be identified by other means, e.g. using simple signatures [34] or more complex pattern matching strategies [20]. Additionally, to the best of our knowledge, there does not exist a publicly available tool which modifies or removes this metadata.

The proposed and implemented extraction schema is designed to handle executables in the ELF (non-PIC) and the PE file format on the x86-64 architecture generated by the Gc/6g compiler starting from Go 1.5 (released in 08/2015 [33]), which introduced the module data structure. However, some parts may also be applicable for older Go versions. For example, the symbol information is present in the current form starting from Go 1.2 [10].

Implementation Remark: The metadata parsing is implemented in Go to ease reuse of compiler and runtime data structures. While it is not possible to use these structures directly as they are private to the runtime, only minor modifications are required. The output of the extraction process is a file in JSON [13] format containing the parsed metadata as well as other information about the binary file, e.g. the compiler version.

4.1.1 Runtime Symbol Information

The easiest way to find the runtime symbol information (internally referred to as *pclntab*, abbreviation for PC-line number table) is to find the symbol `runtime.pclntab` in the binary file. While this method is straight-forward, it does not work when the binary is compiled without symbol information or stripped afterwards. For non-position-independent ELF executables, the *pclntab* can be found at the beginning of a separate section named `.gopclntab`. This, however, is not possible for PIC-compiled code, where the table is placed in the RELRO-section, or other file formats including the PE format used on Windows.

If the *pclntab* cannot be found using symbols or the corresponding section, potential candidates can be found in the `.text` section (for PE files) or the `.data.rel.ro` section (for ELF files) by searching for an 8-byte aligned magic number. Similar to the verification procedure in the runtime[1], we can verify the instruction alignment and pointer size and can ensure that the entries in the function table are sorted by entry address in increasing order to avoid false positives.

The actual parsing and processing of the encoded data is very similar to the handling in the runtime.[2] Although the file table is included in the *pclntab*, the offset is not known directly and must be fetched from the module data structure.

4.1.2 Module Data Structure

As for the runtime symbol information, the easiest way of finding the module data structure is to find the `runtime.firstmoduledata` symbol in the binary. If the symbol is not present, we can use a signature-based approach to find the structure in the appropriate section (`noptrdata` for ELF files, `data` for PE files). This approach is based on the information from the runtime symbol information: the first fields of the module data structure include the known address of the *pclntab* and its (unknown) length, the address and length of the function table, which is stored at the beginning of the *pclntab* and therefore known, and the minimum and maximum code address, which are known from the function table and possibly also from section information from the binary. Additionally, we know that the length and the capacity of slices must be equal as the slices point into read-only memory. Using this information, we were able to find the mentioned structure in executables ranging from Go 1.6 to Go 1.8.

Alternatively, the module data structure can be found via the function `runtime.findmoduledatap`, which references the structure in the code. We did prefer the signature-based approach as we expect that the code generated for this function is more likely to change in future than the layout of the structure.

Beside finding the module data structure, it is critical to distinguish between (currently) three different versions of this structure. The structure was introduced in Go 1.5 and contained information about the function table, the layout of the executable, pointers to some types, and bitmasks for the data section for the garbage collector. Go 1.7 extended

[1] See: `src/runtime/symtab.go:347`
[2] See: `src/runtime/symtab.go`

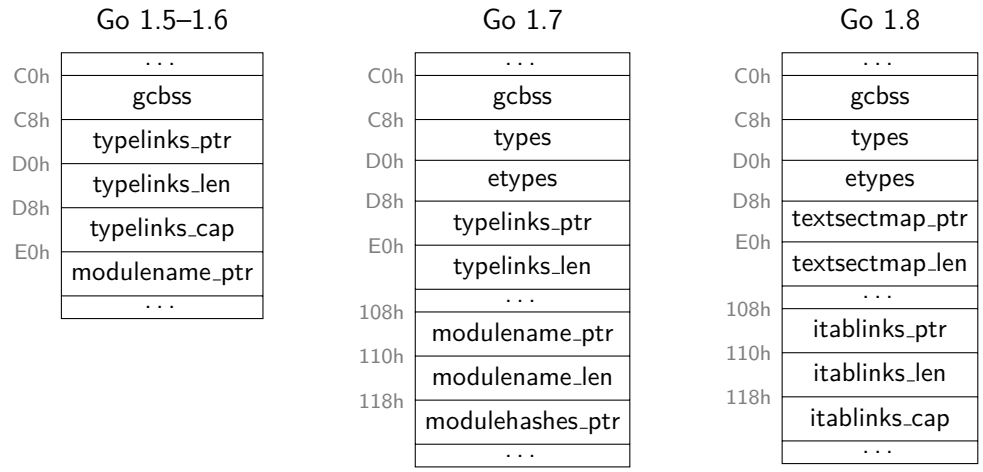


Figure 4.1: *Layout of different versions of the module data structure, distinguish Go 1.5–1.6, Go 1.7 and Go 1.8.*

this structure by links to interface tables and replaced the pointers to the types with offsets to reduce the binary size. In Go 1.8, support for plugins was implemented, which allow to load code dynamically at runtime by wrapping `dlopen`.[3] This required additional fields in the module data structure.

The version identification can be implemented using sanity checks for values at certain offsets in the structure, the layout of the different versions of the structure is shown in Figure 4.1. A summary of the described procedure is shown in Algorithm 1. Distinguishing the structure of Go 1.5–1.6 and later versions is rather simple: at the offset where the `typelinks` slice, being a triple of pointer, length and capacity, is stored in Go 1.5, Go 1.7 and later versions store the start and end address of the types section. As the end address of the section must be larger than the start address of the section, but the pointer to the `typelinks` slice must be larger than the length of the slice, we have a simple way to distinguish Go 1.5–1.6 from later versions. We note that this assumption is a consequence of the binary layout, where the typelinks slice follows after the types section and each type is larger than an entry in the slice.

Distinguishing the structure of Go 1.7 from Go 1.8 is less clear. We observe that

[3] See `https://golang.org/pkg/plugin/`, accessed 2017-08-21

Algorithm 1 Procedure to identify Go version using the module data structure.

if $data\,[0xC8] > data\,[0xD0]$ and $data\,[0xD0] = data\,[0xD8]$ **then**
  return Go 1.5–1.6
else if $data\,[0x110] = 0$ or $data\,[0x110] \neq data\,[0x118]$ **then**
  return Go 1.7
else
  return Go 1.8+
end if

every Go binary has at least one entry in the interface table, namely the built-in `error` type. Furthermore, we note that the `modulename` string is empty (i.e. length zero) for executables. Thus, if the length of the `itablinks` slice in the layout of Go 1.8 would be zero, we know that the module data structure has the layout of Go 1.7.

These simple checks turned out to be sufficient for the released Go versions, starting from version 1.5. If the layout has changes in future versions which cannot be detected using these simple means, more involved approaches can be employed to identify the version of the Go compiler. For example, the presence, absence or code of specific runtime functions can be used for identification as well. We did not implement these more accurate techniques as they seem to be unnecessarily complex without providing any benefit at this time.

Remark Using the described strategy, we are able to distinguish the Go versions 1.5–1.6, 1.7 and 1.8. The versions 1.5 and 1.6 can be easily distinguished using the implementation of Duff's device.

4.1.3 Type Information

Each Go binary contains a significant amount of runtime type information, which are required for interfaces and reflection but also assist in binary analysis. The module data structure stores links to all types, except for basic types like `int`, interfaces and named types. The interface types can be enumerated using the interface table pointers, which are also contained in the module data structures. For named types, it turns out that most types can be found using nested links of processed types. For example, a pointer type to a named structure contains a pointer to the type description of the structure.

Implementation Remark: In the rare event that a type is not detected automatically, the type parsing can be triggered manually by supplying the address of the type using the `-rtti-types` command line flag.

The actual parsing of the type information follows the description and implementation in the runtime and is a straight-forward inversion of the compilation process.

4.2 Intermediate Code Representation

Analysis of binary code typically employs an Intermediate Representation (IR) to reduce the variety of instructions and model side-effects explicitly, e.g. the IR of BAP [6] or VEX as used by Angr [29].

However, existing IRs have major drawbacks when analyzing Go binaries: first, the compiler only uses a small subset of architectural features and rarely relies on instruction side-effects except for status flags (cf. Section 3.7), implying that a high-level IR only provides a marginal benefit. Second, the Go compiler lowers specific language constructs to known sequences of instructions or calls with special calling conventions, which would be harder to identify in an abstract IR (cf. also Section 3.3). Additionally, the IR which

is used by the Go compiler internally is not suitable for analysis, as it does not provide a stable API and requires complete type information, which is not readily available from the binary.

Therefore, we decided to perform analysis of the assembly code on a modified assembly language referred to as Higher-level Go Assembly (HGA), which contains elements of the x86-64 assembly, staying close to the output of the compiler and the IR of the Go compiler, modeling special constructs, e.g. stack bounds checks or Duff's device, appropriately. The scope of the HGA is limited to code generated from Go functions, excluding bindings to C functions and automatically generated wrapper functions. As the lifting process consists of multiple stages operating on the HGA, we distinguish between the *general HGA*, which is merely the x86-64 assembly cut in basic blocks, and the *canonical HGA*, which is used for later analysis and therefore more explicit and simplified.

We did not opt for an SSA form, in contrast to the proposed analysis strategy by van Emmerik [35] and also by the IR used by the Go compiler, as we do not see a benefit outweighing the increased complexity of lifting. We note that transforming the canonical HGA into an abstract SSA form is still possible and also seems advantageous compared to lifting assembly directly as complex instruction sequences are already removed.

4.2.1 Structure

The HGA is used to model a single Go function, consisting of multiple *basic blocks* being a continuous sequence of instructions. Depending on the last instruction, a basic block can have at most two exits (*successors*): one exit which is used when a (conditional) jump is taken, and one exit which is used when a conditional jump is not taken. In special cases, e.g. when a panic function is called, a basic block is allowed to have no successors. A basic block must end with a jump instruction or an "unreachable" indicator. All basic blocks except the entry have at least one predecessor. In the canonical form, the entry basic block must have no predecessors. To ease lifting and analysis by the insertion of fix-up code, the canonical HGA must not contain a branch from a basic block with multiple successors to a basic block with multiple predecessors. (These edges in the Control Flow Graph (CFG) are referred to as *critical edges*.) This can be ensured by replacing critical edges in the CFG with an empty basic block and therefore does not form a limitation (see also Figure 4.4).

4.2.2 Registers

In the HGA, all 16 general purpose registers and 16 SSE registers can be accessed in any valid size. The stack pointer register (`SP`), however, always points to the *top* of the local stack frame, regardless of modifications. Additionally, there is a *static base* register which points to the base address of the image in memory and an explicit flags register containing the status flags. As the Go compiler only uses the flags for comparisons, we model the flags as single abstract register instead of modeling the individual flags. For the purpose of lifting x86-64 assembly into the canonical HGA, there are additional non-architectural special purpose registers, named `TMP1`–`TMP5`.

Even though a register can be written in any size, it must be accessed in the size it was

```
mov rax,[rcx+8*rdi+0x10]
```

```
MOVQ TMP1~8,DI~8
SHLQ TMP1~8,$0x3~8
ADDQ TMP1~8,CX~8
ADDQ TMP1~8,$0x10~8
MOVQload AX~8,[TMP1]~8
```

(a) *Complex memory operands are split into multiple simple instructions.*

```
lea rcx,[rax+0x18]
```

```
MOVQ TMP1~8,AX~8
ADDQ TMP1~8,$0x18~8
MOVQ CX~8,TMP1~8
```

(b) *The* `lea` *instruction is replaced with a plain move.*

```
mov bl,[rsp+rax+0x20]
lea rdi,[rip+<off>]
```

```
MOVBload BX~1,[SP+AX+$0x20]~1
LEAQ DI~8,[SB+$<addr>]~0
```

(c) *The stack and global variables is accessed using dedicated operand types.*

Figure 4.2: *Examples for lifting of x86-64 memory operands into the HGA*

written before in the canonical HGA. Implicit truncations (e.g. accessing `ax` of `rax`) or extensions are not allowed. In the canonical form, the stack pointer (`SP`) must not be used as direct register operand, except for the stack frame setup.

4.2.3 Instruction Operands

The supported instruction operands of the HGA are kept closely to the operands provided by the x86-64 architecture. This includes register and immediate operands. However, to simplify analysis, the HGA does not provide a single complex memory operand but provides three more simple forms instead: first, there are simple memory operands, where the address is stored in a register. Second, there is stack-based memory operands, where the address is computed relative to the stack frame of the function, consisting of a constant offset and an optional register offset. This type of addressing is kept separately to ease stack frame analysis. Third, RIP-relative addressing is lifted to global memory operands, which are relative to the static base register.

Complex x86-64 memory operands, with the exception of stack-relative operands, are lifted by inserting sequences of additions and shifts on the non-architectural register `TMP1`, which is then used as simple memory operand. Examples for this procedure are shown in Figure 4.2. This is also the opposite of the way complex memory operands are constructed while compiling: as shown in Section 3.7.2, this type of operand is formed as combination of the underlying operations.

4.2.4 Instructions

An HGA instruction is defined by an opcode and operands. The instruction types are a mixture of x86-64 instructions, internal instructions of the Go compiler as well as additional instructions. An overview about the existing instructions can be found in Table 4.1, for a complete list of instruction see Appendix D.3. The instruction types can be roughly classified in four categories:

1. **Data Processing Instructions:** These instructions operate on registers and immediate values only and includes most x86-64 instructions, e.g. `ADDQ`. Multiplication

Table 4.1: *Overview of the provided instruction types of the HGA, grouped by functional classification and naming origin. The suffix of most instructions indicates the operand size. Pseudo instructions have no significant effect to the program state.*

Class	x86-64 ISA	Go Compiler	Miscellaneous
Data	`ADDB`, `ADDW`, `ADDL`, `ADDQ`, `SUBQ`, `MOVQ`, `MOVLQZX`, `MOVBWSX`, `SETcc`, ...	`MULB`, `MULQ`, `MULQU2`, `HMULW`, `HMULWU`, `NilCheck`, `GetG`, `GetClosurePtr`, `Trunc16to8`, `Trunc64to32` ...	`CheckIndex`, `CheckSlice`, `CheckDivide`, `GetBP`,`GetSP`,...
Memory	`LEAQ`, `MOVQload`, `REPSTOSQ`*, ...	`MemZero`, `DuffCopy`*, `DuffZero`*	
Control Flow	`JMP`, `Jcc`, `CALL`*, `RET`	`StaticCall`, `GoCall`, `DeferCall`, `ClosureCall`, `InterCall`	`Unreachable`
Pseudo	`NOP`		`MoreStack`, `MoreStackCtxt`, `CmpStackLimit`, `AddressMarker`, `SizeAssert`*

* non-canonical HGA only

and division instructions however follow the internal Go naming convention. Additional instructions for bounds and division checks (`CheckIndex` et al.) serve as replacement for conditional calls to the respective panic functions in the runtime and therefore simplify the control flow. Furthermore, there are special instructions to set the values of registers (e.g. `GetBP`) in the beginning of a function, mainly to ensure that the source of each register is well-defined.

2. **Memory Instructions:** In contrast to the x86-64 instructions, memory can only be accessed using explicit load and store instructions (cf. Section 3.7.2). As there is no complex memory operand in the HGA, the `LEAQ` instruction is only used for address generation of stack-relative or global addresses and replaced with a plain move otherwise. Special sequences of instructions (e.g. for Duff's device) are compacted to a single instruction.

3. **Control Flow Instructions:** These includes the jump instructions `JMP` and `Jcc`. In contrast to the x86-64 ISA, conditional jumps have two targets and no implicit fall-through. Control flow instructions also include function calls and returns. As the Go compiler internally distinguishes five different types of calls (see Section 3.7.4), each type has a different instruction to simplify later analysis. Additionally, there is a special `Unreachable` instruction, which is used after a call to a panic function and indicates that the basic block will not terminate regularly. Although this instruction has similar semantics as the `ud2` instruction on x86-64, we named it differently for

clarity.

4. **Pseudo Instructions:** In addition to the other categories, the HGA also includes non-functional instructions, mainly for simplicity, temporary analysis and debugging. Conditional calls to stack enlargement routines are compacted into one unconditional instruction, which has no further effect in analysis. The `AddressMarker` contains the address and stack frame size of the following instructions in the original binary. However, the address is only used for debugging purposes.

Some instructions like `CALL` can only occur in the non-canonical form and will be removed or replaced during the lifting process.

4.3 Code Lifting

After disassembling the function, the lifting into the HGA happens in multiple *passes*. After the control flow graph is recovered, the first pass lifts the x86-64 assembly into the general HGA while subsequent passes perform further simplifications and clean-ups, resulting in a canonical HGA for the function. The lifting procedure is designed for the newer SSA code generation back-end of the Go compiler, introduced in Go 1.7. The code generator from older Go versions is supported partially: while the described lifting procedure will work in many cases, some rare cases are not be handled yet, see Section 4.3.2.

Implementation Remark: The implementation of the HGA and the lifting process is written in Python 3 as more bindings to other libraries (e.g. Z3 [11]) are available.

4.3.1 Control Flow Recovery

Before the x86-64 instructions can be lifted, it is necessary to recover the control flow graph to form the structure of basic blocks in the resulting HGA representation of the function. For the Go compiler, it turned out that a linear disassembly of the function, whose address and size are known from the function table, is sufficient. A separate basic block is started at the entry of the function, at targets of jumps and after conditional jumps. This is sufficient, as the Go compiler does not emit jump tables. A basic block is terminated on a jump instruction, a return instruction or a `ud2` instruction, which is used to indicate that the current instruction is unreachable. Additionally, a basic block is terminated at an instruction which starts a new basic block, e.g. by being a target of another jump. While this is not strictly necessary for a semantically correct lifting, it prevents code duplication and therefore eases automated as well as manual analysis.

4.3.2 Initial Lifting

The initial lifting pass basically replaces each x86-64 instruction with one or more corresponding HGA instructions, where complex memory operands are replaced with a sequence of simple instructions, see Section 4.2.3. At the boundaries of x86-64 instructions special markers are inserted, which indicate the address of the instruction and the stack frame

Table 4.2: *Special cases while lifting single x86-64 instructions to the HGA for Go 1.7+.*

x86-64 Instruction	HGA Instruction	Comment
`mov <reg>,fs:[-8]`	`GetG <reg>`	Special access to goroutine data
`mov dl,ah`	—	Fix-up code for multiplications
`mov <reg>,rsp`	`LEAQ <reg>,[SP-...]`	Consistency with stack accesses
`xor <reg>,<reg>`	`MOVL <reg>,$0`	Break dependency* of zero-idiom
`xorps <reg>,<reg>`	`MOVO <reg>,$0`	Break dependency* of zero-idiom
`sbb <reg>,<reg>`	`SBBLcarrymask <reg>`	Break dependency* for shift masks
`movzx <reg>,<mem>`	Load + Zero-Extend	Explicit semantics
`movsx <reg>,<mem>`	Load + Sign-Extend	Explicit semantics
`test al,[<reg>]`	`NilCheck <reg>`	Explicit `nil`-checks, see Sec. 3.7.5
`test <reg>,<imm>`	`AND` + `CMP`	Explicit semantics
`cmp <reg>,[rcx+0x10]`	`CmpStackLimit`	Stack bounds checking
`cmp <reg>,[rcx+0x18]`	`CmpStackLimit`	Stack bounds checking
`test <reg>,<reg>`	`CMP <reg>,$0`	Compare with zero

* The original instruction technically has a read dependency on a register, although the result is independent of the original value of that register. The replacement removes this false dependency.

size. Additionally, some special cases of instructions (see Appendix C) are handled during this pass, an overview can be found in Table 4.2. Most notably, the implicit extension of 32-bit registers is handled at a later stage in the lifting process.

Handling Older Go Versions

Older versions of the Go compiler emit different code for some higher-level instructions. We currently handle two common cases: first, in some cases a memory operand is used in combination with an arithmetic instruction. This can be lifted to the HGA by splitting the memory operations and the arithmetic operation and using a temporary register (`TMP5`) to hold the intermediate result. Second, `nil`-checks are implemented using a conditional branch and an explicit load from address zero. This instruction sequence is detected and replaced with a single `NilCheck` instruction. Currently unhandled cases include larger memory moves and big integers, which are left as future work. We note that operations on big integers only occur inside the `math/big` package of the standard library.

4.3.3 Dead-end Block Elimination

The Go compiler lowers bounds and type checks by inserting a comparison followed by a conditional jump to a basic block which calls the appropriate panic function in the runtime, see Figure 4.3. These conditional branches provides important meta information about arrays, slices and types and requires special handling to ease later analysis. Furthermore, most functions contain a stack bounds check in the prologue with a conditional call to a stack enlargement function (see Section 3.5). This additional basic block only indicates

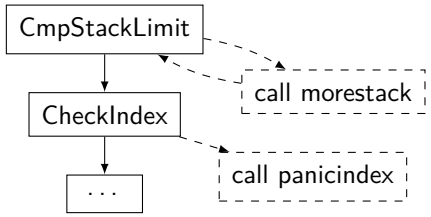


Figure 4.3: *The dashed basic blocks with the call to the panic function for indexing and the call to the stack enlargement function will be removed, simplifying the control flow.*

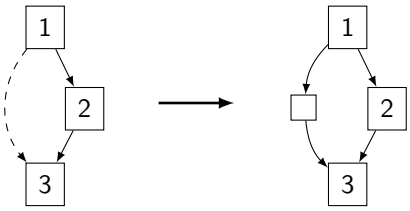


Figure 4.4: *The critical edge from block 1, which has multiple successors, to block 3, which has multiple predecessors, will be replaced with an empty basic block.*

whether the function is a closure (closures call a different stack enlargement function), but has no further use and has to be removed to lift the function into the canonical HGA. Consequently, it can be considered as beneficial to remove these basic blocks and keep the provided meta information in form of additional instructions.

For bounds checks, the panicking basic block can be removed easily. The corresponding jump is simply replaced with an unconditional jump. Behind all instructions which set the flags for the conditional jump to the panicking block, an additional `CheckIndex` or `CheckSlice` instruction is inserted. The operands of these instructions are the same as for the comparison, regarding a potential inversion of the branching condition. We note that a conditional jump can depend on multiple instructions which set the flags (see Section 3.7.3) and therefore multiple check instructions can be inserted.

The basic block which contains the call to the stack enlargement routine can be removed and the conditional branch is replaced by a `MoreStack` or `MoreStackCtxt` instruction.

4.3.4 Critical Edge Elimination

To simplify later stages of lifting and analysis, the canonical HGA does not allow edges in the CFG from a block with multiple successors to a block with multiple predecessors. In the internal SSA form of the Go compiler critical edges are not allowed as well, but may get introduced during the lowering process. Eliminating critical edges can be done in a straight-forward manner (see Figure 4.4): the edge in the CFG is simply replaced by a new block with a single branch instruction to the original branch target. The branch at the end of the preceding block is adjusted to point to the new block instead.

4.3.5 Function Calls

The Go compiler emits different types of calls, see Section 3.7.4. Static calls, which jump to an address known at compile time, are simply replaced by a `StaticCall` instruction. Indirect calls (i.e. `call` *`reg`*) can be either closure or interface calls. It turned out that the Go compiler handles the wrapping interface or closure independently of the contained function value, which implies that these types cannot be distinguished by analyzing instruction sequences only.

However, for closure calls, the closure pointer must be passed in the register `rdx`, as defined by the calling convention. The caller will not use the contents of the closure

pointer, except for loading the actual function pointer once. Therefore, a simple approach to distinguish closure calls and interface calls is to analyze the usage of the `rdx` register before the call: if the register is set and only used to load the address of the call or not used at all, the call *can* be a closure call and must be an interface call otherwise. It might happen that the register `rdx` is set as unused side-effect of another instruction, e.g. `imul`. Thus, we only declare a call as closure call when the closure pointer register is set by a `mov` instruction. Depending on the result of this detection procedure, we replace the call instruction with a `ClosureCall` or `InterCall` instruction. We note that this detection technique is solely based on the code generation habits of the compiler. For future Go versions this heuristic may be inappropriate and more sophisticated detection techniques might be required.

As described in Section 3.3, the Go compiler will emit function calls to some runtime functions which do not follow the Go calling convention. The functions `runtime.newproc` and `runtime.deferproc` have a variably sized argument region, where the first argument on the stack will be set to a constant describing the size of the arguments. Calls to these runtime functions will be replaced with the `GoCall` and `DeferCall` instruction respectively, which contains the argument region as second operand.

4.3.6 Calls to Duff's Device

The remaining functions not following the calling convention are the functions implementing Duff's device for zeroing and copying memory, to which calls are generated automatically by the compiler. Calls into the `runtime.duffcopy` function and the corresponding arguments can be detected easily and are replaced with a `DuffCopy` instruction, which encodes the size of the copied memory region explicitly.

For the `runtime.duffzero` function, the Go compiler emits calls in two different stages during compilation with different assumptions: first, normal memory zeroing instructions emitted in the IR can be lowered to this function. In this case, the side-effects of the function are not used. Second, this function is called to initialize stack memory in the function prologue. An example for this case is shown in Listing 4.1. Here, side-effects on the registers of the function might be used and further care has to be taken. Moreover, the implementation of the Duffzero function requires pointer adjustments in some cases. Thus, lifting the call into a `DuffZero` instruction requires additional analysis of pointer adjustments.

For both types of function calls, the stack frame setup in the caller can be safely removed and the register operands can be moved directly into the instruction.

4.3.7 Instruction Sizes

For some instruction types and sizes, the Go compiler emits an instruction with a wider register size than in the original source, see Listing 4.3 for an example and Section 3.7.2. From the instructions operating on general purpose registers, the register operands of twelve instruction types[4] use a register size of 4-bytes instead of 1-byte or 2-byte registers, and from the SSE instructions, the `movups` instruction is used with 16-byte operands

[4]Instruction types with register widening: `mov`, `add`, `sub`, `neg`, `imul`, `shl`, `not`, `and`, `or`, `xor` and `sbb`

```
1  // Arguments for duffzero
2  MOVO        X0~16,$0x0~16
3  LEAQ        DI~8,[SP+$0x10]~0
4  // Frame pointer setup and call
5  MOVQstore   [SP+$-0x10]~8,BP~8
6  LEAQ        BP~8,[SP+$-0x10]~0
7  CALL        $runtime.duffzero+0x11d
8  MOVQload    BP~8,[BP]~8
9  // Modified register DI is used
10 MOVQ        TMP1~8,DI~8
11 ADDQ        TMP1~8,$0xfffffffffffffff8~8
12 MOVOstore   [TMP1]~16,X0~16
```

Listing 4.1: *Side-effects of calls to Duffzero might be used.*

```
1 DuffZero    [SP+$0x10]~72
```

Listing 4.2: *Lifted code for Listing 4.1.*

```
1 MOVWload    AX~2,[SP+$0xa]~2
2 MOVWload    CX~2,[SP+$0x8]~2
3 ADDL        AX~4,CX~4
4 MOVWstore   [SP+$0x10]~2,AX~2
5 RET
```

Listing 4.3: *Addition of two 16-bit integers. The* `add` *instruction operates on the 32-bit registers.*

instead of moves with 4-byte or 8-byte operands. We will refer to cases where the instruction size is potentially wider than mandated by the source as *widened instructions*.

While this widening does not change the result (the higher part is not used), it increases the difficulty of analysis, as the higher part of the source operands is possibly undefined. On the other side, the canonical HGA requires instructions to only read registers in the size they were written before. Therefore, lifting compiled Go code into the canonical HGA requires a detection and handling of widened instructions.

The main idea to reconstruct the real size of widened instructions is to analyze the maximum use size of the result and the minimum source size of the input operands. As, however, uses and sources may be widened instructions as well, an iterative approach is required, which also handles circular dependencies as they can occur in loops.

The procedure to reduce the size of (potentially) widened instructions to the real instruction size works as follows: while widened instructions exist, we iterate over the unhandled widened instructions. If the result of the instruction is not used, we can safely remove the instruction. If the result of the instruction *is not* used by other widened instructions, then we set the real size to the maximum size of use. If the result *is* used by an widened instruction, we skip the instruction and continue. However, if in the last iteration (over all instructions) no further information was gained, we use the minimum size of the instructions setting the source operands, excluding widened instructions, unless there are no such instructions. If, for some reason, the last two iterations did not provide any information, we give up and use the large instruction size as is. This can happen if two widened instruction mutually depend on each other, e.g. in a loop.

In addition, we have to handle the implicit zero-extension when writing 32-bit registers in the x86-64 architecture: if the result of an instruction writing to a general purpose register is used with a size of 8-bytes, we still limit the size of the instruction to 4-bytes.

4.3.8 Operand Sizes

The x86-64 architecture provides implicit truncations and zero-extensions of the general-purpose registers, which have to be made explicit in the canonical HGA.

To achieve this, an appropriate conversion instruction is inserted before the use of the register. If the original register is not used afterwards, the conversion can take in-place.

Otherwise a temporary register (`TMP2` and `TMP3`) has to be used, as the value might be used in a different size as well.

Basic blocks with multiple predecessors require special handling: it can happen that a the value of a register is set in different sizes, depending on the predecessors. Therefore, the necessary conversions must be done in the predecessors.[5] For this purpose, we iterate over the combinations of basic blocks with multiple predecessors and general-purpose registers. If the register value is (potentially) used, we insert a `SizeAssert` pseudo-instruction with the maximum size of use at the end of all preceding basic blocks. This is possible as critical edges in the CFG were removed and the only successor of all preceding blocks is the current block itself. After the insertion of the instructions, the conversion instructions can be inserted as described. Finally, the pseudo-instructions are removed again as they have no further use.

We note that this does not cover truncations which occur by loading the lower half of an integer from memory. These cases have to be handled during the analysis when memory disambiguation is performed.

4.3.9 Miscellaneous

In addition to the described transformations, some minor modifications are also performed. Special sequences of instructions are replaced with a high-level instruction as in the Go compiler. For example, the sequence `add+rcr`, which is used to compute the average of two numbers, is replaced with an `AVGQU` instruction. Furthermore, instructions which store the address of a type description in a register are replaced with dedicated `TypePtr` instructions to explicitly identify the purpose of the instruction. Zeroing of memory regions is detected and replaced with a special `MemZero` instruction to avoid the problem of finding a type for a zero value which is used for different types. Finally, some basic simplifications of the resulting HGA are performed: unused jumps are eliminated, extensions and truncations are merged where possible, and unused instructions (e.g. comparisons where the jumps have been replaced with bounds checking instructions) are removed.

4.4 Argument Region Analysis

Arguments and return values of functions are both passed on the stack in adjacent regions, which is a rather unique strategy compared to other calling conventions of other compilers and languages. For analyzing functions, it is therefore important to distinguish between the argument and return region. This is a required step to infer the type of the function and also necessary to implement analysis of function calls. In theory, there are two possible ways to get this information: the places where the function gets called (*caller-based* analysis) and the function itself (*callee-based* analysis).

4.4.1 Caller-based Analysis

A caller-based analysis tries to infer information from the call to a function, but suffers from the problem that hardly any information is available. For interface and closure

[5] In the HGA, conversion instructions specify the target size as well as the source size explicitly.

```
// This store is no direct argument,
// only a pointer is passed
MOVQstore  [SP+$0x18]~8,AX~8
LEAQ       AX~8,[SP+$0x18]~0
MOVQstore  [SP+$0x0],AX~8
MOVQstore  [SP+$0x8],$1~8
MOVQstore  [SP+$0x10],$1~8
// Actual signature: func(...int)
CALL       CX~8
```

Listing 4.4: *Indirect arguments, e.g. for variadic functions, are no explicit argument even though they are stored and never read.*

```
// Only stored to keep pointer alive
MOVQstore  [SP+$0]~8,CX~8
CALL       DX~8 // Real: func()
```

Listing 4.5: *The register `CX` is only stored to keep the pointer alive and prevent early garbage collection.*

```
MOVQstore  [SP+$0x8]~8,DX~8
MOVQstore  [SP+$0x0],AX~8
CALL       CX~8 // Real: func(int)
// Is this the value stored above
// or a return value?
MOVQload   DI~8,[SP+$0x8]~8
```

Listing 4.6: *It is not possible to distinguish local variables and return values*

calls, not even the size of the argument and return region is known. The only available information is the usage of the stack frame: if a part in the stack frame is written before a call and never read afterwards (arguments can be overwritten by the callee), it is *probably* a argument. However, even this basic assumption has major problems: for variadic functions, some arguments are stored in the stack and only a pointer to these arguments is passed to the function as argument (Listing 4.4). These variadic arguments are never read after being written but are no explicit arguments of the called function. Furthermore, the case that a register is spilled to the stack only to keep a pointer alive cannot be detected at all (Listing 4.5). Additionally, there is no guarantee that the arguments are set directly before the call: the compiler is allowed to do arbitrary transformations as long as the correctness of the program is maintained. Thus, it can also happen that a argument is set in a different basic block or even in a conditional branch, significantly increasing the difficulty of analysis.

For return values, it turned out to be impossible to infer any reliable information. If a value from the stack frame is read after the call, it can be either a return value of the called function or a local variable of the calling function (Listing 4.6). We were not able to find a way to distinguish these cases.

Given that the information provided by a caller-based analysis would be highly unreliable, especially for calls where the call target is not known, we decided to not implement a caller-based analysis of function types. When the call target is known, it seems to be more reliable to infer information directly from the callee. For interface and closure calls, the necessary information can be inferred from other sources, e.g. when the interface or the closure is constructed.

4.4.2 Callee-based Analysis

The callee-based analysis infers information about arguments and returns from the access pattern of the function and has information about the size of the argument and return region. It is mainly based on two observations: first, all return values must be initialized

by the callee in any case and also must not be used before the initialization. Second, slots in the argument region which can be read before being written must be arguments, as all reads must be well-defined in Go.

While these assumptions generally lead to correct results, this analysis fails in the unusual case that an argument is always written before it is read, i.e. used as additional local variable. If such a region is followed by other used arguments, we can still determine that this is a parameter because arguments and return values are not spatially mixed. Otherwise, if these unusual arguments are located right before the return region, it turned out to be impossible to reason about argument or return value by a simple access analysis. It might be possible to infer information from code generation habits of the compiler, e.g. that return values are usually written at once, either initialized in the beginning or set at the end of the function. We did not implement such detection heuristics as they seem to be too unreliable and note again that the compiler is free to perform arbitrary code transformations as long as the program remains correct.

Despite that the case that the last arguments are only used as local variables is very unlikely to occur in practice, it will also have no significant impact for further analysis: if an unused argument is misclassified as return value, we only lose potential information about the unused value passed as argument. There is no impact from the classification as return value since the value will never be used.

4.5 Basic Type Analysis

The goal of the basic type analysis is to infer information about the parameter and return types of a function, the types of local variables on the stack and the data types stored in memory. We will only focus on basic types, e.g. integers, pointers or floating-point values, and their flat layout in memory, but will not analyze nested structures.

Type Information Sources There are different sources for type information in the binary:

1. When type information is readily available, e.g. from a conversion to an interface or other typed runtime calls, the analysis is eased. However, this information is typically only partially available, and some structures may not have complete type information at all.

2. Beside the runtime type information, calls to functions of the runtime or standard library can provide type information as the type of these functions is known and documented.

3. Additionally, type information can also be inferred from the instructions operating on the value. For example, a signed comparison of two values indicates that the values must be (signed) integers, whereas a value used as address in a load or store instruction must be a pointer.

Approaches for Type Analysis Existing approaches for the problem of type analysis in machine code can be mainly classified in *constraint-based* approaches and *data-flow-based*

approaches. The idea of constraint-based approaches is to reduce the problem of type analysis to a constraint satisfaction problem, where constraints are formulated for each instruction [21, 25, 35]. A solution of the resulting constraint system can be solved using a constraint solver. One problem with constraint-based analysis is that the constraint-system can have multiple solutions, if not enough information is provided such that the resulting types are not unique, or may be unsatisfiable, for instance if the code performs casts which are not modeled in the constraints.

Beside constraint-based approaches, an iterative data flow based approach for type analysis has been proposed by van Emmerik [35], focusing on finding a set of possible types for a value. Here, each instruction restricts the set of possible types. If a value has a type conflict, however, a type cast is inserted, contrary to a constraint-based system which would be unsatisfiable in this case.

Type Analysis of Go Functions For the type analysis of compiled Go functions, we decided to use a constraint-based approach. As Go has a strong but simple type system with very few unsafe type casts, the main advantage of a data flow based approach seems to be not really relevant. In general, we found that the type analysis can be performed in a more straight-forward manner using constraints, as there is no need for complex analysis passes since the analysis is performed implicitly when finding a feasible solution for the constraint set. For example, the maximum dereference offset of a pointer can be simply modeled as variable in the constraint-system.

Furthermore, the Go compiler uses the stack heavily for local variables, and also locates variables or temporary values of different types in the same stack location. While in a constraint-based setting this fact can be modeled by allowing stores to the stack to change the type of the destination, an analysis of the stack frame would have to be performed manually in a data flow based setting. This critical analysis step, however, has been left as future work by van Emmerik [35].

Finally, a constraint system seems to be more flexible for analysis of multiple functions: for inter-procedural type analysis, the constraints of each function can be put in a single constraint system, whose solution may also infer information which could not be inferred from the analysis of a single function.

It turned out that a constraint-based approach for type analysis of Go programs has major deficiencies in terms of performance and in terms of modeling of arrays, slices and other compound types. We will discuss these in Section 4.5.3 in more detail.

Implementation Remark: We formulate the constraints for the type analysis in the SMT-LIB 2.6 language [3] with the addition of the proposal for sequence data types [5]. As solver we used Z3 [11] version 4.5 as it supports the used constraint formulations, most notably the sequence types, sufficiently well and also has Python bindings available.[6]

[6] See: `https://pypi.python.org/pypi/z3-solver`, accessed 2017-07-22

4.5.1 Type Model

In our analysis, we distinguish between numeric types, pointer types and pseudo types. Numeric types include integers with sizes of 8-bits, 16-bits, 32-bits and 64-bits as well as floating-point numbers in 32-bit and 64-bit sizes. Note that we do not distinguish between signed and unsigned types to simplify the resulting constraint system as this information is not visible in machine code since hardware is agnostic of the sign. Integers can always be casted between the corresponding signed and unsigned type.

For pointer types, we not only distinguish between the type of the target object, but also the kind of the pointer:

- **Data Pointers** are pointers to a fixed-length sequence of types, where each type in the sequence represents one byte of memory. Load and store operations can be performed on the target object, but the type of the pointer cannot change.
- **Function Pointers** are pointers to the actual address of a function, contrary to the Go type system, where a function pointer is actually a pointer to a structure with the address being at the beginning. They can only occur in a load from an interface table or on a closure call. We model function pointers as two fixed-length sequences: one sequence stores the type of the arguments while the other sequence stores the return types.
- **Type Pointers** and **Interface Table Pointers** (Itab Pointers) are either used as argument to a runtime function, e.g. a map access, or as first part of an interface. If they are used outside of the stack, they must be succeeded by an interface pointer.
- **Interface Pointers** are pointers to the value of an interface. In contrast to data pointers, these cannot be dereferenced directly but must be converted to a data pointer first using the associated type information.

The type of the `FLAGS` register is modeled as separate flag pseudo-type, which can be the result of a numeric or pointer comparison. Finally, there exists a special *dead* pseudo-type, which indicates that a memory location may not be accessed directly. For example, the seven bytes which directly follows the first byte of a pointer will never be accessed directly as it has no valid value in the Go type system.

To allow recursive type definitions, pointer and function pointer types do not store the type sequences directly but only reference a *type id*, which is mapped to the actual type sequence.

We note that compound types, i.e. interfaces, slices and complex numbers, are not modeled directly but as their individual components, because the components are used as different variables internally and can be stored arbitrarily in the stack frame of the function. As we currently do not track relationships of compound types, we potentially lose information, especially when interfaces are constructed without runtime calls. We leave the analysis of interfaces and slices as future work.

4.5.2 Constraints

In the following, we will describe the strategy for the generation of constraints for a function. A complete example can be found in Figure 4.5.

Registers

We constrain registers to only contain types of the same size as the use of the register. For example, a 64-bit general purpose register can contain 64-bit integers and all kinds of pointers whereas a general purpose register accessed with a size of 32-bits can only contain 32-bit integers. The *flags* type can only be used by the flags register and the *dead* pseudo-type must not occur in used registers. If the value of a register has multiple sources when it is accessed, we require all sources to have the same type, as there are no implicit conversions between the basic types and all conversions between integers are explicit in the HGA.

Floating-point Values

As described in Section 3.7.2, beside memory moves, floating-point values are only processed using the SSE registers, which in turn only process floating-point values. This allows for a straight-forward generation of constraints for these registers.

Assumption 1 *Floating-point values are only stored in SSE registers and in memory, except for memory moves. Additionally, SSE registers accessed in sizes of 4-bytes or 8-bytes are only used for floating-point values.*

Assumption 2 *Floating-point constants are always loaded from a section of the binary.*

Immediate Operands

Non-zero immediate operands of instructions are always treated as integers. For a constant zero, we cannot infer any type information, as the zero value is valid for all types. As a consequence of Assumptions 3 and 2, the problem of typing constants described by van Emmerik [35] does not occur when analyzing Go programs.

Assumption 3 *All variables, constants and functions are always referenced via RIP-relative addressing and not referenced by an immediate address, even for binaries linked to a fixed position, cf. Section 3.7.2.*

Memory

When a register is loaded from memory or stored to memory, the first byte is required to be the type of the register while the remaining bytes must be *dead*. As an exception, we need to allow implicitly truncating loads from integer types in memory. This is reasonable as the Go compiler (currently) does not access higher bytes of a basic type. Additionally, a pointer can be transformed implicitly into an interface pointer. As, however, the only way for such a pointer to leave the scope of the function is a store to memory, we only

allow this conversion to happen on a store instruction. For storing non-zero immediate operands we apply the same handling as for storing registers.

Assumption 4 *When loading from memory, a type is either read completely or may be truncated to a lower part if the type is an integer.*

Assumption 5 *Conversions of pointers to interface pointers can only occur on a store to memory as this is the only way the implicitly converted value can leave the scope of the function.*

Assumption 6 *If a type or interface table pointer is loaded from or stored in non-stack memory, it must be immediately succeeded by an interface pointer. This does not apply for stack memory, see below.*

Stack Memory

A proper modeling of the stack frame in constraints is a difficult problem as the region of local variables in the stack frame does not have a fixed type. Moreover, the components of interfaces, slices and strings can be placed anywhere in the stack frame or may be partially eliminated. For example, the arguments of called functions have different types, multiple temporary variables of different types may be spilled to the same address and the capacity of a function-local slice may not exist at all if it is never used. Therefore, we make some assumptions on the usage of the stack frame before we can formulate constraints.

Assumption 7 *Every direct store to the stack can change the type of the stack frame, except for the argument region, which type is fixed.*

Assumption 8 *Every region of the local stack frame is initialized using a direct store before it is used.*

Whenever a pointer to the stack is taken (using a `LEAQ` *instruction), the target of the pointer is initialized with the correct type, e.g. using a move or zero instruction, as memory is strongly typed and references to uninitialized memory cannot occur in Go. The type of the stack region pointed to cannot change during the lifetime of the pointer.*

Exceptions (Runtime Functions): *Some runtime functions take a buffer of a fixed size to store their result if the buffer is non-*`nil` *and large enough. If the compiler proves that the result of such a function call does not escape the stack, it allocates such a buffer on the stack and passes the pointer without initialization. These cases have to be handled individually.*[7]

Exceptions (Older Versions): *Older versions of the Go compiler occasionally initialize the stack region indirectly using the constructed pointer. These cases need a work-around in the constraint system.*

[7] Affected runtime functions are: `intstring`, `slicebytetostring`, `stringtoslicebyte`, `slicerunetostring`, `stringtoslicerune`, `concatstring`, `chanrecv2`

We will discuss Assumption 8 in Section 4.5.3 in more detail, but already note here that this assumption holds in many cases, but does not hold in the general case.

In the resulting constraint system, we model the argument/return region as a single sequence of types with byte granularity, in the same way we model the target of normal pointers.

Implementation Remark: For the local variables, we cannot do this as the type can change, contradicting the immutability of sequences. Therefore, starting from an initial type sequence for the stack frame, we get a new type sequence on every direct store by replacing the affected region with the type of the source operand. For all subsequent loads or address-taking operations (i.e. `LEAQ`), we use the new sequence (per Assumption 8) until another direct store occurs.

If a basic block has more than one predecessor, the stack frame types of these predecessors may differ, e.g. by different temporary variables stored at the same location. Thus, as the type sequence has byte granularity, we require for each byte that the type of all predecessors is equal or that, if the type differs, the type of this byte in the merge block is *dead*, i.e. unusable. We note that this construction leads to a massive amount of additional constraints for functions with a more complex control flow.

A direct load from the stack frame is handled in the same way as a load from any other memory is handled, that is, we add a constraint ensuring that the type of the loaded value must be equal to the type of the last store to that offset in the stack frame.

A `LEAQ` from the stack frame extracts a sub-sequence from the current type sequence of the stack with a variable length and uses this as sequence for the resulting pointer type. We note that we lose type information as we do not track this information for subsequent stores. We could do this since the type of the region in the stack frame cannot change as long as the pointer is valid, this is left as future work. We additionally note that this handling of `LEAQ` instructions is solely based on Assumption 8.

The work-around for the exception of Assumption 8 for older Go version consists in allowing the stack frame to have an arbitrary type in the beginning instead of forcing all bytes to be dead.

Instructions

For most instructions, the operand types are intrinsic to the instruction type. For example, a sign-extension or multiplication will always operate on integers while a `NilCheck` instruction will only operate on pointers. For move instructions, we can only require that the types of source and destination are equal. Three instruction types, however, require special handling: additions of pointer-sized values, zero-extensions of a constant zero and function calls.

Pointer-sized Additions A pointer-sized addition can be either the addition of two integers or the computation of a new pointer with an offset to an existing pointer, i.e. the addition of a pointer and an integer. If one addend is a constant, it must be an integer per Assumption 3 and the other addend can be either an integer or a pointer. In case of

```
func foo(a *int, b int) int {
    return *a - b
}
```

(a) *Source of the Go function* `foo`

```
481b50 <main.foo>:
  481b50: mov rax, [rsp+0x8]
  481b55: mov rax, [rax]
  481b58: mov rcx, [rsp+0x10]
  481b5d: sub rax, rcx
  481b60: mov [rsp+0x18], rax
  481b65: ret
```

(b) *Disassembly of* `foo`

```
Block @ 0x0
  0: GetSP       SP~8
  1: GetBP       BP~8
  2: GetClosurePtr DX~8
  3: MOVQload    AX~8,[SP+$0x8]~8
  4: MOVQload    AX~8,[AX]~8
  5: MOVQload    CX~8,[SP+$0x10]~8
  6: SUBQ        AX~8,CX~8
  7: MOVQstore   [SP+$0x18]~8,AX~8
  8: RET
```

(c) *HGA representation of* `foo`*, address markers omitted*

```
(declare-sort TyID 0)
(declare-datatypes () ((Ty (dead) (i8) (i16) (i32) (i64) (typeptr) (itabptr)
                           (ifaceptr) (ptr (pointee TyID)) ...)))
(declare-fun tyidfunc (TyID) (Seq Ty))
; ... (Further declarations omitted)
(assert (= (seq.len fnargs) 24))
; Block 0x0, Instruction 3
(assert (= (seq.extract fnargs 0 8) [AX_0_3 7*dead]))
(assert (=> (or (is-typeptr AX_0_3) (is-itabptr AX_0_3))
            (= (seq.extract fnargs 8 1) [ifaceptr])))
; Block 0x0, Instruction 4
(assert (is-ptr AX_0_3))
(assert (>= (seq.len (tyidfunc (pointee AX_0_3))) 8))
(assert (= (seq.extract (tyidfunc (pointee AX_0_3)) 0 8) [AX_0_4 7*dead]))
(assert (=> (or (is-typeptr AX_0_4) (is-itabptr AX_0_4))
            (= (seq.extract (tyidfunc (pointee AX_0_3)) 8 1) [ifaceptr])))
; Block 0x0, Instruction 5 (omitted, similar to instruction 3)
; Block 0x0, Instruction 6
(assert (and (is-i64 AX_0_4) (is-i64 CX_0_5) (is-i64 AX_0_6)))
; Block 0x0, Instruction 7
(assert (or (= (seq.extract fnargs 16 8) [AX_0_6 7*dead])
            (and (is-ptr AX_0_6)
                 (= (seq.extract fnargs 16 8) [ifaceptr 7*dead])))))
(assert (=> (or (is-typeptr AX_0_6) (is-itabptr AX_0_6))
            (= (seq.extract fnargs 24 1) [ifaceptr])))
```

(d) *Excerpt of the resulting constraint system. Sequences are represented in a pseudo-code notation using square brackets for readability. Each load and store ensures that interfaces are always stored as a compound of type or interface table pointer and the actual data pointer. A store allows an implicit conversion of a pointer to an interface data pointer.*

```
(define-fun tyidfunc ((x!0 TyID)) (Seq Ty)
    (ite (= x!0 TyID!val!0) [i64 7*dead typeptr itabptr] []))
(define-fun fnargs () (Seq Ty) [(ptr TyID!val!0) 7*dead i64 7*dead i64 7*dead])
(define-fun AX_0_3 () Ty ptr(TyID!val!0))
(define-fun AX_0_4 () Ty i64)
(define-fun CX_0_5 () Ty i64)
(define-fun AX_0_6 () Ty i64)
```

(e) *One solution of the constraint system. Note that the sequence for the pointer target* `TyID!val!0` *is longer than required and contains two invalid type entries after the used bytes.*

Figure 4.5: *Example of constraint-based analysis of basic types on a simple function.*

a pointer, the target is also a pointer where the type sequence of the source pointer is shifted by the number of bytes given by the constant.

When both operands are registers, there are three possibilities: either all sources and destinations are integers, or exactly one source operand is a pointer, implying that the destination is also a pointer [25]. For pointer additions, we currently do not infer information about the types of the target object as we do not analyze the integral operand. Further analysis could be performed using a Value Set Analysis [1, 21, 28].

Assumption 9 *Pointers are only modified using additions with a (bounded) positive integer unless the unsafe package is used. Special cases of subtractions are handled during the lifting into the canonical HGA.*

Zero-Extension A special case is the constant value zero, which can be used as integer and `nil`-pointer alike. Thus, it can also happen that a register containing the value zero is zero-extended to the pointer size and used for a comparison with a pointer. Therefore, we need to allow `nil`-pointers to have an origin in the `MOVLQZX` instruction.

Function Calls A function call in Go essentially behaves like a combined load and store on the stack. The lowest part of the stack frame must be equal to the argument sequence of the type of the function (load). The return part, which follows immediately after the arguments, is replaced with the return types of the function, like a store to the stack frame.

If the target of the call is known (i.e. a `StaticCall`), we know the size of the argument region and also have information about the size of the argument and return parts of that region per the analysis described in Section 4.4.2. In this case, we can also restrict the sizes of the argument and return sequences of the function type, if the type is not already known by other means. For indirect calls (`InterCall`, `ClosureCall`), the size of the argument region is not directly known but left as variable in the constraint system.

Calls to Runtime Functions A special case of call targets are runtime functions. Not because they have special calling conventions (these functions were already eliminated when lifting to the HGA), but because the signature of the function is known, and, most importantly, some runtime functions also have type information as arguments, as noted in Section 3.7.6. For example, a call to the function `runtime.newobject`, which takes the type of the object to create as its only argument and returns a pointer to a newly allocated object of the given type, allows to correctly specify the type of the return value as the address of the type is always known for compiler-generated calls to this function.[8]

In addition to this, the functions which might have special buffers as arguments require a special handing (see Assumption 8).

Unhandled Cases

There are some cases and language constructs which are not properly handled by the constraint generation procedure. Running pointers in a loop, as they can occur in a

[8] Non-compiler generated calls to such functions can only occur in the `runtime` or `reflect` package.

for-range loop, will currently lead to an unsatisfiable constraint system, because sequences in Z3, which we use to model memory, can only have a finite length. A running pointer in a for-range loop, however, would have to be an infinite repetition of the element type as the iterable might be unbounded. This type of pointer arithmetic has to be detected by a separate analysis. Type assertions and type switches are currently not supported as we do not track the relation of the interface value and the corresponding type field.

Additionally, slices, arrays and, to a lesser extent, strings are only supported in a way that the resulting constraint system is satisfiable, but excluding a significant amount of available type information as pointer additions with non-constant values are not properly handled (see the following section).

4.5.3 Problems

During the design and implementation of a constraint-based type analysis, we encountered several problems and difficulties. While some of these problems can be solved by additional analysis prior to the constraint formulation, other problems are inherent to a constraint-based type analysis.

Pointers to Stack Memory We currently assume that a `LEAQ` into stack memory only points to well-defined stack regions and that the type of the stack region cannot change during the lifetime of the pointer (Assumption 8). As sequences are immutable in Z3, we assign the type of the target region on the stack at the time of the `LEAQ` as target type of the new pointer.

However, as a `LEAQ` instruction is only arithmetic and independent of the memory operations, the compiler might also choose to emit the `LEAQ` instruction before the target region is initialized. Moreover, the compiler is also allowed to use the same address of stack memory for different types for the same reason. Although we have not observed these optimizations so far, the compiler can (in theory) perform such optimizations, implying that the assumption may not hold.

The problem can be solved by evaluating the result of `LEAQ` instructions lazily: instead of evaluating the type of the new pointer at the time of the `LEAQ` instruction, the type can be evaluated when the pointer is actually used. This, however, requires to store and propagate the information along with the pointer. As the pointer, however, can also be stored in the stack frame[9], a complete analysis of the stack frame would be required before the formulation of constraints.

Arrays and Slices The described approach is not able to model arrays, which have a known length, and slices, which essentially are an array with a potentially non-constant length, properly. In addition to an increased modeling complexity, support for arrays also requires quantifiers to apply type information for all elements equally. Z3 currently does not support quantifiers for general sequences[10]. Thus, to include arrays in the described (simple) type systems, major changes are required. Distinguishing the address of single

[9] Pointers to the stack can only be stored on the stack or in registers.

[10] `https://github.com/Z3Prover/z3/issues/1235`, accessed 2017-08-28

elements from sub-slices requires a special analysis tracing the individual components of slices (or compound types in general) for an appropriate result. In addition, an array can become the data pointer of a slice at any point, just as a pointer can become an interface pointer. This has to be handled separately.

Ambiguity In cases where a value is not uniquely defined, the constraint solver can choose *any* value which satisfies the constraint system. For example, if a member of a structure is never dereferenced and the type cannot be inferred by other means, a constraint solver can also choose invalid types including mis-aligned pointers. For indirect function calls, where the size of the argument/return region is not known but left as open variable, the constraint solver can easily choose to define the function type as function with no arguments but a return region size filling the entire stack frame. While this obviously does not make sense in most cases, we cannot disallow this entirely as it is theoretically possible.

Experience shows that this kind of ambiguity is rather common in the constructed constraint systems and is inherent to type analysis solely based on constraints. The ambiguity of constraint systems can be reduced by performing an inter-procedural analysis leading to more constraints, but not removed entirely for reasons explained earlier.

Unsafe Casts For the constraint formulation, we excluded unsafe casts between pointers and integers. When the original program uses unsafe pointer arithmetic, the generation strategy might yield an unsatisfiable constraint system. If we decided to model these casts by allowing the conversion at any time, we would end up with almost no additional information of the analysis with a very high probability: the constraint solver is likely use these casts heavily as this greatly simplifies the solution – values are casted to pointers if they are used as address right before the dereference and otherwise typed as integers.

Performance A major deficiency of the described constraint-based approach is the required time to solve the constraint system. While performance is in general only a secondary aspect in program analysis, the time required for analysis should be still in reasonable limits. We found that for functions with a non-trivial control flow (i.e. multiple basic blocks with more than one predecessor) and a larger stack frame size, which is rather common for Go functions, the analysis requires multiple hours or even days to terminate.

One major reason for this are the constraints which are required to unify the stack frame types at merge blocks, where we currently generate one constraint per byte of the stack frame. As each of these constraints has two possibilities (either type equality or dead), the size of the search space grows exponentially with the size of the stack frame and the complexity of the function. Another source for the increasing complexity is the implicit conversion of pointers to interface pointers, which can happen at every `MOVQstore` instruction.

Debugging Another difficulty of a constraint-based approach is debugging. Due to the performance problems described above, even testing minor changes can take a very long time. More importantly, however, is the debugging of unsatisfiable constraint systems: if a

constraint system does not have a solution, it is possible to retrieve a subset of constraints which proofs the unsatisfiability (the *unsat core*). Depending on the complexity of the function, this might include hundreds of constraints, significantly increasing the difficulty for a human analyst to understand the underlying problem.

5 Evaluation

In the previous chapter, we proposed a strategy to extract metadata from Go binaries, an intermediate representation and an analysis to recover the types of functions. After describing the sample binaries, we evaluate the presented techniques on these binaries and depict our results. Finally, we will discuss our findings in the context of a practical application.

5.1 Targets

For our evaluation, we will use four different binaries, namely the standard library where the source is known and three unknown binaries. An overview of the size of specific sections of the used binaries is shown in Table 5.1.

Standard Library As main target of the evaluation of the described lifting and analysis procedure, we will use the Go standard library of Go 1.8 compiled for x86-64. The standard library is well-suited for this purpose, because it contains functions from a variety domains and with different complexity, including cryptographic functions, compression algorithms, serialization, string formatting, basic image processing, a complete HTTP stack and a parser for Go code.

In the following, when referring to the term *standard library*, we will exclude wrapper functions and functions written in assembly. We additionally exclude special package initialization functions and the runtime package, as some functions to which calls are usually generated by the compiler are also called in other contexts. For example, calls to write barrier functions can occur in other contexts than the usual compiler-generated contexts. Consequently, we also exclude functions of the *reflect* package which are actually defined in the runtime package and only named as functions of the reflection package.

Table 5.1: *Sizes of the binaries used for the evaluation and the sizes of specific sections of these. From the included metadata, the* pclntab *containing information about the functions as well as the Runtime Type Information (RTTI) significantly contribute to the size of the binaries. As the standard library is a shared object and therefore contains position-independent code, it additionally includes relocations.*

	Total	Code	Pclntab	RTTI	Relocations	DWARF
libstd	36.85 MB	6.22 MB	2.71 MB	4.06 MB	5.34 MB	12.86 MB
Goversing	1.96 MB	0.59 MB	0.30 MB	0.18 MB	–	0.67 MB
lets_go	1.85 MB	0.58 MB	0.30 MB	0.18 MB	–	0.57 MB
Lady	8.24 MB	3.91 MB	1.41 MB	1.64 MB	–	–

For the purpose of this analysis, the standard library was compiled as a shared library without inlining.[1]

Unknown Binaries In addition to the standard library, we apply our tools on three binaries where the original source is not available. The first binary is the malware *Lady* (also referred to as *Golad*) for Linux (x86-64), which can send information to a command-and-control server, start programs to mine cryptocurrencies and also replicate itself on other computers reachable via network[2]. The binary has does not contain ELF symbols information or debug information.

The second binary is the *Goversing* task from the Capture-The-Flag competition *Codegate 2017 prequals*[3], which is also a Linux binary for the x86-64 architecture. This binary basically asks for a user name and a password, checks both values using a special routine, and prints a flag if the input is correct.

As third binary, we use the *lets_go* task from the Capture-The-Flag competition *Tokyo Westerns CTF 2017*[4]. This binary is also a reverse engineering challenge for Linux (x86-64), asking for a password and subsequently for the flag, which are both verified using custom procedures. We used our developed tools to solve this challenge during the competition.

5.2 Setup

The analysis was performed on machine equipped with two Intel Xeon E5-2687W v3 processors clocked at 3.10 GHz and 500 GiB memory running Ubuntu 16.04 (64-bit) with Python 3.5.2 and Z3 master, commit `799fb4a`, dated 2017-08-24. We note that the analysis was not performed on a dedicated system and only performed once, implying that the running times only serve as indicators of magnitude. In addition, we limited the CPU time to 5 hours and the usable memory to 8 GiB for the type analysis of each function. These limits appear reasonable in order to provide a significant benefit compared to a manual analysis.

To work around a bug[5] in the Z3 constraint solver, we extracted the generated constraints and solved them separately using the `z3` command line tool. Furthermore, we disabled the generation of unsatisfiability cores for performance reasons.

5.3 Results

5.3.1 Metadata Extraction

The extraction of the meta information about functions, the type information and the module data structure using the described procedure turned out to work quite well on all four binaries and the resulting data did not contain inconsistencies. We additionally verified some random samples of the standard library manually.

[1] Compilation command: `go install -buildmode=shared -gcflags -l std`

[2] `https://vms.drweb.com/virus/?_is=2&i=8400823`, accessed 2017-09-04

[3] Website is not available as of 2016-08-20.

[4] `https://score.ctf.westerns.tokyo/?locale=en`, binary available after login, accessed 2017-09-02

[5] `https://github.com/Z3Prover/z3/issues/1234`, accessed 2017-08-26

For the standard library and the non-stripped *Goversing* and *lets_go* binaries the runtime symbol information and the address of the module data structure were found using symbols. In the stripped *Lady* binary the runtime symbol information was found using the signature-based approach. The information about function names, file names and line numbers was still present and consistent for the three binaries.

We manually verified the compiler version identification by ensuring consistency of the module data structure with the binary layout and additionally compared of some runtime functions with a known binary. The compiler of the *Goversing* binary and the *lets_go* binary was correctly identified to be Go 1.7 and Go 1.8, respectively. For the *Lady* binary the version was detected as Go 1.5–1.6 (is Go 1.6).

It was also possible to extract the type information from both binaries. During later analysis, however, it turned out that for the *Lady* binary 165 types from the 7850 types in total were not extracted automatically for reasons described in Section 4.1.3. The addresses of these types had to be specified manually. From the *Goversing* and *lets_go* binaries all used type information were extracted successfully.

This extracted symbol information also reveals information about used third-party libraries via the naming convention for Go functions: while the *Goversing* and *lets_go* binaries only use the standard library, the *Lady* binary makes use of seven external open source libraries. The metadata indicates that the following external libraries are used:

- `github.com/kardianos/service`[6], a package to install operating system services
- `github.com/naoina/toml`[7], a package to parse and encode TOML[8]
- `github.com/naoina/go-stringutil`[9], string utilities
- `github.com/parnurzeal/gorequest`[10], an HTTP client
- `github.com/shirou/gopsutil`[11], a library to get information about running processes
- `github.com/garyburd/redigo`[12], a client for Redis databases
- `golang.org/x/crypto/ssh`[13], an SSH client

Using this information and the association of file names and functions, we were able to reveal the functions and data types which provide the essential functionality of the binary. For both unknown binaries, we will focus on the unknown functions (*Goversing*: 8, *lets_go*: 8, *Lady*: 30) and data structures (*Goversing*: 1, *lets_go*: 0, *Lady*: 6) in the following, because the source of the other functions is publicly available.

[6] `https://github.com/kardianos/service`, accessed 2017-08-20
[7] `https://github.com/naoina/toml`, accessed 2017-08-20
[8] Tom's Obvious, Minimal Language, `https://github.com/toml-lang/toml`, accessed 2017-08-20
[9] `https://github.com/naoina/go-stringutil`, accessed 2017-08-20
[10] `https://github.com/parnurzeal/gorequest`, accessed 2017-08-20
[11] `https://github.com/shirou/gopsutil`, accessed 2017-08-20
[12] `https://github.com/garyburd/redigo`, accessed 2017-08-20
[13] `https://godoc.org/golang.org/x/crypto/ssh`, accessed 2017-08-20

All relevant functionality of the *Goversing* binary is contained in the package `main`. In addition to the functions, the package contains a single custom data structure `User`, which has two string fields for the name (`id`) and password (`pw`). Also for the *lets_go* binary all notable functionality is included in the `main` package, but no custom structure types are used. The remaining packages of the *Lady* binary are named `main`, `attack`, `ipip`, `minerd`, `redis` and `super`. All six custom data structures are defined in a separate package `st`. Reasoning from the names of the structures, they contain mostly configuration data, e.g. a version and URL for updates and a list of rules for IP addresses.

5.3.2 Code Lifting

All Go functions of the standard library and all functions of the *Goversing* and *lets_go* binaries (excluding runtime functions) were lifted successfully into the canonical HGA using the described procedure. An example for the result of the lifting procedure applied on a function from the *lets_go* binary can be found in Figure 5.1. This example demonstrates the advantage of the HGA: the elimination of the stack frame setup, which contains the conditional call to the stack enlargement routine, combined with the analysis and simplification of memory operations involving Duff's device as well as bounds checks not only simplifies automated analysis, but also supports a human analyst in understanding the relevant functionality.

From the *Lady* binary which uses Go 1.6, all unknown function were lifted successfully and no artifacts from the code generation procedure of that version were observed. Some other functions of external libraries, however, could not be lifted as they contain memory moves involving the `std`, `cld` and `movsq` instructions. These instructions are currently not handled, cf. Section 4.3.2.

5.3.3 Argument Region Analysis

Standard Library To verify the argument region analysis strategy described in Section 4.4.2, we compare the results of our analysis with the signature of the original function for all standard library functions. Therefore, we patched the compiler to additionally emit the offsets and sizes of the parameters and results of the functions and compared the results.

However, a simple comparison turned out to be insufficient for multiple reasons: first, function parameters are not necessarily used, implying that our analysis is not capable to find these arguments. Hence, we only require that the set of bytes identified as parameters must be a subset of the actual parameter bytes. Second, structures may contain holes as a consequence of alignment. These holes might not be written in return values; for simplicity we resorted to a manual analysis of these cases. Third, if a function does not return at all (e.g. by calling *panic* unconditionally), the return values are never written. In this case, our analysis is obviously unable to find written return values. Thus, we exclude the analysis of return regions for functions which do not return ordinarily.

With the described limitations, we found that the analysis of the argument region for parameters and results of the function gives correct results, and note that these limitations have no significant impact for further analysis of any kind.

```
491a80: mov     rcx,fs:[-8]
491a89: lea     rax,[rsp-0x88]
491a91: cmp     rax,[rcx+0x10]
491a95: jbe     0x491b86
491a9b: sub     rsp,0x108
491aa2: mov     [rsp+0x100],rbp
491aaa: lea     rbp,[rsp+0x100]
491ab2: lea     rdi,[rsp]
491ab6: lea     rsi,[rip+0x3ab83]
491abd: mov     [rsp-0x10],rbp
491ac2: lea     rbp,[rsp-0x10]
491ac7: call    runtime.duffcopy+0x2a0
491acc: mov     rbp,[rbp+0x0]
491ad0: mov     rax,[rsp+0x110]
491ad8: mov     rcx,[rax+0x8]
491adc: mov     rdx,[rax]
491adf: test    rcx,rcx
491ae2: jbe     0x491b7f
491ae8: movzx   ecx,byte ptr[rdx]
491aeb: mov     rdx,[rsp+0x120]
491af3: mov     rbx,[rsp+0x118]
491afb: xor     esi,esi
491afd: cmp     rsi,0x20
491b01: jae     0x491b5a
        ...
491b7f: call    runtime.panicindex
491b84: ud2
491b86: call    runtime.morestack_noctxt
491b8b: jmp     0x491a80
```

(b) (left): *Excerpt of the original assembly code.*

```
Block @ 0x0
  0: GetBP      BP~8
  1: MoreStack
     --- 0x491aa2 SP:0x108
  2: MOVQstore  [SP+$-0x8]~8,BP~8
  3: MemMove    [SP+$-0x108]~256,
                  [SB+$0x4cc640]~256
  4: MOVQload   AX~8,[SP+$0x8]~8
  5: MOVQ       TMP1~8,AX~8
  6: ADDQ       TMP1~8,$0x8~8
  7: MOVQload   CX~8,[TMP1]~8
  8: MOVQload   DX~8,[AX]~8
  9: CheckIndex $0x0~8,CX~8
 10: MOVBload   CX~1,[DX]~1
 11: MOVQload   DX~8,[SP+$0x18]~8
 12: MOVQload   BX~8,[SP+$0x10]~8
 13: MOVL       SI~4,$0x0~4
 14: MOVLQZX    TMP2~8,SI~4
 15: CMPQ       TMP2~8,$0x20~8
 16: JAE        $0x11~8,$0xb~8
...
```

(a) (above): *The first basic block of the function lifted to the HGA. Note that* `SP` *is no longer modified.*

Figure 5.1: *Comparison of original assembly code and lifted HGA representation of the function* `main.f1151e71905f3d94b49b0` *of the binary* lets_go. *The stack frame setup including the conditional stack switch is removed, the memory move via Duff's device is compressed into a single instruction and the bounds check is explicit.*

Table 5.2: *Overview of the results of the constraint-based type analysis. Not all functions were analyzed, the number of selected functions and functions in total is indicated next to the name of the binary. Only Sat represents a successful analysis, the other cases are different reasons for failures. For unhandled functions the type constraints could not be generated, e.g. by incomplete handling of runtime type information.*

	Sat	Unsat	Unknown	Time	Memory	Segfault	Unhandled
libstd (4415/8950)	3925	90	32	200	3	44	121
Goversing (8/1831)	4	2	–	2	–	–	–
lets_go (8/1810)	5	3	–	–	–	–	–
Lady (30/8468)	12	7	–	7	–	1	3

Unknown Binaries For the unknown binaries, this approach of verification is not feasible. We resorted to a manual analysis and verified correctness.

5.3.4 Type Analysis

For the evaluation of the basic type analysis based on constraints, as described in Section 4.5, we generate and solve type constraints on a per-function basis. We did not perform an inter-procedural analysis as we do not expect meaningful results after a reasonable amount of time given the issues raised in Section 4.5.3.

Standard Library Due to limitations of the constraint system and current implementation (see Section 4.5.3), we perform the analysis only on a subset of the runtime functions. Specifically, from the 8950 top-level functions in total, we excluded all functions employing indexing and slicing operators, for-range loops, type assertions and type switches as well as closure or interface calls. Moreover, we exclude variadic functions, which are basically functions with a slice as parameter, and function which mostly ($> 70\%$) consist of `MOVQ` instructions, as we do not expect to infer any useful information. This leaves 4415 standard library functions for our type analysis.

Unknown Binaries For the three unknown binaries, we apply our type analysis only on the unknown functions.

An overview of the results can be found in Table 5.2. The segmentation faults in the constraint solver are caused by an assertion failure, we reported this issue to the Z3 developers.[14] For functions marked as *unhandled* no constraint system could be generated, the reason for all cases was a missing lowering of some types of the runtime type information into flat types.

[14] `https://github.com/Z3Prover/z3/issues/1233`, accessed 2017-08-27

Successful Analysis

Standard Library An proper analysis of the cases where a type assignment is found turned out to be a rather difficult problem, especially as functions are analyzed independently of each other. Automating the analysis was not possible because unused values or parts of structures can have any value (cf. Section 4.5.3) and a detection of these cases would either impose a huge analytical effort or based on weak heuristics. Given the other structural problems of the described and implemented constraint-based approach we will only perform a manual analysis of some selected functions with different complexity from different packages. Examples for the results of the type analysis are shown in Table 5.3.

Trivial functions (see 5.3a) without parameters and results which only consist of a return statement are analyzed correctly. Also functions having integers as parameters and return values are analyzed correctly, if the value is actually used. For example, when an argument is modified using a bitwise operation with a constant (Table 5.3b) or two arguments are subtracted from each other (Table 5.3c), the result is correct.

However, whenever a structure or the argument and return region contains alignment holes or unused entries, these parts are filled with arbitrary types. These are not necessarily valid as no constraints are imposed on them. Thus, the resulting type sequences can contain floating-point values or unaligned pointers with a size of one byte. A detection of this cannot be automated in every case: an 8-bit integer, for example, is valid at every position. This issue appears for almost all functions, examples are shown in Table 5.3b-e.

When a string is only compared byte-wise but never used otherwise as string (e.g. as parameter to another function), the string data can also be classified as byte array instead (Table 5.3d). This can be fixed by tracking the relationship of the string length with the string data, see Section 4.5.3. Otherwise, strings are correctly identified as such using the association of the type id with the runtime type information. In the case that entries of memory sequences are only used with move instructions, it can happen for pointer-sized types that the type is assigned wrongly, for example an integer instead of a regular pointer (Table 5.3e) or a function pointer instead of an integer. Except for plain memory moves, floating-point types are correctly classified (Table 5.3f), mainly because they employ different instructions compared to integer and pointer types.

Beside the cases of insufficient information and the ambiguity of unused values, the available type information is used and the resulting information is generally correct.

Unknown Binaries We observe that only simple functions could be analyzed successfully. Package initialization functions, whose signature is known to have neither parameters nor results, could be handled in all cases. In the *Goversing* binary, a byte-wise access to a string was not identified as string, because indexing with non-constant values is currently not handled yet (cf. Section 4.5.2). From the functions of the *lets_go* binary, string arguments were correctly identified as such. However, due to the lack of support for slice index operations with a non-constant index, an arbitrary type is assigned to slice types. From the unknown functions of the *Lady* binary, only simple functions wrapping functions of the standard library could be analyzed in addition to the initialization functions. Beside this, the analysis result of the functions was congruent with a manual analysis, subject to the limitation of available information.

Table 5.3: *Examples of successful results of the type analysis on selected standard library functions. The description refers to the technical implementation but not to the actual semantics. Resulting sequences for the argument region have byte granularity, apparently invalid type information is gray. The pseudo-type dead is represented using the symbol ⊥. The original types have been replaced with an idiomatic replacement for brevity.*

(a) `package net; func sysInit()`
Description: Empty function
Metrics: LOC: 0, IC: 4, FS: 0 — Time: 0.1s, Memory: 18M

```
[] -> []
```

(b) `package os; type FileMode uint32; func (FileMode) Perm() FileMode`
Description: Mask of argument with constant integer
Metrics: LOC: 1, IC: 9, FS: 0 — Time: 0.1s, Memory: 19M

```
[i32 3*⊥ itabptr ifaceptr flags...]  -> [i32 3*⊥ i8 f32 f64...]
```

Remark: Unused parts of structures are filled with arbitrary and likely invalid types in byte granularity, here with interface pointers, status flags and floating-point values. It is impossible to decide whether the `i8` is a return value or invalid.

(c) `package image; type Rectangle [4]int; func (Rectangle) Dy() int`
Description: Subtraction of two fields of a structure
Metrics: LOC: 1, IC: 8, FS: 0 — Time: 0.1s, Memory: 20M

```
[i8 i16 i32...  i64 7*⊥ ptr(Ty!4) ptr(Ty!5)...  i64 7*⊥] -> [i64 7*⊥]
```

(d) `package net; func http2authorityAddr(string, string) string`
Description: Basic string comparison and calls to other functions
Metrics: LOC: 16, IC: 160, FS: 112 — Time: 1471.9s, Memory: 408M

```
[ptr(Ty!6) 7*⊥ i64 7*⊥ ptr(Ty!10) 7*⊥ i64 7*⊥] -> [ptr(Ty!10) 7*⊥ i64 7*⊥]
Ty!6 = [4*i8 ptr(Ty!353) ptr(Ty!347)] — Ty!10 = [i8] (string)
```

Remark: The first argument is only used for a byte-wise comparison and therefore not classified as string but byte array.

(e) `package fmt; func (*ss) scanUint(rune, int) uint64`
Description: Conditionally calling other functions depending on first argument
Metrics: LOC: 25, IC: 168, FS: 112 — Time: 3613.7s, Memory: 647M

```
[i64 7*⊥ i32 3*⊥ ptr(Ty!199) ptr(Ty!200)...  i64 7*⊥] -> [i64 7*⊥]
```

Remark: The receiver is never dereferenced and hence classified as single arbitrary type.

(f) `package math; func Pow(x, y float64) float64`
Description: Implementation of floating-point exponentiation
Metrics: LOC: 98, IC: 385, FS: 48 — Time: 12164.3s, Memory: 590M

```
[f64 7*⊥ f64 7*⊥] -> [f64 7*⊥]
```

Metrics — LOC: Lines of Code — IC: Instruction Count (HGA) — FS: Frame Size (bytes)

Unsatisfiable Constraint Systems

For some functions, the produced constraint system was declared as unsatisfiable. We identified two main reasons for this: first, the usage of casts between pointers and integers using the *unsafe* package leads to contradictions in the constraint system as we did not handle these casts during modeling, see Section 4.5.3. Second, memory moves using registers only work if the type of the moved value has the same size as the register. The possible case of moving an array of eight bytes using one 64-bit general purpose register is not covered. This can be fixed by either allowing multiple types in registers or by detecting and combining memory moves.

The cases of unsatisfiable constraint systems resulting from functions of the unknown binaries also have other reasons: complex slicing operations, type assertions and for-range loops are not supported due to the lack of additional analysis before generating the type constraints, see Sections 4.5.2 and 4.5.3. These issues did not appear in our evaluation of the standard library as we left out functions with these language constructs.

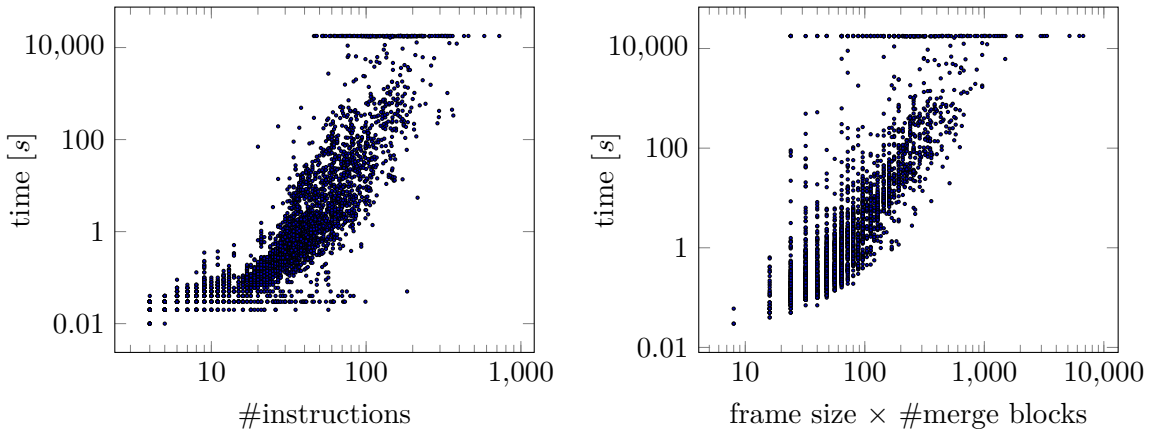
Unknown Result

For some functions, Z3 gave *unknown* as result instead of giving a solution or proving unsatisfiability. We were not able to find compelling reasons for this behavior, simple as well as complex functions are affected by this.

Memory Usage

In total, the memory limit of 8 GiB was exceeded for 3 functions. We identified a single reason for this behavior: the concerned functions allocate larger structures, leading to long sequences in the model due to byte granularity. Especially as long arrays are flattened, functions allocating large buffers of a fixed size or access large indices of slices are affected by this limitation. This problem is inherent to the byte-wise modeling and could be circumvented by modeling arrays in a compact representation.

When enabling the generation of unsatisfiability cores, we observed that the memory consumption increases significantly if a 64-bit addition with a large constant is present and the Python API of Z3 is used. As described in Section 4.5.2, a pointer-sized addition can be either an integer addition or a pointer offset computation. For the pointer offset computation, a subsequence from the original pointer target starting at the value of the constant operand is used as target of the new pointer. Since the generation of unsatisfiability cores implies an increased complexity of constraints in Z3 (see Section 2.2), the case of a large pointer addition will not be pruned, causing an allocation of a sequence whose length is larger than the constant addition operand. Depending on the value of the constant, the resulting sequence can have an any length, leading to an arbitrarily high memory consumption.



(a) *Time required for solving the constraint system by instruction count*

(b) *Time required for solving the constraint system by the product of the stack frame size and the count of blocks with multiple predecessors.*

Figure 5.2: *Analysis time required for solving the system of type constraints for each function. The time increases exponentially with the number of instructions, but also with the size of the stack frame in combination with basic blocks having multiple predecessors.*

Performance

For very simple functions with a small number of instructions, a small stack frame and a low number of branches, we can observe that resulting constraint system is rather simple and solved within less than a second. However, as soon as the number of instructions or the complexity of the control flow increases, the required time grows significantly, see Figure 5.2. In fact, the analysis of more involved functions required several minutes or hours to complete, or was aborted due to the time limitation.

We note that several more complex functions from the standard library were excluded because of the limitations of the constraint-based analysis approach and therefore expect even higher running times for other functions.

5.4 Discussion

In the remainder of this chapter, we will discuss our findings, especially focusing on the application of the proposed techniques in practice.

Metadata Extraction The described procedure the extract metadata works well and can be a significant help for manual and automated analysis of unknown binaries. As the function names typically include a full URL for packages hosted on common open source platform, many external dependencies and functions from the standard library can be easily identified. In addition, the available type information can help a human analyst in understanding the semantics of the program.

We note that obfuscators may try to remove or modify this information. While we do not know of any obfuscation efforts specialized for Go binaries, we expect such tools

to appear with the increasing popularity of Go. However, the range of possibilities for modification is rather limited as most parts of the metadata are required to actually run the program.

Intermediate Representation The HGA intermediate representation had the goal to model special constructs like bounds checking appropriately and concisely, easing manual analysis and also reducing the complexity for automated analysis. Concluding from our observations, we believe that this representation achieves this goal and also serves as a foundation for further analysis of compiled Go code. First experiences in using the intermediate representation for a manual analysis of unknown Go functions confirm this. Especially in combination with a graphical representation of the control flow graph, we found that the HGA provides a significant advantage compared to a linear disassembly or a plain representation of the underlying machine code.

Unfortunately, the information gained by lifting machine code to this representation cannot be used by existing tools easily. Exporting the HGA representation into other commonly used representations would provide additional benefit for practical applications.

Type Analysis For the described type analysis based on constraints, we do not see practical applications in its current form. In addition to the described performance problems, the ambiguity for cases which are not uniquely defined prevents an automated processing of the result, because these cases cannot be identified in general. Moreover, the problem of modeling several language constructs properly, e.g. unsafe casts, pointers to the stack and arrays (see Sections 4.5.2 and 4.5.3), prevents the application on general real-world binaries.

We note that most of the described difficulties can be circumvented by implementing further analysis passes, including an analysis of stack variables, pointer additions and compound types. If this analysis has to be implemented separately even for a constraint-based type analysis, however, there is not much benefit in using a constraint system instead of a data flow based analysis even for a language with a rather simple type system.

On the contrary, a data flow based analysis does not have the problem of ambiguity and would also support casts reasonably well. For an inter-procedural analysis, a constraint system can still be constructed based on the results of the function analysis. This system will, however, be much simpler as it does not involve instruction-level constraints but only constraints to fill the types which could not be determined by an intra-procedural analysis.

6 Summary

With the increasing popularity of the Go programming language, also the analysis of Go binaries becomes increasingly important. Additionally to the generated machine code, the compiler adds a variety of metadata in the binary, which cannot be removed easily. This includes a table with information about functions along with the name of the function and type information for the garbage collector as well as the runtime type information required for garbage collection, reflection and dynamic typing. The included meta information can significantly aid the analysis of binaries and typically allows to distinguish code from the standard library from other source packages and therefore enables to identify important code sections. To this end, we documented important metadata structures such as the runtime symbol information table in addition to other language internals like the layout of data structures and the calling convention. Moreover, we proposed a strategy to extract the relevant metadata from stripped Go binaries, identifying and supporting different compiler versions.

As the standard Go compiler is designed for efficient compilation, it uses a custom code generation procedure together with unique calling conventions, causing problems for existing code analysis tools. We presented the Higher-level Go Assembly (HGA), a low-level intermediate representation based on assembly, to ease automated as well as manual analysis of Go programs. During lifting, many code idioms introduced by the compiler are compressed into explicit instructions or removed entirely. This results in a representation that is significantly easier to understand for a human analyst, but also simplifies automated analysis.

Based on this intermediate language, we explored the possibility of analyzing types of functions by solving SMT formulas by generating type constraints derived from the instruction stream and available type meta information. However, we found that due to structural problems not all language features can be covered appropriately. Furthermore, solving the generated constraint systems incurs major performance problems for complex functions. From this we conclude that additional analysis prior to the formulation of type constraints is required for an effective analysis.

Outlook

The presented approaches can be extended and improved in various directions.

The procedure to extract metadata can be extended to identify and support Go versions prior to Go 1.5 as well. Additionally, new formats added by newer releases can be added. Furthermore, it is possible to support for specific compiler idioms of older and upcoming Go versions in the HGA lifting procedure.

The HGA intermediate representation can also be extended to be agnostic of the underlying architecture to cover other architectures, e.g. ARM, as well.

An analysis of the variables stored in the stack frame combined with an explicit representation of this information in the HGA can ease all kinds of analysis, especially an analysis of the types of variables and function arguments. As a type analysis solely based on type constraints turned out to be difficult, a data flow based analysis can be implemented to reconstruct type information of a single function.

In addition, a control flow recovery algorithm can be implemented. Due to the simplicity of the language and the code generation procedure, it should be possible to recover a human-readable representation of the machine code.

A Usage Instructions

The developed tools consist of two parts: the first tool is named `rego` and implements the extraction of metadata from a given Go binary (ELF or PE, x86-64). It is written in Go and has to be compiled. The second tool is `regopy`, which provides the HGA representation and the type analysis and is written in Python 3[1]. The name *rego* is an abbreviation for *Reverse Engineering GO*.

A.1 Metadata Extraction

The binary has to be compiled first using the Go compiler. It has been tested with Go 1.8.

```
$ cd rego
$ go build
```

After building, the general usage from the command line is the following, processing a given `binary` with an optional list of additional type addresses (cf. Section 4.1.3):

```
$ ./rego -file <binary> [-rtti-types <addr1>[,<addr2>,...]] <command>
```

While processing the binary, some debug information is printed to `stderr`. The following commands are implemented:

- `dump-moduledata`: Dump the module data structure, if found.
- `dump-rtsi`: Dump a list of function address and name.
- `dump-rtti`: Dump type information in human readable format.
- `extract`: Extract all information requires for further analysis into JSON format, a redirection of `stdout` into a file is strongly recommended.

A.2 Lifting and Analysis

The code analysis tool operates on JSON files emitted by the extraction tool described above. The basic usage to process one or more functions is the following:

```
$ ./regopy/rego.py <json file> [-t] [-f <function> [<function> ...]]
```

This will lead to a lifting of the machine code of the specified functions into the canonical HGA. When the option `-t` is given, a type analysis on the specified functions will be performed. Otherwise, the canonical HGA will be printed.

If no functions are specified, all functions will be lifted into the canonical HGA, but *no* further analysis is performed. This can be used to ensure that the lifting procedure works on all functions.

[1] Tested with Python 3.5, earlier versions including Python 2 should also work.

B Implementation Remarks

Before giving an overview of the general structure of the code, we will briefly point out some important aspects of the lifting and analysis procedure of the HGA.

B.1 HGA Passes

The HGA lifting and analysis process is organized in *passes*. Starting from the function description extracted from the *pclntab*, passes usually modify the function (typically using *change sets*) or perform analysis (typically storing the result in the `analysis` field of the function or printing the result). There is a rough distinction between *function passes* and *basic block passes*: the latter operates on the basic blocks of the function instead of the function as a whole and must not add or remove blocks and should only change the assigned block. This distinction simplifies code and allows for further optimizations in future.

Functions are usually modified using *change sets*, which apply multiple changes to a function at once. This has the advantage that the instruction indices, which are used to refer individual instructions, do not change before the whole change set is applied. In addition, when the terminating instruction (e.g. a conditional jump) is modified, the predecessor links of other basic blocks are adjusted automatically.

B.2 Code Structure

The code is structured as follows:

- `rego/` — Extraction of metadata, written in Go
  - `main.go` — Command-line interface
  - `gobinary.go` — Finding and identifying and structures in binary
  - `rtsi.go` — Parsing of the *pclntab*
  - `rtti.go` — Parsing of the type information for Go 1.6 and Go 1.7+
  - `structs.go` — Definition of some structure layouts
  - `elfreader.go` — Helper to read virtual address space of ELF files
  - `pereader.go` — Helper to read virtual address space of PE files
- `regopy/` — Handling of machine code, HGA and analysis
  - `rego.py` — Command-line interface
  - `model/` — Implementation of the HGA
    - `hreg.py` — Definition of registers
    - `hins.py` — Definition of all instruction types

    * `hinstruction.py` — Instruction and instruction operands
    * `structures.py` — Basic blocks, functions and modules
  - `passes/` — Lifting, verification and analysis passes
    * `classes.py` — Definition of the pass classes
    * `passmanager.py` — Pass pipeline, contains default order of passes
    * `lift.py` — Initial lifting of assembly into the HGA
    * `deadendblocks.py` — Removal of Go specific basic blocks
    * `duffdevice.py` — Lifting of calls to Duff's device
    * `calls.py` — Identification of different call types
    * `instructionsizes.py` — Shrinking of widened instructions
    * `conversions.py` — Insertion of explicit extensions and truncations
    * `simplify.py` — Basic code simplification passes
    * `verify.py` — Verification of function to match the canonical HGA
    * `argumentanalysis.py` — Identification of argument and return bytes
    * `simpletypes.py` — Z3-based type analysis
    * ... (only a selection of important passes is listed here)
  - `doc.py` — Script to generate LaTeX documentation HGA as in Section D.3
  - ... (some scripts only serve our evaluation and are left for reference)

C Instructions Used by Go Compiler

The Go compiler (as of version 1.8) only emits a small number of instruction types.

Memory Access Instructions

- Non-atomic: `mov`, `movzx`, `movsx`, `movss`, `movsd`, `movups`, `rep stosq`, `rep movsq`
- Atomic: `xchg`, `lock cmpxchg`, `lock xadd`, `lock and`, `lock or`
- Special Cases:
  - `mov rcx, fs:[-8]` – used to get `g` pointer
  - `test al, [rax]` – used for `nil`-checks
  - `cmp` *reg*`, [rcx+0x10]` – used for small stack bounds checking
  - `cmp` *reg*`, [rcx+0x18]` – used for large stack bounds checking
  - `cmp [rbx], rdi` – used in prefix of wrapper functions

General-Purpose Instructions

- Moves: `mov`, `movzx`, `movsx`, `cmoveq`, `set`*cc*
- Control-flow: `call`, `jmp`, `j`*cc*, `ud2`, `int3`
- Arithmetic: `add`, `inc`, `sub`, `dec`, `imul`, `mul`, `idiv`, `div`, `neg`
- Bitwise: `and`, `or`, `xor`, `not`, `shl`, `shr`, `sar`, `rol`, `bsf`, `bswap`
- Comparison: `cmp`, `test`
- Address Generation/`lea` – local combination of additions and shifts created during optimization (except for `rip`-relative addressing)
- Special Cases:
  - `rcr` – used after `add` to compute average
  - `sbb` – used after shifts for conditional masking
  - `cqo`/`cdq`/`cwd` – used before division

Floating-point Instructions

- Conversions: `cvttsd2si`, `cvttss2si`, `cvtsi2ss`, `cvtsi2sd`, `cvtsd2ss`, `cvtss2sd`
- Arithmetic: `addss`, `addsd`, `subss`, `subsd`, `mulss`, `mulsd`, `divss`, `divsd`, `sqrtsd`
- Comparison: `ucomiss`, `ucomisd`
- Special Cases:
  - `pxor` – used for negation
  - `xorps xmm`*n*`, xmm`*n* – used to zero register

D Higher-level Go Assembly

D.1 Registers

Register	Access Sizes	Description
`AX`,`CX`,`DX`,...,`R14`,`R15`	1,2,4,8	General purpose registers
`X0`,`X1`,`X2`,...,`X14`,`X15`	4,8,16	SSE registers
`SB`	8	Base address of executable in memory
`FLAGS`	–	Flags pseudo-register, only used to mark read and write operations on the flags
`TMP1`	1,2,4,8	Used to simplify x86-64 memory operands
`TMP2`,`TMP3`	1,2,4	Used to make register truncations explicit
`TMP4`	1,2,4,8	Used to eliminate `test` instructions
`TMP5`	1,2,4,8	Used to split out memory operand into separate instructions

D.2 Operands

All immediate operands begin with a dollar sign (`$`) and are followed by the number, usually written in prefixed hexadecimal form (e.g. `$0xabcdef`). An optional negation sign is put immediately after the dollar sign. All immediates are signed and encoded in two's complement in the operand size.

Memory operands are enclosed in square brackets (`[]`), the components are separated using a plus sign (`+`). The first component is the base register. The base register must not be `FLAGS` or an SSE register. If the base register is `SB`, there can exist an immediate offset as second component, written like other immediate operands (e.g. `[SB+$0x401234]`). If the base register is `SP`, beside an immediate as last component there might also exist an offset register as second component. The offset register must not be `SB`, `SP`, `FLAGS` or an SSE register. For all other base registers, no other components are allowed.

Register operands consist of the register name, registers can only be accessed in the allowed access sizes (see above). The registers `SB` and `FLAGS` are not allowed.

Every operand is immediately followed by a tilde (`~`) and the size of the operand written as a decimal number.

D.3 Instruction List

Format	S	U	B	Description
Abbreviations: **S**ets Flags — **U**ses Flags — **B**reaks Control Flow `r`/`reg`=general purpose register – `x`/`xmm`=SSE register – `i`/`imm`=immediate – `mem`=memory If specified, number at the end of the operand indicates size in bits				
`MOVB reg8,ri8`				`reg8` = `ri8`
`MOVW reg16,ri16`				`reg16` = `ri16`
`MOVL reg32,ri32`				`reg32` = `ri32`
`MOVQ reg64,ri64`				`reg64` = `ri64`
`MOVO xmm128,xi128`				`xmm128` = `xi128`
`MOVSS xmm32,xi32`				`xmm32` = `xi32`
`MOVSD xmm64,xi64`				`xmm64` = `xi64`
Memory				
`MOVBload reg8,mem8`				`reg8` = load from `mem8`
`MOVWload reg16,mem16`				`reg16` = load from `mem16`
`MOVLload reg32,mem32`				`reg32` = load from `mem32`
`MOVQload reg64,mem64`				`reg64` = load from `mem64`
`MOVOload xmm128,mem128`				`xmm128` = load from `mem128`
`MOVSSload xmm32,mem32`				`xmm32` = load from `mem32`
`MOVSDload xmm64,mem64`				`xmm64` = load from `mem64`
`MOVBstore mem8,ri8`				store `ri8` to `mem8`
`MOVWstore mem16,ri16`				store `ri16` to `mem16`
`MOVLstore mem32,ri32`				store `ri32` to `mem32`
`MOVQstore mem64,ri64`				store `ri64` to `mem64`
`MOVOstore mem128,xmm128`				store `xmm128` to `mem128`
`MOVSSstore mem32,xmm32`				store `xmm32` to `mem32`
`MOVSDstore mem64,xmm64`				store `xmm64` to `mem64`
`REPSTOSQ`†				Zero 8·`CX` bytes at `DI`
`REPMOVSQ`†				Move 8·`CX` bytes from `SI` to `DI`
`DuffZero mem`†				Zero memory at `mem`
`DuffCopy` $\texttt{mem}_1$`,`$\texttt{mem}_2$†				Move memory from $\texttt{mem}_2$ to $\texttt{mem}_1$
`MemZero mem`				Zero memory at `mem`
`MemMove` $\texttt{mem}_1$`,`$\texttt{mem}_2$				Move memory from $\texttt{mem}_2$ to $\texttt{mem}_1$
`XCHGL mem32,reg32`				swap `mem32` and `reg32` (atomic)
`XCHGQ mem64,reg64`				swap `mem64` and `reg64` (atomic)
`XADDLlock mem32,reg32`				cf. [19], instruction `xadd`
`XADDQlock mem64,reg64`				cf. [19], instruction `xadd`
`CMPXCHGLlock mem32,reg32`	x			cf. [19], instr. `cmpxchg`, `rax` is ignored
`CMPXCHGQlock mem64,reg64`	x			cf. [19], instr. `cmpxchg`, `rax` is ignored
`ANDBlock mem8,reg8`				`mem8` = `mem8` & `reg8` (atomic)
`ORBlock mem8,reg8`				`mem8` = `mem8` | `reg8` (atomic)
Integer Arithmetic				
`ADDB reg8,ri8`				`reg8` = `reg8` + `ri8`
`ADDW reg16,ri16`				`reg16` = `reg16` + `ri16`

Format	S	U	B	Description
`ADDL reg32,ri32`				`reg32` = `reg32` + `ri32`
`ADDQ reg64,ri64`				`reg64` = `reg64` + `ri64`
`SUBB reg8,ri8`				`reg8` = `reg8` − `ri8`
`SUBW reg16,ri16`				`reg16` = `reg16` − `ri16`
`SUBL reg32,ri32`				`reg32` = `reg32` − `ri32`
`SUBQ reg64,ri64`				`reg64` = `reg64` − `ri64`
`NEGB reg8`				`reg8` = −`reg8`
`NEGW reg16`				`reg16` = −`reg16`
`NEGL reg32`				`reg32` = −`reg32`
`NEGQ reg64`				`reg64` = −`reg64`
`MULB reg8,ri8`				`reg8` = `reg8` · `ri8`
`MULW reg16,ri16`				`reg16` = `reg16` · `ri16`
`MULL reg32,ri32`				`reg32` = `reg32` · `ri32`
`MULQ reg64,ri64`				`reg64` = `reg64` · `ri64`
`MULQU2 reg64`$_1$`,reg64`$_2$`,reg64`$_3$				`reg64`$_1$:‡`reg64`$_2$ = `reg64`$_2$ · `reg64`$_3$
`HMULB reg8`$_1$`,reg8`$_2$`,reg8`$_3$				`reg8`$_1$ = (`reg8`$_2$ · `reg8`$_3$) ≫ 8 (signed)
`HMULW reg16`$_1$`,reg16`$_2$`,reg16`$_3$				`reg16`$_1$ = (`reg16`$_2$ · `reg16`$_3$) ≫ 16 (signed)
`HMULL reg32`$_1$`,reg32`$_2$`,reg32`$_3$				`reg32`$_1$ = (`reg32`$_2$ · `reg32`$_3$) ≫ 32 (signed)
`HMULQ reg64`$_1$`,reg64`$_2$`,reg64`$_3$				`reg64`$_1$ = (`reg64`$_2$ · `reg64`$_3$) ≫ 64 (signed)
`HMULBU reg8`$_1$`,reg8`$_2$`,reg8`$_3$				`reg8`$_1$ = (`reg8`$_2$ · `reg8`$_3$) ≫ 8 (unsigned)
`HMULWU reg16`$_1$`,reg16`$_2$`,reg16`$_3$				`reg16`$_1$ = (`reg16`$_2$ · `reg16`$_3$) ≫ 16 (unsigned)
`HMULLU reg32`$_1$`,reg32`$_2$`,reg32`$_3$				`reg32`$_1$ = (`reg32`$_2$ · `reg32`$_3$) ≫ 32 (unsigned)
`HMULQU reg64`$_1$`,reg64`$_2$`,reg64`$_3$				`reg64`$_1$ = (`reg64`$_2$ · `reg64`$_3$) ≫ 64 (unsigned)
`DIVW reg16`$_1$`,reg16`$_2$`,reg16`$_3$				`reg16`$_1$ = `reg16`$_2$ % `reg16`$_3$ (signed) `reg16`$_2$ = `reg16`$_2$ / `reg16`$_3$ (signed)
`DIVL reg32`$_1$`,reg32`$_2$`,reg32`$_3$				`reg32`$_1$ = `reg32`$_2$ % `reg32`$_3$ (signed) `reg32`$_2$ = `reg32`$_2$ / `reg32`$_3$ (signed)
`DIVQ reg64`$_1$`,reg64`$_2$`,reg64`$_3$				`reg64`$_1$ = `reg64`$_2$ % `reg64`$_3$ (signed) `reg64`$_2$ = `reg64`$_2$ / `reg64`$_3$ (signed)
`DIVWU reg16`$_1$`,reg16`$_2$`,reg16`$_3$				`reg16`$_1$ = `reg16`$_2$ % `reg16`$_3$ (unsigned) `reg16`$_2$ = `reg16`$_2$ / `reg16`$_3$ (unsigned)
`DIVLU reg32`$_1$`,reg32`$_2$`,reg32`$_3$				`reg32`$_1$ = `reg32`$_2$ % `reg32`$_3$ (unsigned) `reg32`$_2$ = `reg32`$_2$ / `reg32`$_3$ (unsigned)
`DIVQU reg64`$_1$`,reg64`$_2$`,reg64`$_3$				`reg64`$_1$ = `reg64`$_2$ % `reg64`$_3$ (unsigned) `reg64`$_2$ = `reg64`$_2$ / `reg64`$_3$ (unsigned)
`DIVQU2 reg64`$_1$`,reg64`$_2$`,reg64`$_3$				`reg64`$_1$ = `reg64`$_1$:‡`reg64`$_2$ % `reg64`$_3$ (unsigned) `reg64`$_2$ = `reg64`$_1$:‡`reg64`$_2$ / `reg64`$_3$ (unsigned)
`SARB reg8,ri8`				`reg8` = `reg8` ≫ `ri8` (arithmetic)
`SARW reg16,ri8`				`reg16` = `reg16` ≫ `ri8` (arithmetic)
`SARL reg32,ri8`				`reg32` = `reg32` ≫ `ri8` (arithmetic)
`SARQ reg64,ri8`				`reg64` = `reg64` ≫ `ri8` (arithmetic)
`SHRB reg8,ri8`				`reg8` = `reg8` ≫ `ri8`
`SHRW reg16,ri8`				`reg16` = `reg16` ≫ `ri8`
`SHRL reg32,ri8`				`reg32` = `reg32` ≫ `ri8`
`SHRQ reg64,ri8`				`reg64` = `reg64` ≫ `ri8`

Format	S	U	B	Description
SHLB reg8,ri8				reg8 = reg8 ≪ ri8
SHLW reg16,ri8				reg16 = reg16 ≪ ri8
SHLL reg32,ri8				reg32 = reg32 ≪ ri8
SHLQ reg64,ri8				reg64 = reg64 ≪ ri8
ROLB reg8,ri8				reg8 = reg8 ⋘ ri8
ROLW reg16,ri8				reg16 = reg16 ⋘ ri8
ROLL reg32,ri8				reg32 = reg32 ⋘ ri8
ROLQ reg64,ri8				reg64 = reg64 ⋘ ri8
CMPB reg8,ri8	x			flags = flags of operation reg8 − ri8
CMPW reg16,ri16	x			flags = flags of operation reg16 − ri16
CMPL reg32,ri32	x			flags = flags of operation reg32 − ri32
CMPQ reg64,ri64	x			flags = flags of operation reg64 − ri64
TESTB reg8,ri8†	x			flags = flags of operation reg8 & ri8
TESTW reg16,ri16†	x			flags = flags of operation reg16 & ri16
TESTL reg32,ri32†	x			flags = flags of operation reg32 & ri32
TESTQ reg64,ri64†	x			flags = flags of operation reg64 & ri64
ANDB reg8,ri8				reg8 = reg8 & ri8
ANDW reg16,ri16				reg16 = reg16 & ri16
ANDL reg32,ri32				reg32 = reg32 & ri32
ANDQ reg64,ri64				reg64 = reg64 & ri64
ORB reg8,ri8				reg8 = reg8 | ri8
ORW reg16,ri16				reg16 = reg16 | ri16
ORL reg32,ri32				reg32 = reg32 | ri32
ORQ reg64,ri64				reg64 = reg64 | ri64
XORB reg8,ri8				reg8 = reg8 ⊕ ri8
XORW reg16,ri16				reg16 = reg16 ⊕ ri16
XORL reg32,ri32				reg32 = reg32 ⊕ ri32
XORQ reg64,ri64				reg64 = reg64 ⊕ ri64
NOTB reg8				reg8 = bitwise not of reg8
NOTW reg16				reg16 = bitwise not of reg16
NOTL reg32				reg32 = bitwise not of reg32
NOTQ reg64				reg64 = bitwise not of reg64
Extensions and Truncations				
CWD $reg16_1$,$reg16_2$				$reg16_1$ = sign-extension of $reg16_2$
CDQ $reg32_1$,$reg32_2$				$reg32_1$ = sign-extension of $reg32_2$
CQO $reg64_1$,$reg64_2$				$reg64_1$ = sign-extension of $reg64_2$
MOVBWSX reg16,reg8				reg16 = sign-extended value of reg8
MOVBLSX reg32,reg8				reg32 = sign-extended value of reg8
MOVBQSX reg64,reg8				reg64 = sign-extended value of reg8
MOVWLSX reg32,reg16				reg32 = sign-extended value of reg16
MOVWQSX reg64,reg16				reg64 = sign-extended value of reg16
MOVLQSX reg64,reg32				reg64 = sign-extended value of reg32
MOVBWZX reg16,reg8				reg16 = zero-extended value of reg8
MOVBLZX reg32,reg8				reg32 = zero-extended value of reg8

Format	S	U	B	Description
`MOVBQZX reg64,reg8`				`reg64` = zero-extended value of `reg8`
`MOVWLZX reg32,reg16`				`reg32` = zero-extended value of `reg16`
`MOVWQZX reg64,reg16`				`reg64` = zero-extended value of `reg16`
`MOVLQZX reg64,reg32`				`reg64` = zero-extended value of `reg32`
`Trunc16to8 reg8,reg16`				`reg8` = truncation of `reg16`
`Trunc32to8 reg8,reg32`				`reg8` = truncation of `reg32`
`Trunc32to16 reg16,reg32`				`reg16` = truncation of `reg32`
`Trunc64to8 reg8,reg64`				`reg8` = truncation of `reg64`
`Trunc64to16 reg16,reg64`				`reg16` = truncation of `reg64`
`Trunc64to32 reg32,reg64`				`reg32` = truncation of `reg64`
Miscellaneous				
`NOP`				no operation
`LEAQ reg64,mem`				`reg64` = address of `mem` (stack and global only)
`RCRQ reg64,imm64`[†]				`reg64` = `carry:`[‡]`reg64` $\ggg$ `imm64`
`AVGQU` $\texttt{reg64}_1\texttt{,reg64}_2$				$\texttt{reg64}_1$ = average of $\texttt{reg64}_1$ and $\texttt{reg64}_2$
`BSFL` $\texttt{reg32}_1\texttt{,reg32}_2$				$\texttt{reg32}_1$ = index of least sign. set bit in $\texttt{reg32}_2$
`BSFQ` $\texttt{reg64}_1\texttt{,reg64}_2$				$\texttt{reg64}_1$ = index of least sign. set bit in $\texttt{reg64}_2$
`BSWAPL reg32`				`reg32` = reverse bytes of `reg32`
`BSWAPQ reg64`				`reg64` = reverse bytes of `reg64`
`SET`*cc*[§]`reg8`		x		`reg8` = 1 if *cc* else 0
`SBBBcarrymask reg8`		x		`reg8` = -1 if carry flag set else 0
`SBBWcarrymask reg16`		x		`reg16` = -1 if carry flag set else 0
`SBBLcarrymask reg32`		x		`reg32` = -1 if carry flag set else 0
`SBBQcarrymask reg64`		x		`reg64` = -1 if carry flag set else 0
`GetG reg64`				`reg64` = pointer to `g` in thread-local storage
`GetSP reg64`				`reg64` = stack pointer
`GetBP reg64`				`reg64` = callee-save frame pointer
`GetClosurePtr reg64`				`reg64` = closure pointer
`NilCheck reg64`				segfault if `reg64` is not dereferencable
`CheckIndex` $\texttt{ri}_1\texttt{,ri}_2$				panic unless $\texttt{ri}_1 < \texttt{ri}_2$
`CheckSlice` $\texttt{ri}_1\texttt{,ri}_2$				panic unless $\texttt{ri}_1 \leq \texttt{ri}_2$
Floating-Point Arithmetic				
`Cvt32to32F xmm32,reg32`				`xmm32` = type-conversion of `reg32`
`Cvt32to64F xmm64,reg32`				`xmm64` = type-conversion of `reg32`
`Cvt64to32F xmm32,reg64`				`xmm32` = type-conversion of `reg64`
`Cvt64to64F xmm64,reg64`				`xmm64` = type-conversion of `reg64`
`Cvt32Fto32 reg32,xmm32`				`reg32` = type-conversion of `xmm32`
`Cvt32Fto64 reg64,xmm32`				`reg64` = type-conversion of `xmm32`
`Cvt64Fto32 reg32,xmm64`				`reg32` = type-conversion of `xmm64`
`Cvt64Fto64 reg64,xmm64`				`reg64` = type-conversion of `xmm64`
`Cvt32Fto64F xmm64,xmm32`				`xmm64` = type-conversion of `xmm32`
`Cvt64Fto32F xmm32,xmm64`				`xmm32` = type-conversion of `xmm64`
`UCOMISS` $\texttt{xmm32}_1\texttt{,xmm32}_2$	x			`flags` = float32 compare $\texttt{xmm32}_1$ and $\texttt{xmm32}_2$
`UCOMISD` $\texttt{xmm64}_1\texttt{,xmm64}_2$	x			`flags` = float64 compare $\texttt{xmm64}_1$ and $\texttt{xmm64}_2$

Format	S	U	B	Description
`ADDSS` $\texttt{xmm32}_1$,$\texttt{xmm32}_2$				$\texttt{xmm32}_1 = \texttt{xmm32}_1 + \texttt{xmm32}_2$ (float32)
`ADDSD` $\texttt{xmm64}_1$,$\texttt{xmm64}_2$				$\texttt{xmm64}_1 = \texttt{xmm64}_1 + \texttt{xmm64}_2$ (float64)
`SUBSS` $\texttt{xmm32}_1$,$\texttt{xmm32}_2$				$\texttt{xmm32}_1 = \texttt{xmm32}_1 - \texttt{xmm32}_2$ (float32)
`SUBSD` $\texttt{xmm64}_1$,$\texttt{xmm64}_2$				$\texttt{xmm64}_1 = \texttt{xmm64}_1 - \texttt{xmm64}_2$ (float64)
`MULSS` $\texttt{xmm32}_1$,$\texttt{xmm32}_2$				$\texttt{xmm32}_1 = \texttt{xmm32}_1 \cdot \texttt{xmm32}_2$ (float32)
`MULSD` $\texttt{xmm64}_1$,$\texttt{xmm64}_2$				$\texttt{xmm64}_1 = \texttt{xmm64}_1 \cdot \texttt{xmm64}_2$ (float64)
`DIVSS` $\texttt{xmm32}_1$,$\texttt{xmm32}_2$				$\texttt{xmm32}_1 = \texttt{xmm32}_1 \,/\, \texttt{xmm32}_2$ (float32)
`DIVSD` $\texttt{xmm64}_1$,$\texttt{xmm64}_2$				$\texttt{xmm64}_1 = \texttt{xmm64}_1 \,/\, \texttt{xmm64}_2$ (float64)
`SQRTSD` $\texttt{xmm64}_1$,$\texttt{xmm64}_2$				$\texttt{xmm64}_1 = \sqrt{\texttt{xmm64}_2}$
`PXOR` $\texttt{xmm128}_1$,$\texttt{xmm128}_2$				$\texttt{xmm128}_1 = \texttt{xmm128}_1 \oplus \texttt{xmm128}_2$
`PXORSS` $\texttt{xmm32}_1$,$\texttt{xmm32}_2$				$\texttt{xmm32}_1 = \texttt{xmm32}_1 \oplus \texttt{xmm32}_2$
`PXORSD` $\texttt{xmm64}_1$,$\texttt{xmm64}_2$				$\texttt{xmm64}_1 = \texttt{xmm64}_1 \oplus \texttt{xmm64}_2$
Control Flow				
`J`*cc*[§] $\texttt{imm64}_1$,$\texttt{imm64}_2$		x	x	if *cc* jump to block $\texttt{imm64}_1$ else $\texttt{imm64}_2$
`JMP imm64`			x	jump to block `imm64`
`RET`			x	return
`Unreachable`			x	unreachable
`CALL ri64`[†]				call function at `ri64`
`StaticCall imm64`				call function at `imm64`
`ClosureCall reg64`				call closure `reg64`
`InterCall reg64`				call interface function at `reg64`
`GoCall reg64,mem`				start goroutine with closure `reg64` with arguments `mem` (stack only)
`DeferCall reg64,mem`				defer call to closure `reg64` with arguments `mem` (stack only)
Pseudo Instructions				
`CmpStackLimit`	x			`flags` = comparison of `SP` with stack limit
`MoreStack`				enlarge stack if required and set `DX`=0
`MoreStackCtxt`				enlarge stack if required
`SizeAssert reg`[†]				enforce a register size, cf. Sec. 4.3.8
`---` $\texttt{imm64}_1$,$\texttt{imm64}_2$				marker for address and frame size, ignored

[†] non-canonical HGA only
[‡] `high:low` denotes the concatenation of two registers
[§] condition codes are the same as on x86-64, refer to [19] for a description

D.4 Basic Blocks

A basic block has a number unique within function scope and is a sequence of at least one instruction, control flow breaking instructions must be last. Instructions are zero-indexed, address markers (`---`) do not count, cannot be indexed directly, and are merely treated as comments. Each block should have an address marker before the instruction with index zero to correctly indicate the stack frame size.

Basic blocks are referred to by their number. The basic block with number zero must not have predecessors and is the entry point of the function.

E Vulnerable Safe Go Program

A proof-of-concept of a vulnerable Go binary without using the unsafe or reflection package and without using allowing data races.

```
1  package main
2
3  import (
4      "fmt"
5      "os"
6      "strconv"
7      "syscall"
8  )
9
10 // Source for trick to retrieve address (accessed 2017-08-18):
11 // https://blog.stalkr.net/2013/06/golang-heap-corruption-during-garbage.html
12
13 func main() {
14     // This defer places a structure containing a function pointer on the heap
15     // right before the buffer allocated below.
16     defer fmt.Println("pwn harder")
17
18     var buf *[96]byte = new([96]byte)
19     addr, _ := strconv.ParseUint(fmt.Sprintf("%p", &buf), 0, 0)
20     fmt.Printf("Address is %p\n", buf)
21
22     // Overwrite pointer to buffer and read to the new address
23     syscall.RawSyscall(0, 0, uintptr(addr), 8)
24     os.Stdin.Read(buf[:])
25 }
```

Acronyms

API Application Programming Interface.

ASLR Address Space Layout Randomization.

CFG Control Flow Graph.

DDoS Distributed Denial of Service.

ELF Executable and Linkable Format.

GC Garbage Collector.

HGA Higher-level Go Assembly.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

IP Internet Protocol.

IR Intermediate Representation.

ISA Instruction Set Architecture.

JSON JavaScript Object Notation.

PC Program Counter.

PE Portable Executable.

PIC Position-Independent Code.

RELRO Relocation Read-Only.

RIP Instruction Pointer.

RTTI Runtime Type Information.

SAT Boolean Satisfiability.

SMT Satisfiable Modulo Theories.

SSA Single-Static Assignment.

SSE Streaming SIMD Extension.

SSH Secure Shell.

TLS Thread-local Storage.

URL Uniform Resource Locator.

List of Figures

List of Tables

Listings

Bibliography

[1] Gogul Balakrishnan and Thomas Reps. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*. Springer, 2004.

[2] Clark Barrett, Leonardo de Moura, and Pascal Fontaine. Proofs in Satisfiability Modulo Theories. *All about Proofs, Proofs for All*, 2015.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

[4] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 26, pages 825–885. IOS Press, February 2009.

[5] Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. An SMT-LIB Format for Sequences and Regular Expressions. 2012.

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[7] Russ Cox. Go Data Structures: Interfaces. `https://research.swtch.com/interfaces`, accessed 2017-03-25, December 2009.

[8] Russ Cox. Off to the Races. `https://research.swtch.com/gorace`, accessed 2017-05-10, February 2010.

[9] Russ Cox. Go 1.1 Function Calls. `https://docs.google.com/document/d/1bMwCey-gmqZVTpRax-ESeVuZGmjwbocYs1iHplK-cjo/pub`, accessed 2017-08-28, February 2013.

[10] Russ Cox. Go 1.2 Runtime Symbol Information. `http://golang.org/s/go12symtab`, accessed 2017-03-25, July 2013.

[11] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[12] Nick Diakopoulos and Stephen Cass. IEEE Spectrum: The Top Programming Languages 2017. `http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017`, accessed 2017-08-23, July 2017.

[13] Ecma International. Standard ECMA-404: The JSON Data Interchange Format, October 2013.

[14] A. Fokin, K. Troshina, and A. Chernov. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *14th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 240–243, March 2010.

[15] Andrew Gerrand. The Go Blog: Go Slices: usage and internals. `https://blog.golang.org/go-slices-usage-and-internals`, accessed 2017-04-01, January 2011.

[16] Andrew Gerrand. The Go Blog: Go 2016 Survey Results. `https://blog.golang.org/survey2016-results`, accessed 2017-08-23, March 2016.

[17] Hex-Rays SA. Hex-Rays Decompiler. `https://hex-rays.com/products/decompiler/`, accessed 2017-08-25.

[18] Hex-Rays SA. Interactive Disassembler (IDA). `https://hex-rays.com/products/ida/`, accessed 2017-08-22.

[19] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2200 Mission College Blvd. Santa Clara, CA 95054-1549 USA, April 2016. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, visited 2017-08-19.

[20] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. Labeling Library Functions in Stripped Binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 2011.

[21] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2011.

[22] LLVM Project. The LLVM Compiler Infrastructure. `http://llvm.org`, accessed 2017-08-25.

[23] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.99.7. `http://web.archive.org/web/20160315222117/http://www.x86-64.org/documentation_folder/abi.pdf`, accessed 2017-08-18, November 2014.

[24] Daniel Morsing. How Stacks are Handled in Go. `https://blog.cloudflare.com/how-stacks-are-handled-in-go/`, accessed 2017-04-05, September 2014.

[25] Alan Mycroft. Type-Based Decompilation. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 208–223. Springer-Verlag, 1999.

[26] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. MARX: Uncovering class hierarchies in C++ programs. In *NDSS*, 2017.

[27] Rob Pike. The Go Blog: Strings, bytes, runes and characters in Go. `https://blog.golang.org/strings`, accessed 2017-08-18, October 2013.

[28] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation Recovery from Low-level Code. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 100–111. ACM, 2006.

[29] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[30] Ian Lance Taylor. The Go Blog: Gccgo in GCC 4.7.1. `https://blog.golang.org/gccgo-in-gcc-471`, accessed 2017-08-25, July 2012.

[31] The Go Authors. Frequently Asked Questions (FAQ). `https://golang.org/doc/faq`, accessed 2017-04-06.

[32] The Go Authors. The Go Programming Language Specification. `https://golang.org/ref/spec`, accessed 2017-03-25, November 2016.

[33] The Go Authors. Release History. `https://golang.org/doc/devel/release.html`, accessed 2017-07-29, 2017.

[34] Michael James van Emmerik. Signatures for Library Functions in Executable Files. Technical report, Queensland University of Technology, 1994.

[35] Michael James van Emmerik. *Single Static Assigment for Decompolation*. PhD thesis, University of Queensland, May 2007.

[36] Dmitry Vyukov and Andrew Gerrand. The Go Blog: Introducing the Go Race Detector. `https://blog.golang.org/race-detector`, accessed 2017-05-10, June 2013.