Ugo: Decompilation of Go

Leon Chou*
New York University

Kent Ma*
New York University

Abstract

Accurate recovery of higher level abstraction is the main goal of a decompiler. The Go programming language lacks reverse engineering tools, with the unique type system and calling convention of compiled Go programs foiling industry standard decompilers. In the past year, malware and capture-the-flag challenges have been written in Go as a form of obfuscation. In this paper, we describe our IDAPro Hex-rays plugin which recovers function metadata using information left by the Go compiler, correctly handles Gos calling convention and restores type information of data items.

1 Introduction

The readability of binary programs is one of the key principles to reverse engineering and dynamic analysis. A decompiler allows for analysts to abstract away lower level concepts and create output that more closely matches higher level source code. Go malware, as well as Go binaries in the current CTF meta, have become much more common in recent years. Linux.Rex.1 and gopherz from CSAW CTF are good examples of these, respectively.

Go is a popular language for binary obfuscation because of the lack of debugging and reverse engineering tooling for the language. Its abstractions and unique mechanics hinder current binary analysis tooling.

In this paper, we present a methodology of analyzing Go binaries based on the Go compiler leaving information in the binary that allows us to retrieve information about functions and their return types. We preprocess that information, and when we reach calls in the binarys execution we do a lookup to retrieve information that allows us to assign return values to local variables.

Our implementation of these decompiler extensions is a plugin on IDAPros Hex-Rays decompiler. Our name for this is *ugo*.

In summary, we make the following contributions:

- Recover function metadata for go binaries
- Modify generated ASTs to match calling convention
- Locate new functions that are type information sinks
- Use calling convention recovery to improve type recovery

2 Related Works

There has been extensive work on decompilation of C binaries. This comes primarily in two forms: abstract data type recovery and control flow structuring.

Lee et al. have done work for recovering types in binary programs [5] based on type sinks. This involves solving two primary problems with decompilation of abstract data structures: type recovery and variable recovery. Once these primary problems are solved, their inferred type can be propagated to subsequent untainted uses of the data.

IDAPro Hex-rays and Schwartz et al. have previously done work for recovering high-level control structures [9]. These techniques primarily involve taking the control-flow graphs and recovered types from TIE and BAP and matching predefined graph schemas.

Alternatively, Yakdan et al. have a pattern independent control-flow structuring algorithm [**?**]. However, the DREAM decompiler has yet to be released.
The industry standard for decompilation of C binaries is the IDA Hexrays plugin [3]. There is the IDA FLIRT feature for detecting known library subroutines based on matching signatures [6]. This can be applied to Go binaries, as those are statically linked with the Go runtime.

There hasnt been any previous work on explicitly decompiling Go - work on C decompilation fails to recover

key layers of abstraction used in Go. However, there has been some work on reverse engineering malware that exist as stripped Go binaries. This work is focused entirely on type information recovery.

Tim Strazzeres work is the first published on type recovery of Go programs. He determines the entrypoint for Go programs as the main_main function and recovers function boundaries using signature detection based on stack allocation properties of Go functions. He also correctly labels string boundaries in the binary, which he uses to recover symbol names of stripped functions [10]. In doing so, he additionally discovered a list of type information structures (rtypes) for functions (called the .gopclntab) left as a linker artifact [11].

Using the type information structure described by Strazzere, Sergi Martinez discovered symbols for type names inside of the rtype structure, and used Strazzeres string boundary recovery to recover them [7]. He then performed type recovery with the runtime.newobject function as a type sink. As its name suggests, this function initializes an object and one of its parameters is an rtype of the to-be-initialized type.

George Zaytsev further expands on both Martinez and Strazzeres work for type information recovery [12]. He discovers a list of members of custom struct types included in the rtype structure, and writes an IDA plugin to recover struct layouts. He additionally identifies two additional linking artifacts in the binary as promising future areas of information recovery: the moduledata segment and the .typelink segment.

Another data structure unique to Go is the channel. This is a typed conduit which can concurrently send and receive values. Common Go programs extensively use this data structure in switch statement structures, which decompile illegibly. However, there has been work for reverse engineering them for a CTF competition problem [1].

3 Background

Go is a compiled programming language created at Google that has garbage collection and structural typing [2].

It is possible for Go functions to return multiple values, with each value assigned to a set of variables. This is frequently used for error handling in the language, with functions frequently having one of many return values be an error value.

```
func Atoi(s string) (int, error);
i, err := Atoi(s)
```

Conformance to structural typing is statically checked at compile time. However, it implements runtime polymorphism with interfaces. Any type that implements all methods of an interface conforms to that interface.

One key corner case is the empty `interface{}` type. This can refer to any data with a concrete type, including primitives. At runtime, code using this `interface{}` type can convert it into more useful types with typecasts or can inspect the type information with Gos builtin reflect package.

Go is designed to be concurrency-friendly with built-in types to support multithreading and a keyword that instantiates a goroutine - gos internal implementation of threading that prevents wholesale register swapping by the cpu, in favor of slightly higher level memory management by the runtime. Each Go function is designed to be independent of other functions, and each function is able to be treated as its own separate goroutine.

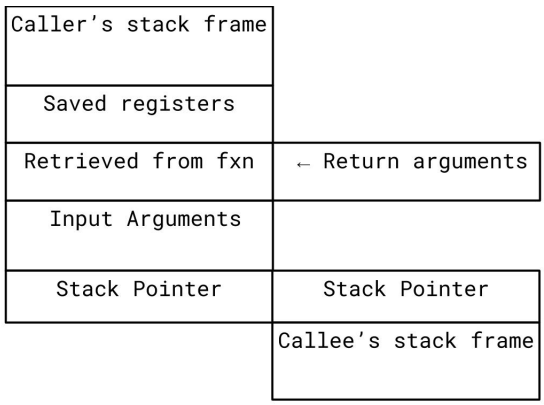
4 Design

4.1 Function Recovery

The Go runtime allocates a contiguous piece of memory for the stack, which is grown by reallocation/copy when filled. There is a stack overflow check called at the start of every function. If triggered, the runtime calls the function runtime_morestack_noctxt, which allocates a new stack of exponentially larger size for the routine and copies over its old stack contents. These check and allocation calls are removed from the final decompiled output, as they are implicitly called by every Go function.

Multiple return of functions is implemented at the assembly level by Gos calling convention. There are two types of arguments: input arguments and return arguments, with return arguments above input arguments on the stack. Callers put input arguments directly above the stack frame of the called function. Registers are also caller saved, and stored on the stack above the return arguments. This stack space is offset starting from the base of the frame for the function, and reused by subsequent function calls (starting again from the base of the frame). Callee functions put return arguments on the stack allocated by their parent. It is necessary to label which stack variables are input arguments and which are return arguments.

Metadata for functions are stored in a segment of the binary called the runtime.pclntab. The runtime.pclntab section of the Go binary is a program counter-line table of all function definitions used by the runtime for error logging when the program panics. Its header starts with the magic signature 0xFFFFFFFB, two zeros, a pc quantum, and the pointer size - which is then followed by a value that represents how many pcln table entries for functions exist. We use this number to enumerate all the functions that exist in the binary and retrieve metadata about those functions.



Each entry in the pcln table is structured as a two word pair, with the first word being the address of a function and the second word being the offset from the pclntab to the functions metadata structure.

4.2 Type Recovery

Every data item in Go is an `interface{}` type. Internally, this means that each type is actually a pair of pointers, the first resolving to an itab type and the second pointing to binary data for the struct.

The itab struct holds pointers that point to more metadata about the data item. It also contains an array of methods for the matching type, fun

. The most significant one here is the _type pointer, that points to a _type struct.

```
type itab struct {
  inter     *interfacetype
  _type     *_type
  hash      uint32         // copy of
  _         [4]byte
  fun       [1]uintptr     // variable in length
}
```

Every data item in Go has an associated rtype structure (also named the _type structure) which contains more metadata about the data, primarily used by the Go runtimes garbage collector.

This rtype structure contains information about the name of the name and size of the type. This allows for recovery of custom data structures such as user-defined structs even for symbol-stripped binaries.

```
type _type struct {
  size         uintptr
  ptrdata      uintptr
  hash         uint32
  tflag        tflag
  align        uint8
  fieldalign   uint8
  kind         uint8
  alg          *typeAlg
  gcdata       *byte
  str          nameOff
  ptrToThis    typeOff
}
```

Although rtype data is included by the linker for every symbol, they are scattered throughout the .rodata segment of the binary program and there is no explicit segment in the binary for tying data and rtype structures together.

Instead, the linker inserts the exact address of the rtype structure of data as an additional parameter to functions that require them.

One major type sink is the runtime.convT2E family of functions, which casts data into the `interface{}` type. Gos runtime uses this family of functions when passing an item of any type to a function whose parameter is an `interface{}` type.

There is a different convT2E function for each builtin type, but all functions match the following declaration:

```
func convT2E(t *_type, elem unsafe.Pointer) (e eface) {  }
```

This function directly matches a data element with its associated rtype structure as arguments. Ugo uses this type recovery technique of rtypes described by Martinez [7] but using the runtime.convT2E family of functions as an additional type information sink.

5 Implementation

The focal point of this paper is on improving the readability of decompiler-created code for Go binaries. Several problems we encountered when attempting to recover information created by the Go compiler:

- Go implements strings differently - Go treats strings as a struct containing a length field and a second pointer referencing the actual data that exists elsewhere in the binary. This makes it difficult to detect and retrieve strings properly.

- Offsets instead of pointers - Go internal structs use offsets instead of pointers, making it difficult to determine the values of fields without reading Gos dense linker source.

Ugo is implemented as a plugin for the IDAPro Hexrays decompiler, leveraging IDAs analyses. IDAs api allows us to search for the beginning of runtime.pclntab using a lookup or the magic address with IDAs LocByName function or idaapi.getsegmbyname. It is then easy

to define static structs for pclnentry and pclntab in IDA itself, and use the size field provided by pclntab to enumerate each individual pclnentry. IDA also allows for defining structs at locations with MakeStrucEx, but because Go defines many fields as offsets from the beginning of pclntab, we do some preprocessing beforehand to grab the func metadata struct out as we parse each individual pclnentry. The preprocessing generates a lookup table from function pointer to function metadata, allowing for a quick grab whenever a call is detected.

For the sake of xrefs, we also use define func structs using IDAs AddStrucEx and AddStrucMemberEx functions because IDA allows us to define fields as offsets from symbols, it made navigating and reversing the binary much easier. Another minor note, Go has 4 byte sized ints, so using IDAs api call `GetLongPrm(INFCOMPILER).sizei = 4` , we can tell the decompiler to treat ints as if they were 4 bytes instead of 8, which is the default for 64 bit binaries.

IDAs builtin calling convention labels are not applicable to Gos calling convention. They all assume that return values are singular and are placed in a register - the API prevents otherwise [4]. Ugo relabels the types and calling conventions of all functions in a Go binary as an IDA void `__usercall<>()`. This causes IDA to build its initial abstract syntax tree with all functions having no return value and both parameters and return values as arguments.

Ugo then uses the number of parameters and return values recovered to label which stack variables are return values. It restructures the abstract syntax tree in IDA (internally called the ctree) of the program. It defines a custom comma operator for grouping variables together, and sets the lvalue of the function to these grouped variables. This allows Hexrays to display the line of code in correct Go syntax by properly implementing calling convention.

Ugo performs its type recovery while restoring function calling conventions. It checks the symbol of the function against the whitelist of known type sinks and recovers type names from all rtypes passed as arguments for the function. This type name is then used to label the associated stack variables.

The whitelist consists of the following functions:

- `runtimenewobject`
- `runtimeconvT2E`
- `runtimeconvT2E16`
- `runtimeconvT2E32`
- `runtimeconvT2E64`
- `runtimeconvT2Eslice`
- `runtimeconvT2Enoptr`

6 Evaluation

We plan to evaluate ugo by conducting a study with OSIRIS security research lab members. The tasks for evaluation will be based on reverse engineering on real CTF challenges and will revolve around number of solves and feedback from members.

Another option for evaluation is Stanfords MOSS (Measure of Software Similarity) [8] test to measure parity from decompiler output against Go source code. Because of the amount of information the Go compiler leaves in the binary, it is theoretically feasible to extract direct source from go binaries. This is noted as a future option, after further development.

Although there is no explicit segment in the binary for tying data and rtype structures together, it may still be possible to recover rtype structures for data even when it is not explicitly used.

There are two potential approaches for doing so: the runtime garbage collector must reference the rtype for each data entry. Therefore, it may have a scheme in the .gcdata segment of the binary for connecting a data entry with its rtype. Additionally, the linker may have a scheme for placing rtypes in the .rodata. Therefore, there may be a deterministic placement order that is undocumented but used in implementation.

While ugo is performing calling convention recovery, the ctree representation of the program source can additionally be modified for other components of Go syntactic sugar. Channels, goroutines, defer, and panics are all compiled to and implemented by runtime function calls.

7 Conclusion

In this paper we made two contributions. We discovered additional commonly-used functions to use as type sinks for type information recovery. We additionally recovered metadata that we used for restoring calling convention of a a Go function call. This allows Hexrays decompilation to display Go source under IDAPro. Our plugin is the first of its kind to use linker artifacts and reflection metadata to decompile into Go source. We hope that our work can serve as inspiration and documentation for similar attempts to reverse engineer compiled Go.

8 Availability

Ugo is free software, available via Github from

`github.com/isislab/ugo`

References

[1] inbincible writeup - golang binary reversing.

[2] DONOVAN, A. A., AND KERNIGHAN, B. W. *The Go Programming Language*. Addison-Wesley Professional, 2015.

[3] EAGLE, C. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, 2 ed. No Starch Press.

[4] IDA. *Set function/item type*, 2017.

[5] LEE, J., AVGERINOS, T., , AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs. *Proceedings of the Network and Distributed Systems Symposium (NDSS)* (2011).

[6] MAKTM, M. *FLIRT Signature File Database*. IDA, 2017.

[7] MARTINEZ, S. Reversing linux malware. NCCGroup, R2con.

[8] SCHLEIMER, S., WILKERSON, D. S., AND AIKEN, A. Winnowing: Local algorithms for document fingerprinting. *ACM* (2003).

[9] SCHWARTZ, E. J., LEE, J., WOO, M., , AND BRUMLEY, D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. *Proceedings of the USENIX Security Symposium* (2013).

[10] STRAZZERE, T. Reversing go binaries like a pro.

[11] STRAZZERE, T. Go forth and reverse. Cloudflare, BSides Las Vegas.

[12] ZAYTSEV, G. Reversing golang.