Go ReStruct

Matthew LaCorte

Georgia Institute of Technology

mlacorte3@gatech.edu

Abstract – Go-based malware is a growing trend among bad actors, while reverse engineering tools for Go binaries have significant gaps in them. This paper will discuss Go ReStruct, a Ghidra plugin designed to extract structs from stripped Go binaries. With Go ReStruct, reverse engineers can better analyze stripped Go binaries, providing a better understanding of structs used by developers of the code. Go ReStruct is a Ghidra script written in Python and is available at www.github.com/matthewlacorte/go_restruct.

Keywords

Go; Golang; Structs, Reverse Engineering; Binary Analysis; Data Structures; Ghidra

1. INTRODUCTION

In recent years, the Go (Golang) programming language has become increasingly popular among malware authors **[1]**. With libraries being statically linked within the binaries, allowing for portability, and a single codebase for use across all major operating systems and architectures, it is an appealing language for use in malware development and distribution **[2]**. With the relative newness of the language, the reverse engineering and analysis of Go binaries lacks in-depth tooling. Due to this deficiency, malware can better avoid analysis, and increase the complexity of the reverse engineering process.

1.1 Recent Threats

The Mirai Botnet is a very well-known example of Go-based malware, with the entirety of the command and control (CNC) portion written in Go **[3]**. Additional functions such as the administrative and database logins were handled via Go. Go was also used to interface with the bots across the botnet **[4]**.

The Mustang Panda APT group was also observed to be using a Golang Loader in 2020, with the Go-based binary used to decrypt and load the "adobeupdate.dat" file onto systems **[5]**.

1.2 Increased Usage by Malware Developers

In 2020, the Go programming language saw an increase of approximately 2000% in new malware written in Go, discovered "in the wild". This malware originates both from nation-state and non-nation-state threat actors **[1]**.

CrowdStrike saw that from June to August of 2021, there was nearly an 80% increase in Go-based malware, with 70% of these malware samples observed to be for coin miners. "Golang's versatility in enabling the same codebase to be compiled for all major operating systems, coupled with the financial incentive offered by coin miners, could be one of the driving factors behind the recent wave of Go-written malware. However, we will likely see more Go-based malware as it is becoming more popular with developers." **[6]**.

CrowdStrike also observed that one motivating factor behind the rise in Go-based malware can be attributed to the increased performance Go offers over other languages such as Python, with some benchmarking tests showing increases by as much as 40 times faster.

1.3 Previous Work

In 2022, Dorka Palotay, a senior threat researcher at CUJO AI, presented her work on Type Extraction within stripped Go binaries at FIRSTCON22, an annual FIRST conference on computer security incident handling **[7]**. Through her presentations, methods were discussed on how to locate Type definitions within stripped Go binaries, and how to parse these definitions for useful information held within. Her work served as a starting point for the development of Go ReStruct, extending functionality she had previously discussed and worked on. The GitHub repository is available at **https://github.com/getCUJO/ThreatIntel** **[8]**, though it should be noted that these scripts were unable to be ran within Ghidra, on Go version 1.22 binaries.

1.4 Proposal

With the size and complexity of Go binaries, there is a growing need for tools to better assist in the reverse engineering efforts of them. This paper will describe Go ReStruct, a Ghidra plugin developed to detect structs in stripped Go binaries, create comments within the Ghidra project for a better understanding of the structs, and recreate the structs as custom Ghidra data structures. This allows for a reduction in manual efforts when reverse engineering Go binaries. While the tool developed is specific to Ghidra, algorithms and detection methods used can be ported to other disassembly tools as well.

This paper assumes a 64-bit architecture when discussing binaries and assembly code (i.e. RAX register, pointers are 8-bytes, etc.), though if used as a reference for 32-bit architectures, registers are interchangeable (RAX → EAX), and offsets may vary (two pointers immediately following one another are a combined 8-bytes in length, altering where following offsets may align).

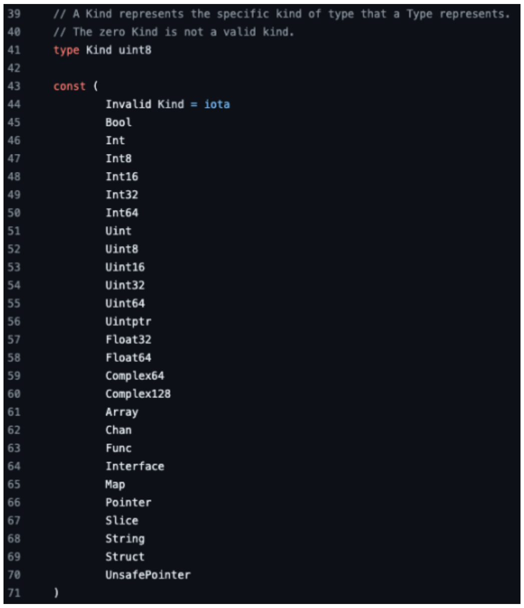
2. RESEARCH

Background research was performed to gain a deeper understanding of Go and its source code, Kinds and Types within the language, structs, naming conventions, and more. This information proved to be highly beneficial when developing the Go ReStruct tool, with major points discussed throughout this section. Source code references were taken from the official Go GitHub repository **[9]**. This section serves to compile all research performed.

2.1 Go Kinds & Types

The Go language contains 29 Kinds, as defined within the Go source code. Additionally, Types are used within the language and will reflect custom data types defined. For example, while a struct is a Kind, a developer's custom struct definition is classified as a Type, with a Kind attribute of struct.

A Kind is a custom Type within the Go language and is mapped to a "uint8" Kind. This Kind Type holds an integer representation of all different Kinds available within the language.

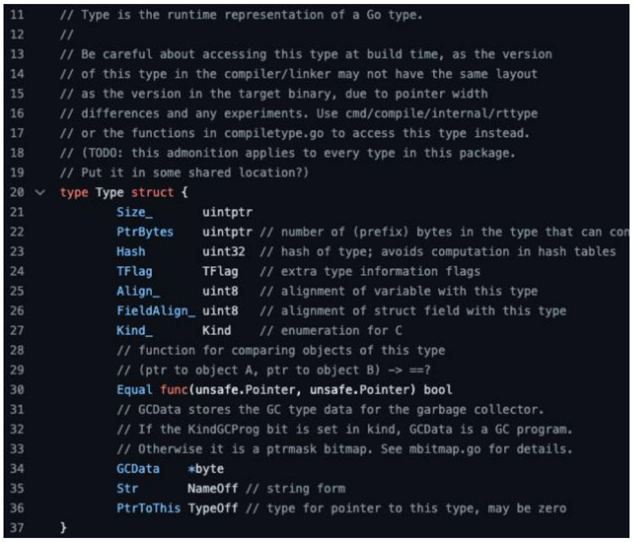


```go
// A Kind represents the specific kind of type that a Type represents.
// The zero Kind is not a valid kind.
type Kind uint8

const (
        Invalid Kind = iota
        Bool
        Int
        Int8
        Int16
        Int32
        Int64
        Uint
        Uint8
        Uint16
        Uint32
        Uint64
        Uintptr
        Float32
        Float64
        Complex64
        Complex128
        Array
        Chan
        Func
        Interface
        Map
        Pointer
        Slice
        String
        Struct
        UnsafePointer
)
```

Figure 1. List of available Go Kinds, from internal/abi/type.go

With this ordered list of available Kinds, a Kind attribute will hold the Type's Kind index value within the list. For example, "String" has an index value of 0x18. Therefore, a String's Kind value is 0x18.

A Type is a custom struct Type within the language, and will hold a multitude of variables within, referring to specific attributes of the Type created, including the Type's Kind.



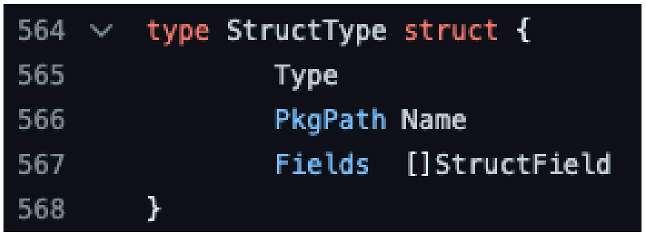
```go
// Type is the runtime representation of a Go type.
//
// Be careful about accessing this type at build time, as the version
// of this type in the compiler/linker may not have the same layout
// as the version in the target binary, due to pointer width
// differences and any experiments. Use cmd/compile/internal/rttype
// or the functions in compiletype.go to access this type instead.
// (TODO: this admonition applies to every type in this package.
// Put it in some shared location?)
type Type struct {
        Size_       uintptr
        PtrBytes    uintptr // number of (prefix) bytes in the type that can con
        Hash        uint32  // hash of type; avoids computation in hash tables
        TFlag       TFlag   // extra type information flags
        Align_      uint8   // alignment of variable with this type
        FieldAlign_ uint8   // alignment of struct field with this type
        Kind_       Kind    // enumeration for C
        // function for comparing objects of this type
        // (ptr to object A, ptr to object B) -> ==?
        Equal func(unsafe.Pointer, unsafe.Pointer) bool
        // GCData stores the GC type data for the garbage collector.
        // If the KindGCProg bit is set in kind, GCData is a GC program.
        // Otherwise it is a ptrmask bitmap. See mbitmap.go for details.
        GCData    *byte
        Str       NameOff // string form
        PtrToThis TypeOff // type for pointer to this type, may be zero
}
```

Figure 2. Type definition within Go, from internal/abi/type.go

Notable attributes within the Type type are the Size (uintptr), Kind (uint8), and NameOff (int32) variables. The Size will hold the overall size of the Type in bytes, the Kind will hold the Type's Kind index value, and the NameOff will hold an integer offset from the ".rodata" section, where the Type's name is stored as a string.

2.2 Struct Definitions in Go

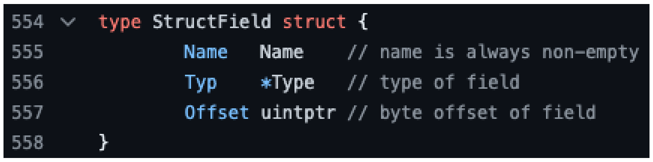
A struct is a Kind within Go and has a Type definition of StructType. This StructType stores information on the specific struct, such as the Type, name of the package it is associated with, and an array of fields within the struct.



```go
type StructType struct {
        Type
        PkgPath Name
        Fields  []StructField
}
```

Figure 3. StructType definition, from internal/abi/type.go

A struct's fields are noted as an array of StructField objects. This StructField type contains three values – Name, a Type pointer, and Offset (uintptr). With these values, each field of a struct can be understood. The Name points to a String of the field's name (as defined by the developer), the Type is a pointer to the generic Type definition, and the Offset is an integer representing the field's offset within the struct's definition.



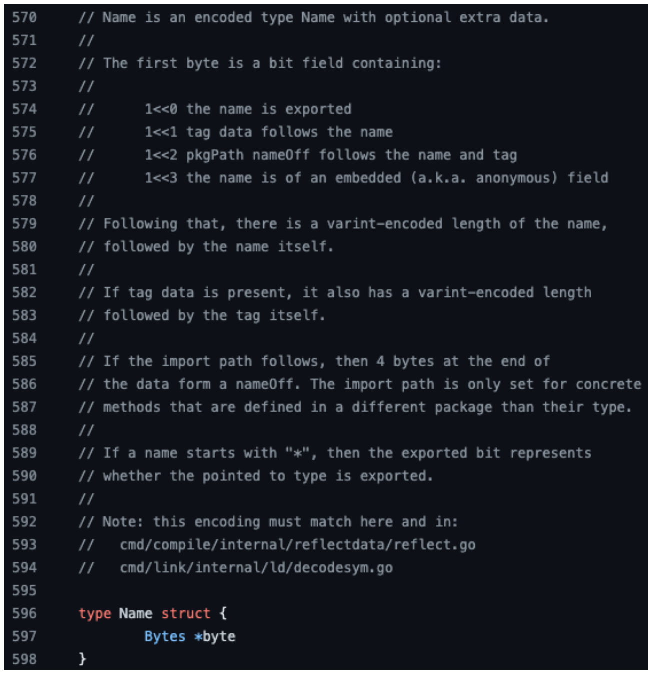
```go
type StructField struct {
        Name   Name    // name is always non-empty
        Typ    *Type   // type of field
        Offset uintptr // byte offset of field
}
```

Figure 4. StructField definition, from internal/abi/type.go

With the above definitions, an understanding of how a struct is defined can be developed. A developer's struct definition (custom Type) consists of a Type definition, followed by a pointer to the struct's Name, followed by a pointer to an array of StructField objects. The Type definition contains insight into the struct's Size, Kind, and full name (NameOff), and more. The PkgPath shows the package the struct Type belongs to, and the fields can be gathered from the array of StructFields. These StructField objects further an understanding of the struct, as each object shows the name of the field, the type of the field, and the offset of the field within the struct.

2.3 Understanding Names

With the StructField type containing a Name field, it is important to be able to interpret this field. The "reflect" package makes use of "internal/abi/type.go", where it can be seen how Names are defined and used.



```go
// Name is an encoded type Name with optional extra data.
//
// The first byte is a bit field containing:
//
//      1<<0 the name is exported
//      1<<1 tag data follows the name
//      1<<2 pkgPath nameOff follows the name and tag
//      1<<3 the name is of an embedded (a.k.a. anonymous) field
//
// Following that, there is a varint-encoded length of the name,
// followed by the name itself.
//
// If tag data is present, it also has a varint-encoded length
// followed by the tag itself.
//
// If the import path follows, then 4 bytes at the end of
// the data form a nameOff. The import path is only set for concrete
// methods that are defined in a different package than their type.
//
// If a name starts with "*", then the exported bit represents
// whether the pointed to type is exported.
//
// Note: this encoding must match here and in:
//   cmd/compile/internal/reflectdata/reflect.go
//   cmd/link/internal/ld/decodesym.go

type Name struct {
        Bytes *byte
}
```

Figure 5. Name definition, from internal/abi/type.go

From these comments, it can be observed that the Name type has three parts – a bit field value on "metadata" in the first byte, a varint-encoding of the name's length, and a byte array of the name's string format.

"The varint functions encode and decode single integer values using a variable-length encoding; smaller values require fewer bytes" **[10]**.

For simplicity's sake, the varint-encoding is assumed to be 1-byte in length in Go ReStruct, allowing for names up to 256 characters in length. With this, a Name can be made up of 1 byte for metadata, 1 byte for the Name's string length, and the string.

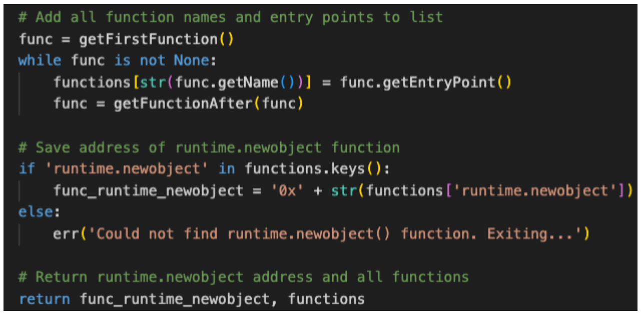
3. ANALYZING A GO BINARY

The following methods were used to analyze separate pieces of a stripped Go binary, looking for key sections, definitions, and components within. These methods were implemented in the Go ReStruct source code.

3.1 Finding Struct Definitions in a Binary

Stripped Go binaries present a few issues in determining all structs used within them, as the struct's definition may not always be within the binary. How the author has chosen to use the structs within the code determines if the struct's definition is present. If the author utilizes the Go "reflect" package **[11]** on the struct, the definition is likely present.

To make use of the struct definitions available within the binary, Go ReStruct looks for the uses of the `runtime.newobject()` function call. When called, a new object is created, as defined by a Type definition within the binary. This Type definition is passed to the `runtime.newobject()` function as a parameter within the RAX register. By first locating calls to `runtime.newobject()`, then looking backwards through instructions to determine the value passed in RAX, the location of all struct definitions can be found within the binary. This method will also include calls to `runtime.newobject()` that are not for just structs. These non-struct definitions can be parsed out to ensure only structs are being evaluated.



```
# Add all function names and entry points to list
func = getFirstFunction()
while func is not None:
    functions[str(func.getName())] = func.getEntryPoint()
    func = getFunctionAfter(func)

# Save address of runtime.newobject function
if 'runtime.newobject' in functions.keys():
    func_runtime_newobject = '0x' + str(functions['runtime.newobject'])
else:
    err('Could not find runtime.newobject() function. Exiting...')

# Return runtime.newobject address and all functions
return func_runtime_newobject, functions
```

Figure 7. Locating the `runtime.newobject()` function within Go ReStruct

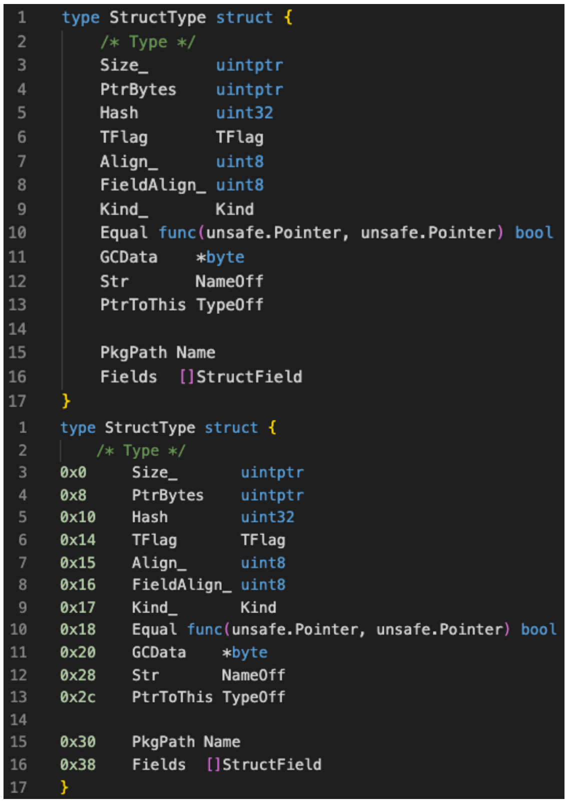
To find the calls to `runtime.newobject()`, all functions within the binary are parsed, looking for a function who's name equals "runtime.newobject". If found, the function's entry point is saved. Then when parsing all instructions, if the instruction's mnemonic string is "CALL" and the first operand is an address equivalent to the `runtime.newobject()` address previously found, it can be assumed that the instruction calls the `runtime.newobject()` function.

Without the use of the "reflect" package by the author, a struct's definition cannot be found within a stripped Go binary. Go ReStruct works to find all instances of structs within a stripped binary who's Type definitions are saved.

3.2 Analyzing Struct Definitions

By taking the previously discussed definitions on Kinds, Types, StructTypes, and StructFields, a struct object's definition can be parsed from within a Go binary. From previous sections, it is known that the StructType object has three fields within it – the Type, the PkgName, and an array of StructFields. Since the Type field is an object, rather than a pointer to an object, all fields of the Type are present within the struct's definition. Therefore, the Type can be replaced with all fields within the Type object. The Name and array of StructFields are both pointers to the locations.

These definitions translates directly within the binary. By knowing the size of each field, it's location can be found as an offset from the base definition address. Due to working with 64-bit binaries, pointers are 8 bytes in length.



```
type StructType struct {
    /* Type */
    Size_       uintptr
    PtrBytes    uintptr
    Hash        uint32
    TFlag       TFlag
    Align_      uint8
    FieldAlign_ uint8
    Kind_       Kind
    Equal func(unsafe.Pointer, unsafe.Pointer) bool
    GCData    *byte
    Str       NameOff
    PtrToThis TypeOff

    PkgPath Name
    Fields  []StructField
}
type StructType struct {
    /* Type */
0x0     Size_       uintptr
0x8     PtrBytes    uintptr
0x10    Hash        uint32
0x14    TFlag       TFlag
0x15    Align_      uint8
0x16    FieldAlign_ uint8
0x17    Kind_       Kind
0x18    Equal func(unsafe.Pointer, unsafe.Pointer) bool
0x20    GCData    *byte
0x28    Str       NameOff
0x2c    PtrToThis TypeOff

0x30    PkgPath Name
0x38    Fields  []StructField
}
```

Figure 8. Combined fields of StructType, including Type. Offsets shown.

Significant offsets to look for are the "Kind", "NameOff", "PkgPath", and array of StructFields, located at offsets 0x17, 0x28, 0x30, and 0x38, respectively. These offsets are used from the struct's base address, found when parsing a binary for calls to `runtime.newobject()`. The base address was passed as a parameter via the RAX register.

3.2.1 Kind

The Kind of the Type is defined at offset 0x17 from the base address, and is stored as an unsigned-integer pointer (uintptr). By retrieving the data at this offset, the Type's Kind can be known. Making use of the Kind definition previously discussed, this is an integer value mapped to the indices within a list of all available Kinds. From Figure 1, a struct's Kind value is 0x19. By parsing the base address for an offset of 0x17 equal to 0x19, it can be determined that the Type definition is for a struct.

3.2.2 NameOff

The NameOff field within a Type definition is an integer offset from the ".rodata" section of the binary. By converting the data held at offset 0x28 to an integer, this value can be added to the start of the ".rodata" section to determine the Type's name, as defined by the developer.

Once found, the memory address calculated can be parsed for the string held within. This string will be the name of the Type.

3.2.3 PkgPath

The PkgPath value is a pointer to a string value, where the package name is held. The address within the pointer can be taken to read the string at the given address, providing the package name associated with the Type.

3.2.4 StructField Array

The StructField array is a pointer to the start of an array of StructField objects. Each object being 0x18 bytes in length. Within each object is a pointer to the field's name, as defined by the developer, a pointer to the field's type, and an integer offset value from within the struct object.

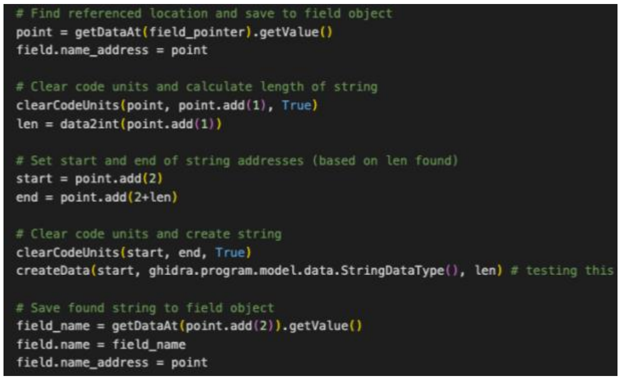
This array of StructFields can be parsed to find all fields within the struct. This allows the string holding the field's name to be determined, as well as finding the Kind value of the Type pointer, and extracting the offset value of the field.

3.2.5 Reading Names

Reading the data at the Name's base address plus an offset of one ($base\_address + 1$) as an integer will give the length of the Name string. The string itself is contained from Name base address plus two to Name base address plus two plus the length, as shown in Equation 1 below. Within Ghidra, these two addresses can be used to assign a string data type to it, allowing the Field's Name to be read.

$$(base\_address + 2\ \rightarrow base\_address + 2 + len)$$

Equation 1. Location of Name string.

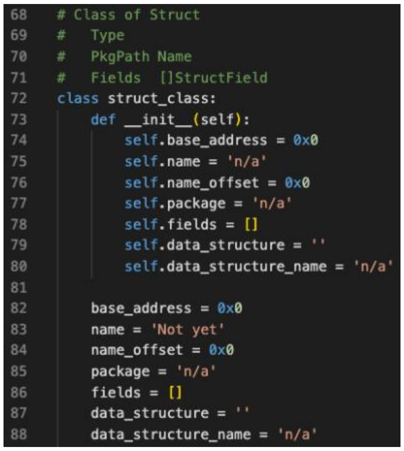


```
# Find referenced location and save to field object
point = getDataAt(field_pointer).getValue()
field.name_address = point

# Clear code units and calculate length of string
clearCodeUnits(point, point.add(1), True)
len = data2int(point.add(1))

# Set start and end of string addresses (based on len found)
start = point.add(2)
end = point.add(2+len)

# Clear code units and create string
clearCodeUnits(start, end, True)
createData(start, ghidra.program.model.data.StringDataType(), len) # testing this

# Save found string to field object
field_name = getDataAt(point.add(2)).getValue()
field.name = field_name
field.name_address = point
```

Figure 9. Calculating field name within Go ReStruct

3.3 Putting it All Together

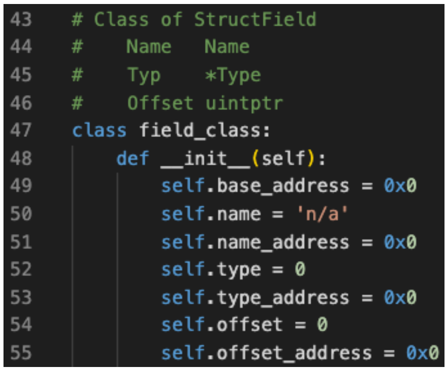
Within Go ReStruct, classes are created for both Fields and Structs, allowing easy manipulation of each object. When parsing a struct from the base address determined via the `runtime.newobject()` parsing, that base address is saved to a struct class. From there, the offset values can be calculated in memory as well, with any data found saved to the class as well.



```
# Class of Struct
#   Type
#   PkgPath Name
#   Fields  []StructField
class struct_class:
    def __init__(self):
        self.base_address = 0x0
        self.name = 'n/a'
        self.name_offset = 0x0
        self.package = 'n/a'
        self.fields = []
        self.data_structure = ''
        self.data_structure_name = 'n/a'

    base_address = 0x0
    name = 'Not yet'
    name_offset = 0x0
    package = 'n/a'
    fields = []
    data_structure = ''
    data_structure_name = 'n/a'
```

Figure 10. Struct class definition within Go ReStruct

Similarly, for fields within structs, a Field class is created to contain information such as the address of the field within the array, the name of the field, the type of the field, and the offset of the field. These Field class objects are then stored within an array in the Struct class object.



```
# Class of StructField
#    Name    Name
#    Typ     *Type
#    Offset uintptr
class field_class:
    def __init__(self):
        self.base_address = 0x0
        self.name = 'n/a'
        self.name_address = 0x0
        self.type = 0
        self.type_address = 0x0
        self.offset = 0
        self.offset_address = 0x0
```

Figure 11. Field class definition within Go ReStruct

With the above steps taken, a programmatic understanding of structs within a Go binary can be recreated.

All struct definitions used can be found via calls to the `runtime.newobject()` function, with these definitions saved to Struct class objects. Within them, the struct's name, package, and an array of fields are saved. The array of fields contains all fields discovered from the struct definition, with the field's name, type, and offset saved to the Field class object.

These Struct and Field objects can be used throughout Go ReStruct for additional programmatic parsing of the binary and feature implementation.

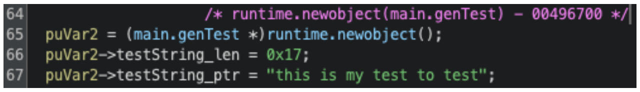
4. FEATURES OF THE TOOL

4.1 Comments from Go ReStruct

Go ReStruct works to add in-line comments within a Ghidra workspace to allow for a better understanding of the binaries by reverse engineers. These comments focus on the creation of new struct objects, the definitions of structs, and partially in read/write locations of struct fields.

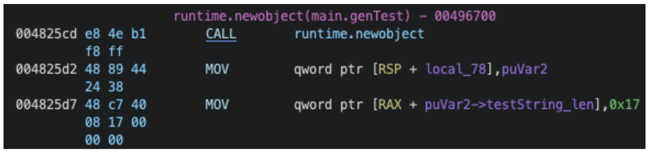
4.1.1 Commenting New Objects

After having parsed binaries for calls to the `runtime.newobject()` function and saving the location of struct definitions used within these calls, Go ReStruct will create Pre Comments detailing the name of the structs created, as well as the addresses where the StructType is defined at.



```
                   /* runtime.newobject(main.genTest) - 00496700 */
puVar2 = (main.genTest *)runtime.newobject();
puVar2->testString_len = 0x17;
puVar2->testString_ptr = "this is my test to test";
```

Figure 12. Pre Comment in decompiled view, showing creation of main.getTest struct, defined at memory address `0x00496700`



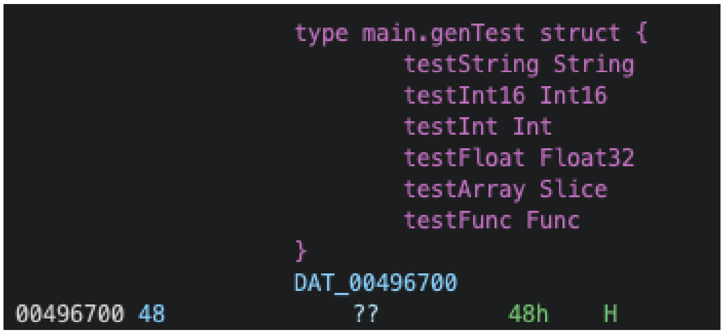
```
                     runtime.newobject(main.genTest) - 00496700
004825cd e8 4e b1        CALL       runtime.newobject
         f8 ff
004825d2 48 89 44        MOV        qword ptr [RSP + local_78],puVar2
         24 38
004825d7 48 c7 40        MOV        qword ptr [RAX + puVar2->testString_len],0x17
         08 17 00
         00 00
```

Figure 13. Pre Comment in disassembly view, showing creation of main.getTest struct, defined at memory address `0x00496700`

Clicking on the address shown in the Pre Comment will jump to the address, allowing for easy viewing of the struct definition if needed.

4.1.2 Commenting StructType Definitions

From having saved name information and all fields within the struct previously, Go ReStruct also works to recreate the struct's definition as it would be written in Go. This source code definition is added as a Pre Comment at the location where the StructType is defined, showing all field names and their types.

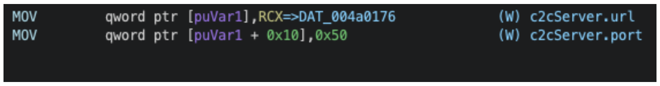


```
                  type main.genTest struct {
                          testString String
                          testInt16 Int16
                          testInt Int
                          testFloat Float32
                          testArray Slice
                          testFunc Func
                  }
                  DAT_00496700
00496700 48                ??          48h    H
```

Figure 14. Pre Comment in disassembled view, showing source code definition of main.getTest, including all fields and their types

4.1.3 Commenting Read/Write Instructions of Fields

While not fully implemented, some effort was put towards creating End-of-Line comments within the disassembly view, allowing the tracking of read and write locations of struct fields. This was made redundant due to the creation of custom Ghidra data structures, discussed in following sections.



```
MOV        qword ptr [puVar1],RCX=>DAT_004a0176            (W) c2cServer.url
MOV        qword ptr [puVar1 + 0x10],0x50                  (W) c2cServer.port
```

Figure 15. End-of-Line comments for read locations of the c2cServer StructType, for fields "url" and "port".

4.2 Custom Data Structures in Ghidra

Ghidra allows for a useful feature in manually creating custom data structures as needed. This can be done in the GUI via the Data Type Manager window, or programmatically via APIs. When created, data structures are specific to the individual files being worked on, but can be copied across to other files as well. They do not have a project-wide scope.

The Data Type Manager allows for folders as well, allowing data types and structures to be separated into logical groupings. There is typically a folder for "/BuiltInTypes", types that are defaults in Ghidra, a folder for the project, as well as "/generic_clib" and "/generic_clib_64". These are referred to as "root" folders within the Data Type Manager.

Go ReStruct works to recreate all structs discovered within the binaries as custom data structures, allowing the assignment to variables within the disassembled and decompiled views in Ghidra.

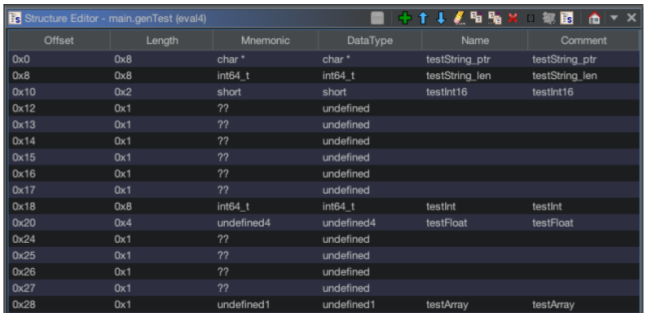
4.2.1 Creating Custom Data Structures

From the previously determined struct definitions and having been saved in the Struct and Field classes, the attributes of each can be used to create a data structure within Ghidra, allowing fields to accurately map to their offsets.

When creating new data structures, Ghidra requires an address of where that data structure is defined. However, due to how Go defines these structs, the address required by Ghidra does not create a strong correlation. An address is required regardless, so the lowest address within the binary is used. Once created, any fields automatically assigned by Ghidra are removed, leaving an empty data structure definition.

The fields within a struct are parsed, taking the field name, field's type, and the offset of the field. These values are then used to create each field within the new data structure. Certain Kinds within Go have multiple parts within them, such as strings. Strings have a "string" component, as well as an integer length. To account for this, two fields are added, one noting the string address and one noting the length.

Additionally, to account for offset alignments, Go ReStruct will add undefined bytes to fill any spaces required. If an int32 Kind (4-byte size) is at offset 0, followed by in int64 Kind (8-byte size) at offset 0x8, there are unassigned bytes between 0x4-0x7. These 4 bytes are filled with 1-byte unassigned types, ensuring all offsets are filled in within the data structure.

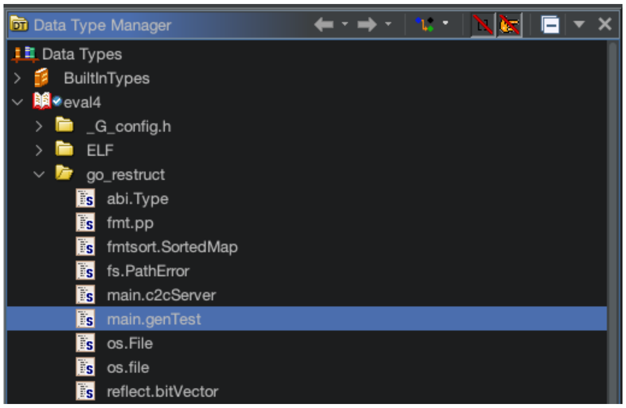


Structure Editor - main.genTest (eval4)

Offset	Length	Mnemonic	DataType	Name	Comment
0x0	0x8	char *	char *	testString_ptr	testString_ptr
0x8	0x8	int64_t	int64_t	testString_len	testString_len
0x10	0x2	short	short	testInt16	testInt16
0x12	0x1	??	undefined		
0x13	0x1	??	undefined		
0x14	0x1	??	undefined		
0x15	0x1	??	undefined		
0x16	0x1	??	undefined		
0x17	0x1	??	undefined		
0x18	0x8	int64_t	int64_t	testInt	testInt
0x20	0x4	undefined4	undefined4	testFloat	testFloat
0x24	0x1	??	undefined		
0x25	0x1	??	undefined		
0x26	0x1	??	undefined		
0x27	0x1	??	undefined		
0x28	0x1	undefined1	undefined1	testArray	testArray

Figure 16. Partial data structure created by Go ReStruct for main.genTest struct.

These data structures are created, then added to a "/go_restruct" folder under the binaries root folder, allowing for easy segmentation of data structures, as well as ease of use by reverse engineers when referencing or altering data structures.



Data Type Manager

```
Data Types
> BuiltInTypes
v eval4
  > _G_config.h
  > ELF
  v go_restruct
      abi.Type
      fmt.pp
      fmtsort.SortedMap
      fs.PathError
      main.c2cServer
      main.genTest
      os.File
      os.file
      reflect.bitVector
```

Figure 17. Data Type Manager showing /go_restruct subfolder with custom data structures.

Go ReStruct makes use of data types in the "/generic_clib_64" folder, namely the "types.h" and "stdint.h" libraries. It is recommended to copy these folders to the project's root folder, though if not present, Go ReStruct will instead use "undefinedX" data types, respective to the Go Kind's size in bytes. For example, if an "int32" Kind is not available for use within Ghidra, an "undefined4" type will be used in its place, as both are 32-bits/4-bytes in size.

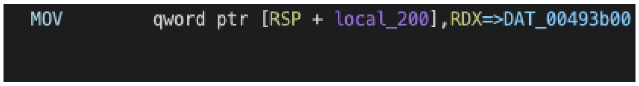
4.2.2 Data Structure Assignments to Local Variables

Go ReStruct works to automatically assign the custom created data structures to local variables in functions within the disassembly and decompiled views. This is not a perfect feature, as it is dependent on some default variable name assignments. These names can oftentimes be changed by Ghidra's default rules or altered manually. In either of those cases, custom data structures may not be assigned automatically.

As the newly created object from `runtime.newobject()` is returned in the RAX register, the first MOV instruction where RAX is the second operand is searched for after the call to `runtime.newobject()`. This instruction signifies where the new object is saved to. If the instruction saves RAX to a location on the stack, that stack variable is the target for the data structure assignment.

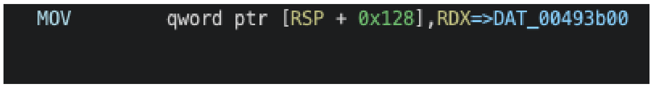
When analyzing instructions in Ghidra, instructions refer to stack variables as an offset from the RSP register (positive offset), though Ghidra defaults to naming variables as an offset from the RBP register (negative offset). For example, if a variable is given the default name of "local_200", this means that it has an offset from the RBP register of -0x200, though the assembly instruction would instead refer to it as an offset from the RSP register. If the stack's depth at that instruction is 0x328, the assembly instruction would refer to it as "RSP + 0x128" – 0x128 higher than RSP, but 0x200 lower than RBP.

This variable naming scheme can present some issues when attempting to automatically assign data structures to variables, as there is no native way to retrieve local variables used within an Instruction in Ghidra's APIs. To attempt to resolve this, Go ReStruct will look for variables who's assigned name matches the difference in stack depth versus RSP offset. For the above example, with an offset of 0x128 and stack depth of 0x328, a variable of "local_200" would be searched in the list of local variables to the function. If a variable matching the requirements is found, the associated data structure will be assigned. Otherwise, no data structure is assigned automatically.



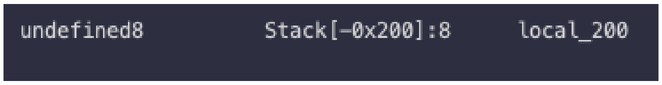
```
MOV        qword ptr [RSP + local_200],RDX=>DAT_00493b00
```

Figure 18. Local variable "local_200" being used in an instruction, with the default Ghidra label assigned



```
MOV        qword ptr [RSP + 0x128],RDX=>DAT_00493b00
```

Figure 19. "local_200" label removed, showing an offset of 0x128 from RSP, with a stack depth of 0x328



```
undefined8         Stack[-0x200]:8      local_200
```

Figure 20. "local_200" assigned as an offset of -0x200 from the RBP register

In situations where a variable cannot be automatically assigned to the associated data structure, there is the ability to manually assign it, as the data structure still exists within the project. With the name of the struct given in the Pre Comment associated with the `runtime.newobject()` call, this is a simple and fast process when needed.

4.3 Struct Definitions Output to File

Go ReStruct presents the user with the option to save all discovered structs to a text file of their choosing, displaying definitions in a manner as if they were defined within the Go code. This allows the user to refer back to any and all structs used throughout a binary, and easily search for any struct names or field names that may be present.

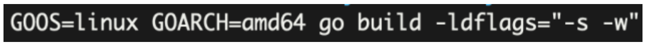
5. EVALUATION

A multitude of binaries were compiled, with 1-3 user defined structs across 1-2 functions. Structs were designed to encompass a wide range of Types within the Go language, allowing for a thorough testing of Type detection within them. These Types encompassed integers and unsigned integers, strings, arrays, functions, and nested structs.

All base structs, meaning non-nested structs, were used with the "reflect" package in some form to make use of the `runtime.newobject()` function call to ensure each struct being detected by Go ReStruct.

When looking at results, there are two main measurements being assessed – effectiveness and accuracy. Effectiveness looks to quantify the number of structs detected and analyzed, while accuracy looks to quantify the correctness of structs and their fields within.

Each binary was built with the arguments shown in Figure 21 below.



```
GOOS=linux GOARCH=amd64 go build -ldflags="-s -w"
```

Figure 21. Go build command for evaluations.

"GOOS=linux" builds for a Linux OS, while "GOARCH=amd64" builds for an AMD64 (64-bit) architecture. Additionally, there are ldflag options set. The “-s” flag builds the binary with the symbol table disabled, while the “-w” flag builds without DWARF generation. DWARF generation is commonly used for debugging information **[12]**. The “-s -w” flags are commonly used to create stripped binaries in Go.

All source code files used for analysis and evaluation are available on the Go ReStruct GitHub page.

5.1 Results

To generate the outputs within Ghidra of the Go ReStruct tool, a few steps were taken per binary. The binaries were first built with the options in shown Figure 21 above and imported into Ghidra. From there, Ghidra’s auto-analysis features were run on the binaries. The Trellix Advanced Research Center (ARC)'s Ghidra script for Go function name recovery was then run **[13]**. The “generic_clib_64/ stdint.h” and “generic_clib_64/types.h” libraries were then copied into the Data Type Manager. Lastly, the Go ReStruct plugin was run.

Visual outputs of the Go ReStruct tool can be found in section 10.1 of the Appendix.

5.1.1 Effectiveness

Of the seven user-defined base structs across all binaries, all were identified by Go ReStruct. Each struct’s object creation was detected, with the `runtime.newobject()` call properly commented to show the struct used. Additionally, each struct’s definition was observed to have the source code definition commented at its location in memory. All structs had the associated custom Ghidra data structure created and added into the “/go_restruct” subfolder within the Data Type Manager, however not all custom data structures were applied to the associated local variables. Only one of seven had the data structure applied automatically.

With the variable names not following the default naming conventions discussed in section 4.2.2 earlier, the application of the data structures is lacking. The custom data structure still being created means that the user can manually set the variable’s data type, allowing an accurate representation of the struct.

5.1.2 Accuracy

All field names and types within each data structure were accurately identified and recorded within the custom data structures. Additionally, all `runtime.newobject()` calls within the binary were commented correctly, showing the name of the struct type and the memory location of its definition. Lastly, all struct definitions were commented with the Go source code definition correctly, showing the struct name, field names, and field types accurately.

All custom Ghidra data structures were created, with each field’s name and initial offset set as defined in the binary.

Issues were presented with more complex Go Types, such as the Array, where there are custom definitions per entry, rather than a single point of data. Where edge cases were identified with String Types, these strings were correctly recreated within the data structures. Functions were correctly identified, as they are pointer values within a struct to a separate function. Nested structs were not correctly identified, as they were not used with the `runtime.newobject()` calls.

5.1.3 Additional Results

27 structs from imported libraries were identified, and a custom data structure added to the “/go_restruct” subfolder within the Data Type Manager. 66 instances of these structs being used were identified and commented.

Each run of Go ReStruct took anywhere from 0.75-1.25 seconds per run, on a 14” MacBook Pro with M3 Pro processor and 18GB RAM, running on the 11.1 Ghidra DEV build.

5.2 Efficiency Increases

One initial datapoint thought to be used to measure the results of Go ReStruct was the efficiency increase for reverse engineers when analyzing Go binaries. Having a tool capable of not only detecting and marking up locations where structs are used throughout a binary, but also creating the custom data structures associated, the time required to analyze a binary would be reduced.

While efficiency increases may be an intuitive thought, throughout this evaluation, a quantitative method measuring increases in efficiency was not established, though still warrants recognition and discussion.

6. LIMITATIONS

Go ReStruct is currently developed for 64-bit binaries, though there is an underlying framework to allow for the ability to analyze 32-bit binaries in the future. This limitation is primarily due to how fields and variables are discovered within definitions, such as where the NameOff value is stored within a StructType definition. As there are pointer values between the base address of the StructType definition and the NameOff field, the offset of the fields change. A 32-bit binary would have 4-byte lengths for all pointers, while a 64-bit binary would use 8-byte pointers. Similarly, integers are treated the same way. A 32-bit binary uses integers of 4-bytes in length, while a 64-bit binary would use 8-byte integers.

Additionally, Go ReStruct is only developed for AMD architectures. Due to the nature of how registers are used within the tool, ARM architectures are not currently supported.

Complex Go Types, such as arrays, maps, slices, and nested structs are not accurately depicted within custom data structures. This comes down to a lack of discovery on these individual data types, and would require a similar process for each Kind as the struct Kind.

The largest limitation to Go ReStruct currently is the need to use external scripts to locate and assign function names within the binaries. As they are stripped, with no symbol table available, function names are not automatically set via Ghidra's built-in analysis tools. These function names still exist within the binary though, and scripts exist to locate and assign them. Development of Go ReStruct was done with the Golang scripts from the Trellix ARC Team **[13]**, with the primary script being "**GolangFunctionRecovery.java**".

Lastly, to make full use of Go ReStruct's automation, libraries need to be imported to the Data Type Manager to create the custom Ghidra data structures to model the Go structs. While this is a trivial step, it is one that is needed on every binary analyzed. Without it, data structures are still created, though they may be less useful as data types are not assigned, only the sizes of each field are held within the data structure.

7. FUTURE WORK

Future work to be done on Go ReStruct highly aligns with the limitations of the tool.

As there is already some underlying infrastructure to support 32-bit binaries, a minor amount of effort would be needed to fully implement this. This can be done either via separate plugin scripts for 32-bit versus 64-bit, or by using variables within the script to account for the differences. While 32-bit programs are less common nowadays, there is still use for the analysis of them, especially when the level of effort needed to add in the feature is low.

There is also room to improve the tool from more than just AMD architectures, as ARM processors are growing in market share and are expected to cover 25% of the market by 2027 **[14]**. With this growing trend, malware is more commonly built for these architectures, requiring an equally in-depth level of analysis. As Go is a language where a single codebase can support a wide range of operating systems and architectures, effort should be put into developing analysis tools with a similarly wide range of support.

Finally, room exists to extend Go ReStruct's functionality beyond just that of struct Kinds within Go, extending to more complex kinds such as arrays, maps, and slices. Similar concepts can be used to parse the definitions of these complex data types, retrieving valuable information from them, comparable to that of structs.

8. CONCLUSIONS

Through this paper, the benefits Go ReStruct presents to reverse engineers analyzing stripped Go binaries has been shown. This is a tool capable of accurately and effectively parsing stripped binaries to discover structs within, creating comments in Ghidra to benefit the analysis process. Go ReStruct has also proven to be capable of recreating discovered structs as custom Ghidra data structures in a manner representative of the original structs, as defined by the developer.

Go ReStruct is a useful addition when reverse engineering stripped Go binaries, and is a tool with the future potential to improve the reverse engineering process beyond what has already been proven.

9. REFERENCES

[1] Intezer. 2021. Year of the Gopher: 2020 go malware round-up. (February 2021). Retrieved April 17, 2024 from https://intezer.com/blog/malware-analysis/year-of-the-gopher-2020-go-malware-round-up/

[2] Josh Grunzweig. 2019. The Gopher in the room: Analysis of GoLang malware in the wild. (July 2019). Retrieved April 17, 2024 from https://unit42.paloaltonetworks.com/the-gopher-in-the-room-analysis-of-golang-malware-in-the-wild/

[3] Jgamblin. 2016. JGAMBLIN/mirai-source-code: Leaked Mirai source code for Research/IOC development purposes. (2016). Retrieved April 17, 2024 from https://github.com/jgamblin/Mirai-Source-Code

[4] CJ Barker. 2016. Mirai (ddos) source code review. (October 2016). Retrieved April 17, 2024 from https://medium.com/@cjbarker/mirai-ddos-source-code-review-57269c4a68f

[5] The Proofpoint Threat Research Team. 2020. TA416's Golang plugx malware loader: Proofpoint us. (November 2020). Retrieved April 17, 2024 from https://www.proofpoint.com/us/blog/threat-insight/ta416-goes-ground-and-returns-golang-plugx-malware-loader

[6] Anmol Maurya. 2021. Financial motivation drives golang malware adoption: CrowdStrike. (November 2021). Retrieved April 17, 2024 from https://www.crowdstrike.com/blog/financial-motivation-drives-golang-malware-adoption/

[7] Anon. Annual conferences. Retrieved April 17, 2024 from https://www.first.org/conference/

[8] getCUJO. 2020. GetCUJO/threatintel. (June 2020). Retrieved April 17, 2024 from https://github.com/getCUJO/ThreatIntel

[9] Golang. 2008. Golang/GO: The go programming language. (March 2008). Retrieved April 17, 2024 from https://github.com/golang/go

[10] Anon. Package binary. Retrieved April 17, 2024b from https://www.cs.ubc.ca/~bestchai/teaching/cs416_2015w2/go1.4.3-docs/pkg/encoding/binary/index.html

[11] Golang. Reflect. Retrieved April 17, 2024 from https://pkg.go.dev/reflect#TypeOf

[12] Golang. Dwarf. Retrieved April 17, 2024a from https://pkg.go.dev/cmd/internal/dwarf

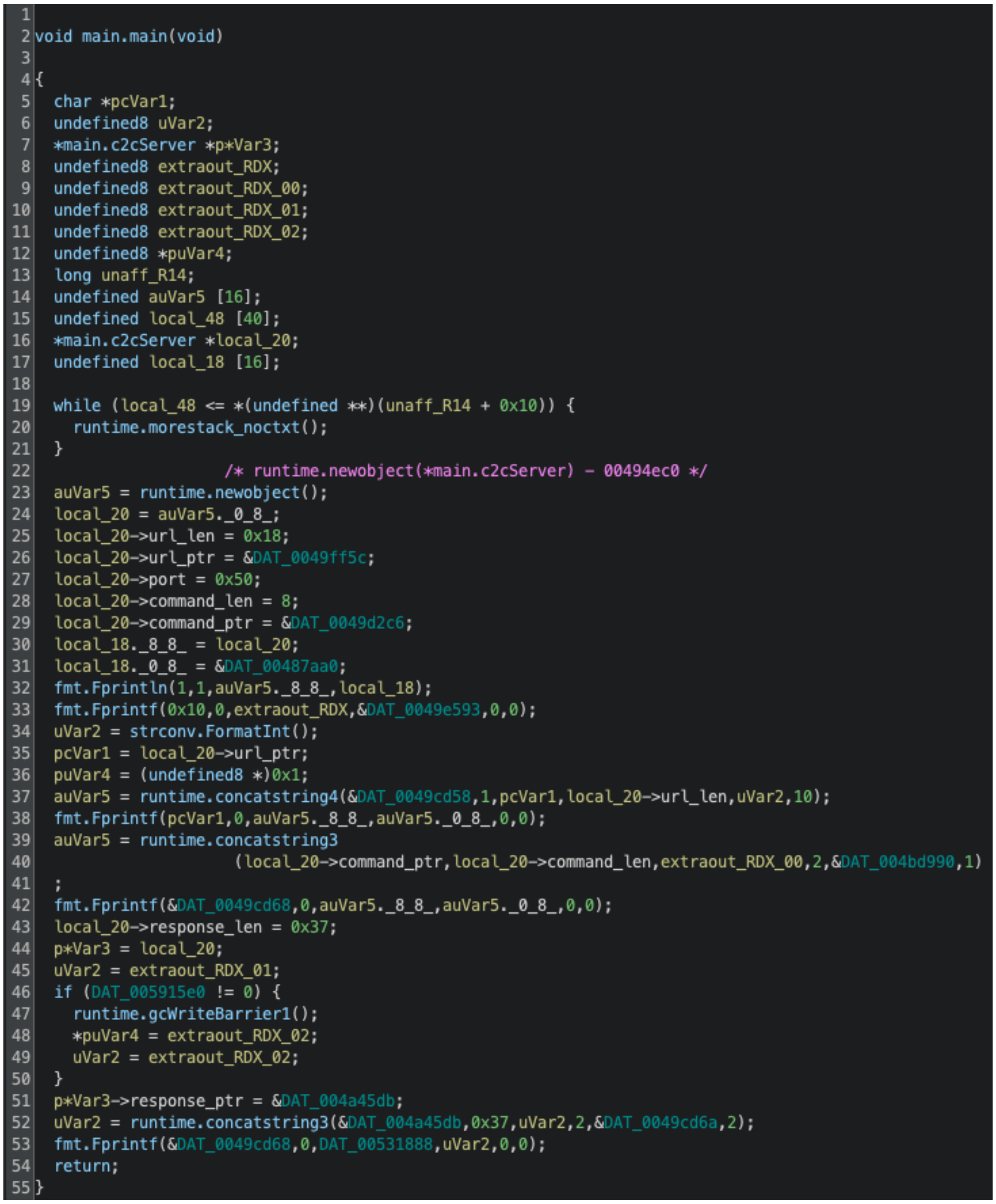
[13] Advanced-Threat-Research. 2023. Advanced-threat-research/ghidrascripts: Scripts to run within Ghidra, maintained by the Trellix Arc Team. (April 2023). Retrieved April 17, 2024 from https://github.com/advanced-threat-research/GhidraScripts

[14] Brady Wang. 2023. ARM-based pcs to nearly double market share by 2027. (April 2023). Retrieved April 17, 2024 from https://www.counterpointresearch.com/insights/arm-based-pcs-to-nearly-double-market-share-by-2027/

10. APPENDIX

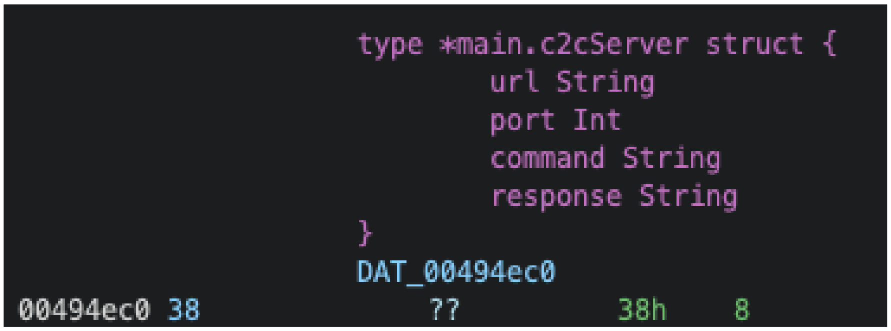
Go ReStruct GitHub - www.github.com/matthewlacorte/go_restruct

10.1 Visual Outputs of Go ReStruct



```

void main.main(void)

{
  char *pcVar1;
  undefined8 uVar2;
  *main.c2cServer *p*Var3;
  undefined8 extraout_RDX;
  undefined8 extraout_RDX_00;
  undefined8 extraout_RDX_01;
  undefined8 extraout_RDX_02;
  undefined8 *puVar4;
  long unaff_R14;
  undefined auVar5 [16];
  undefined local_48 [40];
  *main.c2cServer *local_20;
  undefined local_18 [16];

  while (local_48 <= *(undefined **)(unaff_R14 + 0x10)) {
    runtime.morestack_noctxt();
  }
                  /* runtime.newobject(*main.c2cServer) - 00494ec0 */
  auVar5 = runtime.newobject();
  local_20 = auVar5._0_8_;
  local_20->url_len = 0x18;
  local_20->url_ptr = &DAT_0049ff5c;
  local_20->port = 0x50;
  local_20->command_len = 8;
  local_20->command_ptr = &DAT_0049d2c6;
  local_18._8_8_ = local_20;
  local_18._0_8_ = &DAT_00487aa0;
  fmt.Fprintln(1,1,auVar5._8_8_,local_18);
  fmt.Fprintf(0x10,0,extraout_RDX,&DAT_0049e593,0,0);
  uVar2 = strconv.FormatInt();
  pcVar1 = local_20->url_ptr;
  puVar4 = (undefined8 *)0x1;
  auVar5 = runtime.concatstring4(&DAT_0049cd58,1,pcVar1,local_20->url_len,uVar2,10);
  fmt.Fprintf(pcVar1,0,auVar5._8_8_,auVar5._0_8_,0,0);
  auVar5 = runtime.concatstring3
                   (local_20->command_ptr,local_20->command_len,extraout_RDX_00,2,&DAT_004bd990,1)
  ;
  fmt.Fprintf(&DAT_0049cd68,0,auVar5._8_8_,auVar5._0_8_,0,0);
  local_20->response_len = 0x37;
  p*Var3 = local_20;
  uVar2 = extraout_RDX_01;
  if (DAT_005915e0 != 0) {
    runtime.gcWriteBarrier1();
    *puVar4 = extraout_RDX_02;
    uVar2 = extraout_RDX_02;
  }
  p*Var3->response_ptr = &DAT_004a45db;
  uVar2 = runtime.concatstring3(&DAT_004a45db,0x37,uVar2,2,&DAT_0049cd6a,2);
  fmt.Fprintf(&DAT_0049cd68,0,DAT_00531888,uVar2,0,0);
  return;
}
```

Figure 22. Go ReStruct output on main.main function, in eval_1.go file



```
                        type *main.c2cServer struct {
                                url String
                                port Int
                                command String
                                response String
                        }
                        DAT_00494ec0
00494ec0 38                 ??          38h     8
```

Figure 23. Go ReStruct Pre Comment on struct definition, in eval_1.go file