Memory Forensics Techniques for Automated Detection and Analysis of Go Malware

Hala Ali[a,*], Andrew Case[b] and Irfan Ahmed[a]

[a]Department of Computer Science, Virginia Commonwealth University, USA
[b]Volatility Foundation, USA

ARTICLE INFO

Keywords:
Go
Golang
Memory Forensics
Malware Analysis
Volatility 3

ABSTRACT

The Go programming language has become increasingly popular among malware developers due to its ability to produce statically linked, cross-platform executables that challenge traditional analysis techniques. These binaries embed a substantial runtime and compiler-generated metadata and are compiled with aggressive optimizations that discard type information for function parameters and local variables. Go's design further complicates analysis by representing strings as pointer–length pairs rather than null-terminated sequences, employing a caller-allocated stack model that obscures argument boundaries, and fragmenting program state across concurrent goroutines. Although existing static analysis and reverse engineering tools provide Go-specific support, they remain limited to compile-time artifacts and cannot recover runtime execution state and artifacts that persist solely in memory. To address this gap, we present the first memory forensics framework for runtime analysis of Go binaries. By parsing Go's internal structures, our framework reconstructs type and function metadata, recovers heap-allocated and static strings, and distinguishes application-level functions. Through ABI-aware backward analysis, it derives execution paths and argument values from call sites. To capture runtime state beyond what static analysis reveals, it analyzes goroutine stacks to identify actively executing functions and recover their runtime argument values. We implemented all capabilities as Volatility 3 plugins and evaluated them against malware observed in recent incidents, such as the *BRICKSTORM* backdoor, *Obscura* ransomware, and *Pantegana* RAT, as well as open-source samples for reproducibility. The framework successfully recovered C2 endpoints, persistence mechanisms, encryption keys, ransom notes, and execution state, including critical runtime artifacts that were absent from published threat intelligence.

1. Introduction

The *Go* programming language has become increasingly popular among malware authors due to its cross-platform compilation capabilities, which allow a single codebase to produce executables targeting multiple platforms (Grunzweig, 2019, Maurya, 2021, Stevens, 2025). This adoption has created significant challenges for malware analysts and incident responders. Go produces statically linked binaries that embed a substantial custom runtime and extensive compiler-generated metadata, resulting in large executables (d4rksystem, 2025). Moreover, such binaries are compiled with aggressive optimizations that discard type information for function parameters and local variables (InstaTunnel Team, 2025). These properties reduce the effectiveness of traditional detection mechanisms and complicate reverse engineering.

Go's design introduces further obstacles for analysis. Unlike C and C++, strings are represented as pointer–length pairs rather than null-terminated byte sequences, eliminating explicit delimiters and thus making reliable extraction of string-based artifacts difficult (Guerrero-Saade, 2021). Dynamic analysis is similarly challenging, as Go employs a caller-allocated stack model in which arguments and return values reside in caller-reserved memory regions, obscuring argument boundaries and complicating parameter recovery (Calvet, 2019). These inherent challenges are compounded when binaries are stripped of symbols, lack debug information, or are obfuscated using tools such as Garble, which hashes identifiers and removes metadata to disrupt static analysis (Raimbaud and Rascagneres, 2025).

Although reverse-engineering tools, including *IDA Pro* and *Ghidra*, have improved their support for Go binaries (Hex-Rays, 2025, Trellix, 2023), they rely on heuristic-based disassembly without fully parsing the embedded metadata. This results in incomplete function boundaries, fragmented call graphs, and prototypes lacking type annotations. Specialized tools, such as *GoReSym*, address these limitations by parsing metadata structures derived from the Go runtime source code (Mandiant, 2026, Stephen Eckels). Nevertheless, all existing tools remain limited to compile-time artifacts and cannot reconstruct runtime program state or recover critical runtime artifacts that reside solely in memory, such as decrypted strings, encryption keys, and command-and-control configurations.

To address this gap, this paper introduces the first memory forensics framework for runtime analysis of Go binaries. By parsing Go's internal structures, including `pclntab` and `moduledata`, it recovers function metadata, type descriptors, and interface tables. Leveraging this metadata, the framework identifies heap objects, extracts dynamically allocated and static strings, and classifies functions by origin. Through ABI-aware backward analysis, it derives execution paths and argument values from call sites. Goroutine stack analysis extends this to runtime state, identifying actively executing

*Corresponding author
alih16@vcu.edu (H. Ali); andrew@dfir.org (A. Case); iahmed3@vcu.edu (I. Ahmed)

functions and recovering argument values that exist only at execution time. By relying on structural validation rather than symbolic information, the framework remains effective on stripped binaries and maintains robustness against obfuscation that corrupts version identifiers and magic bytes within `pclntab`.

We implemented our framework as a suite of Volatility 3 plugins (Volatility Foundation, 2025), supporting multiple Go versions on both Linux and Windows. We evaluated it against real-world malware recently observed in the wild, including the *BRICKSTORM* backdoor (Cybersecurity and Infrastructure Security Agency (CISA) et al., 2025, Mandiant, 2025a), *Pantegana* RAT (Hunt.io, 2025b,a), *Obscura* ransomware (Carvey et al., 2025, Coveware by Veeam, 2025), as well as the open-source *Screenshotter* application for reproducibility (omaidf, 2025). Results demonstrate that `go_strings` successfully extracted string artifacts with clear boundaries and memory locations, such as hard-coded credentials from *Screenshotter*, which appear only as concatenated data in standard `strings` output. From *Obscura*, it recovered the ransom note, Curve25519 public key, and threat actor communication channels, including a *Tor* hidden service URL and *TOX* messaging ID. The `go_functions` plugin recovered complete binary paths and classified functions by source file, while its ABI-aware backward analysis of *BRICKSTORM* revealed C2 endpoints, persistence paths, and DNS-over-HTTPS resolver settings. Finally, `go_goroutines` analyzed *Pantegana* goroutine stacks, recovering deployment-specific C2 configuration from active function arguments, such as server address, API endpoints, and beacon intervals. We summarize the contributions of this paper as follows:

- We present the first memory forensics framework for reconstructing the runtime behavior of Go binaries, enabling analysis beyond compile-time artifacts.
- We parse Go's internal structures to extract strings from heap and static sections, classify functions by origin, derive execution paths and argument values, and analyze goroutine stack to recover execution state.
- We implement the proposed framework as Volatility 3 plugins and evaluate it on real-world Go malware, demonstrating accurate recovery of critical runtime forensic artifacts.

The paper is organized as follows. Section 2 summarizes related work. Section 3 explains the framework. Section 4 presents the evaluation, Section 5 discusses limitations and future directions, and Section 6 concludes the paper.

2. Related Work

Prior work has shown that malware frequently abuses userland runtimes, motivating memory forensics techniques that reconstruct high-level execution state from process memory. The Android runtime, for example, has been widely targeted to access sensitive resources, such as call history, text messages, microphones, and cameras (Smmarwar et al., 2024). In response, multiple memory analysis approaches have been proposed to acquire Android memory and detect runtime-resident malicious behavior (Sylve et al., 2012, Case and Richard III, 2017, Ali-Gombe et al., 2020, 2019, Tam et al., 2015). On macOS, the Objective-C and Swift runtimes have similarly been analyzed using Volatility plugins to detect suspicious method invocation, keystroke logging, and runtime manipulation (Case and Richard III, 2016). Subsequent research extended these capabilities to enumerate loaded classes and instances, decode type information, and identify suspicious API usage and malicious function calls (Manna et al., 2021). The .NET and .NET Core runtimes have also been subject to analysis that extract memory-resident assemblies and recover runtime components, such as classes, fields, methods, and native imports (Manna et al., 2022). More recently, researchers analyzed the V8 JavaScript runtime by extracting and tracing V8 objects via internal structures such as *MetaMap* (Wang et al., 2022), and analyzed the Python runtime by recovering modules, classes, functions, and execution context to capture malicious behavior (Ali et al., 2025b,a).

3. Memory Analysis Framework

This section presents our memory forensics framework, which recovers Go's embedded runtime metadata as the foundation for all subsequent analysis.

3.1. Go Runtime Metadata Extraction

Go binaries embed linker-generated metadata that supports core runtime functionality, including stack unwinding and garbage collection. A primary component of this metadata is `pclntab` (program counter line table), which begins with a `pcHeader` structure. The `pcHeader` contains version-identifying magic bytes, architectural parameters, such as the minimum instruction size (`minLC`) and pointer size (`ptrSize`), counts of functions and source files (`nfunc`, `nfiles`), and relative offsets to several internal tables. These tables include `funcnametab`, which stores function name strings; `filetab`, which stores source file paths; `cutab`, which maps compilation units to ranges of file indices; `pctab`, which encodes PC-value data used for stack frame size and line number resolution; and `pclntable`, which contains the function table (`ftab`) and associated `_func` structures. Each `ftab` entry maps a function's entry program counter to its corresponding `_func` structure, which references the function name, PC-value streams, and source file information.

When a Go binary is loaded, the runtime initializes `moduledata`, a linker-emitted structure located in a writable data section. This structure references the same internal tables as `pcHeader`, but uses absolute addresses resolved at process initialization rather than relative offsets. In addition, `moduledata` defines memory region boundaries for the loaded binary, including `text`, `data`, `bss`, `rodata`, and `types`. It also contains two slice fields: `typelinks`, which store offsets to type descriptors, and `itablinks`, which store pointers to interface tables (`itab`) that bind concrete types with implemented interfaces. Figure 1 summarizes the metadata extraction pipeline. The primary path recovers metadata from process memory, while a supplementary path extracts paged-out function names and source file paths from the cached binary.

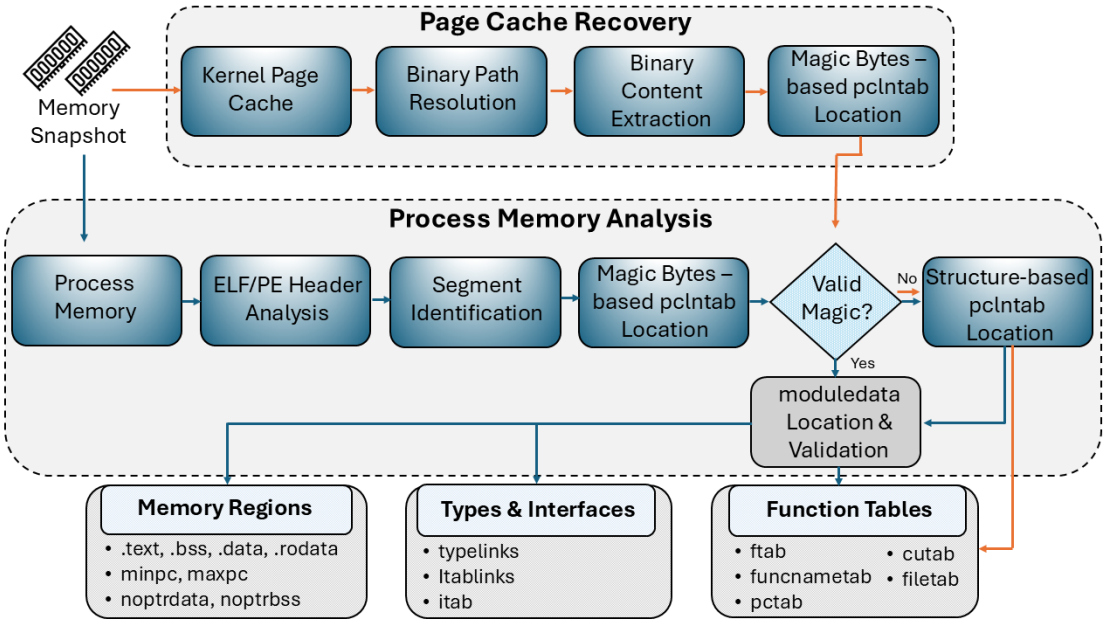

Figure 1: Go runtime metadata extraction

3.1.1. Process Memory Analysis

The framework parses the executable header to identify program segments and then scans read-only segments for `pclntab` using version-specific magic bytes: `\xf1\xff\xff\xff` (Go 1.20+), `\xf0\xff\xff\xff` (Go 1.18–1.19), `\xfa\xff\xff\xff` (Go 1.16–1.17), and `\xfb\xff\xff\xff` (Go 1.2–1.15). Each candidate match is validated by enforcing `pcHeader` invariants, requiring zeroed padding bytes, `minLC` $\in \{1, 2, 4\}$, `ptrSize` $\in \{4, 8\}$, and function and file counts within reasonable bounds. When magic-byte detection fails due to obfuscation, the framework transitions to structure-based discovery, evaluating aligned addresses using these same invariants. Once `pclntab` is validated, the framework locates `moduledata` by scanning writable segments for a pointer equal to the recovered `pcHeader` address, as the first field of `moduledata` references `pcHeader`. Candidate structures are then validated by enforcing ordered memory regions (e.g., `data` $<$ `edata`) and well-formed slice headers satisfying `len` $\leq$ `cap`.

3.1.2. Page Cache Recovery

Function names and source file paths are primarily used for diagnostics (i.e., panic messages and stack traces) and are rarely accessed during normal execution, as the runtime operates directly on program counters. Consequently, pages containing `funcnametab` entries may be paged out under memory pressure, particularly for infrequently executed code. To recover complete metadata, the framework additionally extracts `pclntab` from the kernel page cache. In ELF binaries, `pclntab` resides in the `.gopclntab` section, and, in PE binaries, it is embedded as the `runtime.symtab` symbol within the `.symtab` section. In contrast, `moduledata` cannot be reconstructed from page cache alone, as it is only partially initialized in the static binary and is finalized by the runtime during process startup.

3.1.3. Obfuscation Handling

Go obfuscation tools, such as *Garble*, explicitly target symbolic and metadata-level artifacts (Martí, 2024), including function names, file paths, type names, and static strings, to hinder static analysis and reverse engineering. These transformations do not modify the core components required by the Go runtime for correct execution. Accordingly, our framework avoids reliance on symbolic information and instead analyzes runtime structures directly, as discussed earlier in this section. Heap spans and goroutine objects are recovered based on their structural layout rather than by resolving the `runtime.mheap` or `runtime.allgs` symbols. Consequently, recovery of critical forensic artifacts, such as function bodies, argument layouts, type descriptors, heap objects, and goroutine stacks, remains robust under obfuscation. In addition, dynamically allocated strings located in the heap remain in plaintext at runtime and are therefore unaffected by compile-time obfuscation, representing a key advantage of memory forensics over static analysis.

Given that obfuscation can corrupt or randomize version identifiers and magic bytes within the `pclntab`, our framework identifies the Go version based on changes in data structures introduced across Go releases. For example, the `textStart` field in `pcHeader`, as well as the `rodata` and `gofunc` fields in `moduledata`, are introduced in Go 1.18. Coverage-related fields (`covctrs`, `ecovctrs`) and the `inittasks` field appear in Go 1.20, while the `sys.NotInHeap` marker is introduced in Go 1.21. However, when symbolic deobfuscation is required, existing tools can be applied to executables extracted from memory. For example, *GoReSym* (Mandiant, 2026) reconstructs function and package symbols using Go runtime metadata and heuristic analysis; *GoStringUngarble* (Mandiant, 2025b) recovers strings obfuscated by *Garble*; and *GoResolver* (Volexity, 2025) resolves obfuscated standard library functions via control-flow graph similarity.

3.1.4. Type Metadata Extraction

Each entry in `typelinks` references a type descriptor beginning with a common `_type` header, which encodes size, pointer metadata (`ptrdata`), alignment requirements (`align`, `fieldAlign`), type flags (`tflag`), and a type kind (`kind`). Go defines 26 kinds spanning primitive, composite, and abstract types (Calvet, 2019). Following the common header, descriptors include kind-specific metadata sufficient to traverse and reconstruct typed values, such as pointers, slices, arrays, structs, interfaces, functions, and methods. For type methods, the descriptor is further followed by an `uncommonType` structure containing method-related metadata, including the package path, method counts, and references to method descriptors. Each method descriptor specifies the method name, the number of input parameters and return values, the concrete types of each, and the entry points for direct (`tfn`) and interface-based (`ifn`) invocation.

3.1.5. Interface Metadata Extraction

Each `itab` entry stores pointers to interface type descriptor (`inter`), concrete type descriptor (`_type`), cached type hash, and function pointer array (`fun`) used for method dispatch. The `fun` array contains one entry per interface method, ordered according to the interface's method set, with each entry holding the program counter of the concrete implementation. We parse each `itab` by resolving `inter` to recover the interface method set and `_type` to identify the implementing type. The `fun` entries then provide direct mappings from interface methods to concrete implementations, enabling resolution of polymorphic calls executed via interface dispatch.

3.2. String Recovery

Go represents strings as a structure comprising a pointer to the underlying UTF-8 byte array and an integer specifying the length. This separates headers from backing data, with each potentially residing in different memory regions. For instance, heap-resident and stack-resident headers reference data in either the heap (for dynamically created strings) or .rodata (for static constants). In contrast, headers in static sections (.rodata, .data, .bss) always reference data located only in .rodata, as these represent compile-time literals or global variables initialized with constant strings.

3.2.1. Heap-Based String Recovery

Recovering heap-resident strings requires identifying allocated heap objects and interpreting them using runtime type metadata. Figure 2 summarizes this process.

Heap Structure Parsing. The Go heap is organized into spans containing fixed-size slots for objects of a given size class. While span recovery relies on locating the global `runtime.mheap` symbol, this structure is large, vary across versions, and may be absent in stripped or obfuscated binaries. Instead, we locate the `allspans` slice header within the `.data` or `.bss` sections and validate candidates using alignment checks, length–capacity constraints, and verification that referenced entries point to valid `mspan` structures. Each `mspan`

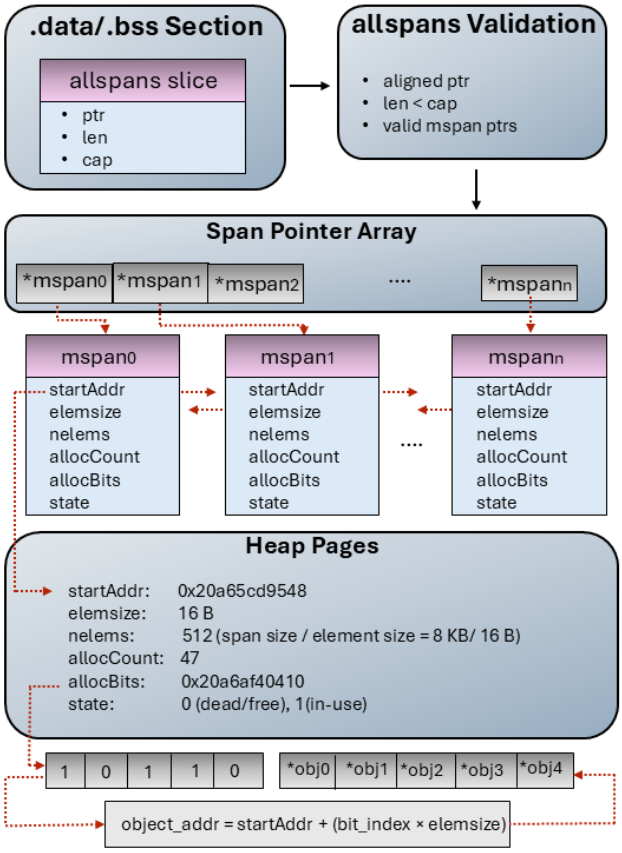

Figure 2: Heap object recovery

describes a contiguous memory region divided into fixed-size slots. Key fields include the base address (`startAddr`), slot size (`elemsize`), total capacity (`nelems`), number of allocated objects (`allocCount`), allocation bitmap (`allocBits`), and span state. For active spans, allocated objects are recovered by iterating the allocation bitmap and calculating object addresses using their `bit_index` (position in the bitmap).

Size-to-Type Mapping. Heap objects lack type annotations, preventing direct identification of their types. We therefore infer candidate types by correlating the object size specified by each span's `elemsize` with recovered type descriptors. By extracting `_type.size` from each descriptor, we group types by size and map heap object sizes to candidate types. Since multiple types may share the same size, a single object size may correspond to several candidates. To reduce ambiguity, candidates are restricted to types whose layout may directly contain or indirectly reference string values. These include strings; arrays or slices of strings; composite types that reference string-typed elements or fields; maps whose key or value type involves strings; and interfaces, whose concrete types are determined dynamically at runtime and therefore cannot be excluded statically.

Type-Guided Object Interpretation. After identifying candidate types for each object, we interpret the object under each candidate type. For objects whose size equals two machine words, we first disambiguate interface and string representations. If the first word points to a valid `itab` address, the object is treated as an interface value and its

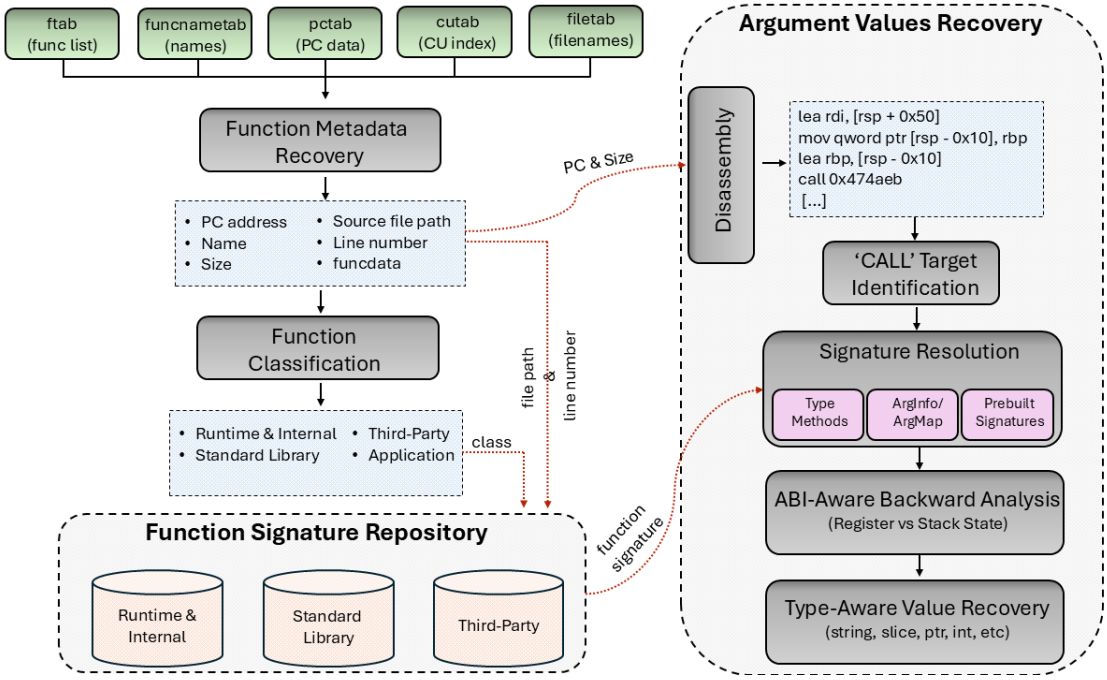

Figure 3: Function analysis workflow

concrete payload is processed recursively. Otherwise, we treat the first word as a potential string data pointer and the second as a potential length. We then validate that the pointer references accessible memory, the length is within reasonable bounds, and the referenced bytes form valid UTF-8 with predominantly printable characters. Objects that fail string validation are excluded and evaluated under the remaining candidates. Pointer-typed objects, slices, structures, and maps (hash tables) are analyzed using the same algorithm against their contained fields.

3.2.2. Static Section String Recovery

Unlike the heap, static sections (`.rodata`, `.data`, and `.bss`) store data sequentially without size metadata or explicit object boundaries. To recover strings from these regions, we scan for patterns matching Go string headers. For each aligned candidate, the first word is interpreted as a potential data pointer and the second as a potential length, and candidates are validated using the same pointer, length, and content checks described earlier. This pattern-based approach enables recovery of compile-time string literals and global string variables that reference constant data.

3.3. Function Analysis

Figure 3 illustrates the function analysis workflow to recover function metadata, classify functions by origin, infer argument types, and apply ABI-aware backward analysis to recover argument values at call sites.

3.3.1. Function Metadata Recovery

We recover function metadata by parsing the internal tables referenced by `moduledata` (Section 3.1). Each function is described by a `_func` structure, which references the function name via `nameoff` (an offset into `funcnametab`), identifies its compilation unit via `cuOffset`, and provides offsets into PC-relative metadata streams in `pctab` for stack frame sizes (`pcsp`), source file indices (`pcfile`), and line numbers (`pcln`). These streams encode metadata as compact sequences of value and program-counter deltas, allowing metadata to be resolved at arbitrary instruction addresses within the function. Although a function is defined in a single source file, inlined and compiler-generated code may originate from other files. Consequently, source file paths are not stored directly in `_func`. Instead, the file index is decoded from the `pcfile` stream at the target program counter and combined with `cuOffset` to index into `cutab`, yielding the corresponding offset into `filetab` where the source file path resides.

3.3.2. Function Classification

Once the source file path is recovered, we use it to classify functions. Paths beginning with `runtime/` are classified as runtime core, while paths under `internal/` are split into runtime-critical internal packages (e.g., `internal/abi`, `internal/cpu`) and other standard-library internal packages (e.g., `internal/poll`). Paths whose first segment matches a public standard library package (e.g., `net`, `os`, `fmt`) are classified as public standard library. On the other hand, third-party functions are identified by domain-based prefixes (e.g., `github.com`, `gitlab.com`) or by paths under `vendor/` directories. Remaining functions, including those in the `main` package and other user-defined packages within the same module, are classified as application-level code.

3.3.3. Argument Type Inference

We leverage the recovered source file path and entry line number to resolve function identities via signature repositories, particularly when function names are unavailable due to paging. Each signature associates a function name with its defining file, entry line number, and parameter types. For type methods, signatures are recovered directly from the

uncommonType structure. For runtime, internal, and standard library functions, we parse the official Go source distribution for the detected Go version, extracting signatures from both .go source files and compiler-generated .s assembly files. For third-party dependencies, which vary across applications, signatures are generated on demand by parsing the corresponding source packages.

Application-level functions present a distinct challenge as their signatures are not known a priori and type information is frequently eliminated by aggressive compiler optimizations. To address this, we infer argument structure from compiler-generated metadata stored in the funcdata associated with each _func. For binaries compiled with Go 1.17 and later, we parse ArgInfo bytecode, while for earlier versions we rely on ArgsPointerMaps. Both structures describe the layout of arguments within the call frame. ArgInfo encodes argument boundaries explicitly, representing scalar arguments by their frame offsets and sizes, and composite arguments (e.g., strings, slices, structs) as aggregates delimited by start and end markers. Unlike ArgInfo, ArgsPointerMaps encodes only a bitmap indicating which stack slots within the argument area contain pointers, without explicit argument boundaries. We therefore infer boundaries using Go's type layout conventions.

3.3.4. ABI-Aware Backward Analysis

Once signatures are resolved, we map parameters to their locations based on the calling convention. Go 1.17 and later use a register-based ABI, passing arguments through general-purpose registers and spilling excess arguments to the stack (The Go Authors, 2024). Earlier versions use a stack-based ABI, placing all arguments at consecutive word-aligned stack offsets. To recover argument values, we disassemble the caller function and traverse backward from each CALL instruction to identify the last write to each argument location. Immediate mov instructions define constant values, register-to-register mov instructions propagate values through copy chains, RIP-relative lea instructions compute addresses of static data, and xor reg, reg patterns indicate zero initialization. Traversal terminates at preceding call instructions to avoid crossing call boundaries, as register contents may be modified by callees.

3.4. Goroutine Stack Analysis

Go maintains a global slice, runtime.allgs, containing pointers to all goroutine structures. As stripped binaries lack this symbol, we scan writable memory regions for candidate slice headers and validate them by ensuring that the referenced array contains pointers to well-formed g structures. A candidate g is considered valid if its stack bounds are properly aligned and ordered, its status field encodes a correct runtime state, and its goroutine identifier falls within reasonable bounds.

Stack Unwinding. Each goroutine's execution state is stored in the g.sched field, which stores the current stack pointer (sp) and program counter (pc). We reconstruct the call stack by iteratively unwinding frames starting from these values. At each iteration, the corresponding function is identified by searching the function table (ftab) for the entry whose PC range encompasses the current pc. The pcsp stream of the function is then decoded to obtain sp_delta, representing the frame size at that pc. Adding sp_delta to the current sp locates the caller's return address, which resides at the boundary between frames. Since Go's calling convention places callee arguments in caller-allocated stack space (The Go Authors, 2024), the argument area begins immediately above the return address at sp + sp_delta + ptrSize, where ptrSize denotes the pointer size. To advance to the caller's frame, the return address becomes the new pc and the argument base becomes the new sp. Unwinding continues until an invalid return address is encountered or the stack bounds are exceeded. Once the argument base is determined, individual argument boundaries are established using the argument type reference techniques described in Section 3.3.

4. Experimental Evaluation

We implemented our framework as a suite of Volatility 3 plugins, namely go_strings, go_functions, and go_goroutines, and evaluated them using memory images collected from controlled executions of precompiled Go binaries[1]. Page cache recovery leveraged existing Volatility 3 plugins, *linux.pagecache* and *windows.dumpfiles*, while function disassembly was performed using *Capstone* (Capstone Engine Team, 2024). Our evaluation includes Linux memory images of the *BRICKSTORM* backdoor (Go 1.16.3) and *Pantegana* RAT (Go 1.25.5), a Windows memory image of the *Obscura* ransomware (Go 1.15), and an open-source *Screenshotter* application compiled with Go 1.24.10.

4.1. Screenshotter Memory Analysis

We demonstrate the advantages of go_strings over the standard strings utility on the *Screenshotter* sample. In statically linked Go binaries, compile-time string literals are stored as contiguous blobs without null terminators. These blobs combine strings not only from application code, but also from the Go runtime and third-party libraries. As a result, strings produces a single concatenated output stream from which individual artifacts are difficult to recover. As shown in Figure 4a, application-specific artifacts, including the username, password, and remote host used for SFTP-based screenshot exfiltration, are indistinguishable within the output. In contrast, Figure 4b shows that go_strings reconstructs Go string headers together with their backing data and memory locations. This enables reliable extraction of sensitive forensic artifacts and identifies whether strings are compile-time constants or generated at runtime.

4.2. Obscura Ransomware Memory Analysis

Obscura is a Go-based ransomware first observed in August 2025, targeting enterprise environments through domain controller compromise and NETLOGON share distribution (Carvey et al., 2025). It employs XChaCha20 encryption

[1] The plugins are available at https://github.com/HalaAli198/Go-Memory-Forensics.

```
user1@ubuntu:~/go_malware/Screenshotter$ strings -a screenshotter | grep -E "username|user1password|192.168.181.138"

writetofips140SHA-224SHA-256SHA-384SHA-512Ed25519MD5-RSAavx512fos/execruntimetls3desanswersAES-CBCusernameGoStringThursday
SaturdayFebruaryNovemberDecember%!Month(3des-cbcpasswordhostkeynistp256nistp384nistp521runtime:sp=abimismatchwrongtimersil
legalseekinvalidslothostisdownmultipathtcp127.0.0.1:53nosuchhostunknownportinvalidportinvalidbaseECDSA-SHA256ECDSA-SHA384E
CDSA-SHA512randautoseednotpollablegotypesaliashttpmuxgo121tlsunsafeekmRCodeSuccessRCodeRefused/dev/urandomunknown
sizeuser1password/etc/zoneinfoistoolargenoconnectionSSH_FXP_CLOSESSH_FXP_WRITESSH_FXP_LSTATSSH_FXP_FSTATSSH_FXP_MKDIRSSH_F
XP_RMDIRSSH_FXP_ATTRSsftp:%q(%v)hmac-sha2-256hmac-sha2-512filtermethodButtonPresscrypto/fips140mime/multipartRCodeNameErro
rResourceHeaderunknowndsbyte192.168.181.138,M3.2.0,M11.1.0packetoolongconnectionlostSSH_FXP_VERSIONSSH_FXP_SETSTATSSH_FXP_
OPENDIRSSH_FXP_READDIRSSH_FXP_SYMLINKssh
```

(a) Concatenated output produced by the Linux `strings` utility

```
user1@ubuntu:~/volatility3$ python3 vol.py -f /mnt/hgfs/Screenshotter/Snapshot.vmem linux.go_strings.Go_Strings --pid 36796
Volatility 3 Framework 2.27.1
Struct Addr     Data Addr       Length      Value                                   Struct Location     Data Location
0xc00002ea90    0xc0002d7740    37          "Wednesday, 07-Jan-26 23:52:18 EST.png" heap                heap
0xc00002eac0    0xc0000182a0    17          "/tmp/.X11-unix/X0"                     heap                heap
0xc00002eaf0    0xc0002d7710    37          "Wednesday, 07-Jan-26 23:34:18 EST.png" heap                heap
                                        [Snip]
0c000188010    0x664706         14          "ssh-connection"                        heap                rodata
0xc000188070    0x662da8        8           "password"                              heap                rodata
0xc000188088    0x6641c4        13          "user1password"                         heap                rodata
0xc0000e4250    0xc000010390    10          "aes128-ctr"                            heap                heap
0xc00007a4f0    0xc0000103e0    9           "hmac-sha1"                             heap                heap
0xc000012040    0x664ada        15          "192.168.181.138"                       heap                rodata
                                        [Snip]
0xc00007a560    0xc000018468    20          "umac-128@openssh.com"                  heap                heap
0xc00007a590    0xc000010410    9           "hmac-sha1"                             heap                heap
0xc000188000    0x662d60        8           "username"                              heap                rodata
0xc0001998d0    0x6f7bac        31          "image/png.(*encoder).writeImage"       heap                rodata
```

(b) Type-aware strings recovered by the `go_strings` plugin

Figure 4: Comparison of the `go_strings` plugin and the Linux `strings` utility

with Curve25519 key agreement, terminates security software prior to encryption, and deletes volume shadow copies to prevent recovery. We analyzed a memory image from an infected Windows host containing a live Obscura process (PID 1100). Using `go_strings`, we reconstructed heap allocation state (112 of 942 spans active) and recovered 778 strings from 550 heap objects, along with 728 additional strings from non-heap regions. Table 1 presents runtime artifacts recovered from memory, compared against public threat intelligence reports on Obscura (Carvey et al., 2025, Meskauskas, 2025). Highlighted entries denote artifacts absent from these reports: the 32-byte Curve25519 public key value, which enables decryption if the corresponding private key is recovered; exact Go module dependencies revealing the cryptographic implementation, which link samples compiled from the same build environment; and the Go toolchain (`GOROOT`) path, which exposes the threat actor's toolchain configuration on a VeraCrypt-encrypted volume. This table also includes previously reported indicators, such as domain-role detection strings (standalone, domain member, BDC, PDC), which provide direct evidence of Obscura's propagation logic, escalating from workstations to domain controllers and subsequently targeting all domain-joined systems when executed on a primary domain controller.

Using the `go_functions` plugin, we recovered Obscura's functions together with their source file paths. Similar to compiler-generated PDB paths, file paths are forensically valuable, as they expose characteristics of the build environment that may persist across malware samples (Wilhoit, 2025). In Obscura, the recovered build root (`/run/media/veracrypt1/Backups/Obscura/Locker/windows/`) indicates compilation from a VeraCrypt-mounted volume. Grouping recovered functions by their file paths indicates a modular internal organization. The main module (`locker/locker.go`) implements file discovery (`scanDisk`), parallel encryption (`worker`), ransom note deployment (`dropNote`), and anti-analysis checks (`runtimeCheck`); the encryption module (`api/xchacha20.go`) implements full and partial encryption routines (`EncryptFull`, `EncryptPart`); and `api/api.go` implements Windows API wrappers for privilege checks (`IsRunAsAdmin`), drive enumeration (`GetWindowsDrives`), and process termination (`KillProcess`).

4.3. BRICKSTORM Memory Analysis

BRICKSTORM is a Go-based backdoor attributed to the China-nexus threat actor *UNC5221* and has been observed in intrusions targeting VMware vCenter infrastructure, with activity reported through December 2025 (Mandiant, 2025a, Cybersecurity and Infrastructure Security Agency (CISA) et al., 2025). Public threat intelligence characterizes its persistence mechanisms, DNS-over-HTTPS (DoH) command-and-control, and post-exploitation capabilities, while providing limited insight into execution-time state and actionable memory-resident artifacts. We analyzed a Linux sample (PID 2004) masquerading as a VMware Java component.

Table 2 summarizes quantitative metadata recovered using the `go_functions` plugin, including the number of recovered functions and source file paths from both process

Table 1
Runtime forensic artifacts recovered from Obscura memory using `go_strings`. Highlighted cells indicate the novel artifacts

Artifact	Value
Curve25519 Public Key (Base64)	`Z7oIV9mbFaMuePnpNBvdcweN+FIFLFVaK1a8TmVMTUs=`
Encryption / Key Exchange Algorithms	`XChaCha20` / `Curve25519`
TOX ID (from the note)	`AE55FC0EB1C25A5B081650108F9081E236DECE1CE08D2E185A6F15B9FB48E700210BED374643`
Tor Hidden Service (from the note)	`http://obscurad3aphckihv7wptdxvdnl5emma6t3vikcf3c5oiiqndq6y6xad.onion`
Victim ID (from the note)	`148061707215636819`
Crypto Dependency	`golang.org/x/crypto v0.0.0-20201203163018-be400aefbc4c`
System Dependency	`golang.org/x/sys v0.0.0-20201018230417-eeed37f84f13`
Go Toolchain (GOROOT) Path	`/run/media/veracrypt1/Locker Deps/go1.15.linux-amd64/go`
Module Path	`main/windows/locker`
Domain Role Detection Strings	`[+] detect standalone pc., [+] detect pc in domain. run transfer to dc., [+] detect BDC. run transfer to PDC., [+] detect PDC. run transfer to all pc in domain.`
Runtime Status Messages	`[!!!] user not admin. exit [!!!], [-] failed to delete shadow copies., [-] failed to run as daemon!, [+] encryption start. you may close this window, Error calling DsRoleGetPrimaryDomainInformation:`
Security Software	`WinDefend, MsMpEng, MpCmdRun, CSFalconService, SentinelAgent`
Antivirus	`bdagent, McAfee, SymCorpUI, ccSvcHst, AMService, Emsisoft*`

Table 2
Recovered BRICKSTORM metadata using `go_functions`

Attribute	Value
Process ID (PID)	2004
Cached ELF Path	`/usr/java/jre-vmware/bin/upgrademgr`
ELF Architecture	64-bit, Little Endian
ELF Size (Extracted)	5,844,992 bytes
Go Version	`go1.16.3hijacke`
Cached Function Names	5,524
Cached File Paths	5,524
Recovered Functions	5,524
Recovered Types	3,597
Recovered Interfaces	360
Recovered Type Methods	1,695

memory and the cached binary, in addition to the number of recovered types, interfaces, and type methods.

Given that the analyzed sample was compiled with Go 1.16, argument recovery relies on parsing `ArgsPointerMaps` (see Section 3.3). Despite this challenge, Table 3 shows that `go_functions` successfully identifies *BRICKSTORM*'s main functions and execution paths and recovers argument values. We compared all recovered artifacts against the CISA malware analysis report (Cybersecurity and Infrastructure Security Agency (CISA) et al., 2025) and Mandiant's threat intelligence publication (Mandiant, 2025a). Highlighted entries in Table 3 denote artifacts absent from those reports. These include the configuration file `/etc/sysconfig/vmp` (CISA documents only the parent directory `/etc/sysconfig/`), the `KENSO` and `ENVLOG` environment variables, the utility function `wssoft/libs/utils.HasExistProc` (distinct from the documented `wssoft2` C2 package), raw XOR key material used to recover DoH resolver addresses (Table 4), a beacon timer interval of `0x2260ff9290000` (≈ 7 days), randomized sleep and jitter parameters (0–600 and 0–7200 seconds), the C2 termination keyword `"exit"`, and exit code `200`.

Analysis of `main.startNew` revealed the complete persistence workflow. The malware first checks the `KENSO` environment variable as a kill-switch; if set to `"true"`, *BRICKSTORM* removes itself and terminates. Otherwise, it reads an embedded payload from `/etc/sysconfig/vmp`, deletes any existing binary at `/usr/java/jre-vmware/bin/upgrademgr`, writes the payload with executable permissions (`0755`), and modifies the `PATH` environment variable to prioritize this directory. This masquerades the backdoor as a legitimate VMware Java Runtime component. Notably, these file paths were recovered while analyzing the `main.copyFile` function (Figure 5). Persistence is maintained via `main.selfWatcher`, which implements a watchdog that monitors process liveness and reinstalls the payload if necessary. Recovered beacon delay and jitter parameters indicate timing-based evasion, explaining BRICKSTORM's resistance to sandbox analysis. Prior to C2 communication, six deobfuscation routines (`main.main.func1–func6`) apply bytewise XOR over stacked key arrays to recover DNS-over-HTTPS resolver endpoints. Figure 6 illustrates the XOR key pairs recovered via ABI-aware backward analysis of `main.main.func1`, while Table 4 enumerates the complete set of recovered keys and shows their decrypted endpoints.

4.4. Pantegana Memory Analysis

Pantegana is a fully-featured, Go-based remote access trojan that supports command execution, file transfer, and system fingerprinting. Originally released as an open-source project, it was later abused by the China-nexus group RedNovember in a global espionage campaign targeting government ministries, defense contractors, and critical infrastructure, after which the repository was removed from GitHub (Hunt.io, 2025b,a). To conduct our experiment, we obtained a preserved copy of the original source code, compiled the client and server following the same configuration reported in prior analyses, and executed *Pantegana* within a controlled local network environment. We captured memory from the "victim" system after executing several reconnaissance commands from the server. We then used `go_functions` to reconstruct the execution flow that starts at `main.main` and proceeds to `client.RunClient`, which orchestrates the RAT's core functionality by calling `client.RunFingerprinter` for

```
0x6e4561: lea      rax, [rip + 0x9d83d]          ; "/etc/sysconfig/vmp"
0x6e4568: mov      qword ptr [rsp], rax
0x6e456c: mov      qword ptr [rsp + 8], 0x12     ; = 18
0x6e4575: call     0x4d1700                      ; os.ReadFile
              [Snip]
0x6e45ae: lea      rax, [rip + 0xa4c2e]          ; "/usr/java/jre-vmware/bin/upgrademgr"
0x6e45b5: mov      qword ptr [rsp], rax
0x6e45b9: mov      qword ptr [rsp + 8], 0x23     ; = 35
0x6e45c2: call     0x4d25c0                      ; os.Remove
              [Snip]
0x6e45c8: lea      rax, [rip + 0xa4c14]          ; "/usr/java/jre-vmware/bin/upgrademgr"
0x6e45cf: mov      qword ptr [rsp], rax
0x6e45d3: mov      qword ptr [rsp + 8], 0x23     ; = 35
              [Snip]
0x6e45fa: mov      dword ptr [rsp + 0x28], 0x1ed ; = 493
0x6e4602: call     0x4d1aa0                      ; os.WriteFile
              [Snip]
```

Figure 5: ABI-Aware Backward Analysis of `main.copyFile`

Table 3
BRICKSTORM main functions with recovered arguments. Highlighted cells indicate novel artifacts discovered by `go_functions`

Depth	Function	Recovered Arguments	Forensic Significance
main.main (Entry Point) [0x6e4da0]			
1	main.startNew	–	Initialization and persistence
	→ os.Getenv	`"PATH","TERM","USER", "KENSO"`	Environment reconnaissance / kill switch
	→ main.copyFile	–	Payload installation
2	→ os.ReadFile	`"/etc/sysconfig/vmp"`	Embedded payload retrieval
	→ os.Remove	`"/usr/java/jre-vmware/bin/upgrademgr"`	Pre-installation binary removal
	→ os.WriteFile	`"/usr/java/jre-vmware/bin/upgrademgr", 0755`	**Trusted-path binary replacement**
1	→ os.Setenv	`"PATH", "/usr/java/jre-vmware/bin/"`	PATH hijacking
	→ os/exec.Command	`"upgrademgr"`	Payload execution
	→ os.Exit	`0`	process termination
main.selfWatcher (Process Watchdog) [0x6e39a0]			
1	os.Getenv	`"ENVLOG"`	**Operator-controlled runtime mode**
	wssoft/libs/utils.HasExistProc	`"upgrademgr"`	Process existence verification
	os.ReadFile	`"/etc/sysconfig/vmp"`	Payload retrieval for **self-reinstallation**
	os.WriteFile	`"/usr/java/jre-vmware/bin/upgrademgr", 0755`	self-reinstallation
	os.Setenv	`"PATH", "/usr/java/jre-vmware/bin/"`	Execution path restoration
	os/exec.Command	`"upgrademgr"`	Watchdog respawn
	time.NewTimer	`0x2260ff9290000` (≈ 7 days)	Long-delay scheduling
	math/rand.Int63n	`600` (0-10min), `7200` (0-2hr)	Randomized sleep and jitter
	runtime.selectgo	3 cases, block=true	Scheduler-mediated multiplexing
	os.(*Process).signal	–	Inter-process signaling
main.main *(continued) — C2 Task Loop*			
1	main.main.func1–func6	XOR key pairs (Table 4)	**String deobfuscation**
	wssoft/core/task.DoTask	`0x798af8` (task config)	C2 command execution
	strings.Index	`"exit"`	C2 termination keyword
	time.Sleep	calculated from rand	Evasion delay
	os.Exit	`200`	Distinctive termination exit code

system reconnaissance, `client.ClientSetup` for HTTP client initialization, and `client.RequestCommand` for the C2 polling loop. The latter constructs and dispatches HTTP requests via `net/http.NewRequestWithContext`, `net/http.(*Client).Do`, and `client.Middleware`.

However, `go_goroutines` revealed that these application-level functions were absent from the captured call stacks. While awaiting a C2 response, the goroutine is parked by the Go scheduler with a `select` state, and the underlying network file descriptor concurrently enters an `IO`

Table 4
Complete XOR Key Pairs Recovered from *BRICKSTORM* Memory

Function	Len	Key A (hex, little-endian qwords)	Key B (hex, little-endian qwords)	Decrypted String
func1	25	d88eebd0f9d5fc3a, c3cb8be38823922c, b1fd2c591f81753c, 30	f7a1d1a389a18852, ecf3a5dba61bbc14, c398592832f21b58, 49	https://8.8.8.8/dns-query
func2	25	1b67d2765502f10c, 54cfcf26fdae3805, 9ac28998a6fea591, 56	3448e80525768564, 7bfbe112d396163d, e8a7fce98b8dcbf5, 2f	https://8.8.4.4/dns-query
func3	25	74c4b1463d036b04, 26816391cf54809c, 2273df9d12618a88, a7	5beb8b354d771f6c, 09b84da8e16daea5, 5016aaec3f12e4ec, de	https://9.9.9.9/dns-query
func4	26	da62e026514ae0cb, 6f7aeb38cfedb01f, 74af8230987fe0c5, 139f	f54dda55213e94a3, 5e4bc501e1d49e26, 11daf31deb1184ea, 6aed	https://9.9.9.11/dns-query
func5	25	c209a43124860499, 461eb55f0c04f616, b95f447c0bef12fe, 15	ed269e4254f270f1, 692f9b6e2235d827, cb3a310d269c7c9a, 6c	https://1.1.1.1/dns-query
func6	25	8d93c87a64ed5031, 072864390872e841, eff656a7257d0c4e, ea	a2bcf20914992459, 28194a092642c670, 9d9323d6080e622a, 93	https://1.0.0.1/dns-query

Table 5
Runtime forensic artifacts recovered from Pantegana HTTP/TLS goroutines using `go_goroutines`

Goroutine ID	Wait Reason	Function (Frame)	Field	Recovered Value
C2 Server Configuration				
25, 29, 31, 33, 39, 57	select	`net/http.(*persistConn).writeLoop`	`cacheKey.{addr; scheme}`	`192.168.20.123:1337; https`
HTTP Request Buffers (Runtime-Constructed)				
39	select	`net/http.(*persistConn).writeLoop`	`bw.buf`	`POST /sysinfo HTTP/1.1\r\nHost: 192.168.20.123:1337`
25, 29, 31, 57	select	`net/http.(*persistConn).writeLoop`	`bw.buf`	`POST /cmdoutput HTTP/1.1\r\nHost: 192.168.20.123:1337`
33	select	`net/http.(*persistConn).writeLoop`	`bw.buf`	`GET /getcmd HTTP/1.1\r\nToken: 5112caedfb3a`
HTTP Response Buffers (Network-Received)				
24, 28, 30, 38, 56	select	`net/http.(*persistConn).readLoop`	`br.buf`	`HTTP/1.1 200 OK\r\nDate: Sat, 27 Dec 2025 23:49`
TLS Session State (Negotiated at Handshake)				

25, 29, 31, 33, 39, 57	select	`net/http.(*persistConn).writeLoop`	`tlsState.Version`,
			`tlsState.CipherSuite`,
			`tlsState.HandshakeComplete`
				`772 (TLS 1.3)`
				`4865 (TLS_AES_128_GCM_SHA256)`
				`True`

C2 Server Certificate (Network-Received)				
32	IO wait	`internal/poll.(*pollDesc).waitRead`	`[]uint8` (DER)	Subject: C=`US`, ST=`Hawaii`
Connection Metadata (Kernel State)				
32	IO wait	`net.(*netFD).Read`	`net; family; isConnected; Sysfd`	`tcp; AF_INET; True; fd=10`
Main Client State				
1	select	`net/http.(*persistConn).roundTrip`	–	Main HTTP client blocked awaiting C2 response

```
Caller pc: 0x6e56c0, Caller Name: main.main.func1 [depth=1]
0x6e56c0: mov      rcx, qword ptr fs:[0xfffffffffffffff8]
0x6e56c9: cmp      rsp, qword ptr [rcx + 0x10]
0x6e56cd: jbe      0x6e57d8
0x6e56d3: sub      rsp, 0x68
0x6e56d7: mov      qword ptr [rsp + 0x60], rbp
0x6e56dc: lea      rbp, [rsp + 0x60]                ; &stack[0x60]
0x6e56e1: xorps    xmm0, xmm0
0x6e56e4: movups   xmmword ptr [rsp + 0x47], xmm0
0x6e56e9: movups   xmmword ptr [rsp + 0x50], xmm0
0x6e56ee: movabs   rax, 0xd88eebd0f9d5fc3a
0x6e56f8: mov      qword ptr [rsp + 0x47], rax
0x6e56fd: movabs   rax, 0xc3cb8be38823922c
0x6e5707: mov      qword ptr [rsp + 0x4f], rax
0x6e570c: movabs   rax, 0xb1fd2c591f81753c
0x6e5716: mov      qword ptr [rsp + 0x57], rax
0x6e571b: mov      byte ptr [rsp + 0x5f], 0x30      ; = 48
0x6e5720: movups   xmmword ptr [rsp + 0x2e], xmm0
0x6e5725: movups   xmmword ptr [rsp + 0x37], xmm0
0x6e572a: movabs   rax, 0xf7a1d1a389a18852
0x6e5734: mov      qword ptr [rsp + 0x2e], rax
0x6e5739: movabs   rax, 0xecf3a5dba61bbc14
0x6e5743: mov      qword ptr [rsp + 0x36], rax
0x6e5748: movabs   rax, 0xc398592832f21b58
0x6e5752: mov      qword ptr [rsp + 0x3e], rax
0x6e5757: mov      byte ptr [rsp + 0x46], 0x49      ; = 73
```

Figure 6: ABI-Aware Backward Analysis of `main.main.func1`

`wait` state until data becomes available, unwinding application frames and leaving only blocked standard library frames, such as `net/http.(*persistConn).roundTrip`. Despite this, `go_goroutines` recovered critical runtime artifacts by inspecting `http.persistConn` structures referenced from blocked goroutine stack arguments. It identified 31 goroutines (23 valid), of which 13 were actively involved in HTTP/TLS communication with the C2 server. Table 5 summarizes the recovered artifacts, including the C2 address (`192.168.20.123:1337`), API endpoints used for system fingerprinting (`/sysinfo`), command output exfiltration (`/cmdoutput`), and command polling (`/getcmd`), a runtime-generated authentication token (`5112caedfb3a`), server responses, negotiated TLS 1.3 parameters, and X.509 certificate fields extracted from TLS buffers in Goroutine 32. These artifacts are constructed dynamically at runtime and are therefore inaccessible to static analysis.

As shown in Figure 7, `go_strings` recovered further heap-resident artifacts, including the C2 IP address, API endpoints used for system fingerprinting (`/sysinfo`), all of our reconnaissance commands (*whoami*, *id*, *who*, *cat /etc/shadow*), and their complete output. In total, `go_strings` recovered all the dynamic aspects of the malware, including C2 configuration, attacker-issued commands, exfiltrated outputs, and HTTP requests and responses.

5. Limitations and Future Work

Our framework currently targets x86-64 architecture, as it covers the majority of environments where Go malware is observed. Supporting ARM64 would require adapting the ABI-aware backward analysis to different calling conventions and register sets. To this end, large language models present a promising direction for generalizing across architectures. Moreover, the current analysis focuses on argument recovery at call sites and within goroutine frames. Thus, extending dataflow tracking to local variables and return values would enable richer semantic analysis, including taint propagation and more complete execution reconstruction.

6. Conclusion

This paper presented our efforts in analyzing Go binaries in memory. Our proposed framework parses Go's embedded structures to recover function metadata, type descriptors, and interface tables, considering stripped and obfuscated binaries. Leveraging this metadata, it identifies heap objects, extracts dynamically allocated and static strings, and classifies functions by origin. Through ABI-aware backward analysis, it further reconstructs execution paths and argument values from call sites. Goroutine analysis extends these capabilities by capturing active functions and their runtime argument values. We implemented these capabilities as Volatility 3

```
user1@ubuntu:~/volatility3$ python3 vol.py -f /mnt/hgfs/pantegana/Snapshot.vmem linux.go_strings.Go_Strings --pid 2795
Volatility 3 Framework 2.27.1
Struct Addr     Data Addr         Length      Value                                                    Struct Location   Data Location
0xc000200180    0xc00023a068      19          "192.168.20.123:1337"                                    heap              heap
0xc000182880    0xc0000141e0      16          "5112caedfb3a4a17"                                       heap              heap
0xc0000b6158    0xc0000f804b      7           "/getcmd"                                                heap              heap
0xc0001a0c98    0xc0001b637b      10          "/cmdoutput"                                             heap              heap
0xc00021e1e8    0xc00023a07b      8           "/sysinfo"                                               heap              heap
0xc0001804b0    0xc000194590      6           "whoami"                                                 heap              heap
0xc0001806b0    0xc00019463c      2           "id"                                                     heap              heap
0xc000180840    0xc000194740      3           "who"                                                    heap              heap
0xc000192870    0xc0001a5200      15          "cat /etc/shadow"                                        heap              heap
0xc00018d6e0    0xc0000d9400      39          "uid=0(root) gid=0(root) groups=0(root)"                 heap              heap
0xc00007d9e0    0xc0000d8000      5           "root\n"                                                 heap              heap
0xc00018dad0    0xc000229400      298         "x        pts/1        2025-12-27 17:49 (192.168.20.42)..." heap          heap
0xc00018dec0    0xc000220800      748         "root:$y$j9T$FLXb8uv9M83ZY/ROMEAL21$AdX.RJtCqn52jn..."    heap              heap
                                        [Snip]
0xc0001876e0    0xc000298000      4096        "HTTP/1.1 200 OK ... Successfully got system information" heap              heap
0xc000272660    0xc000278000      4096        "HTTP/1.1 200 OK ... Successfully posted output"         heap              heap
0xc000198080    0xc0001c8000      164         "{"os":"linux","arch":"amd64","name":"rkclient",...}"    heap              heap
```

Figure 7: Runtime forensic artifacts recovered from Pantegana memory using `go_strings`

plugins and evaluated them against real-world malware, such as *BRICKSTORM*, *Obscura*, and *Pantegana*. The results demonstrated recovery of critical forensic artifacts, such as C2 configurations, cryptographic keys, persistence mechanisms, and threat actor communication channels, including indicators and behaviours not reported in published threat intelligence.

References

Ali, H., Case, A., Ahmed, I., 2025a. Leveraging memory forensics to investigate and detect illegal 3d printing activities. Forensic Science International: Digital Investigation 53, 301925. doi:10.1016/j.fsidi.2025.301925.

Ali, H., Case, A., Ahmed, I., 2025b. Memory analysis of the python runtime environment. Forensic Science International: Digital Investigation 53, 301920. doi:10.1016/j.fsidi.2025.301920.

Ali-Gombe, A., Sudhakaran, S., Case, A., Richard III, G.G., 2019. {DroidScraper}: A tool for android {In-Memory} object recovery and reconstruction, in: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp. 547–559.

Ali-Gombe, A., Tambaoan, A., Gurfolino, A., Richard III, G.G., 2020. App-agnostic post-execution semantic analysis of android in-memory forensics artifacts, in: Proceedings of the 36th Annual Computer Security Applications Conference, pp. 28–41.

Calvet, J., 2019. Analyzing Golang Executables. https://www.pnfsoftware.com/blog/analyzing-golang-executables/. Accessed: 2026-01-16.

Capstone Engine Team, 2024. Capstone: The ultimate disassembly framework. https://github.com/capstone-engine/capstone. Accessed: 2026-01-16.

Carvey, H., O'Donnell-Welch, L., Schmidt, A., Pham, A., 2025. Obscura, an obscure new ransomware variant. https://www.huntress.com/blog/obscura-ransomware-variant. Accessed: 2026-01-16.

Case, A., Richard III, G.G., 2016. Detecting objective-c malware through memory forensics. Digital Investigation 18, S3–S10.

Case, A., Richard III, G.G., 2017. Memory forensics: The path forward. Digital investigation 20, 23–33.

Coveware by Veeam, 2025. Obscura ransomware: A case study in ransomware data loss. https://www.coveware.com/blog/2025/11/18/obscura-ransomware-data-loss-validation. Accessed: 2026-01-16.

Cybersecurity and Infrastructure Security Agency (CISA), National Security Agency (NSA), Canadian Centre for Cyber Security, 2025. BRICKSTORM Backdoor: Malware Analysis Report AR25-338A. https://www.cisa.gov/news-events/analysis-reports/ar25-338a. Accessed: 2026-01-16.

d4rksystem, 2025. Go Big or Go Home (and Other Terrible Go Puns): Tips for Analyzing GoLang Malware. https://securityliterate.com/go-big-or-go-home-and-other-terrible-go-puns-tips-for-analyzing-golang-malware/. Accessed: 2026-01-15.

Grunzweig, J., 2019. The Gopher in the Room: Analysis of Golang Malware in the Wild. https://unit42.paloaltonetworks.com/the-gopher-in-the-room-analysis-of-golang-malware-in-the-wild/. Accessed: 2026-01-15.

Guerrero-Saade, J.A., 2021. AlphaGolang: A Step-by-Step Go Malware Reversing Methodology for IDA Pro. https://www.sentinelone.com/labs/alphagolang-a-step-by-step-go-malware-reversing-methodology-for-ida-pro/. Accessed: 2026-01-15.

Hex-Rays, 2025. Stop guessing and start going: Cleaner golang decompilation in ida 9.2. https://hex-rays.com/blog/stop-guessing-and-start-going. Accessed: 2026-01-16.

Hunt.io, 2025a. Gh0st and pantegana: Two rats that refuse to fade away. https://hunt.io/blog/gh0st-and-pantegana-two-rats-that-refuse-to-fade-away. Accessed: 2026-01-16.

Hunt.io, 2025b. Pantegana rat malware family. https://hunt.io/malware-families/pantegana-rat. Accessed: 2026-01-16.

InstaTunnel Team, 2025. Rust and go malware: Cross-platform threats evading traditional defenses. https://medium.com/@instatunnel/rust-and-go-malware-cross-platform-threats-evading-traditional-defenses-d7fddf127d32. Accessed: 2026-01-15.

Mandiant, 2025a. Another brickstorm: Stealthy backdoor enabling espionage into tech and legal sectors. https://cloud.google.com/blog/topics/threat-intelligence/brickstorm-espionage-campaign. Accessed: 2026-01-16.

Mandiant, 2025b. Gostringungarbler: Tools for recovering strings from obfuscated go binaries. https://github.com/mandiant/gostringungarbler. Accessed: 2026-01-16.

Mandiant, 2026. Goresym: Go symbol recovery tool. https://github.com/mandiant/GoReSym. Accessed: 2026-01-16.

Manna, M., Case, A., Ali-Gombe, A., Richard III, G.G., 2021. Modern macos userland runtime analysis. Forensic Science International: Digital Investigation 38, 301221.

Manna, M., Case, A., Ali-Gombe, A., Richard III, G.G., 2022. Memory analysis of. net and. net core applications. Forensic Science International: Digital Investigation 42, 301404.

Martí, D., 2024. Garble: Changelog (v0.15.0). https://github.com/burrowers/garble/blob/6dab979/CHANGELOG.md. Accessed: 2026-01-16.

Maurya, A., 2021. Golang malware is more than a fad: Financial motivation drives adoption. https://www.crowdstrike.com/en-us/blog/financial-motivation-drives-golang-malware-adoption/. Accessed: 2026-01-15.

Meskauskas, T., 2025. How to remove obscura (.obscura) ransomware. https://www.pcrisk.com/removal-guides/33788-obscura-ransomware. Accessed: 2026-01-16.

omaidf, 2025. Go-malware: Golang malware examples. https://github.com/omaidf/go-malware/tree/master. Accessed: 2026-01-16.

Raimbaud, K., Rascagneres, P., 2025. GoResolver: Using Control-flow Graph Similarity to Deobfuscate Golang Binaries, Automatically. https:

//www.volexity.com/blog/2025/04/01/goresolver-using-control-flow-graph-similarity-to-deobfuscate-golang-binaries-automatically/. Accessed: 2026-01-15.

Smmarwar, S.K., Gupta, G.P., Kumar, S., 2024. Android malware detection and identification frameworks by leveraging the machine and deep learning techniques: A comprehensive review. Telematics and Informatics Reports , 100130.

Stephen Eckels, . Ready, Set, Go — Golang Internals and Symbol Recovery. https://cloud.google.com/blog/topics/threat-intelligence/golang-internals-symbol-recovery/. 2022, Accessed: 2026-01-11.

Stevens, E., 2025. BitSight Threat Intelligence Briefing: Key Malware Trends Shaping Cyber Risk in 2025. https://www.bitsight.com/blog/current-malware-trends-2025. Accessed: 2026-01-15.

Sylve, J., Case, A., Marziale, L., Richard, G.G., 2012. Acquisition and analysis of volatile memory from android devices. Digital Investigation 8, 175–184.

Tam, K., Edwards, N., Cavallaro, L., 2015. Detecting android malware using memory image forensics, in: Engineering Secure Software and Systems (ESSoS) Doctoral Symposium.

The Go Authors, 2024. Go Internal ABI Specification (ABIInternal). https://go.dev/src/cmd/compile/abi-internal. Accessed: 2026-01-18.

Trellix, 2023. Feeding gophers to ghidra. https://www.trellix.com/blogs/research/feeding-gophers-to-ghidra/. Accessed: 2026-01-16.

Volatility Foundation, 2025. Volatility 3: The volatile memory extraction framework. https://github.com/volatilityfoundation/volatility3. Accessed: 2026-01-16.

Volexity, 2025. Goresolver: Context-aware recovery of go malware semantics. https://github.com/volexity/GoResolver. Accessed: 2026-01-16.

Wang, E., Zurowski, S., Duffy, O., Thomas, T., Baggili, I., 2022. Juicing v8: A primary account for the memory forensics of the v8 javascript engine. Forensic Science International: Digital Investigation 42, 301400.

Wilhoit, K., 2025. Bookworm to stately taurus using the unit 42 attribution framework. url = https://unit42.paloaltonetworks.com/bookworm-stately-taurus-attribution-framework/. Accessed: 2026-01-21.