Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.34, NO.4, Aug. 2024 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2024.34.4.707

GoAsap: 정적분석 관점에서 바라보는 Golang 신버전 탐지·분석시스템 제안*

강 형 민,[1†] 원 유 재[2‡]
[1,2]충남대학교 (대학원생, 교수)

GoAsap: A Proposal for a Golang New Version Detection and Analysis System from a Static Analysis Perspective*

Hyeongmin Kang,[1†] Yoojae Won[2‡]
[1,2]Chungnam National University (Graduate student, Professor)

요 약

최근 Golang은 크로스 컴파일이 가능하고 코드 생산성이 높다는 특성으로 프로그래밍 언어 점유율 순위가 매년 지속적 상승하고 있다. 하지만 최근 악성코드 개발자들 또한 랜섬웨어, 백도어 등 악성코드 배포에 자주 활용하고 있다. 특이한 점으로 오픈소스 언어인 Golang은 새로운 버전이 출시될 때, 삭제된 심볼 복구에 필요한 중요한 값들이 포함된 Pclntab이라는 구조체의 값과 구성순서가 상시적으로 변경되고 있다. 개발자 측면에서는 코드 가독성 및 생산성을 위해 구조를 자주 변경하는 것이 문제는 아니나, 사이버보안 측면에서는 구조가 변경된 새로운 버전이 악성코드에 활용될 수 있는 문제점이 존재한다. 따라서 본 논문에서는 Golang 신버전 대상 실행파일 탐지·분석시스템(GoAsap)를 제안하고 기존 바이너리 분석 도구 6종과 비교·평가하여 제안한 시스템의 성능을 검증하였다.

ABSTRACT

Recently, Golang has been gaining attention in programming language rankings each year due to its cross-compilation capabilities and high code productivity. However, malware developers have also been increasingly using it to distribute malware such as ransomware and backdoors. Interestingly, Golang, being an open-source language, frequently changes the important values and configuration order of a crucial structure called Pclntab, which includes essential values for recovering deleted symbols whenever a new version is released. While frequent structural changes may not be an issue from a developer's perspective aiming for better code readability and productivity, it poses challenges in cybersecurity, as new versions with modified structures can be exploited in malware development. Therefore, this paper proposes GoAsap, a detection and analysis system for Golang executables targeting the new versions, and validates the performance of the proposed system by comparing and evaluating it against six existing binary analysis tools.

Keywords: Golang, Static Detection, Static Analysis, Malware

I. 서 론

Golang은 구글에서 2007년 설계·개발을 시작하여 2012년 버전 1.0이 공개된 오픈소스 프로그래밍 언어이다[1]. Golang이 다른 언어와 비교하여 가지는 특징으로는 △정적 링크 타입 △

Received(06. 26. 2024), Modified(07. 22. 2024), Accepted(07. 23. 2024)
* 이 논문은 2023년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(IITP-2022-0-01200)
† 주저자, hmkang@cnu.ac.kr
† 교신저자, yjwon@cnu.ac.kr(Corresponding author)

CSP(Communicating Sequential Processes) 기반의 Concurrent(동시성 프로그래밍) △Garbage Collection 기능이 있다. Golang은 수년 동안 프로그래밍 언어 점유율이 50개 중 상위권에 소속이 될 정도로 최근 시스템 개발자들 사이에서 큰 인기를 얻고 있다[2]. 또한, Golang은 라이브러리가 풍부하고 크로스 컴파일이 가능해 윈도우(x86/x86-64), 리눅스(ELF), 맥(Mach-O) 등 이기종 환경에서 쉽게 개발이 가능한 장점이 있다.

반면, Golang을 악성코드 개발자의 시각에서 바라보면 △크로스 컴파일을 통한 다형성 악성코드 제작 △오픈소스 라이브러리인 OSS(Open Source Software) 변조 등 Golang에서 제공하는 편리기능을 악용할 수 있다. 특히, 최근 Golang 악성코드 제작자들은 악성 행위가 포함된 OSS를 사전에 깃허브에 업로드 후 다운로드 받는 방식으로 악용하고 있다[3][4][5][6]. 또한, 악성코드 백도어 통신에 자주 사용되는 KCP 프로토콜이 처음으로 22년도에 Golang 라이브러리 배포 및 악성코드에서 사용됐다[7][8][9]. 즉, 악성기능을 하는 다수의 OSS를 다운받아 커스텀 기능을 추가・조합한다면 악성코드 개발자의 생산성이 크게 높아질 수 있는 상황이다.

Pclntab(Program Counter Line Table)은 가상 메모리 주소를 심볼 이름으로 매핑하여 스택 추적을 용이하게 하는 목적으로 만들어졌으며, 심볼 복구를 위한 △함수 이름 및 코드 위치 오프셋 △함수 시작 및 종료 주소 등 메타데이터들을 저장하기 때문에 정적분석시 중요한 테이블이다. Golang은 특정 버전별로 고유 헤더와 주요 정보들이 포함된 Pclntab이 변동하는 경우가 있다. 즉, 새로운 버전이 출시되면 고유정보들이 변경돼 알려진 Golang의 특징을 기반으로 탐색하는 기존 분석도구들은 새로운 버전에 대한 실행파일을 분석하지 못 할 가능성이 크다. 만약 악성코드 제작자가 Golang 배포부터 분석도구 수정까지의 시간 사이에 새로운 버전에 대한 악성코드를 제작한다면 기존 분석도구에서는 제대로 대응하지 못해 선제적으로 탐지・분석 등 대비할 수 있는 프로세스가 필요하다.

이에, 본 논문에서는 새로운 Golang 버전이 배포됐을 때 악성코드에 적용되기 전 이를 선제적으로 탐지・수집하고 정적분석할 수 있는 시스템(GoAsap)을 제안한다. 현재 Golang 24년 8월경에는 최신 버전(1.23)을 발표할 것으로 보이지만, 현재 최신 버전은 1.22로, 본 논문에서 언급하는 새로운 버전은 1.22 이후 버전을 의미한다. 주요 기능으로는 △(신규버전 탐지)실시간 Golang 실행파일 모니터링 △주요정보가 포함된 Pclntab 탐색(4가지 방안) △삭제된 심볼 복구 △문자열 난독화(gobfuscate) 탐지・해제 등을 수행한다.

또한, 본 논문에서 제안하는 시스템의 성능을 평가하기 위해 Golang 정적분석에 자주 사용되는 도구 6개를 선정했으며, 평가에 사용된 샘플(기존 버전・변조한 새로운 버전)을 대상으로 △Pclntab 버전 및 위치(오프셋) 탐지 △삭제된 심볼복구 △문자열 난독화 해제 관점에서 평가를 진행했다. 그 결과, 알려진 Golang 버전에 대해서 기존 도구 2개를 제외하고는 Pclntab 탐지 및 심볼 복구가 가능했으나 새로운 Golang 버전은 모든 도구에서 탐지가 불가했다.

제안하는 시스템은 기존 Golang 정적분석 도구들이 고유 헤더 정보로만 수동 탐색하는 방안에 비해 능동적 탐색을 통해 찾기 때문에 새로운 Golang 버전에 대한 Pclntab의 정확한 값과 위치를 찾을 수 있어 기존 분석도구 대비 성능이 뛰어났다.

본 논문의 하위 내용은 다음과 같이 구성되어 있다. 2장에서는 Golang 실행파일 탐지・분석 관련 연구를 알아보고, 3장에서는 Golang 파일의 구조 및 특징 분석을 다룬다. 4장에서는 Golang 신버전 탐지・분석 시스템을 제안하고 5장에서는 제안 방법에 대한 실험 결과를 기술한다. 6장에서는 제안 방법의 한계점 및 향후 계획을 설명하고, 마지막 7장에서는 본 연구의 결론을 끝으로 논문을 마친다.

II. 관련 연구

2.1 Golang 실행파일 분석 관련 연구 동향

L. Fróes는 Golang 파일을 효율적으로 분석하기 위한 동적분석 도구를 개발했다[10]. 다만, 탐지율 향상보다는 분석가의 동적분석 효율성을 높이기 위해 API 후킹 기반 사용함수 추적을 했으나 설정하지 않은 API는 확인하지 못해 커버리지가 높지 않았다.

R. M. Yasir 등은 Golang에서 사용된 God Class를 탐지하는 도구(GodExpo)를 최초로 제안했다[11]. 해당 도구를 이용해 모든 버전에서 Weighted Method Count, Tight Class 등 외부 데이터 접근과 같은 항목을 수집・측정하여 God

구조를 성공적으로 발견했다.

J. Lauinger 등은 Github에서 가장 인기있는 상위 500개 오픈소스 Golang 프로젝트 및 관련 의존성 코드(내·외부 라이브러리)를 대상으로 unsafe 패키지 내 메모리 안전성을 위협하는 모듈(7종) 탐지도구(go-geiger)를 제안했다[12][13].

A. Engelke는 Golang 1.7버전 실행프로그램을 대상으로 어셈블리 기반의 하위 중간언어인 HGA(Higher-level Go Assembly)를 사용, 코드 타입 및 함수 인수를 간결화 하는 방안에 대해 제안했다[14].

M. Praveen 등은 Golang과 비슷한 시기에 발표된 Rust 프로그래밍 언어로 작성된 악성코드에 대한 연구가 부족하다고 판단했다. 이에, C언어와 Rust로 작성된 악성코드를 대상으로 안티바이러스 탐지율을 비교했으며, Rust 악성코드를 쉽게 분석할 수 있는 전용 프레임워크를 제안했다[15]. 저자는 Rust가 전통적인 시그니처 분석으로는 탐지가 어려워 △의존성 라이브러리 △난독화 코드 △MITRE ATT&CK 프레임워크와 샌드박스를 융합한 하이브리드 분석 등 차별화된 동적분석을 통해 탐지를 해야한다고 강조했다. Golang은 Rust와 마찬가지로 오픈소스 프로젝트로 개발자들에 의해 새로운 버전이 지속적으로 출시되고 있다. 기존 바이너리 분석 도구들 또한 신속하게 새로운 버전에 맞는 분석 기능을 추가해 패치하지만 다른 저수준 언어(C·C++) 대비 중요한 정보가 포함된 구조가 버전별로 크게 변경돼 신속한 대응이 어렵다. 이에, 새로운 Golang 버전이 악성코드로 배포되는 문제점을 사전에 예방하기 위해서는 새로운 버전을 감시·대비할 수 있는 체계가 필요하다.

2.2 바이너리 정적탐지 도구

2.2.1 YARA

YARA(Yet Another Recursive Acronym)는 악성코드 시그니처를 이용해 식별·분류 목적으로 사용하는 오픈소스 바이너리 탐지 도구이다[16]. 바이러스토탈[17]의 개발자인 Victor Alvarez에 의해 개발되었으며, 안티바이러스 제품, 분석 도구 등 다양한 보안 분야에서 활용되고 있다. 본 논문에서 제안하는 시스템에서는 Golang 실행파일을 수집하기 위해 관련 특징을 Yara 룰로 작성해 선별 수집한다.

2.3 바이너리 정적분석 도구

2.3.1 IDA Pro

IDA Pro는 1991년 커맨드라인 버전(0.1)부터 2024년도 GUI 버전(8.4)까지 다양한 CPU 아키텍처 분석을 위한 Hex-Rays Decompiler와 같은 유용한 바이너리 역분석 도구를 개발하고 있다[18]. 도구 자체적으로 2021년 릴리즈된 IDA 7.6버전부터 Golang 분석을 지원하기 시작했다. 또한, 사용자들의 IDC·IDAPython 플러그인을 통한 Golang 분석 스크립트도 깃허브에 공유되는 등 지속적인 연구가 진행되고 있다.

2.3.2 Ghidra

Ghidra는 미국 국가 안보국(NSA)에서 2019년 RSA 컨퍼런스에서 오픈소스 공개·발표된 바이너리 역분석 도구이다[19]. 이는 JAVA 프로그래밍 언어로 개발됐으며, 오픈소스 특성상 무료이고 도구 업데이트 등 관리가 꾸준하게 잘 되고 있다. 또한, IDA Pro에서 유료로 판매중인 Hex-Rays Decompiler와 유사한 기능을 무료로 사용할 수 있어 사용자들의 인기가 높다. 다만, Ghidra는 IDA Pro와 다르게 도구 자체적인 Golang 분석 기능이 없어 사용자들이 만든 플러그인을 통해 구조 분석을 수행할 수 있다.

2.3.3 Radare2

Radare2는 2014년 오픈소스로 공개된 바이너리 역분석 도구이다[20]. 이 도구는 다양한 바이너리 분석도구들과 연동하여 분석할 수 있는 플러그인이 많은 것이 특징이다. 또한, 윈도우·리눅스별 GUI·CLI 버전을 모두 지원하지만, 리눅스 환경에서 과거 많이 사용했던 GDB 대신 최근 주로 사용된다. 다만, Ghidra와 마찬가지로 도구 자체적인 Golang 분석 기능이 없어 사용자들이 만든 플러그인을 통해 구조 분석을 수행할 수 있다.

2.4 Golang 구조 탐지·분석 도구

D. Palotay 등은 Golang에서 사용하는 난독화

를 해제하는 방안을 제안하였다[21]. 실행파일 분석 도구 Ghidra 스크립트를 사용해 난독화한 문자열을 복구하지만 특정 버전에 한해서만 분석을 진행했다. S. Eckels는 Golang의 Pclntab 구조를 탐색해 Symbol을 복구하는 분석도구를 제안했다[22]. 제안된 도구는 Pclntab 영역을 찾기 위해 섹션이름 탐색 및 버전별 매직헤더 바이트스캔 방법을 사용했다. 하지만, 새로운 버전 출현시 고유값이 변경된다면 미탐이 발생 할 가능성이 있어 한계가 있었다.

2.4.1 GoReSym

Google의 자회사인 미국 사이버 보안회사 Mandiant가 개발한 Golang 모든 버전 내 삭제된 심볼을 복구하는 도구이다[23]. Golang으로 개발됐으며, 주 기능으로는 프로그램·함수 메타데이터 및 내장된 구조·타입 추출 기능이 있다. 또한, IDA Pro와 Ghidra 도구에 연동하여 추가 분석을 수행할 수 있는 강점이 있다.

2.4.2 IDAGolangHelper

IDA Pro에서 제공하는 IDAPython을 이용해 개발됐으며, Golang 1.2~1.17 버전 구조 파싱 및 심볼 복구 기능을 수행하는 플러그인이다[24]. 현재 최신 버전에 대해서는 기능이 정상적으로 실행되지 않는다.

2.4.3 redress

Golang 구조 파싱 및 심볼 복구 기능을 수행하는 도구이다[25]. 특징으로는 Radare2에서 제공하는 r2 플러그인을 이용해 디버깅 연동 및 분석할 수 있다. 또한, Golang 소스코드가 포함된 프로젝트 폴더경로를 파라미터로 전달하면 세밀한 정적분석을 수행하는 장점이 있다.

2.3, 2.4장에서 소개한 기존 도구들은 알려진 Golang 버전별 실행파일 구조 분석 및 심볼 복구에 대한 연구를 진행했다. 하지만, 유지보수 단절 또는 새로운 버전에 대한 대응이 없어 최신 Golang 버전은 분석이 불가능했다.

본 논문에서는 기존 분석도구에서 새로운 Golang 버전에 해당하는 Pclntab 미탐지, 추가 분석하지 못하는 단점을 보완하기 위해 Golang 실행파일을 선제적 탐지·분석하는 실용화된 프레임워크를 제안한다.

III. Golang 실행파일 상세 분석

3.1 Golang 구조 및 PCLNTAB 소개

Golang은 2012년 1.0 버전을 시작으로 2024년 2월 1.22 버전까지 빠른속도로 신규기능이 개발되고 있다. Golang에는 메타데이터, 변수 타입 등 중요한 항목들이 포함된 Pclntab 구조체가 있다[26]. 1.2 버전 이전에는 runtime.pclntab 데이터 구조가 압축된 상태로 저장됐으나, 이후 버전에서는 압축되지 않은 상태로 저장하여 실행파일의 크기가 커지게 되었다. Pclntab은 가상 메모리 주소를 가장 근접한 Symbol Name으로 매핑하여 함수 및 파일 이름이 포함된 스택을 트레이싱 하는 데 주로 사용되며, 가비지 컬렉터에서도 주로 참고하는 중요한 테이블이다. 이를 활용하면 구조체 내부 특정 값을 참조해 심볼이 삭제된 경우에도 복구할 수 있는 강점이 있다.

Moduledata 구조는 파일 레이아웃에 대한 정보와 가비지 수집 및 리플렉션과 같은 핵심 기능을 지원하는 데 사용되는 런타임 정보를 저장하는 내부 런타임 테이블이다. Moduledata 배열의 주요 항목으로는 런타임 심볼 정보인 Pclntab의 포인터와 리플렉션 및 가비지 컬렉터에 주로 사용되는 타입링크 리스트, 타입인포 등 다양한 항목들이 존재한다.

Pclntab 내 테이블 버전 식별 지표인 Magic Header는 △1.1 버전(0xfffffffd) △1.2~1.15(0xfffffffb) △1.16~1.17(0xfffffffa) △1.18~1.19(0xfffffff0) △1.20~1.22(현재 최신 버전)이 있으며, 각 Go 버전별로 차이가 있는 것을 확인 할 수 있다. 이는 Table 1.에서 보는 바와 같이 Golang 깃허브 내 symtab 파일에 Pclntab 코드·구조체가 저장돼있다.

Table 1. Information on Golang Version Values

Golang Version	Magic Header Value of Pclntab
1.1	0xfffffffd
1.2~1.15	0xfffffffb
1.16~1.17	0xfffffffa
1.18~1.19	0xfffffff0
1.20~1.22(new version)	0xfffffff1

Fig. 1. PclnTab Structure in Golang

Fig. 1.은 바이너리 편집도구를 통해 확인한 Golang 1.22 버전(64비트) PclnTab 구조이다. 테이블 구조 내 주요 항목으로는 테이블 시그니처 값(magic), 운영체제 비트 수를 나타내는 포인터 사이즈(ptrSize), 바이너리 내 저장된 함수 개수(nfunc), 코드섹션 주소(textStart), 헤더에서 함수 문자열 배열까지의 오프셋(funcnameOffset), 헤더에서 함수정보 관련 링크가 포함된 배열까지의 오프셋(pclnOffset) 등이 있다.

3.2 프로그래밍 언어 비교 분석(Go vs C++)

24년 美 백악관은 C와 C++와 같은 취약한 프로그래밍 언어의 사용을 중단하고 메모리 안전 프로그래밍 언어를 사용하라고 권고했다[27]. 또한, 22년 국가안보국(NSA)도 Golang, Rust를 사용해 메모리 보안을 강화하라는 권고를 내렸었다[28]. 역사적으로 오래되고 사용 분포도가 높은 C와 C++ 언어를 단기간에 다른 언어로 교체하는 것은 쉽지 않겠지만, 추후에는 Golang과 같은 메모리 안전 언어를 주로 사용할 것으로 추정된다. 본 장에서는 Golang과 C++ 언어를 비교하면서 각 특징을 살펴본다.

두 언어를 비교하면 주요 특징은 Table 2.와 같다. Golang은 가비지 컬렉터를 사용해 C++과 다르게 메모리 관리를 JAVA와 같이 사용할 수 있다. 또한, Golang은 리턴값을 2개 이상 사용해 데이터와 오류값을 반환하여 효율적인 프로그래밍을 할 수 있으나, 역공학 관점에서 보면 수십 개의 리턴 값을 반환하는 경우 분석이 난해할 수 있다.

또한, 컴파일 시 Golang은 정적링킹 방식을 통해 필수 라이브러리를 실행파일 내 모두 포함시킨다. 장점으로는 모듈 방식의 의존성 관리를 하기 때문에 C++보다 의존성 이슈가 없는 편이다. 단점으로는 기능이 많이 포함될수록 컴파일된 실행파일의 크기가 지속적으로 커진다. 관련 내용을 각 언어로 덧셈 기능을 하는 함수를 작성해 비교분석을 수행한 결과, Fig. 2.에서 보는 바와 같이 정적링킹을 사용하는 Golang(1,503)이 C++(331)에 비해 함수 개수가 상당한 것으로 확인됐다. 또한, Fig. 3.과 같이 Golang은 C++과 달리 문자열이 null(0x00)로 끝나지 않고 Space(0x20)으로 구분을 하는 차이점도 있었다.

Table 2. Comparison of Key Features Between Golang and C++

	Golang	C++
Pointer	O	O
Number of Return Values	2	1
Class	X	O
Garbage Collection	O	X
Inheritance	X	O
Overloading	X	O
Linking	static	static, dynamic

Golang은 윈도우, 리눅스 64비트에서 컴파일된 바이너리와 다른 호출 규약(Calling Convention)을 가지고 있다. 윈도우는 5개의 레지스터(RCX, RDX, R8, R9, 스택), 리눅스는 6개의 레지스터(RDI, RSI, RDX, RCX, R8, R9)를 사용하고

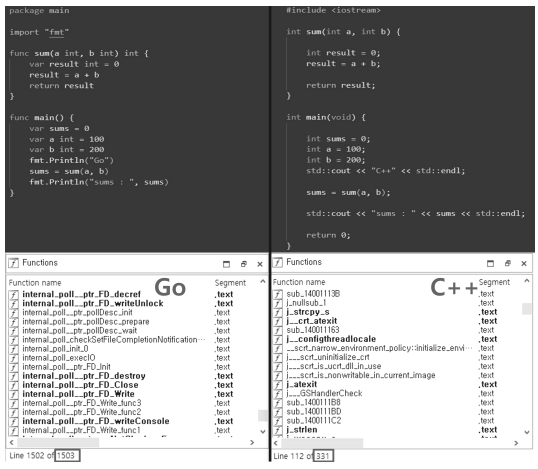

Fig. 2. Differences Between Functions in Golang and C++

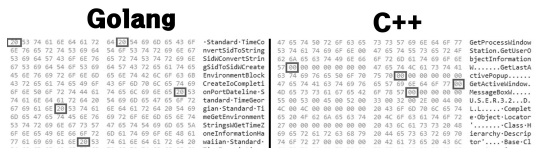

Fig. 3. Comparison of String Handling in Golang and C++

파라미터가 7개 이상일 경우 스택을 활용하며, 함수 리턴 값 전달은 RAX 레지스터를 사용한다.

1.17버전 이상인 Golang 실행파일은 파라미터와 리턴 값 전달에 9개의 레지스터(RAX, RBX, RCX, RDI, RSI, R8, R9, R10, R11)를 활용하고, Go(1.1~1.16 버전)에서는 윈도우 32비트와 같이 스택을 활용해 파라미터와 리턴 값을 전달한다.

3.3 심볼 복구를 위한 아키텍처별 Pclntab 탐색

Golang은 다른 언어들과 마찬가지로 main 함수부터 시작된다. 차이점으로는 패키지 형태로 코드를 모듈화 하고 있어 프로그램의 시작점인 엔트리포인트가 main 패키지 내부에 있는 main() 함수부터 시작된다. 즉, 심볼이 제거되지 않았다면 main.main() 함수를 탐색해 해당 지점부터 분석을 수행하면 된다. 다만, 심볼이 제거됐거나 문자열 난독화로 인해 main 함수가 보이지 않는다면 정적·동적 분석 및 자동화 스크립트를 통해 Golang에서 가장 중요한 정보가 담겨 있는 구조체인 Pclntab 탐색 후 심볼을 복구하는 것이 우선이다.

Table 3.에서 보는 바와 같이 Pclntab을 찾기 위한 방법으로는 파일형식에 따라 다르다. ELF, Mach-O 파일형식의 경우에는 각각 'gopclntab'과 '__gopclntab'섹션 내부에 테이블을 저장한다. 이 경우에는 섹션명을 탐색하고 주소를 알아내면 Pclntab 영역의 주소를 쉽게 알 수 있다. PE 파일의 경우에는 특정 섹션에 테이블이 저장된 구조가 아니라 여러 경로를 참조해 주소를 알아내야 한다.

Table 3. Searching Pclntab Across Architectures

	Symbol Exists	Symbol Deleted
PE	'.symtab' Searching	Pclntab Feature-Based Binary Search
ELF	'gopclntab' Searching	
Mach-O	'__gopclntab' Searching	

첫 번째 방법으로는 PE 구조 내 FileHeader.PointerToSymbolTable 필드의 값으로 .symtab 영역을 탐색하고 runtime.pclntab 필드를 찾아내야 한다.

두 번째로는 ModuleData 항목에서 runtime.pclntab 주소를 찾는 방법이다.

위 두 가지 방법을 통해 PE파일 내 Pclntab 구조체를 찾는 데 성공할 수도 있지만 한계점도 있다. 첫 번째 방법의 경우 심볼테이블이 제거된 PE파일에서는 .symtab 데이터가 삭제되어 찾을 수 없었고, 두 번째 방법의 경우에는 ModuleData 항목을 심볼이름으로 찾을 수 없으며, 버전이 변경되면서 항목이 지속적으로 변경돼 탐색에 큰 어려움이 있었다. 또한, 사이즈를 줄이기 위해 패커를 적용했다면 섹션명과 압축된 크기로 변경돼 시그니처 탐색은 미탐이 발생할 확률이 높다. 이에, 심볼이 제거되거나 헤더가 변조됐을 경우 시그니처 탐지를 통한 방식은 더 이상 완전한 방법이 아니다. 이를 해결하기 위해 본 논문에서는 능동적 탐지를 통해 Pclntab을 탐색하고 제거된 심볼을 복구하는 등 능동적 탐지·분석을 수행하는 시스템을 제안한다.

IV. Golang 신버전 탐지·분석 시스템 제안

4.1 제안 시스템 개요

본 장에서는 Golang 신버전 배포시 신속하게 대응하기 위한 목적으로 「Golang 신버전 탐지·분석 시스템」 GoAsap를 제안한다. Fig. 4.은 제안하는 시스템의 전체적인 과정을 설명한 것으로 크게 △탐지 △수집 △분석 △저장 총 4단계로 이루어진다.

첫 번째로, 바이러스토탈, MalwareDB 등 분석 서비스들을 다양하게 활용해 Golang 샘플을 탐지하기 위한 정교한 룰(Yara 작성)을 기반으로 탐지한다. 단, 수집은 위협분석 서비스 기반으로 API 통신 및 WEB Crawling 방식으로 진행, 라이선스 및 IP 계정 블록으로 한계가 있을 수 있어 시스템 내 수집 기능은 강제적이 아닌 선택사항으로 둔다.

두 번째로 분석 서비스에서 제공되는 API를 이용해 수집하거나 제공하지 않을시에는 웹 크롤링 기능을 통해 샘플을 수집한다. 또한, 새로운 버전 실행파일 수집을 위해 Golang 깃허브 저장소 내 패치 내

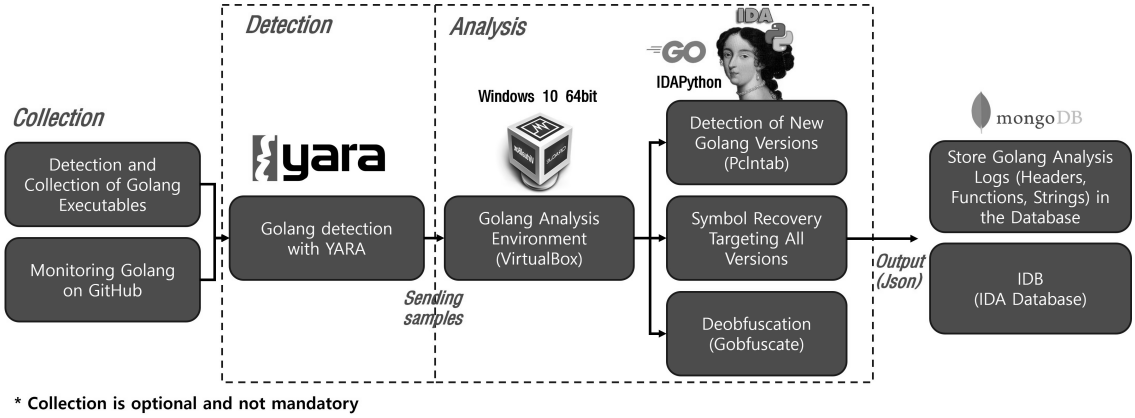

Fig. 4. GoAsap System Overview

역을 상시 모니터링한다.

세 번째로, 수집한 샘플을 자체 구축한 Golang 분석 가상환경(VirtualBox)에 전달하고 IDA에서 제공하는 스크립팅(Python 기반) 기능을 활용해 심볼복구, 난독화 탐지, 신버전 Pclntab 구조체 탐색 등 Golang 분석을 진행한다.

마지막으로는 위 과정에서 분석·저장된 Json 형식의 데이터를 Host로 전송하여 데이터베이스(MongoDB)에 저장 및 분석했던 IDB(IDA Database)도 저장·추출해 분석가의 편의를 돕는다.

4.2부터는 위 기술한 4가지 단계의 상세내용을 설명한다.

4.2 Golang 실행파일 수집 및 탐지

4.2.1 Golang 실행파일 선별 탐지 및 수집

악성코드를 수집할 수 있는 방법으로는 백신회사의 경우 엔드포인트 환경에 설치된 각 사용자들의 PC 내 의심파일을 수집한다. 개인의 경우에는 유/무료 위협분석 서비스를 활용, 악성코드 DB에서 파일 해시를 검색해 파일을 수집할 수 있다. 특히, 유료버전의 경우에는 바이너리 스캔 도구인 Yara 룰 및 다양한 키워드(백신탐지수, 파일명, 컴파일언어 등)을 통해 선별 수집이 가능한 장점이 있다.

제안하는 시스템에서는 3개 서비스(VirusTotal, MalShare, VirusShare) 대상으로 Golang 실행파일을 수집하는 크롤링 모듈을 개발했으며, 스케줄링 형태로 동작(시간설정 필요) 및 직접 샘플을 업로드하도록 선택사항을 두었다. VirusTotal의 경우 Golang 실행파일만 수집하기 위해 Yara룰을 등록한 상태에서 수집하는 절차가 필요하다. Table 4.는 Golang으로 컴파일된 peexe, elf 파일형식만 필터링하는 탐지규칙이며, 이를 통해 바이너리, 해시, 파일정보, 백신탐지명, 관련 OSINT 내용 등이 수집된다.

Table 4. The Rules for Detecting Golang on VirusTotal

detectiteasy:"**Compiler: Go**" and **positives: {Number of detections by antivirus}**+ and (**type:peexe or type:elf**)

4.2.2 Golang 산출물 감시

Golang은 오픈소스 프로젝트로 관리되는 언어로 분산 버전 관리 도구인 깃허브(Github)에서 관리되고 있다. Golang의 특정 코드 변경 및 신버전이 출시될 경우 배포 버전별로 코드가 따로 관리된다. 이에, 새로운 버전을 탐지하기 위해 공식 Golang 깃허브 저장소를 모니터링하고 관련 실행파일을 수집하는 크롤링 모듈을 개발했다.

Pclntab 구조체 코드가 존재하는 symtab 파일의 절대경로를 깃허브에서 버전별로 확인한 결과, Table 5.와 같이 파일명과 버전명을 제외하고는 유사했다. 이를 토대로 새로운 버전에 해당하는 URI를 기댓값으로 설정하고 상시 모니터링을 수행한다. 새로운 버전이 배포됐다면, symtab.go 파일을 다운로드 후 Pclntab의 Magic Header 및 나머지 구성요소들이 변경됐는지 레거시 버전들과 비교·분석을 통해 업데이트 내역을 확인한다. 마지막으로,

Table 5. Golang Source Code URI and Function Name with Pclntab for each version

Version	URI	Function Name
1.0	github.com/golang/go/blob/release-branch.go1/src/pkg/runtime/symtab.c	walksymtab
1.1	github.com/golang/go/blob/release-branch.go1.1/src/pkg/runtime/symtab.c	
1.2	github.com/golang/go/blob/release-branch.go1.2/src/pkg/runtime/symtab.c	runtime.symtabinit
1.3	github.com/golang/go/blob/release-branch.go1.3/src/pkg/runtime/symtab.goc	
1.4~1.15	https://github.com/golang/go/blob/release-branch.go1.{4-15}/src/runtime/symtab.go	moduledataverify1
1.16~1.17	github.com/golang/go/blob/release-branch.go1.{16-17}/src/runtime/symtab.go * Code changed from functions to structures starting from 1.16	pcHeader
1.18~1.22	github.com/golang/go/blob/release-branch.go1.{18-22}/src/runtime/symtab.go * Added pcHeader.textStart field starting from 1.18	
N.N	github.com/golang/go/blob/release-branch.go{N}.{N}/src/runtime/symtab.go	Expected value

4.3.5에서 제안한 신버전 탐지와 결과 값을 비교하여 최종 검증을 통해 판별한다.

4.3 Golang 실행파일 자동 분석기능 개발

본 장에서는 4.1에서 Golang 샘플이 수집된 후 사전 구축한 분석환경에 샘플 자동 전송 및 분석 관련 상세내용을 기술한다. 특히, 알려진 버전에 대한 △삭제된 심볼 복구 △문자열 난독화 탐지·해제 △분석시간 감소를 위한 표준·사용자 정의 함수 분류에 대해 설명한다. 마지막으로는 새로운 Golang 버전이 배포됐을 때 탐지·분석할 수 있는 시스템의 핵심 기능을 소개한다.

4.3.1 실행파일 분석을 위한 분석환경 구축

본 논문에서 제안하는 시스템은 자동화 분석 기반으로, 수집된 Golang 실행파일을 자동 분석하기 위해 VirtualBox[29] 가상환경에 샘플을 전송한다. 가상환경에서 실행파일을 분석하는 이유는 미탐(False-Negative)을 유발하는 Golang 난독화 코드 변종에 대비하기 위해 동적탐지가 필요하기 때문이다. 해당 환경에서는 에이전트를 통해 샘플 상세분석을 시작한다.

4.3.2 삭제된 심볼 복구

Golang 1.1~1.22 버전에서 심볼이 삭제된 경우 모두 복구할 수 있는 기능을 개발했다.

본 장에서는 최신버전인 1.22 실행파일을 대상으로 Fig. 5.에서 소개한 Pclntab 구조 도식화를 기반해 심볼이 저장된 함수들이 위치한 위치를 탐색하고 삭제된 심볼을 복구한다. 복구를 위해서는 크게 함수의 이름이 순차적으로 저장된 위치(A), 함수가 실제 구현된 코드영역 시작주소(B) 2가지가 필요하다. 각 버전별로 Pclntab 내 구성요소 순서 및 항목 추가·삭제 등 변동사항이 있어 탐색방법은 미세하게 차이가 있지만 전체적인 탐색 메커니즘은 동일하다.

심볼 복구에 필요한 영역들을 찾기 위한 전제조건으로 총 3개의 구조체(pcHeader, functab, _func)의 시작 포인터 주소가 필요하다. pcHeader 구조체는 Table 1.에서 소개한 각 버전별 Pclntab.Magic 헤더를 시그니처 탐색을 진행해 알아낼 수 있으며, Fig. 6.과 같이 0xFFFFFFF1로 1.20~1.22 버전 중 하나인 것으로 확인 가능하다.

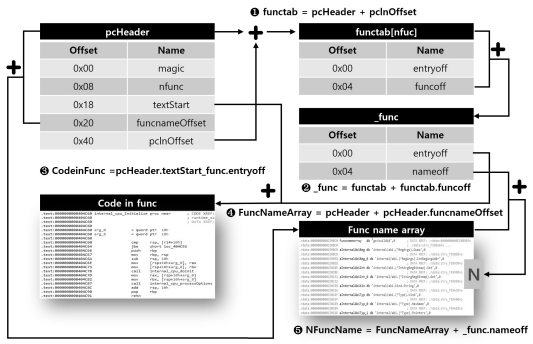

Fig. 5. Pclntab Structure Diagram

PcHeader Start Offset

```
F1 FF FF FF            dword_548840    dd 0FFFFFFF1h
00                                     db    0
00                                     db    0
01                                     db    1
08                                     db    8
FA 08 00 00 00 00 00 00                dq 8FAh
1A 01 00 00 00 00 00 00                dq 11Ah
00 10 40 00 00 00 00 00                dq offset sub_401000
60 00 00 00 00 00 00 00                dq 60h
80 32 01 00 00 00 00 00                dq 13280h
C0 3F 01 00 00 00 00 00                dq 13FC0h
E0 56 01 00 00 00 00 00                dq 156E0h
C0 D0 05 00 00 00 00 00                dq 5D0C0h
```

Fig. 6. PcHeader Start Offset

pcHeader 구조체 시작 포인터 주소를 알아냈다면, 내부 포함된 오프셋 항목들을 참고해 functab, _func 구조체의 위치를 찾을 수 있으며, 이를 기반으로 최종 심볼 복구까지 수행하는 계산식은 아래와 같다.

Fig. 7.은 함수이름 배열과 실제 코드영역 위치 계산에 필요한 각 함수별 오프셋이 포함된 영역이다.

❶ functab[nfuc] = pcHeader + pcHeader.pclnOffset

Fig. 8.은 각 함수별 함수이름 오프셋, 스택 관련 pcsp, pcfile, pcln 등 중요한 메타데이터들이 포함된 구조체(_func) 영역이다.

❷ _func = functab[nfuc] + functab[nfuc].funcoff

Functab Offset

```
00 00 00 00            dword_5A5C00    dd 0

D8 47 00 00                            dd 47D8h
80 00 00 00                            dd 80h
08 48                                  dw 4808h
00                                     db    0
00                                     db    0
E0                                     db 0E0h
00                                     db    0
00                                     db    0
00                                     db    0
60                                     db  60h
48                                     db  48h
00                                     db    0
00                                     db    0
40                                     db  40h
06                                     db    6
00                                     db    0
00                                     db    0
B8                                     db 0B8h
48                                     db  48h
```

Fig. 7. Fuctab Offset

Fig. 9.는 함수별 _func.nameoff를 통해 찾은 함수이름 배열 영역이다.

❸ A = pcHeader + pcHeader.funcnameOffset + _func.nameoff

Fig. 10.는 함수별 _func.entryoff와 실행파일의 textStart 영역을 더해 함수의 코드영역 시작주소를 찾은 영역이다.

❹ B = pcHeader.textStart + _func.entryoff

❺ ❸~❹와 같이 삭제된 함수이름과 매칭되는 코드를 알아낼 수 있다면, 모든 함수를 대상으로 IDAPython의 idc.MakeName 함수를 이용해 변경을 진행해 삭제된 심볼을 모두 복구할 수 있다. 복구된 결과는 아래 Fig. 11.과 같이 함수

_func Offset

```
80 00 00 00                            dd 80h
0B 00 00 00                            dd 0Bh
10 00 00 00                            dd 10h
00 00 00 00                            dd 0
01 00 00 00                            dd 1
08 00 00 00                            dd 8
0B 00 00 00                            dd 0Bh
04 00 00 00                            dd 4
00 00 00 00                            dd 0
7D                                     db  7Dh
00                                     db    0
00                                     db    0
00                                     db    0
00                                     db    0
00                                     db    0
00                                     db    0
07                                     db    7
17                                     db  17h
```

Fig. 8. _func Offset

Func Name Array

```
67 6F 3A 62 75 69 6C 64+aGoBuildid     db 'go:buildid',0
69                                     db  69h ; i
6E                                     db  6Eh ; n
74                                     db  74h ; t
65                                     db  65h ; e
72                                     db  72h ; r
6E                                     db  6Eh ; n
61                                     db  61h ; a
6C                                     db  6Ch ; l
2F                                     db  2Fh ; /
63                                     db  63h ; c
70                                     db  70h ; p
75                                     db  75h ; u
```

Fig. 9. Func Name Array Offset

Code in Func

```
                        sub_401080      proc near               ; COD

                        var_8           = qword ptr -8
                        arg_0           = qword ptr  8
                        arg_8           = qword ptr  10h

49 3B 66 10                             cmp     rsp, [r14+10h]
76 38                                   jbe     short loc_4010BE
48 83 EC 18                             sub     rsp, 18h
48 89 6C 24 10                          mov     [rsp+18h+var_8], rbp
48 8D 6C 24 10                          lea     rbp, [rsp+18h+var_8]
48 89 44 24 20                          mov     [rsp+18h+arg_0], rax
48 89 5C 24 28                          mov     [rsp+18h+arg_8], rbx
66 90                                   xchg    ax, ax
E8 9B 05 00 00                          call    sub_401640
48 8B 44 24 20                          mov     rax, [rsp+18h+arg_0]
48 8B 5C 24 28                          mov     rbx, [rsp+18h+arg_8]
E8 2C 00 00 00                          call    sub_4010E0
48 8B 6C 24 10                          mov     rbp, [rsp+18h+var_8]
48 83 C4 18                             add     rsp, 18h
C3                                      retn
```

Fig. 10. Code Section Start Address of the Found Function

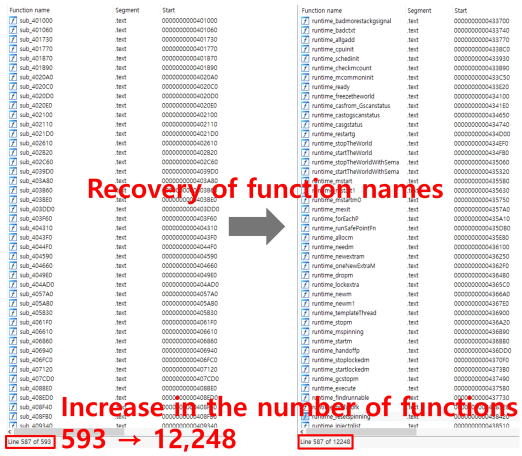

Fig. 11. The Result of Symbol Recovery

개수 증가(593→12,248) 및 심볼이 삭제된 함수명을 모두 복구할 수 있었다.

4.3.3 문자열 난독화 탐지 및 해제

4.3.3.1 Gobfuscate 난독화 사례

문자열 난독화는 함수·변수 등 주요정보를 읽기 어렵게 만드는 기술을 의미하며, 프로그래밍 언어별로 다양한 기법을 가지고 있다. 주요 목적으로는 누군가 자신이 만든 프로그램을 역공학 하려고 할 때, 분석 시간을 지연시키기 위해 사용한다. C#, JAVA와 같은 고수준 언어의 경우에는 C, C++ 과 같은 저수준 언어보다 훨씬 많은 심볼(Symbol)을 가지고 있어 난독화를 하지 않는다면 바이트 코드는 디컴파일러를 통해 컴파일 전 코드 그대로 확인이 가능하다. 이에, 프로그래밍 언어들은 함수명, 제어문, 문자열 인코딩, 안티 덤프 등 다양한 난독화 방식을 통해 코드를 보호하고 있다.

Golang은 문자열을 난독화 하는 오픈소스 프로젝트(Gobfuscate)를 주로 사용한다[30]. 이를 사용하면 패키지 이름, 전역 변수, 함수·타입·메소드 이름, 데이터섹션 내 문자열을 특정 커스텀 암호화 알고리즘(XOR 기반)을 통해 난독화를 수행한다. 해당 난독화는 소스코드 기반으로, 컴파일 전 난독화를 수행할 프로젝트 폴더경로를 파라미터로 입력해 난독화를 수행한 후 컴파일을 진행한다. 한 예로, 4.3.2에서 심볼이 삭제된 함수명을 모두 복구했지만, Fig. 12.와 같이 함수명이 난독화된 경우가 있었다.

이에, 본 연구에서는 해당 프로젝트를 목표로 문자열 난독화 알고리즘을 분석해 탐지 및 자동 해제하는 모듈을 개발했다.

난독화는 Fig. 13.과 같은 알고리즘을 사용한다. 먼저 4개의 HEX형 데이터로 나눈 뒤 길이가 같은 데이터별로 1Byte씩 XOR을 수행한다.

다음으로 분류된 N개의 결과 값들마다 HEX형에서 CHR형으로 변경(실제 코드에서는 runtime_slicebytetostring 함수 사용). 마지막으로 문자열 순서를 거꾸로 변환한 뒤 더하는 방식으로 최종 문자열을 도출한다. 참고로, 다른 난독화 함수를 탐색한 결과, 10개의 HEX형 데이터로 이루어진 것도 확인

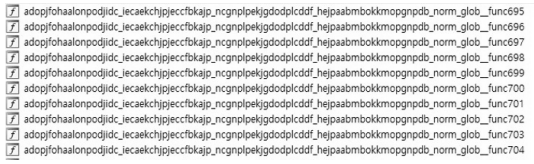

Fig. 12. Obfuscated Golang Functions

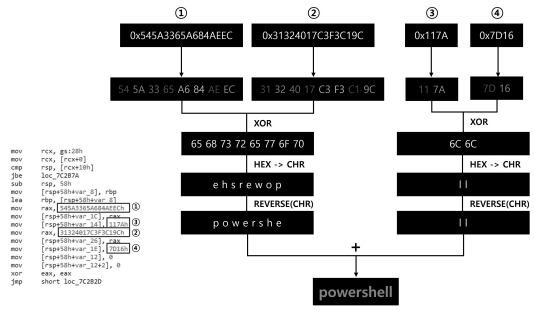

Fig. 13. Gobfuscate Obfuscation Algorithm

할 수 있었다. 즉, 난독화 형태를 시그니처 방식으로 탐지하면 미탐 확률이 높아질 수도 있다는 가정을 세웠다.

4.3.3.2 Gobfuscate 난독화 탐지

Gobfuscate는 컴파일 전 소스코드가 포함된 패키지 경로를 파라미터로 받아 재귀적으로 탐색하면서 문자열 난독화를 시작한다. 사용법은 Table 6.과 같으며, 'noencrypt' 옵션을 통해 암호화 하지 않을 패키지명을 지정할 수도 있다.

Fig. 14.는 Gobfuscate 난독화에서 사용하는 함수 내 코드이다. 이를 정적으로 탐지하기 위해서는 대표적인 특징을 파악한 뒤 아래 코드의 어셈블리어를 정교하게 탐지하는 알고리즘이 필요하다.

본 논문에서는 상시적인 코드 변화에 따른 미탐지 우려로 정적분석 기반이 아닌 동적분석 기반 탐지를 수행한다. 실질적인 기능은 난독화 기능을 호출하는 함수 내부 어셈블리 인스트럭션 특징을 기반으로 동적분석을 통한 난독화 탐지 알고리즘을 설계하고 개발했다.

난독화 탐지 알고리즘은 각 절차에 따라 동적으로 탐색한다. 우선, Fig. 15. 좌측 하단에 있는 난독화 특징 어셈블리어 5가지를 선정했다. 이를 기반으로 변종탐지 또한 고려하여 고정 명령어와 가변적인 특정 인자를 구분해 탐지 규칙(우측 하단)을 만들었다.

알고리즘 첫 번째로, 난독화 함수에서 고정적으로 사용되는 타입변환 함수(slicebytetostring)를 특징으로 사용하여 해당 문자열을 포함한 함수의 주소

```
mov     rcx, gs:28h
mov     rcx, [rcx+0]
cmp     rsp, [rcx+10h]
jbe     loc_7C2B7A
sub     rsp, 58h
mov     [rsp+58h+var_8], rbp
lea     rbp, [rsp+58h+var_8]
mov     rax, 545A3365A684AEECh
mov     [rsp+58h+var_1C], rax
mov     [rsp+58h+var_14], 117Ah
mov     rax, 31324017C3F3C19Ch
mov     [rsp+58h+var_26], rax
mov     [rsp+58h+var_1E], 7D16h
mov     [rsp+58h+var_12], 0
mov     [rsp+58h+var_12+2], 0
xor     eax, eax
jmp     short loc_7C2B2D
```

```
movzx   ecx, byte ptr [rsp+rax+58h+var_1C]
movzx   edx, byte ptr [rsp+rax+58h+var_26]
xor     ecx, edx
mov     byte ptr [rsp+rax+58h+var_12], cl
inc     rax

                        ; CODE XREF: main_mgj
cmp     rax, 0Ah
jl      short loc_7C2B1A
mov     [rsp+58h+var_58], 0 ; __int64
lea     rax, [rsp+58h+var_12]
mov     [rsp+58h+var_50], rax ; __int64
mov     [rsp+58h+var_48], 0Ah ; __int64
mov     [rsp+58h+var_40], 0Ah ; __int64
call    runtime_slicebytetostring
mov     rax, [rsp+58h+var_38]
mov     rcx, [rsp+58h+var_30]
mov     [rsp+58h+arg_0], rax
mov     [rsp+58h+arg_8], rcx
mov     rbp, [rsp+58h+var_8]
add     rsp, 58h
retn
```

Fig. 14. The Code Inside Functions Used in Gobfuscate Obfuscation

Table 6. Usage of Gobfuscate

gobfuscate.exe [**option**] [**Package path to obfuscate**] [**Obfuscated package storage path**]

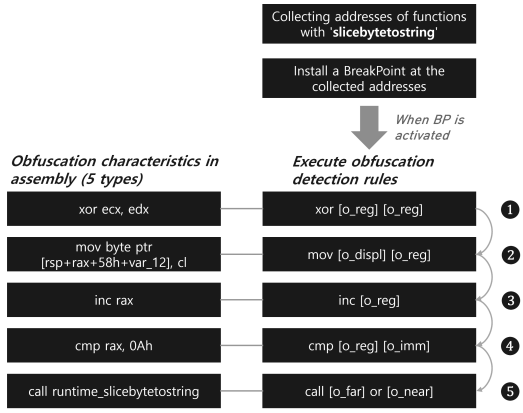

Fig. 15. Gobfuscate Obfuscation Detection Algorithm

를 모두 수집한다. 다음으로는 수집된 주소에 BP(BreakPoint)를 설정해 활성화 되기를 기다린다.

두 번째로, BP가 활성화 됐다면, 실제 난독화 함수인지 오탐인지 여부를 가리기 위해 난독화 함수를 구성하는 주요 어셈블리어(5가지)를 탐색해 정탐 여부를 최종 판별한다.

❶ XOR을 통해 데이터를 디코딩 하는 기능
❷ 디코딩된 데이터를 특정 메모리 주소에 저장
❸ 디코딩된 데이터의 길이를 측정하기 위해 반복시 1씩 증가
❹ 현재 반복된 수와 인코딩된 데이터(하드코딩) 길이를 비교
❺ runtime.slicebytetostring 함수를 호출

4.3.3.3 Gobfuscate 난독화 해제

난독화 함수 탐지가 모두 완료됐다면, 해제된 문자열을 얻기 위해 slicebytetostring 함수를 실행한 다음 RIP까지 BP를 설정한다. 해당 RIP까지 도착했다면, 현재 스택의 2가지 부분을 확인해 추출한다. Fig. 16.은 디버거 도구(x64dbg)를 통해 난독화가 해제되는 시점을 모니터링 위치이다.

모니터링 지점에 BP가 실행됐을 때, Fig. 17.과 같이 스택에서 해제된 문자열의 포인터 주소와 길이를 확인 할 수 있었다. 제안하는 시스템에서는 이와 같이 해제된 문자열을 난독화된 함수명 포함 모든 심볼을 복구해 분석가의 가독성을 높인다.

Fig. 16. The Point Monitored by the Debugger Tool (x64dbg) for Detecting Deobfuscation

Fig. 17. Check the Pointer Address and Length of the Deobfuscated String from the Stack

❶ 난독화 해제된 문자열의 길이(스택 위치 : RSP + 0x18)

❷ 난독화 해제된 문자열의 포인터(스택 위치 : RSP + 0x20)

4.3.4 Golang 표준·사용자 정의함수 분류

본 장에서는 Golang 실행파일 내 표준함수와 사용자 정의함수를 분류하는 방안에 대해 기술한다.

외부 라이브러리를 사용하지 않은 상태로 기본적인 코드만 작성 후 윈도우·리눅스 아키텍처별로 컴파일시 정적링크 방식으로 표준함수가 동일하게 리소스 영역에 저장되는 것을 확인했다. 이를 통해 Golang은 바이너리를 생성할 때 동일한 표준함수가 항상 존재한다는 사실을 인지했다. 한편, Fig. 11.에서와 같이 삭제된 심볼이 복구된 함수는 12,248개로 실제 분석에 필요한 사용자 정의함수는 100여개 이하였다. 즉, 나머지 함수들은 Golang 표준함수 및 OSS 라이브러리 내 모듈에서 가져온 코드로, 분석가가 정적분석을 수행하는 데 많은 시간이 소요돼 신속한 대응이 어렵다.

제안하는 시스템에서는 분석가의 분석속도와 코드 가독성을 위해 Golang 표준함수와 개발자가 작성한 사용자 정의 함수를 분류하는 기능을 개발했다.

분류를 위해 우선 실행파일 내 함수명과 코드 일부분을 2-depth 형태로 모두 수집하고, 함수 이름에 사전에 수집한 Goalng 표준함수 및 깃허브 관련 문자열 포함시 같이 수집한 코드 일부분과도 비교해 오탐(False Positive)을 방지한 상태로 제외시키는 방식으로 분류했다.

4.3.5 Pclntab 신버전 탐지 방안 제안

본 장에서는 새로운 Golang 버전에 대한 Pclntab을 효율적으로 찾기 위한 방법을 제안한다.

Pclntab을 찾는 방법으로는 4.1에서 기술한 것과 같이 각 파일형식별 찾는 방법의 대부분은 기존에 알려진 레거시 Golang 버전의 Pclntab.MagicHeader를 시그니처 탐색에 이용한다. 하지만, 값이 변경된 상태로 새로운 버전이 배포되면 기존 도구들의 Pclntab 탐색 기능은 미탐(False Negative)이 발생할 확률이 높다. 한 예로 분석가들이 자주 사용하는 정적분석 도구 IDA(8.4 최신버전)을 통해 △기존 악성샘플 원본 △동일 샘플 대상 새로운 버전으로 변조한 바이너리 2종을 비교·분석한 결과, Fig. 18.의 오른쪽 위치와 같이 코드, 구조체 기능에서 정상적인 동작을 수행하지 못했다.

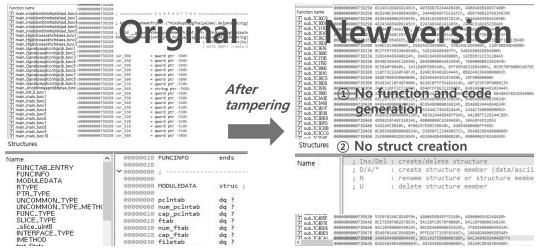

Fig. 18. The Analysis Results of Golang New Version and Stable Version in IDA Pro

4.3.5.1 Pclntab 신버전 탐지 첫 번째 방법(4가지 단계)

Pclntab 신버전을 탐지하는 알고리즘(4가지 단계)는 Fig. 19.와 같다. 해당 알고리즘은 개별이 아닌 Pclntab 위치를 정확도 높게 찾기 위한 4단계로 이루어진 하나의 알고리즘이며, 자세한 설명을 위해 4개로 나눠 설명한다.

❶ 코드 영역 내 모든 lea [o_reg] [o_mem] 가젯

```
Algorithm 1: Golang New Version Detection Algorithm
Input: detected_golang_version, start_text_seg_address,
  end_text_seg_address, start_rdata_seg_address, end_rdata_seg_address
Output: final_pcln_header_offset symbol_address_offset
legacy_pclntab_version_list = [FFFFFFFD, FFFFFFFB, FFFFFFFA,
  FFFFFFF0, FFFFFFF1]
regex_magic_header = [0-9a-f]2FFFFFF
if detected_golang_version in legacy_pclntab_version_list then
    return legacy_analysis_action
while True do
    line = start_text_seg_address
    if GetMnem(line) == lea and GetOpType(line, 0) == o_reg and
      GetOpType(line, 1) == o_mem then
        data_address = Dword(GetOperandValue(line, 1))
        for data_address ≤ offset ≤ data_address + 256 do
            count = 1
            offsets + = hex(offset)
            if count % 8 == 0 then
                suspicious_magic_header=regex_magic_header.find(offset)
                return suspicious_magic_header_offsets
            offsets = 0
            count + = 1
    line = NextAddr(line)
    if line ≥ end_text_seg_address then
        break
text_addr = start_text_seg_address
for start_rdata_seg_address ≤ addr ≤ end_rdata_seg_address do
    filtered_addr = Dword(addr)
    if filtered_addr == text_addr then
        return suspicious_text_offsets
    start_rdata_seg_address + = 4
arg_list_1 = suspicious_magic_header_offsets
arg_list_2 = suspicious_text_offsets
for suspicious_header in arg_list_1 do
    for suspicious_text_addr in arg_list_2 do
        distance_value suspicious_header  suspicious_text_addr
        if distance_value ≥ 0 and distance_value < 256 then
            return suspicious_pcln_header_offsets
go_build_string = 676f3a6275696c64
total = " "
for header_offset in suspicious_pcln_header_offsets do
    line = header_offset
    for line ≤ addr ≤ line+256 do
        one_step_offset = header_offset + Byte(line)
        for one_step_offset ≤ addr2 ≤ one_step_offset+8 do
            total += str(hex(Byte(addr2)))
        if total == go_build_string then
            return (final_pcln_header_offset symbol_address_offset)
        else
            total = " "
```

Fig. 19. Algorithm for Detecting the Pclntab in the New Version

을 탐색 후 0x100 범위 내 Magic Header 정규표현식('[0-9a-f]{2}ffffff)에 부합하는 주소와 값을 {addr, value}와 같은 형식으로 수집한다.

❷ Pclntab에서는 코드 영역(.text) 주소 항목이 포함된다. 이 값이 포함된 주소를 찾기 위해 리소스 영역(.rdata)을 스캔하면서 코드 영역이 저장된 주소를 모두 수집한다.

❸ ❶에서 수집한 주소와 ❷에서 수집한 코드 영역 주소가 유사한 위치에 있는지 검증하는 단계이다. ❷에 저장된 주소와 ❶에 저장된 주소를 각각 연산(❷-❶)을 통해 0x0~0x100인 값을 저장한다.

❹ 마지막 Pclntab을 검증하는 단계로, ❸에서 수집한 주소부터 8Byte씩 추출해 0x100 범위 내 심볼이 저장된 시작 값'676f3a6275696c64(go:build)'이 존재하는지 확인한다. 최종적으로 남은 ❶의 주소와 ❹에서 구한 심볼 시작 주소를 통해 삭제된 심볼을 모두 복구한다.

V. Golang 실행파일 신규 탐지·분석 모델 성능 평가

5.1 성능 평가 개요

본 장에서는 4장에서 기술한 Golang 탐지・분석 기법을 적용하여 개발한 시스템의 성능을 기존도구 6종과 3단계 비교 항목을 통해 성능 평가를 하였다. 이를 위한 실험셋은 바이러스토탈, MalShare, VirusShare에서 수집한 Golang 실행파일(PE・ELF 포맷) 악성 6,610건, 정상 15,899건 총 22,509건을 대상으로 진행한다.

첫 번째로는 70여개의 안티바이러스 엔진을 통합 관리・분석하는 바이러스토탈을 대상으로 변경 전/후 실험셋을 업로드 후 결과를 비교한다.

두 번째로는 지속적으로 업데이트 개발·배포되고 있는 대표적인 바이너리 역공학 분석 도구 3종, 파일 시그니처 분석 도구 2종과 제안 시스템의 정적분석 기능의 성능을 비교한다.

5.2 평가를 위한 사전 준비

5.2.1 분석환경 구축

정적분석 성능 검증용 평가에 사용된 환경은 Table 7.과 같다.

Table 7. Analysis Environment Specifications

		Specification
PC	OS	Windows 10 Pro (64bit)
	Memory	32GB
	Storage	SSD 2TB
S/W	VirtualBox	7.0.18 (64bit)
	Python	2.7.10 (64bit)

5.2.2 평가를 위한 실험셋 생성 논리

실험셋은 Golang 실행파일 Pclntab.magic 값이 버전별로 변경된다는 사실을 통해 새로운 버전을 모사・예측하여 수집한 실행파일 △악성(6,610) △

정상(15,899)개의 Pclntab.magic 값과 Pclntab의 위치를 변경한다.

위 기술한 내용을 기반으로 실험셋을 만드는 과정은 다음과 같다.

❶ Fig. 20.에서 작성한 YARA 룰은 Table. 1.에서 소개한 공개된 리틀·빅엔디언 형식의 Pclntab.magic 값을 기반으로 Golang 모든 버전에 해당하는 Pclntab 구조체를 탐색하는 기능을 수행한다.

❷ 탐색된 오프셋에 접근하여 Golang Pclntab을 각 버전의 구조체 크기에 따라 추출한다. 구조체 크기는 Table 8.과 같다. 다음으로, 구조체 시작 오프셋부터 4바이트 떨어진 위치에 바이너리 패치를 진행한다.

❸ 이미 알려진 Magic 값 중 하위 8비트(0xFA, 0xFB, 0xFD, 0xF0, 0xF1)을 제외한 값을 랜덤으로 생성한다. 다음으로, 생성된 값을 ❷와 같은 방법으로 기존 값에 바이너리 패치를 Fig. 21.과 같이 진행한다.

```
rule GoPclntab {
    strings:
        //little endian
        $suspicious_header_l = {F? FF FF FF 00 00 (01|02|04) (04|08)}

        //big endian
        $suspicious_header_b = {FF FF FF F? 00 00 (01|02|04) (04|08)}

    condition:
        $suspicious_header_l or $suspicious_header_b
}
```

Fig. 20. YARA rule for exploring Pclntab structure applicable to all versions of Golang

Table 8. Size of the Golang Pclntab

Version	Size
1.1~1.15	10Byte
1.16~1.17	38Byte
1.18~1.22	40Byte

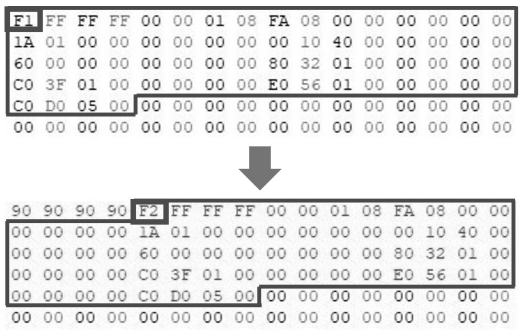

Fig. 21. Changing the Location(4byte) and Magic Value(1byte) of Golang Pclntab

5.3 Golang 실행파일 구조 변경을 통한 바이러스토탈의 탐지율 변화 비교

안티바이러스 탐지율 관련 연구로 L. Jeongho 등은 국내외 9종의 안티바이러스 제품 성능 시험을 위해 △기능성 △효율성 △신뢰성 △사용성 △부가기능 △공급업체 지원 등 총 6가지 평가지표를 수립하여 제품 성능을 검증했다[31]. 특징으로는 실험을 가상머신이 아닌 복원시점을 설정한 실제 PC에서 탐지 테스트를 진행했다.

C. Leka 등은 바이러스토탈과 상업용 안티바이러스간 탐지율이 일부 제품에서 다르다는 의견을 시작으로 다운로드·실행 등 다양한 실험을 토대로 비교분석을 진행했다[32]. 실험 결과, 아비라와 멀웨어바이츠는 바이러스토탈과 안티바이러스 탐지율이 같았으며, Windows Defender가 유일하게 바이러스토탈에서 탐지율이 더 높았다. 그 나머지는 안티바이러스 탐지율이 더 높았다. 저자는 Windows Defender가 탐지율이 더 높았던 이유로, 다른 안티바이러스 엔진에 비해 클라우드 기반 탐지기능이 포함돼있다고 판단했다. 즉, 바이러스토탈에서 운영중인 70여개 안티바이러스 엔진 중 Windows Defender를 제외하고는 정적분석 기반으로 탐지하고 있었다.

본 장에서는 안티바이러스 엔진의 기본 성능 테스트를 위해 5.2.2에서 Pclntab.magic 값을 변조한 실험셋 악성샘플(6,610개)를 대상으로 바이러스토탈에 분석요청을 수행했다. 그 결과, Table 9.에서 첫 번째 항목에서 두 번째 항목의 차이와 같이 안티바이러스 탐지 수가 원본 샘플에 비해 평균 20~30건씩 감소했다.

추가로, Golang은 정적링크 방식을 이용하기 때문에 크기가 크다는 특징이 있다. 이 사실을 기반으로 파일의 끝 부분에 40~100MB 크기의 패딩을 추가한 뒤 업로드한 결과, Table. 9.에서 두 번째 항목에서 세 번째 항목의 차이와 같이 Pclntab.magic 값을 변조한 것보다 탐지율이 평균 10건 이상 추가로 감소한 것을 확인 할 수 있었다.

즉, 정적분석 시 미리 정의한 magic 등 고유

Table 9. The Change in VirusTotal Detection Rates Due to Golang Structural Modifications

Original (Malware)	Pclntab Modify	Adding Bulk Padding
		

정보가 다르거나 실행파일 크기가 크다면 일부 안티바이러스 제품에서는 미탐이 발생할 가능성이 있다고 판단했다. 또한, 이를 악성코드 개발자가 이용하면 충분히 안티바이러스를 우회할 수 있는 가능성이 있다는 가정을 세울 수 있었다.

5.4 정적분석 도구와 Golang 신버전 탐지・분석 기능 비교

정적분석 도구는 파일을 실행하지 않고 코드・데이터・리소스 영역 등 파일 헤더와 구조를 파싱해 분석가에게 필요한 정보를 제공한다.

자체 개발한 시스템(GoAsap)과 Golang 실행파일 탐지・분석기능을 비교하기 위해 Table 10.과 같이 최근 악성코드 분석가들이 많이 사용하고 있는 대표적인 바이너리 정적분석 도구(3종)과 Golang 심볼 복구용 도구(3종) 총 6종을 선정했다. 평가를 위한 실험셋은 Golang 실행파일 대상 △(변조 전) 22,509 △(변조 후) 22,509 총 샘플 45,018개를 대상으로 진행했다.

성능 평가 항목은 5단계로 세부설명은 아래와 같다.

(A) 레거시 버전 Pclntab 탐지 : Golang 1.1~1.22버전 대상 실행파일 내 Pclntab 탐지 여부
(B) 레거시 버전 삭제된 심볼 복구 : Golang 1.1~1.22 버전 대상 실행파일 내 삭제된 심볼 복구 여부
(C) 신버전 Pclntab 탐지 : 새로운 Pclntab 탐지 여부
(D) 신버전 삭제된 심볼 복구 : 새로운 버전 배포 시 삭제된 심볼 복구 여부
(E) 문자열 난독화(Gobfuscate) 해제 : 문자열 난독화 탐지 및 해제 여부

Table 10. List of tools to be used for performance evaluation

Binary Reverse Engineering Analysis Tools	Version	Plugin (Golang Symbol Recovery)	Version
IDA Free	8.4	GoReSym	2.7.4
Ghidra	11.0.3	IDAGolang Helper	0.1
Radare2	5.9.0	redress	1.2.0

각 분석도구의 평가결과는 Table 11.과 같다. 대부분의 도구가 기존 Golang 버전에 대한 탐지・분석은 잘 수행되는 한편, 새로운 버전 및 자주 사용되는 난독화 기법에 대해서는 탐지・분석하지 못했다. 이는 알려진 시그니처 탐지 기반으로 Pclntab 구조체를 탐색했기 때문에 신버전 실행파일을 탐지・분석하지 못했다고 판단했다. 이에 반해, 제안한 GoAsap는 Pclntab이 변조되더라도 A・B・C・D 평가에서 45,018개 모두 탐지・분석이 가능했다.

IDA Free, GoReSym, IDAGolangHelper, redress는 이미 알려진 버전에서 구조체 구조분석 및 심볼 복구를 지원해 기능이 정상 동작했지만, 신버전 분석시 Golang 관련 정적분석 기능이 정상동작을 하지 않았다. 특히, IDAGolangHelper는 1.2~1.17버전까지만 지원을 해 1.18 버전부터는 지원하지 못했다.

Ghidra, Radare2는 Golang 버전, 빌드관련 내용 등 메타정보를 출력하는 풋프린팅 기능만 존재했으며, Pclntab 구조분석 및 심볼 복구 등 정적분석 기능을 지원하지 않았다. 다만, 도구에서 지원하는 플러그인 기능을 기반으로 개발된 Golang 정적분석용 스크립트가 존재했으나 신버전은 분석하지 못했다.

본 논문에서 제안한 GoAsap 시스템은 이미알려진 버전을 시그니처 탐색하는 방식이 아닌, 4.3.5.1에서 제안한 탐색방안을 통해 Pclntab 구조체를 찾는다. 이를 통해 신규 Golang 버전에서 삭제된 심볼 복구 및 문자열 난독화 해제 기능을 수행할 수 있었다.

Table 11. Comparison Results of Existing Static Analysis Tools and Golang Executable Detection and Analysis Features

	IDA Free	Ghidra	Radare2	GoReSym	IDAGolang Helper	redress	GoAsap (Our Method)
A	O	X	X	O	△	O	O
B	O	X	X	O	△	O	O
C	X	X	X	X	X	X	O
D	X	X	X	X	X	X	O
E	X	X	X	X	X	X	O

VI. 한계점 및 향후 계획

본 논문에서 제안한 GoAsap 시스템은 윈도우(AMD64), 리눅스(ARM) 등 각 아키텍처별로 탐지·분석하는 방법이 달라 새로운 아키텍처에 대한 코드 수정이 필요한 한계점이 있다. 최근 Golang은 악성코드 배포를 목적으로 크로스컴파일을 통해 다양한 멀티플랫폼 실행을 목적으로 아키텍처별 실행파일을 생성한다. 이에, 모든 아키텍처를 단순 대응하기는 쉽지 않다[33].

이를 대응하기 위해 향후에는 LLVM(Low-Level Virtual Machine)을 이용해 △Golang에서 크로스컴파일된 모든 아키텍처별 실행파일을 IR로 변환 △공통된 특징을 분석하여 Golang Pclntab 탐지 및 삭제된 심볼을 더 효과적으로 복구하는 방안에 대해 연구를 진행할 예정이다.

또한, 본 논문에서 제안한 시스템 내 난독화(Gobfuscate) 탐지기능을 다른 형태의 난독화에 적용·고려했을 때는 다른 알고리즘이 사용됐을 가능성이 높아 미탐(False-Negative)이 발생할 수 있다. 추후에는 Golang에 자주 사용되는 난독화 알고리즘을 지속적으로 파악해 시스템에 신속 적용하려고 한다.

VII. 결 론

기존 바이너리 분석도구들은 Golang 분석시 이미 알려진 특정 값을 통해 시그니처 탐색을 수행해 신규버전에 해당하는 Pclntab 구조체를 찾지 못하는 미탐 관련 문제점 있었다. 또한, 악성코드가 Golang 신규버전을 사용하거나 Pclntab 변조를 수행한다면 대응이 쉽지 않을 것이다. 이에, 신속한 대응을 수행할 수 있는 체계적인 탐지·분석 시스템이 필요하다.

따라서 본 논문에서는 정적분석 관점에서 Golang 신규버전 배포시 실시간으로 탐지·분석하는 시스템(GoAsap)을 제안했다. GoAsap는 기존 도구와 다르게 시그니처 탐색이 아닌 4가지 탐색방안을 통해 능동적으로 Pclntab 구조를 찾아 신속한 탐지·분석을 수행할 수 있다.

References

[1] Golang, "Golang Homepage" https://go.dev/, accessed May. 2024.

[2] TIOBE, "TIOBE Index" https://www.tiobe.com/tiobe-index/, accessed Oct. 2023.

[3] BLACKBERRY, "Old Dogs New Tricks: Attackers Adopt Exotic Programming Languages" https://blogs.blackberry.com/en/2021/07/old-dogs-new-tricks-attackers-adopt-exotic-programming-languages/, accessed Oct. 2023.

[4] S2W, "Kimsuky disguised as a Korean company signed with a valid certificate to distribute Troll Stealer" https://medium.com/s2wblog/kimsuky-disguised-as-a-korean-company-signed-with-a-valid-certificate-to-distribute-troll-stealer-cfa5d54314e2/, accessed Mar. 2024.

[5] GITHUB, "HackBrowserData" https://github.com/moonD4rk/HackBrowserData/, accessed Mar. 2024.

[6] GITHUB, "screenshot" https://github.com/kbinani/screenshot, accessed Mar. 2024.

[7] VB2023, "LET'S GO DOOR WITH KCP" https://www.virusbulletin.com/conference/vb2023/abstracts/lets-go-door-kcp/, accessed Mar 27. 2024.
[8] GITHUB, "kcp" https://github.com/skywind3000/kcp, accessed Jan. 2024.
[9] GITHUB, "kcp-go" https://github.com/xtaci/kcp-go, accessed Jan. 2024.
[10] BLACKHAT, "Tracing Golang Windows API calls with gftrace" https://www.blackhat.com/eu-23/arsenal/schedule/#tracing-golang-windows-api-calls-with-gftrace-35376/, accessed Jan 4. 2024.
[11] R. M. Yasir, M. Asad, A. H. Galib, K. K. Ganguly and M. S. Siddik, "GodExpo: An Automated God Structure Detection Tool for Golang," 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), pp. 47-50, May. 2019.
[12] J. Lauinger, L. Baumgärtner, A. -K. Wickert and M. Mezini, "Uncovering the Hidden Dangers: Finding Unsafe Go Code in the Wild," 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 410-417, Dec. 2020.
[13] GITHUB, "go-geiger" https://github.com/jlauinger/go-geiger, accessed Feb. 2024.
[14] Alexis Engelke, "Reconstructing Program Semantics from Go Binaries," M.S. Thesis, Technical University of Munich, Dec 10. 2023.
[15] M. Praveen and W. Almobaideen, "The Current State of Research on Malware Written in the Rust Programming Language," 2023 International Conference on Information Technology (ICIT), pp. 266-270, Aug. 2023.
[16] YARA, "Yara Doc" https://yara.readthedocs.io/en/latest/, accessed Dec 8. 2023.
[17] VIRUSTOTAL, "Virus Detection" https://www.virustotal.com/, accessed Dec 8. 2023.
[18] IDA Pro, "Disassembler Tool" https://hex-rays.com/ida-pro/, accessed Oct 20. 2023.
[19] Ghidra, "Disassembler Tool" https://ghidra-sre.org/, accessed Oct. 20. 2023.
[20] Radare2, "Disassembler Tool" https://rada.re/n/, accessed Oct 20. 2023.
[21] VB2021, "Reversing Golang Binaries with Ghidra" https://vblocalhost.com/uploads/2021/09/VB2021-04.pdf, accessed May 5. 2023.
[22] MANDIANT, "Ready, Set, Go – Golang Internals and Symbol Recovery" https://cloud.google.com/blog/topics/threat-intelligence/golang-internals-symbol-recovery/?hl=en/, accessed Jul 13. 2023.
[23] GITHUB, "GoReSym" https://github.com/mandiant/GoReSym, accessed Jan. 2024.
[24] GITHUB, "IDAGolangHelper" https://github.com/sibears/IDAGolangHelper, accessed Jan. 2024.
[25] GITHUB, "redress" https://github.com/goretk/redress, accessed Jan. 2024.
[26] GOOGLE, "Go 1.2 Runtime Symbol Information" https://docs.google.com/document/d/1lyPIbmsYbXnpNj57a261hgOYVpNRcgydurVQIyZOz_o/pub, accessed Dec 12. 2022.
[27] THE WHITE HOUSE, "BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE" https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, accessed Mar 26. 2024.
[28] National Security Agency, "Software Memory Safety" https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF/, accessed Mar 27. 2024.

〔29〕 VIRTUALBOX, “Virtual Environment” https://www.virtualbox.org/, accessed Oct 20, 2023.
〔30〕 GITHUB, “gobfuscate” https://github.com/unixpickle/gobfuscate, accessed Dec. 2023.
〔31〕 Jeongho Lee, Kangsik Shin, Youngrak Ryu, Dong-Jae Jung and Ho-Mook Cho, “A Study on Establishment of Evaluation Criteria for Anti-Virus Performance Test,” Journal of the Korea Institute of Information Security & Cryptology, 33(5), pp. 847-859, Oct. 2023.
〔32〕 C. Leka, C. Ntantogian, S. Karagiannis, E. Magkos and V. S. Verykios, “A Comparative Analysis of VirusTotal and Desktop Antivirus Detection Capabilities,” 2022 13th International Conference on Information, Intelligence, Systems & Applications (IISA), Corfu, Greece, pp. 1-6. Jul. 2022.
〔33〕 KISA, “TTPs #11: Operation An Octopus” https://thorcert.notion.site/TTPs-11-Operation-An-Octopus-d875862055ca4b7b815b5e496b219671, accessed Jul. 2024.

〈저자소개〉

강 형 민 (Hyeongmin Kang) 정회원
2017년 2월: 건양대학교 정보보호학과 졸업
2016년 12월~2019년 12월: 국가보안기술연구소 기술원
2018년 3월~현재: 충남대학교 컴퓨터공학과 석사과정
〈관심분야〉 악성코드 분석, 취약점 분석, IoT 보안

원 유 재 (Yoojae Won) 종신회원
1985년 2월: 충남대학교 계산통계학과 학사
1987년 2월: 충남대학교 계산통계학과 석사
1998년 2월: 충남대학교 컴퓨터공학과 박사
1987년 2월~2001년 2월: 한국전자통신연구원(ETRI) 팀장
2001년 3월~2004년 8월: 안랩유비웨어, 안철수연구소 CTO
2004년 9월~2014년 2월: 한국인터넷진흥원 인터넷침해대응센터 본부장
2014년 2월~현재: 충남대학교 컴퓨터융합학부 교수
〈관심분야〉 사이버 침해대응, 시스템 및 네트워크 보안, IoT 보안 등