NTT Security Holdings

Golang マルウェアに対する新たなアプローチ gimpfuzzy の実装と評価

NTT セキュリティ・ジャパン株式会社

本レポートの目的

NTT セキュリティ・ジャパン株式会社のセキュリティオペレーションセンター（以下 SOC）は、グローバルにおけるお客様システムを 24 時間体制で監視し、迅速な脅威発見と最適な対策を実現するマネージド・セキュリティ・サービス（以下 MSS）を提供しています。最新の脅威に対応するための様々なリサーチ活動を行い、その結果をブラックリストやカスタムシグネチャ、IOC（Indicator of Compromise）、アナリストが分析で使用するナレッジとしてサービスに活用しています。

SOC では、Golang 製のマルウェアを用いた攻撃を多く観測しています。Golang 製のマルウェアの情報は様々な組織から公開されていますが、SOC では特にクラスタリングに着目し、新たなツールを開発しました。本レポートでは、Golang マルウェアの現状と、開発したツールの概要および評価とケーススタディを紹介し、今後の Golang マルウェア対策のアプローチの一つとして活用していただくため、ホワイトペーパーを公開します。

目次

概要

NTT セキュリティ・ジャパン株式会社の SOC では、Golang 製のマルウェアを用いた攻撃を観測しています。本レポートでは、Golang 製のマルウェアに対するアプローチとして、以下の通り調査した結果を報告します。

- Golang 製のマルウェアの現状について
- gimpfuzzy という新たなアプローチの提案
- gimpfuzzy の評価実験およびケーススタディ

Web サイトでは gimpfuzzy を計算可能な YARA モジュールを公開しています。今後の Golang 製のマルウェアに対するアプローチの一つとしてご活用ください。

1.はじめに

Golang は 2012 年にはじめのバージョンがリリースされて以来、開発者に高い人気を得ているプログラミング言語です。現代的な言語設計であり、高速に動作し、コードの記述がシンプルで、かつ豊富なライブラリを有します。また、クロスプラットフォームにバイナリをコンパイル可能で、同一のソースコードから複数のプラットフォーム向けのバイナリをビルド可能です。

こうした特徴はマルウェア作成者にとっても同じで、Golang 製のマルウェアは年々増加傾向にあります。金銭を目的としたような攻撃に限らず、国家支援型の標的型攻撃でも採用されており、日本でも観測されています。

Golang 製のバイナリは独自の構造となっており、従来のような C/C++製のバイナリに対する解析手法はそのまま適用できるわけではありません。Golang 製バイナリはライブラリが静的リンクされており、また Golang 製バイナリ特有の挙動が知られておらず、かつ解析ツールがまだまだ少ない状況です。こうした事情から、Golang 製バイナリの性質を正しく理解していなければ、解析のハードルは高くなります。

SOC では増加し続ける Golang マルウェアに対して、効率的に解析リソースを割り当てるために、Golang マルウェアのクラスタリングを行うためのツールを開発しています。これを使用することで、未知の Golang 製バイナリに対してのみ解析リソースを割り振ることが可能となり、増加し続ける Golang マルウェアを効率的に対処することができます。

本稿では、まず Golang マルウェアの状況と、解析のためのツールを紹介します。その後、現状抱える課題を示し、それを解決するためのアプローチとして gimpfuzzy を提案します。更に、gimpfuzzy を用いたいくつかの評価実験を行い、その結果とケーススタディを示します。最後に、現在の gimpfuzzy が抱える課題と今後の展望について扱います。

本稿によって、Golang マルウェアを解析する現場で、解析リソースを適切に割り振り、組織を守るための一助となることを目指しています。

2.Golang マルウェアの状況

2012 年に Golang のはじめのバージョンがリリースされて間もなく、Encriyoko というマルウェアが観測されています。これが最初の Golang マルウェアであると言われていますが、その後も図 1 のように次々と Golang マルウェアが登場しています。

それらの中でも特に有名なものとして、APT29 が使用した WellMess と Zebrocy の Golang 実装が 2018 年に報告されています。それ以降、国家支援型の標的型攻撃においても Golang マルウェアが増加しており、日本に対する攻撃でも使用されています。

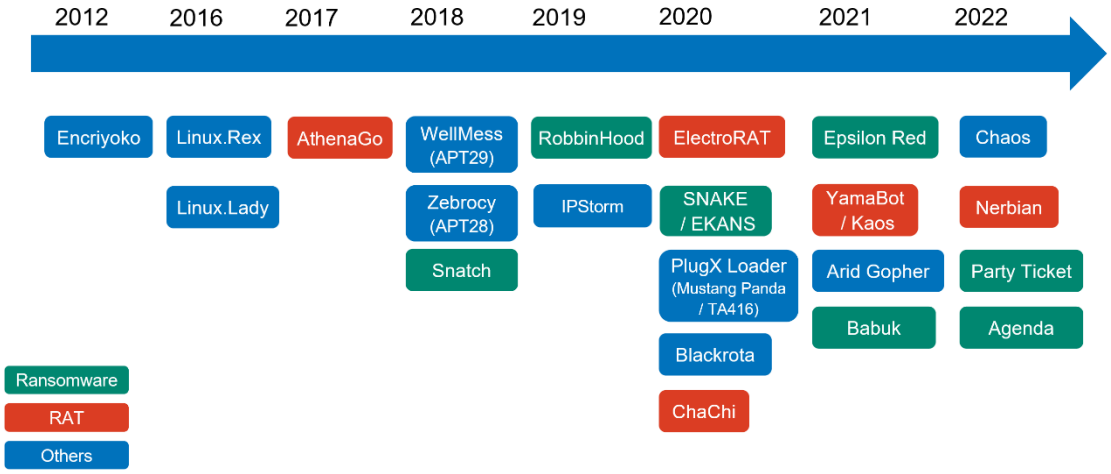


図 1 代表的な Golang マルウェアの変遷

また、昨今では Golang で実装された Post-Exploitation フレームワークや、マルウェア作成フレームワークが存在しており、GitHub などで公開されています。これらは非常に活発に開発が行われており、攻撃者に一定の人気があります。

Golang 製バイナリはライブラリが静的リンクされており、バイナリサイズが大きく、そのままの状態では解析が困難であることが知られています。現在では解析ツールにプラグインなどが開発されつつありますが、解析情報は依然として少なく、その参入ハードルは高くなっています。

3.関連ツールと課題

Golang マルウェアの特徴はこれまでに説明した通り、相対的にリバースエンジニアリングのハードルが高く、攻撃者有利な状況ではありますが、解析の補助となるツールが活発に開発されています。本章ではそのツールの一部を紹介します。

3.1. ライブラリ関数の識別

Golang マルウェアに限らず Golang 製バイナリはライブラリが静的リンクされており、ファイル単体で動作します。よって IDA や Ghidra といった disassembler ツールでは多くの関数が検出されます。そのためライブラリ関数の判別が出来ないツールの場合は、解析対象の関数が多く検出される事で、そのプログラムの主要な処理部分の絞り込みが難しい状況にありました。

図 2 は"Hello World"を出力する単純なプログラムを C 言語と Golang（以下 HelloGo バイナリ）で実装し、IDA の Version 7.7 で開いた際に検出された関数リストを示しています。

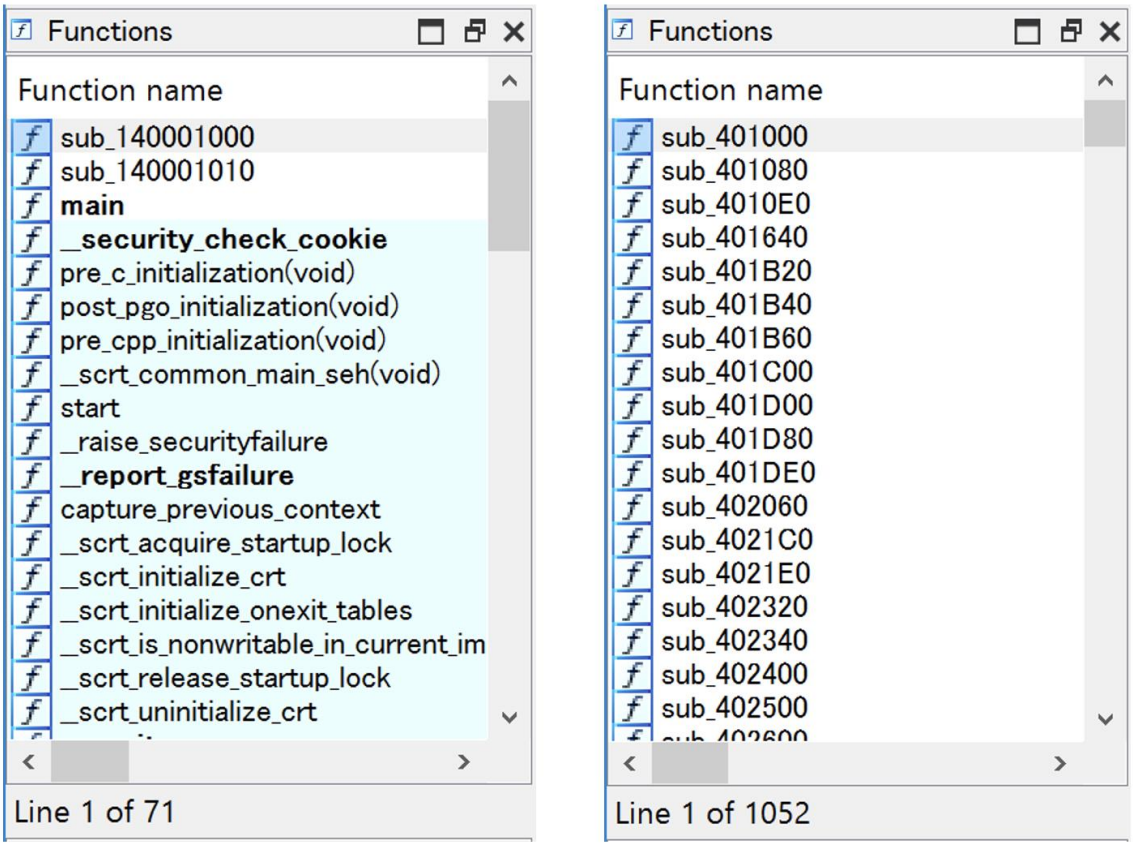


図 2 C 言語と Golang のバイナリで検出された関数の比較

左側が C 言語、右が Golang です。図 2 の下部は検出された関数の数が表示されており、C 言語に比べ Golang では極端に多いことがわかると思います。

また、Function name ですが、右側の Golang はすべての関数が"sub_～"という名前で関数として認識されており、ライブラリ関数が認識できておりません。この中からプログラムの本質的な部分を探して解析する必要があることから、分析の難易度が高いことがわかるかと思います。

しかし、最近ではツールの改善によりライブラリ関数の識別は容易になりました。

3.1.1. IDA の例

図 3 は図 2 の HelloGo バイナリを IDA 8.2 で開いたときの画面です。

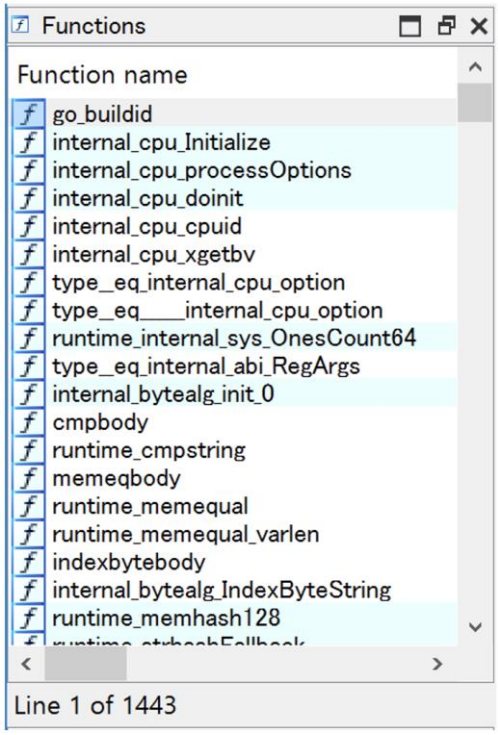


図 3 IDA 8.2 HelloGo バイナリの関数リスト

図 2 の右側と比べると、ライブラリ関数などが認識されていることがわかると思います。しかしデコンパイラは Golang のソースコードを復元するまで至らず、アセンブリ言語または生成された疑似的な C 言語のコードで解析する必要があります。

```
1  // main.main
2  void __cdecl main_main()
3  {
4    int v0; // r10d
5    int v1; // r11d
6    __int64 v2; // r14
7    void *retaddr; // [rsp+8h] [rbp+0h] BYREF
8
9    while ( (unsigned __int64)&retaddr <= *(_QWORD *)(v2 + 16) )
10     runtime_morestack_noctxt();
11   fmt_Fprintf((unsigned int)off_4C7F88, qword_53CA08, (unsigned int)"Hello World\n", 12, 0, 0, 0, v0, v1);
12 }
```

図 4 IDA 8.2 HelloGo バイナリのデコンパイル結果

3.1.2. Ghidra の例

Ghidra でも様々なツールが開発されています。

以下の図 5 は GolangAnalyzerExtension [1] という Ghidra プラグインを導入し、先ほどの HelloGo バイナリを開いた時の様子です。右側がプラグインの適用後ですが、左側の適用前と比べて関数が認識できていることがわかると思います。

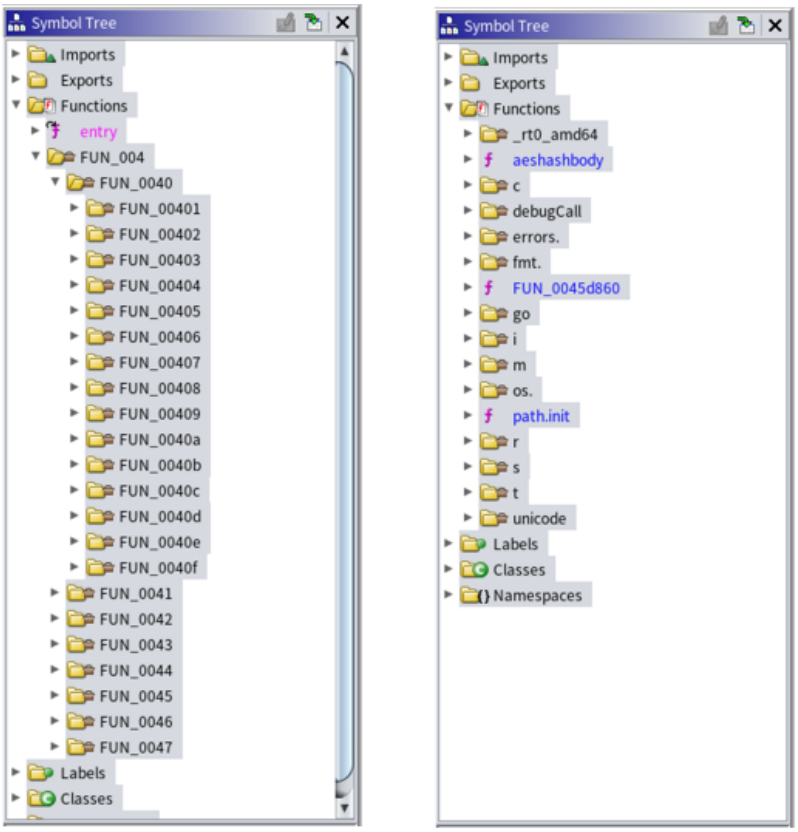


図 5 GolangAnalyzerExtension の適用

また図 6 はデコンパイル結果です。GolangAnalyzerExtension によってソースコードのファイルパスのコメントが追記されており、このファイル名から機能が推測できる場合もあります。

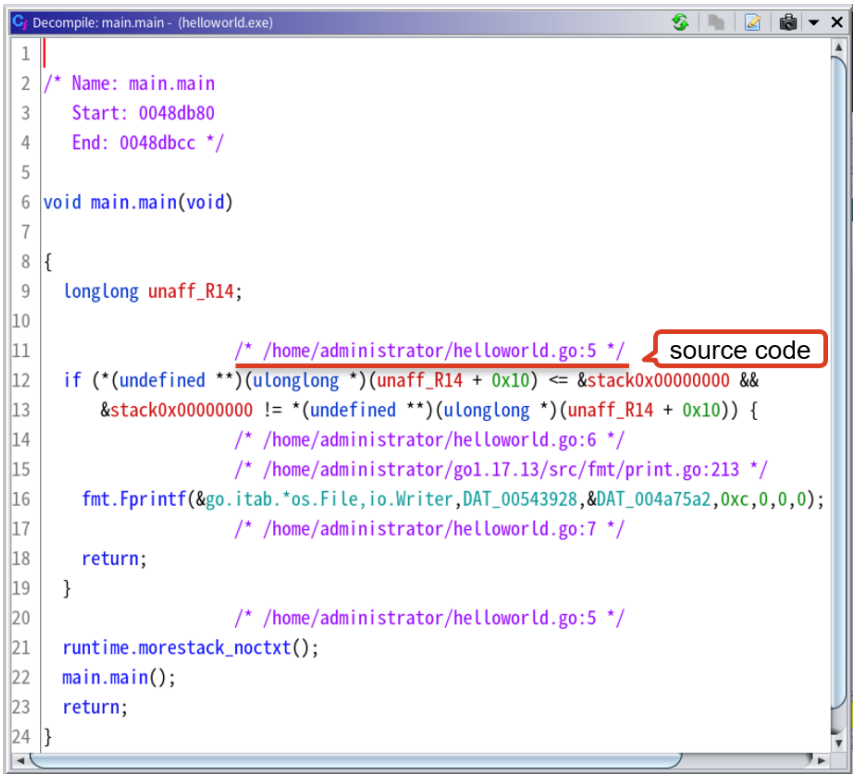


図 6 GolangAnalyzerExtension 適用時のデコンパイル結果

3.2. gobfuscate

gobfuscate [2] は Golang のプログラムに特化した、ソースコードに含まれるパッケージ名、関数名、変数名などの文字列情報を難読化し実行ファイルを作成するツールです。

重要なアルゴリズムなどがリバースエンジニアリングによって漏れることを防ぐような正当な使われ方をする一方で、マルウェアの解析を難しくする目的でも使われています。

特定の Golang マルウェアはこのツールを使い難読化が施されたものが存在しますが、難読化を解除し情報を復元するツール degobfuscate.py [3] も公開されています。

図 7 は GolangAnalyzerExtension を適用している状態で、先ほどの HelloGo バイナリに gobfuscate を適用した結果です。図中の赤線＊1 からファイル名以外のパス情報は難読化されている状況が確認できます。赤線＊2 は難読化された文字列を復元するための関数が追加されました。

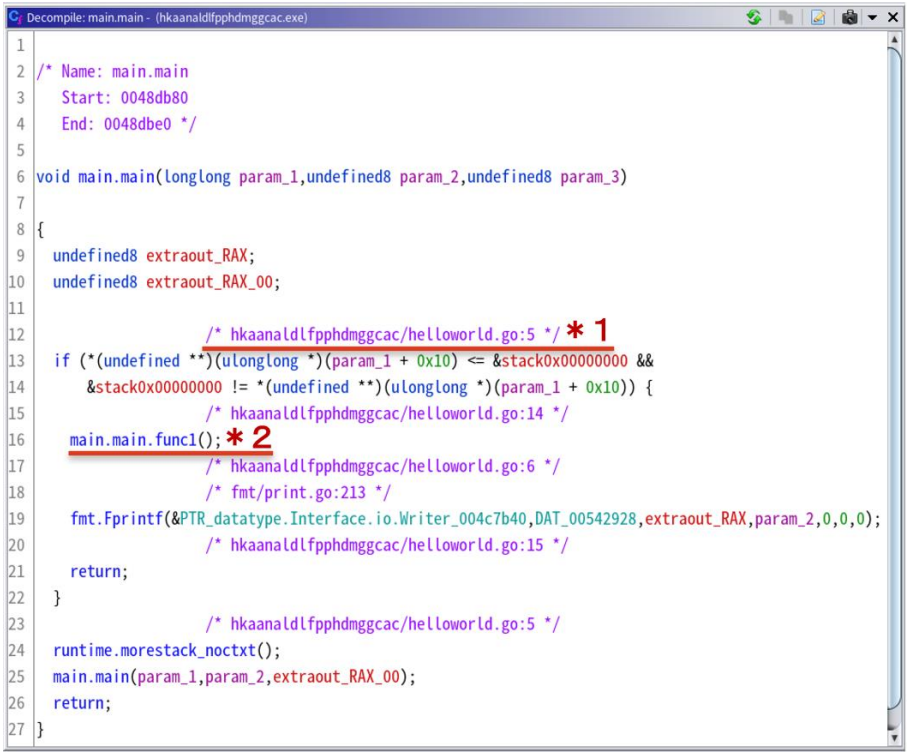


図 7 HelloGo バイナリに gobfuscate を適用した結果

図 8 degobfuscate 適用した結果の赤線＊3 は難読化を解除し情報を復元するスクリプト degobfuscate.py を適用したことで、文字列が復元され、解析の補助となるように関数名が変更されました。

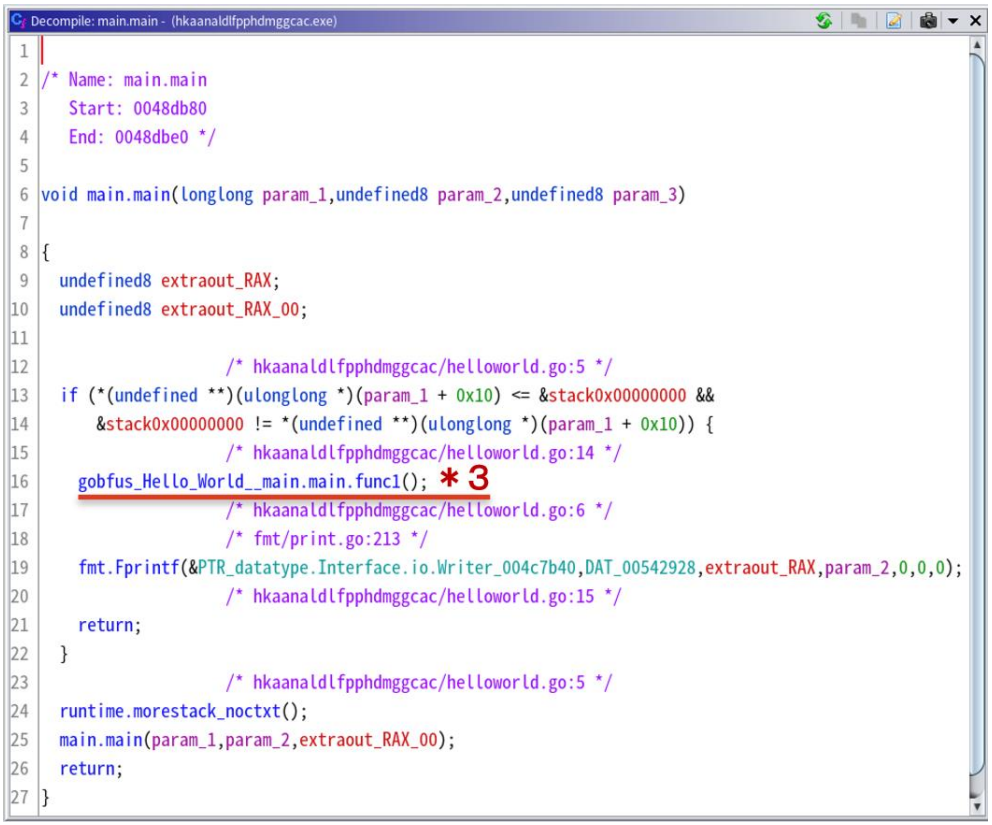


図 8 degobfuscate 適用した結果

3.3. gimphash

VirusTotal [4] や MalwareBazaar [5] 、Triage [6] などのマルウェア検体の共有プラットフォームでは検体のプロファイリングや分類・探索のために様々な指標が取り入れられています。下記はその代表的な例になります。

- filehash
- TLSH
- imphash
- impfuzzy
- ssdeep
- gimphash

Golang 製バイナリの特徴に着目した指標として gimphash（Go-import-hash)があります[7]。Golang 製バイナリはプラットフォームに依存せず、実行可能ファイルの一部に pclntab [8] という構造を持ち、そのプログラムに依存する標準ライブラリや外部パッケージの情報を含んでいます（以下この情報を gimp と記載します)。

図 9 は HelloGo バイナリより gimp の情報を出力した結果となります。gimphash はこの gimp から汎用的な関数名などを削除した文字列に対し SHA256 でハッシュ値を計算したものです。

```
$ ./gimp helloworld.bin
internal/cpu.Initialize
internal/cpu.processOptions
internal/cpu.doinit
internal/cpu.cpuid
internal/cpu.xgetbv
type..eq.internal/cpu.option
type..eq.[...]internal/cpu.option
runtime/internal/sys.OnesCount64
type..eq.internal/abi.RegArgs
internal/bytealg.init.0
cmpbody
runtime.cmpstring
memeqbody
runtime.memequal
runtime.memequal_varlen
indexbytebody
internal/bytealg.IndexByteString
runtime/internal/syscall.Syscall6
runtime.memhash128
```

図 9 HelloGo バイナリから依存する関数を抽出した一部

この算出された gimphash を使って、同一ファミリのマルウェア検体の分類や探索などに利用されています。

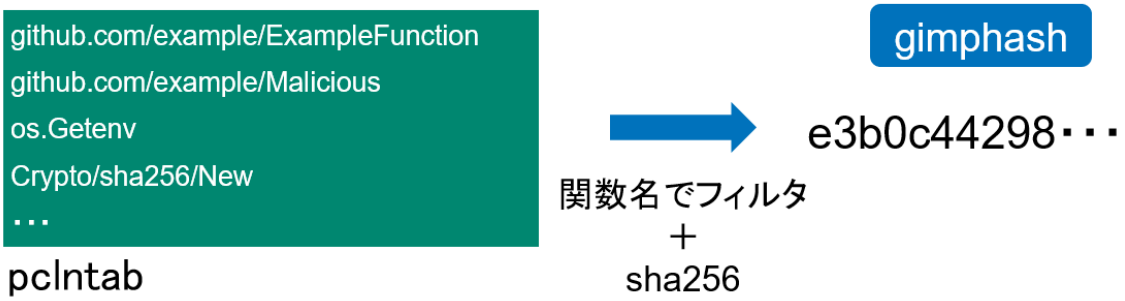


図 10 gimphash 計算過程

3.4. 課題

これまで紹介したように Golang マルウェアに対する解析ツールの環境は改善されつつありますが、以下のような課題があります。

- これまでの C/C++製バイナリと同等の解析方法は難しい
  - 解析者が Golang 製バイナリ特有の構造や挙動に慣れていない
    - Golang 特有のデータ構造や関数呼び出し規約など
  - デコンパイル結果も C 言語コードとなり、Golang へは対応できていない
- 言語がアップデートされ新しいバージョンでビルドされた実行ファイルについてツールが対応できない場合がある
  - ツール側で新しいバージョンに対応しなければツールが使えない

そうした課題からマルウェア解析者の負担を減らすため、分析済みの Golang マルウェアの分類や探索を効率よく実施できることが求められますが、Golang 製バイナリに特化した指標 gimphash には、gimp が 1bit でも異なればハッシュ値が異なり、類似度の判定が不可能となってしまうという大きな課題があります。

4.gimpfuzzy

gimpfuzzy は前章で挙げた gimphash の課題を解決できます。gimpfuzzy の説明の前に以下の基本となる概念について簡単に説明します。

- Fuzzy Hashing アルゴリズム
- 編集距離（レーベンシュタイン距離)

4.1. Fuzzy Hashing

Fuzzy Hashing は似ている入力には、ある程度同じ値を返す大雑把なハッシュ値を計算するものです。

図 11 は Fuzzy Hashing の計算例を示します。左側の入力でそれぞれの文字に対する Fuzzy Hash の結果が右側の文字列となります。赤字部分は情報の差異を表しており、右側の結果を見てわかるように同じような入力に対しては、同じような結果の文字列が出力されます。

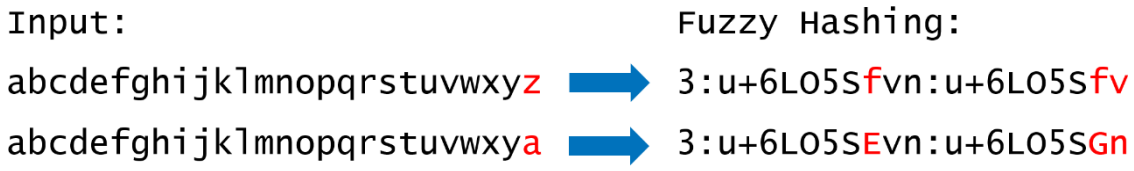


図 11 Fuzzy Hashing の計算例

4.2. 編集距離（レーベンシュタイン距離）

編集距離は別名レーベンシュタイン距離とも呼ばれており、２つの文字列の類似度を示す方法で、１文字の挿入・削除・置換によって一方の文字列からもう一方の文字列へ変換する最小の回数と定義されます。

図 12 は文字列"Security"と"Secret"の編集距離の計算例です。

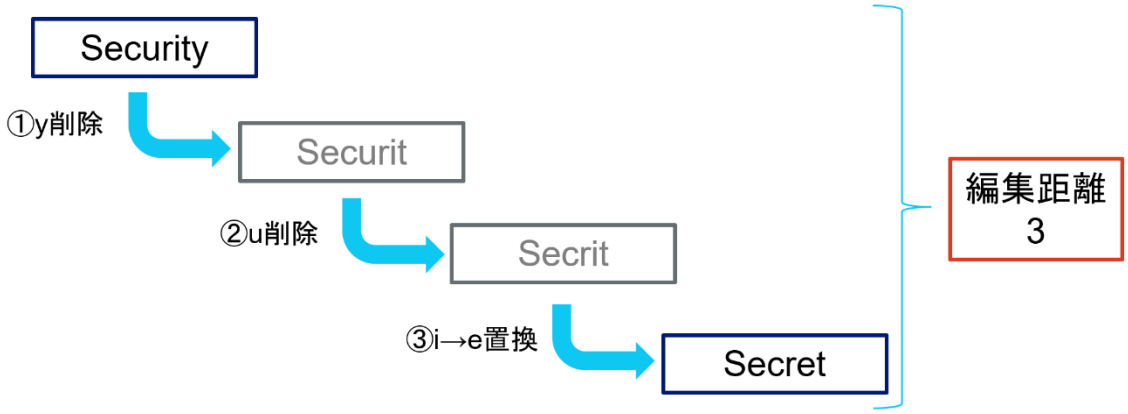


図 12 編集距離の計算例

手順の回数が小さいほど、２つの文字列の類似度は高いと判断できます。

4.3. gimpfuzzy

前述したように、Golang 製バイナリには pclntab という構造を持っており、依存するパッケージや呼び出す関数名などの情報（gimp)を抽出可能です。

gimphash は gimp に対し SHA256 のハッシュ値を計算した値です。gimp の値が少しでも変わるとハッシュ値が全く異なるため、バイナリ同士の類似性の検証には利用できません。我々は Golang 製バイナリについて、この gimp を抽出し、その情報に対して Fuzzy Hashing を計算する gimpfuzzy を提案しました。

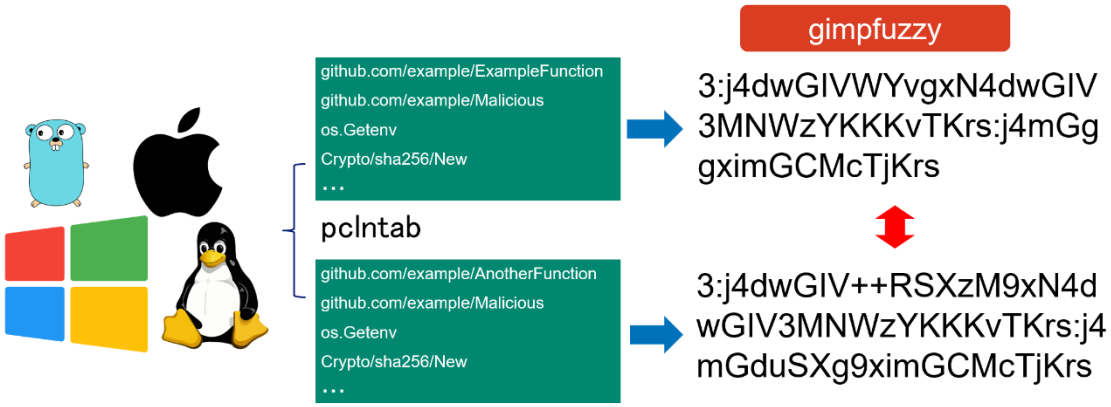


図 13 gimpfuzzy の概要

Golang 製バイナリそれぞれについて gimpfuzzy を計算し比較することで、バイナリの類似度を算出します。

図 14 は Fuzzy Hashing の実装の一つである ssdeep [9] によって、編集距離による類似度スコアを計算する様子を表しています。この数値が 100 に近いほど類似度が高いことを示しています。

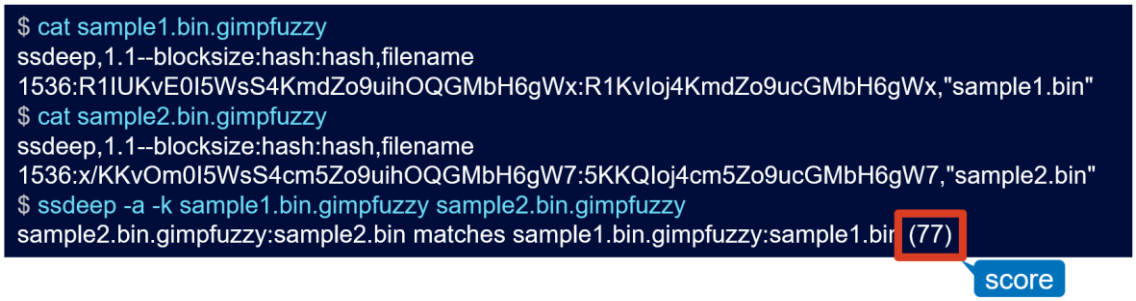


図 14 ssdeep による類似度スコア計算イメージ

5.YARA モジュール

5.1. YARA について

YARA [10] は VirusTotal により開発された、主にマルウェアをターゲットにしたファイル（以下、検体）の分類、検索を行うためのツールキットです。具体的には、YARA ルールと呼ばれるルールを記述することにより、そのルールにマッチする検体の検索が行えます。例えば、YARA の開発リポジトリには SilentBanker と呼ばれる検体を検索するルールの例が記載されています。

```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        threat_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        $a or $b or $c
}
```

図 15 SilentBanker を分類する YARA ルールの例

図 15 は検体のバイナリに出現する特徴的なビット列や文字列を基にしたルールの例ですが、他にも様々なルールに基づいた検索が行えます。

例えば、PE ファイルを扱う YARA モジュールを利用すれば、アーキテクチャ情報、インポート関数、エクスポート関数、エントリポイントのアドレスなどが検索ルールとして利用可能です。

また、YARA は C 言語で実装されており、マルチスレッドに対応しています。そのため、多数の検体を高速に処理することが可能です。

5.2. Golang 製バイナリを対象にした YARA モジュール

Golang によってビルドされた検体は、それぞれのアーキテクチャに応じた YARA モジュール（PE、Mach-O、ELF）を利用可能です。

一方で、前述のとおり Golang 製バイナリには pclntab などの解析に有用な情報が独自に存在しますが、それらを専門的に扱えるモジュールは現時点で存在しません。さらに、我々が提案する gimpfuzzy を扱うモジュールも存在しません。このような情報を利用可能な Golang 製バイナリの解析に特化したモジュールを開発し利用することにより、より詳細な検体の分類や検索が可能になります。

5.3. 開発したモジュールの紹介

今回、我々は新たに「go」モジュールと「fuzzy」モジュールを開発しました。「go」モジュールは Golang によりビルドされた PE ファイルを扱うモジュールです。pclntab から抽出した関数名や、関数名を基にした gimpfuzzy を計算可能です。

一方、「fuzzy」モジュールは ssdeep を用いた Fuzzy Hashing を扱うモジュールです。文字列から Fuzzy Hash を計算したり、ハッシュ値の類似度を計算したりすることが可能です。

上記 2 つのモジュールを利用することにより、図 16 のルール例のように gimpfuzzy の類似度を基にした検体の検索、分類が可能になります。

```
import "go"
import "fuzzy"
import "pe"

rule GoFuzzyTest
{
    condition:
        pe.is_pe
        and
        fuzzy.score(go.gimpfuzzy(),"96:O5iaa8UdGAq27F92HQvI//ssMin64K+v+:OIaMsAq27F92HQvI//ssMin641v+")>80
}
```

図 16 gimpfuzzy を基に検体を分類する YARA ルールの例

上記のルールでは、検体の pclntab から gimpfuzzy を計算し、文字列 "96:O5iaa8UdGAq27F92HQvI//ssMin64K+v+:OIaMsAq27F92HQvI//ssMin641v

+"との類似度が 80 以上だった検体を抽出することが可能なルールです。このルールを適用することにより、ある検体と機能が類似する検体を検索・分類することが可能です。

今回開発した 2 つの YARA モジュールを利用することにより、大規模な Golang の検体群について、利用する関数名の類似度に基づいた検体の抽出、検索を高速に行うことができます。その結果、大量の Golang マルウェアのファミリ分類や、特定の検体に絞ったマルウェアのハンティングなどが行えるようになります。

6.評価

6.1. 評価方法について

本ホワイトペーパーでは、大きく分けて 2 つの評価を行います。

- 解析済みの、ファミリ名が判明している検体について gimpfuzzy による分類を行い、精度及びクラスタリングを行う閾値の決定・評価を行います。
- 収集した最新の未解析の検体について gimpfuzzy による分類を行い、どのようなクラスタが生成されるか評価を行います。

6.2. 解析済み検体における評価

6.2.1. データセット

本節では、Palo Alto Networks 社が公開している Golang マルウェアのデータセット[11]（以下、PaloAlto データセット）を用います。本データセットには、実際に Palo Alto Networks 社のアナリストが収集したマルウェア検体のハッシュ値と、検体を解析した結果に基づいて作成した YARA のルールで分類した結果が表形式で記載されています。

本リストからファミリ名が判明しており、かつ、VirusTotal 上でハッシュ値がヒットし、検体がダウンロード可能だった約 7,900 件、53 ファミリの検体に基づいて評価を行います。これらの検体のうち、実際に gimpfuzzy を計算可能だった検体は 5590 件でした。

6.2.2. 評価指標

評価指標として、組み合わせの確率を用います。同一ファミリ内の検体を 2 つ選んだ時に、その 2 つの検体の組み合わせにおいて類似度スコアが閾値以上である確率を考えます。

下記に示す図 17 は、同一ファミリの検体 6 個 (A-F) について、それぞれ gimphash の類似度スコアを計算した例を示しています。この場合、検体 2 つの組み合わせのうち類似度スコアが閾値以上である確率は

$$\frac{{}_4C_2 + {}_2C_2}{{}_6C_2} \approx 0.467$$

となります。これを判別率とします。

	A	B	C	D	E	F
A	100	100	70	80		
B	100	100	65	90	0	0
C	70	65	100	70	0	0
D	80	90	70	100	0	0
E	0	0	0	0	100	100
F	0	0		0	100	100

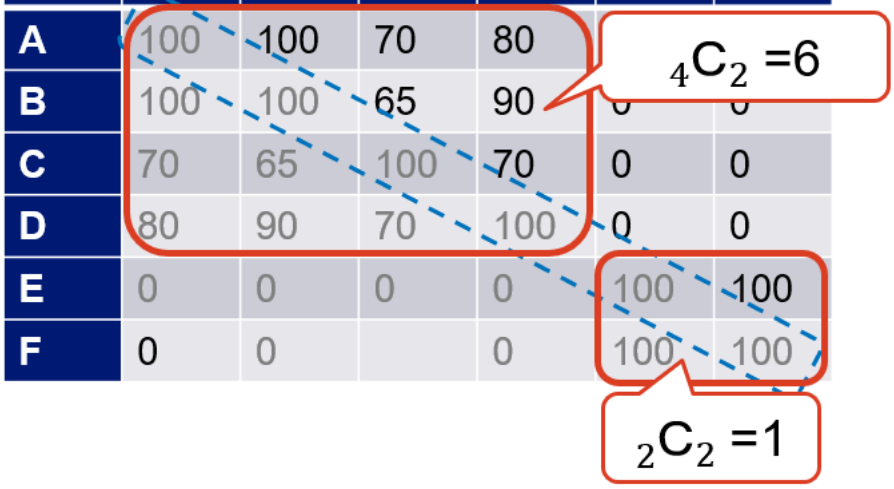
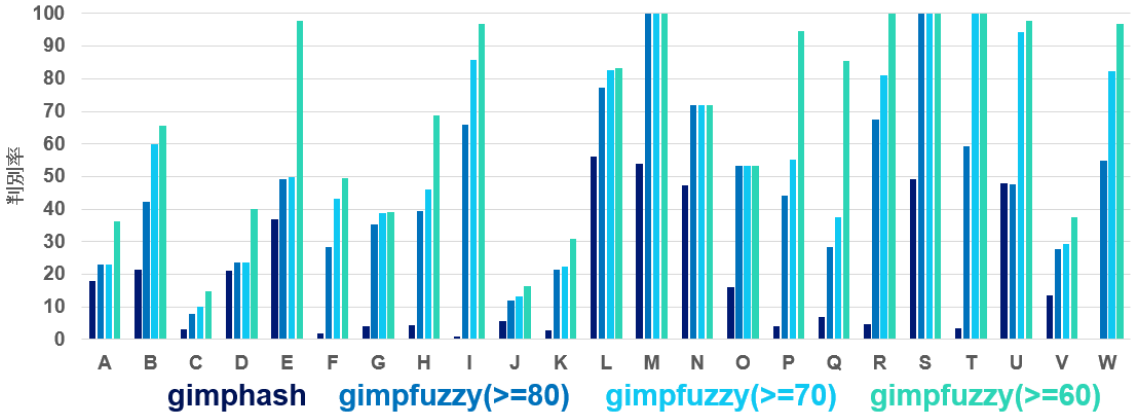


図 17 判別率の計算例

ここで、gimphash の場合、ハッシュ値の性質から元の入力値が 1bit でも異なれば全く違う値が出力されます。つまり、類似度のスコアは 0 (全く一致しない) か 100 (完全一致) の二択になることに注意が必要です。

PaloAlto データセットに含まれる検体に gimpfuzzy 及び gimphash を適用し、ファミリごとに判別率を計算し平均をとった結果を図 18 に示します。



	gimphash	gimpfuzzy(>=80)	gimpfuzzy(>=70)	gimpfuzzy(>=60)
平均判別率	7.32	47.85	67.37	79.98
(veil除外)	17.64	37.89	46.13	55.94

図 18 PaloAlto データセットにおける判別率の計算結果

図 18 の各グラフは、gimhash 及び gimpfuzzy における判別率を示しています。さらに、gimpfuzzy において分類を決めるスコアの閾値は自由に決められるので、閾値を 80、70、60 と変化させたときの精度を示しています。特に、「veil」ファミリ[12] は共通する性質から gimpfuzzy との相性が良く、gimphash と比較して大幅に精度が向上します。判別率に大きく影響するため、veil ファミリを除外した判別率の平均を分けて記載しています。なお、veil ファミリにおいて分類精度が大きく向上する要因は、7 章で考察します。

6.2.3. 最適な閾値の決定

本評価ではファミリ分類の結果を正解として評価を行っていますが、何を分類の正解とするかは目的によって変わり、適切な閾値も変わります。例えば、gimpfuzzy の活用方法として、以下のような使用例が考えられます。

- マルウェアの機能などおおまかな分類
- マルウェアファミリの分類
- マルウェアファミリのうち亜種や細かいバージョン違いまでの細かい分類

大まかな分類を行うときは閾値を低めに取り、逆に細かい分類を行いたいときは閾値を高めに設定する事が可能です。

一方で、図 18 の評価では閾値を下げれば判別率が向上しますが、単純に閾値を下げればよいというものではありません。ファミリ内における組み合わせの確率を基にした評価では、ファミリ外の検体を間違えて紐づけてしまう可能性を考慮できていません。本節では、ファミリ分類において両者のトレードオフを考慮した最適な閾値について調査します。評価に用いるデータセットについては引き続き PaloAlto データセット[11]を使用します。

まず、ファミリ内の検体同士の組み合わせ全てについて、gimpfuzzy の類似度スコアの分布を調べます。類似度スコアの分布を示すと、図 19 のようになります。

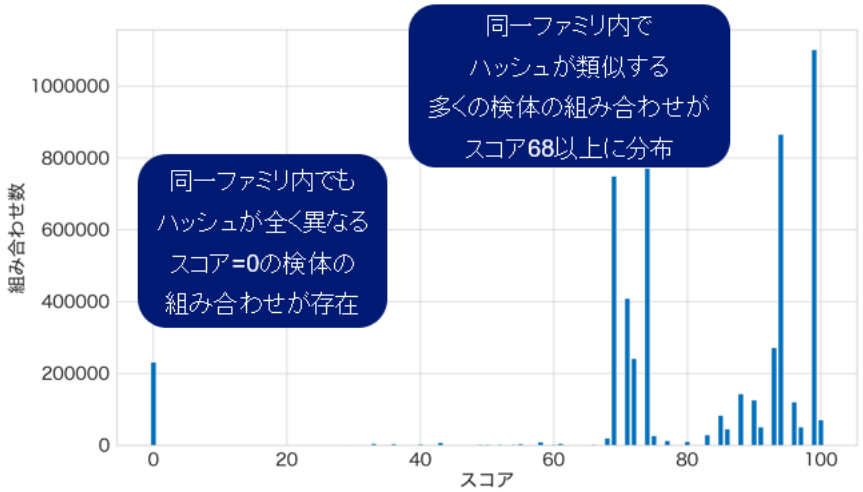


図 19 ファミリ内の検体の組み合わせにおける類似度スコアの分布

一方で、ファミリ外の検体同士の組み合わせ全てのうち、75%の組み合わせでは、類似度スコアが 0 になります。一方で、25%の検体の組み合わせでは、類似度スコアが 0 より大きくなります。0 より大きい類似度スコアを持つ、25%の検体の組み合わせのスコアの分布を示すと、図 20 のようになります。

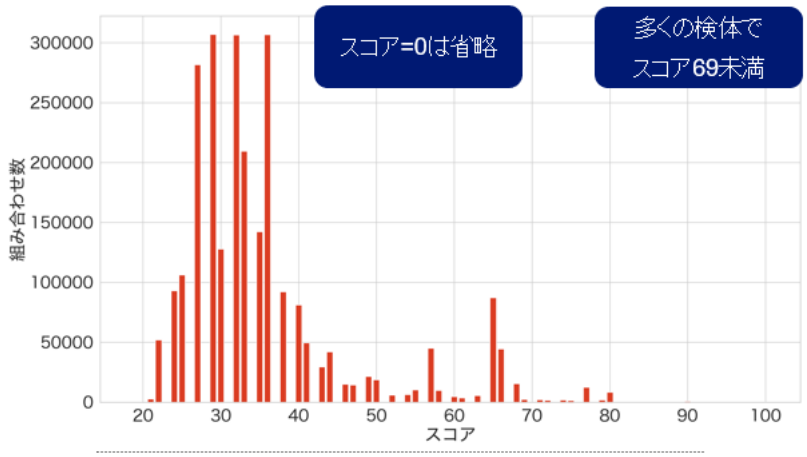


図 20 ファミリ外の検体の組み合わせにおける類似度スコアの分布

図から直感的に、閾値を 60～70 に設定すれば精度良くファミリの分類ができそうであることがわかりますが、より定量的な評価を試みます。ただし、実際は省略されている閾値=0 の組み合わせが圧倒的に多く、単純な正解率の平均の最大値を考えると閾値は限りなく 0 に近くなり、精度の良い分類が行えなくなります。

本評価では、両者のトレードオフを公平に評価するため、F-measure [13] を評価に用います。組み合わせの類似度スコアに基づいてファミリの分類を行い、本当にファミリ内であったかに注目すると、以下の 4 つのケースが考えられます。

- TP： 閾値以上で実際にファミリ内である組み合わせ
- TN： 閾値以下で実際にファミリ外である組み合わせ
- FP： 閾値以上で実際はファミリ外である組み合わせ
- FN： 閾値以下で実際はファミリ内である組み合わせ

図 21 は、これらのケースを図に示したものです。

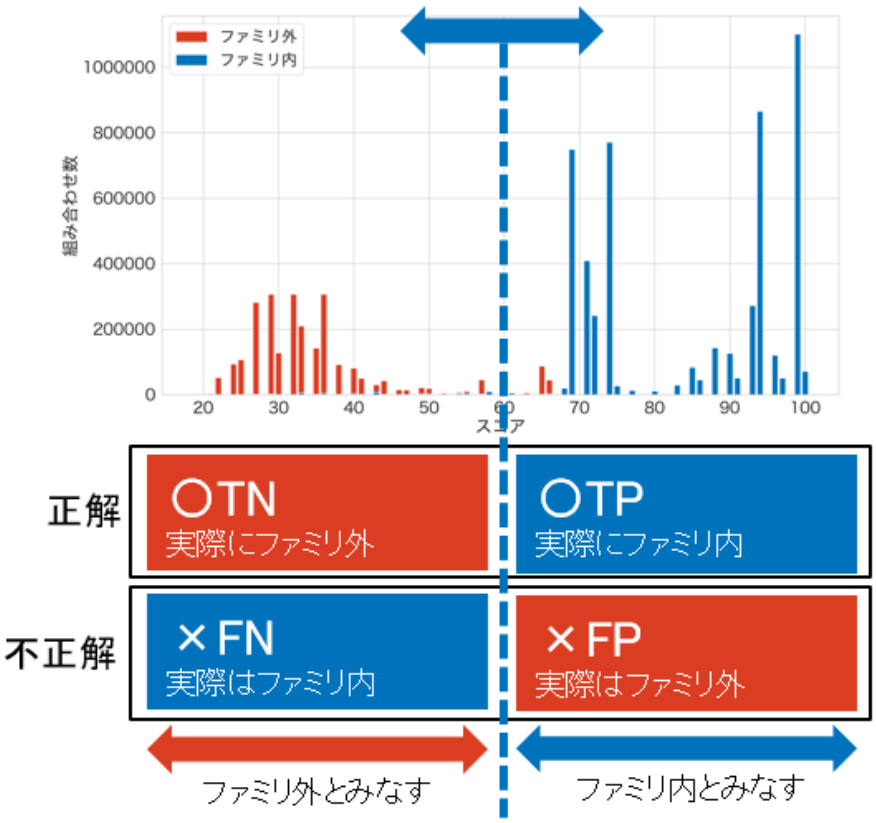


図 21 類似度スコアに基づいた分類のケース例

これら分類の正解、不正解を公平に評価する指標として、以下の 2 つの指標があります。

- 適合率（Precision： ファミリ内と分類したもののうち、本当にファミリ内だった割合)

$$\frac{TP}{TP + FP}$$

- 再現率（Recall： 実際はファミリ内である組み合わせを、正しくファミリ内として分類できた割合)

$$\frac{TP}{TP + FN}$$

この 2 つの指標の調和平均を取ったものが F-measure です。実際に計算すると以下の図 22 のような結果となり、閾値=68 で F-measure が最大となり、一番精度良くファミリの分類ができることがわかります。

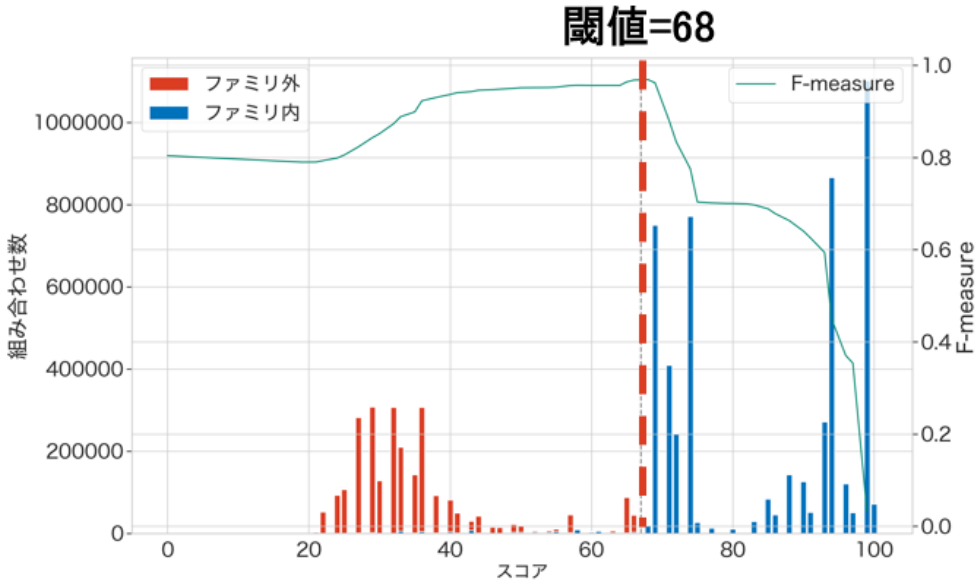


図 22 F-measure スコアの計算結果

6.3. 未解析の検体における評価

6.3.1. 検体の収集

本節では、2022 年 9 月から 3 カ月間、図 23 の YARA ルールにマッチする PE ファイルの Golang 製バイナリを収集しました。

```
rule go_language_pe
{
   strings:
      $go1 = "go.buildid" ascii wide
      $go2 = "go.buildi\\" ascii wide
      $go3 = "Go build ID:" ascii wide
      $go4 = "Go buildinf:"
      $go5 = "runtime.cgo"
      $go6 = "runtime.go"
      $go7 = "GOMAXPRO"
      $str1 = "kernel32.dll" nocase
   condition:
      uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and 2 of ($go*) and all of ($str*)
}
```

図 23 PE ファイルの Golang 製バイナリを分類する YARA ルール

6.3.2. 解析結果

収集した検体のうち、gimp を取得可能な検体が 7735 件でした。さらに、そのうち gimpfuzzy を計算可能な検体は 6372 件でした。ここで、収集した検体は UPX によるパッキングのみ YARA で検知し、機械的にアンパックを行っています。6372 件の検体を gimpfuzzy によりクラスタリングした結果、1093 個のクラスタが生成されました。

図 24 は VirusTotal 上の検体が最初に投稿された日付を基に、2022 年 9 月 1 日から、クラスタの出現順を可視化したものです。図の横軸は日付を示し、縦軸は検体の数を示します。また、各色のグラフは各クラスタを示しています。

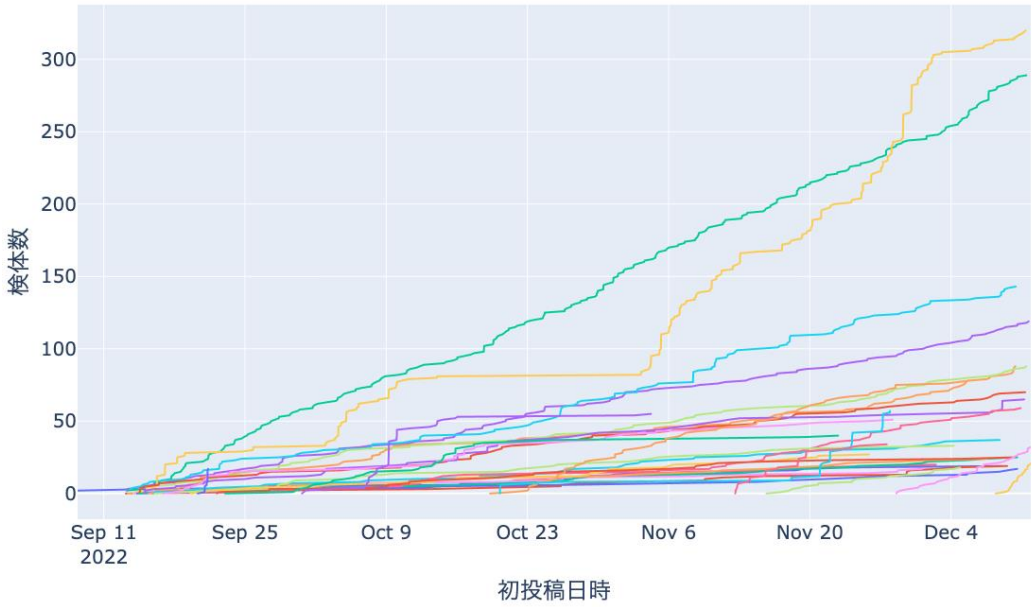


図 24 クラスタリングの結果と出現の様子

6.3.3. 出現するパッケージ名

gimp を抽出可能な 7735 件の検体から関数名を抽出すると、GitHub 上で開発、配布が行われているパッケージ名も復元が可能です。図 25 は、VirusTotal 上で悪性判定が 10 件以上ある検体が持つ関数の名前から、出現数が多い上位 20 件の GitHub のパッケージ名を抽出した結果です。全 806 件のリポジトリが抽出されました。

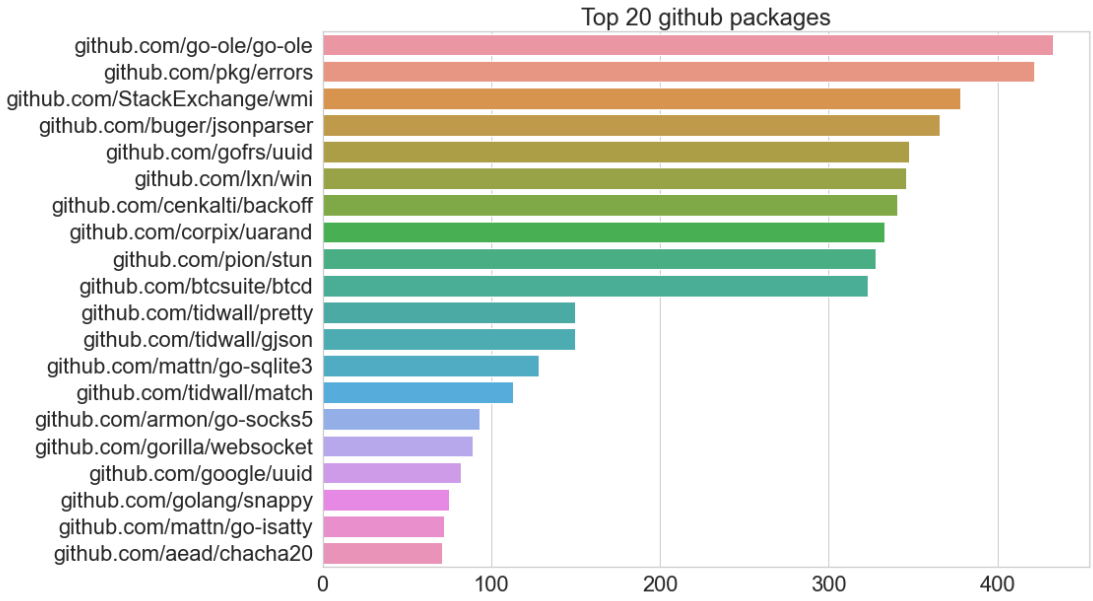


図 25 悪性検体に共通して出現するパッケージ

下記に観測されたパッケージの簡単な説明を列挙します。

- go-ole：COM を操作するパッケージ
- uarand：UserAgent をランダムにするパッケージ
- btcd：bitcoin にかかわるライブラリ
- mousetrap：Process Invoking を行うライブラリ

また、実際に抽出されたリポジトリ名を基に GitHub 上のリポジトリにアクセスしたところ、38 件のリポジトリで 404 レスポンスが返ってくることがわかりました。消去されたリポジトリ、フォークやコピーを含むプライベートリポジトリがこれらに該当すると考えられます。

7.ケーススタディ

本章では、6 章の評価で行った解析済み検体に対する分類、未解析検体に対するクラスタリングにおいて、特に gimpfuzzy が効果的に機能した事例について紹介します。また、gimpfuzzy による分類がどのように検体の特徴を捉えているかについて、クラスタリングにおける結合の順序をもとに考察していきます。

7.1. クロスプラットフォーム検体の分類

攻撃者が Golang を利用してマルウェアを作成する理由の一つに、クロスプラットフォームにバイナリをコンパイルできる点が挙げられます。幅広いプラットフォームを標的とする検体を容易に生成することが可能なため、解析者は異なる環境を標的とする検体でも同一の機能を持つ検体は同じファミリとして分類したいという需要があります。gimpfuzzy のハッシュ値の類似度を用いた分類では、こうした検体を同一のマルウェアファミリとして分類することが可能です。

例として、2019 年 5 月に発見された IPStorm と呼ばれる P2P のボットネット型マルウェアの検体を紹介します。表 1 は、異なる環境を標的とする 2 検体について、算出した gimphash、gimpfuzzy を示しています。同一のコードから複数のプラットフォームを標的とするような検体が生成された場合、従来の gimphash による分類では関数名の一部が異なると同一のハッシュ値が生成されないため、類似する検体と判断することは困難となります。それに対し、gimfuzzy を利用した分類では検体間の類似度が計算できるため、同一のソースコードから生成された検体を特定することが可能です。

今回の事例では gimpfuzzy の類似度が 90 であり、検体 X と Y は同一のマルウェアファミリである可能性が高いと判断することが可能です。表 2 は検体から抽出された関数名について、検体 X, Y で共通する割合を示したものです。全体の約 85%で関数名が一致しており、これにより gimpfuzzy の類似度が高い値になったことが推測されます。このように、gimpfuzzy では共通する関数名を基に類似度を計算することで、検体間の類似性を推測することが可能となります。

表 1 標的環境の異なる IPStorm の検体

	標的環境	gimphash	gimpfuzzy
X	Linux	4e92f61bb61e08947f457e73cbe72348c1dce312323a652397c85731144a8088	3072:qZosQIop4rzLYr62Xb+iwzh7RXnZWCy7IxXkzAEmZMpg1DtNg+rhHWPvXvAYiMsP:bA/KT7439mV/Wrz
Y	Mac	37da5b52a7c1577b7e0f3cfc99d059ffefe4def03ee840a8b9becd192ca79291	3072:lZosCIop4szLYr62Xb+iSzh7RXnZWCy7IxXyzAKmZMpg1DtNg+rhHWPvXvAYiMs0:1AQT7439mV/WrOt
		Not Matched	Similarity: 90

表 2 検体から抽出された関数

	# func
Common (X & Y)	5744
Only X	194
Only Y	817

次に、gimpfuzzy で測定される検体間の類似度の妥当性について検証します。図 26, 27 は、先ほど gimpfuzzy によって類似する検体と判断した検体 X, Y に対し、main 関数をディスアセンブルしてその結果を視覚的に示したものです。

Golang 製バイナリを実行すると main パッケージ内の main 関数が実行されるため、main 関数は検体の特徴を表していると考えられます。検体 X, Y はどちらもほぼ同一のコードとなっており、storm パッケージの starter 関数を呼び出すという main 関数の構造が類似しています。また、これは IPStorm の検体の特徴とも合致しており [14]、starter 関数に含まれるメインロジックが実行されていきます。

図 28 は Bindiff によって検体間の差分比較を行った結果を示しています。大多数の関数で類似度が高いことからも、検体 X, Y はほぼ同じ構造を有していることが推測されます。以上の通り、検体 X, Y は標的とするプラットフォームは異なりますが、同じ機能を持っていると考えられます。したがって、gimpfuzzy を利用することで、その

検体が標的とするプラットフォームの差異によらず、検体を正確に分類することが可能となります。

通常こうした bindiff を用いた解析はバイナリ間の比較に時間がかかるのに対し、gimpfuzzy による分類は検体間のハッシュ値の類似度を算出することで短時間での分類が実現可能です。特に、多くの Golang マルウェアを分類する状況において、gimpfuzzy による分類は非常に効果的な手法と考えられます。

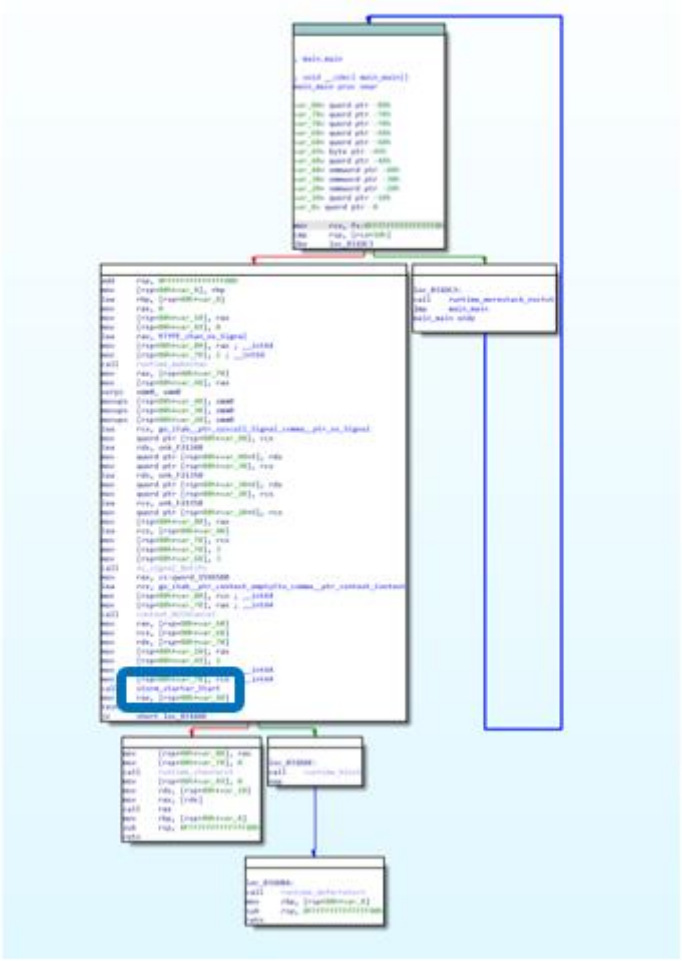


図 26 main 関数のディスアセンブル結果（X）

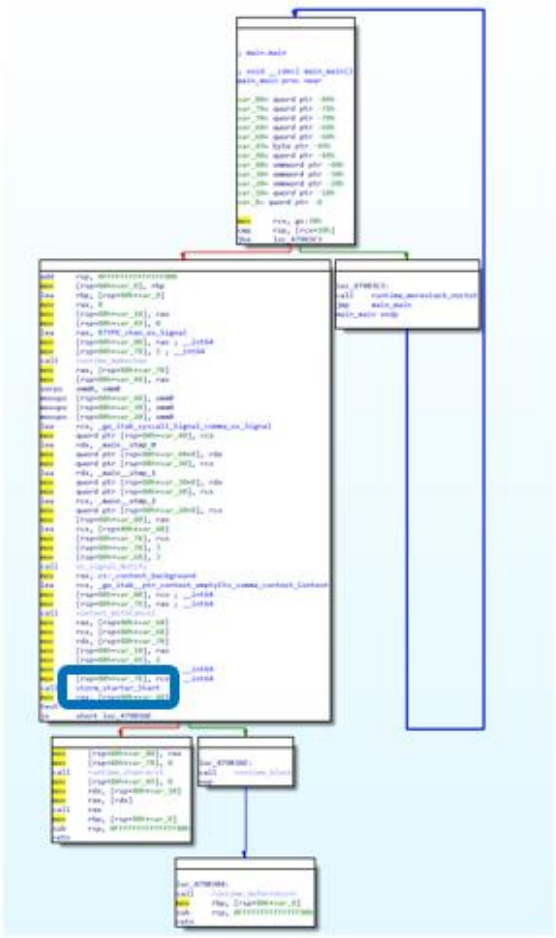


図 27 main 関数のディスアセンブル結果（Y)

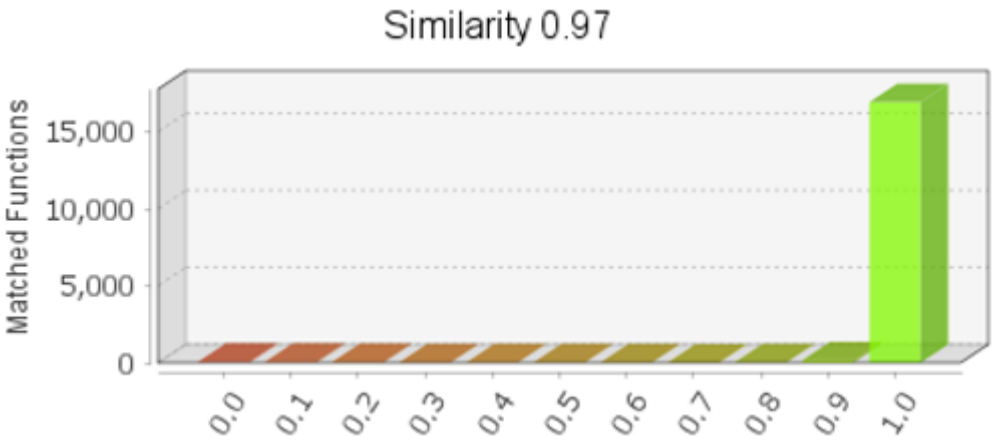


図 28 検体 A, B の bindiff

7.2. 悪性検体の機能変化の検出

gimpfuzzy によるクラスタリングは検体が利用する関数名の類似度に基づいて行われます。このクラスタリングをマルウェアに適用することにより、gimpfuzzy のハッシュ値によるマルウェアファミリの高速な分類が可能になります。

また、マルウェアファミリ内の機能の追加や挙動の変更を検知することは、マルウェアの効率的な解析のために有用であると言えます。しかし、従来は機能の追加や挙動の変更などを検知するのは動的解析などに頼らざるを得ず、時間がかかるものでした。

一方、gimpfuzzy によるクラスタリングは、関数名の類似度に基づくため、クラスタ内部の gimpfuzzy のハッシュ値の変化を検体ごとの活動時期データなどと組み合わせることによってマルウェアファミリ内の関数名の変更や関数の追加を追跡することができます。

例として、RanumBot と呼ばれるマルウェアファミリのクラスタを紹介します。この RanumBot クラスタの各検体はハッシュ値が異なるものの、二種類の gimpfuzzy のハッシュ値のみで構成されていました。

それぞれの gimpfuzzy のハッシュ値の VirusTotal への投稿日時ごとにグラフ化したのが図 29 になります。

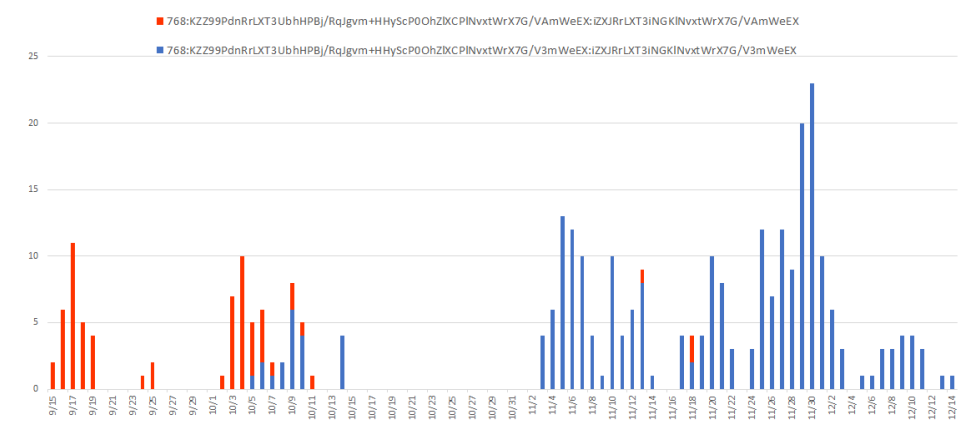


図 29 RanumBot の投稿数の変化

この図から、10 月初旬ごろから 2 つの gimpfuzzy のハッシュ値が入れ替わっていることがわかります。

次に、gimpfuzzy のハッシュ値の変化がどのような関数に由来しているのかを調査します。Bindiff を用いて検体間の差分比較を行ったところ、wincert.GetPostalCode という関数が wincert.Extract という名称に変更されていることがわかりました。gimpfuzzy のハッシュ値の変化はこの関数名の変更に起因すると考えられます。さらに、名前が変更されていない関数 main.reportInstallFailure にも機能の変更が行われていました。上記の関数以外は名称が変更されていないだけではなく、Similarity も高く、ほとんど変更が加えられていないと考えられます。

Similarity	Confidence	Address	Primary Name	Type	Address	Secondary Name	Type	Basic Blocks			Jumps		
0.26	0.98	006A4080	main_reportInstallFailure	No...	006A3DC0	main_reportInstallFailure	No...	0	10	58	4	9	84
0.45	0.97	0069DA10	main_getCampaignID	No...	0069D790	main_getCampaignID	No...	11	3	2	15	3	2
0.81	0.96	0067B610	application_pesignature...	Normal	0067B610	application_pesignature...	Normal	7	39	0	22	32	13
0.86	0.97	006B01B0	main_extractDistributor...	Normal	006B05E0	main_extractDistributor...	Normal	0	10	2	1	11	3
0.91	0.99	006943D0	main_initializeConfig	Normal	006941D0	main_initializeConfig	Normal	0	21	3	2	22	6

図 30 Bindiff による関数の比較

それぞれの関数に対し、どのような変更が行われているか解析を行ったところ、wincert.GetPostalCode/wincert.Extract には一部ロジックの変更が行われていることを確認しました。一方、さらに多くの変更が加えられていると思われる main.reportInstallFailure では通信機能が実装されるなど大幅な機能変更が確認できました。

```
234   v42 = (http_Request *)net_http_NewRequestWithContext(
235                           (int)&go_itab__ptr_context_emptyCtx_comma__ptr_context_Context,
236                           dword_C76630,
237                           (int)"POSTQEMU",
238                           4,
239                           (int)"https://fulusus.com/api/install-failure",
240                           39,
241                           v39,
242                           v40);
243   if ( !v43 )
244   {
245     Header = (runtime_hmap *)v42->Header;
246     v56 = net_textproto_CanonicalMIMEHeaderKey((int)"Content-Type", 12);
247     v55 = (_DWORD *)runtime_newobject((int)&RTYPE__1_string);
248     v55[1] = 33;
249     *v55 = "application/x-www-form-urlencoded";
```

図 31 挿入された通信機能

gimpfuzzy のハッシュ値の変更と main.reportInstallFailure の変更は同じタイミングで起きており、この時期に何らかのバージョンアップが行われていることがわかります。

以上の通り、マルウェアファミリ内部で gimpfuzzy のハッシュ値の変更を追跡することで、関数の変更を検出することが可能です。マルウェアファミリの分類のみではなく、マルウェアファミリ内部の変化を追跡するうえでも gimpfuzzy が効果的な手法であると考えられます。

7.3. 関数名が難読化された検体の分類

Golang マルウェアの解析妨害の一つに、関数名の難読化があります。特に、一部のマルウェアファミリは検体毎にランダムな文字列を関数名として使用しており、関数名から機能を推測することを妨害します。そのため、同一のソースコードから生成された検体であっても関数名が異なっており、gimphash では適切な分類ができません。このような状況においても、gimpfuzzy による分類が効果的に機能することがあります。

図 32 は、マルウェア作成ツール「veil」によって生成された検体に対して gimpfuzzy を計算する過程を示しています。左右でそれぞれ異なる検体から関数を抽出しており、これが Fuzzy Hashing の入力に利用されます。

この検体は main パッケージの関数名の一部が難読化されており、この難読化された文字列は検体ごとに異なります。したがって、gimphash の場合は同時期に作成された検体であったとしても、異なる関数名となるため同じハッシュ値が生成できず、分類精度が大きく低下します。

しかし、この難読化は検体のごく一部の関数に対して行われており、そのほかの大部分の関数名は難読化されていません。したがって、gimpfuzzy ではこの残りの関数名が一致することによって類似度が高くなり、同一のマルウェアファミリと判定することが可能となっています。

このように、関数名の一部が難読化されている検体に対して、Fuzzy Hashing が有効に機能することで gimpfuzzy による分類が効果的となります。

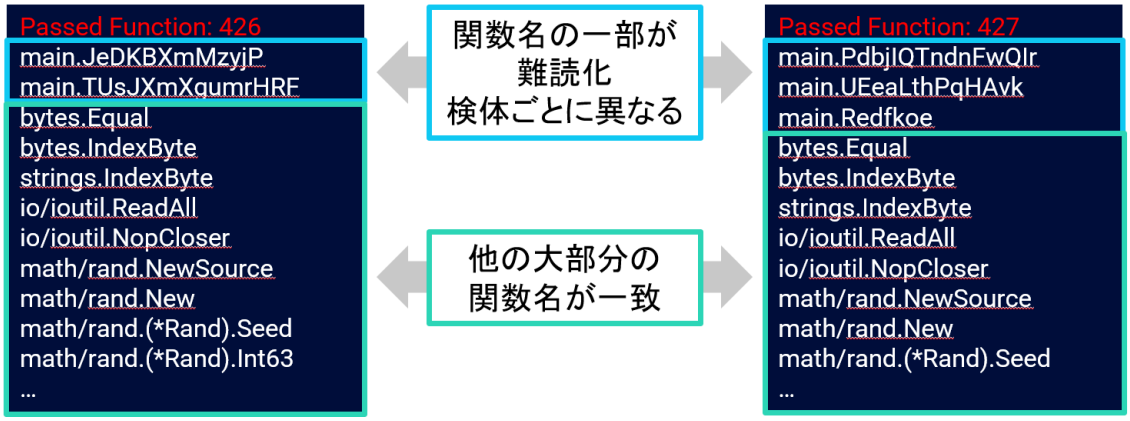


図 32 抽出された関数の比較

7.4. クラスタリングによる正規検体に偽造した悪性検体の発見

検体解析において、解析対象となるバイナリがどのような性質をもつか識別することは重要です。正規のプログラムのソースコードをベースに改造されたマルウェアが存在することは広く知られています。gimpfuzzy によるクラスタリングでは、関数名の類似する検体間でクラスタが形成されるため、正規のプログラムと、そのソースコードをベースにしたマルウェアの間でクラスタが形成されることが期待できます。

以下のクラスタは正規のプログラムと悪性検体を含む未解析検体群に対しクラスタリングを行った結果、生成されたクラスタです。

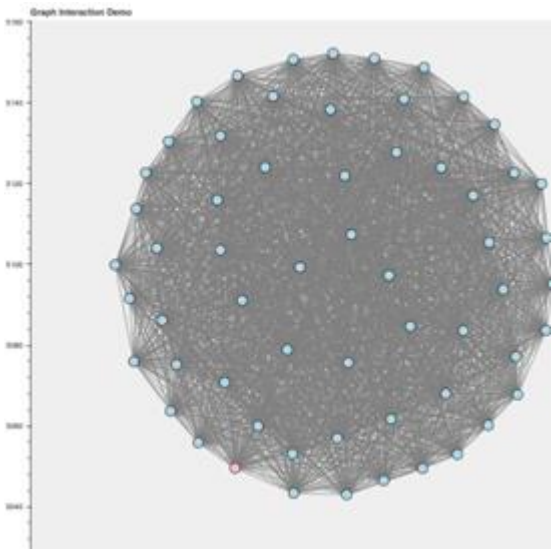


図 33 正規プログラムと悪性検体が混在するクラスタ

このクラスタを構成する検体のほとんどは良性検体ですが悪性検体が一件存在します。このクラスタの良性検体に対し解析を行ったところ、psiphone-tunnel という検閲回避ツールであることがわかりました。

一方、悪性検体は VirusTotal 上ではマルウェア Floxif と判定されています。悪性検体の解析を行ったところ、psiphone-tunnel-core とコードを共有しており、psiphone-tunnel に偽装した検体であると推定されました。

psiphone-tunnel と Floxif の関連性は報告されておらず、今回の調査によって初めて関連性が発見されました。通常、外部の Reputation 評価に頼らず、クラスタリングのみで正規プログラムと悪性検体を識別することは、このような正規プログラムに偽装するマルウェアが存在するために不可能です。しかし、悪性検体の機能・特徴の評価という観点からはクラスタリングという手法が有用であると言えます。

7.5. クラスタリングによる結合

最後に gimpfuzzy を利用したクラスタリングにおける、検体間の類似性について考察します。gimpfuzzy によるクラスタリングでは、算出された類似度に対する閾値を変更することで、クラスタ数を変化させて分類することができます。ここでは、どのような検体が同一のクラスタに分類されるのか、結合に影響を及ぼす要素について考察します。

分類する検体の対象には WellMess と呼ばれる Golang マルウェアの検体を使用します。WellMess は 2018 年にはじめて観測されたマルウェア[15]で、2020 年には APT29 がコロナワクチンの情報を狙った攻撃キャンペーンにおいて利用したことが報告されています[16]。

今回の調査ではハッシュ値の異なる WellMess の検体を 13 検体収集しました。各検体に対し gimphash を計算したところ 7 種類の値が計算されたため、各 gimphash の値に対し最も観測時期の早い検体を 1 つ選ぶことで、代表的な 7 検体のクラスタリングについて考察することにしました。表 3 はクラスタリングに使用した検体の情報を示しており、観測時期の早い方から順にアルファベットを割り当てています。

表 3 クラスタリングに使用した WellMess の検体

	観測時期	標的環境	gimphash	gimpfuzzy
A	2018/03	Windows	14821564ff52051b823c97f8aafb276787b8632438c165dcc5cf24b9a0e526aa	384:CR5Ma9P2nV0gWWRJgvmX+HHyyScP0OhZuv+rS0u4yssYDwSWK:fa9P2nV1RJgvmX+HHyyScP0OhZg1YD1x
B	2018/05	Linux	5662f393dc40199c4e908fa936b26fceca9594a4482813b8c2a62943ee7b4a57	384:XR5th9P2nVTNWWRJgvmX+HHyyScP0OhZkvzrFRG045ssYDwSWK:hh9P2nV7RJgvmX+HHyyScP0OhZue1YDr

C	2018/10	Windows	5662f393dc40199c4e908fa936b26fceca9594a4482813b8c2a62943ee7b4a57	384:XR5th9P2nVTNWWRJgvmX+HHyyScP0OhZkvzrFRG045ssYDwSWK:hh9P2nV7RJgvmX+HHyyScP0OhZue1YDr
D	2019/07	Linux	3402371b4ad89140fc2668fb65da61aa84fd79e5a426ad19814347a5fa353103	384:ZR5th9P2nVTNWWRJgvmX+HHyyScP0OhZxvzrhG045ssYDwSnC:Dh9P2nV7RJgvmX+HHyyScP0OhZs1YD1C
E	2020/02	Windows	c37d909a059e459d1181ce130d64a37502e3627c6cf6104fe83093c06d4fb6f7	384:md5oa9P2nVR/xtShr+WWRJgvmX+HHyyScP0OhZPvk0u4kbWZsaYDwSaq:1a9P2nVdrBRJgvmX+HHyyScP0OhZ6bWS
F	2020/04	Linux	64100a1bcb1fa7664a1eca52b4e1ef9cca05c4780efa5ded1adc844c7115ea13	384:od5th9P2nVV/xtSor+WWRJgvmX+HHyyScP0OhZLvZG04ubWGsaYDwSaT:Yh9P2nVhryRJgvmX+HHyyScP0OhZLbWg
G	2020/06	Windows	966066fea7eadb9b06d258882cd601d97fed8846e2a8f999adaeb9c5d9641949	384:6d5Kj59P2nVR/xtS0RJgvmX+HHyyScP0OhZYrLwk0u4kbWZsaYDwSGe+:r59P2nVdrlRJgvmX+HHyyScP0OhZ6bWL

図 34 は、gimphash, gimpfuzzy による分類を基にクラスタリングを行った際のクラスタ数を示しています。gimphash では 7 種類に分類されていた検体が、gimpfuzzy で閾値を変化させることでさらに少ないクラスタ数で分類できていることが分かります。また、類似度の閾値を下げることでクラスタ同士が結合していき、閾値 70 ではすべての検体が 1 つのクラスタに分類されています。

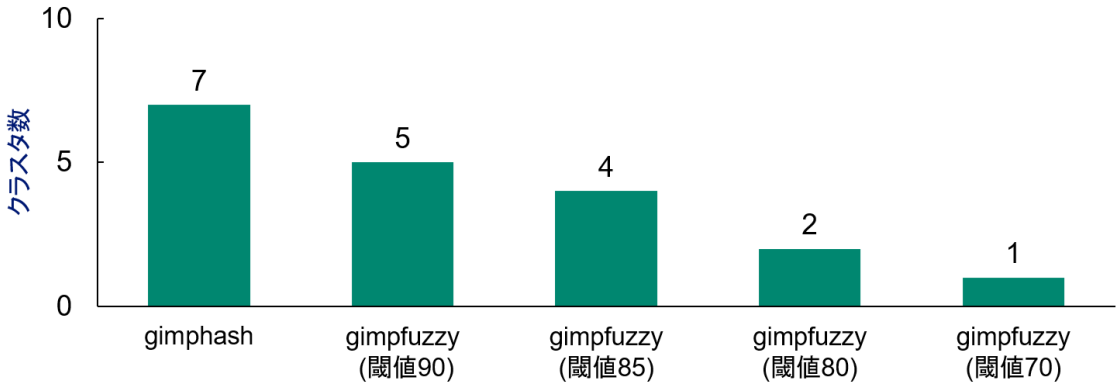


図 34 類似度の閾値とクラスタ数の関係

次に、gimpfuzzy による分類において類似度の閾値を変化させた際、クラスタがどのように収束していくか確認します。図 35 は 7 検体の gimpfuzzy における類似度を基にクラスタリングを構成した時のデンドログラム（樹形図）を示しています。

図中の数字は gimpfuzzy による類似度の値を示しており、上に行くほど閾値が低くなりクラスタの結合が進行していきます。

この図からは、閾値を下げた時にクラスタがどの順番で結合していくかを読み取ることができます。例えば閾値を 80 に設定した場合、クラスタは一方が A から D を含むもの、もう一方が E から G を含むものの 2 つに分類されます。つまり、観測された時期が 2020 年より前の検体によるクラスタ、2020 年以降の検体によるクラスタと、観測時期によってクラスタが分けられていることが分かります。

同様に、各検体が最初に結合する組み合わせ（A-C, B-D, E-G)に着目すると、観測時期が近い検体から先に結合していることが読み取れます。また、標的とするプラットフォームも結合に影響を与えていることが分かります。

前述の各検体が最初に結合する組み合わせに着目すると、同じ環境を標的とする検体の組み合わせほど類似度が高くなり、閾値が高い段階で結合が行われています。一

方で、閾値が下がるにつれ、同一のクラスタでも異なるプラットフォームを標的とする検体が含まれるようになります。このことから、標的プラットフォームの要素は観測時期ほど大きな影響力を持っていないことが分かります。

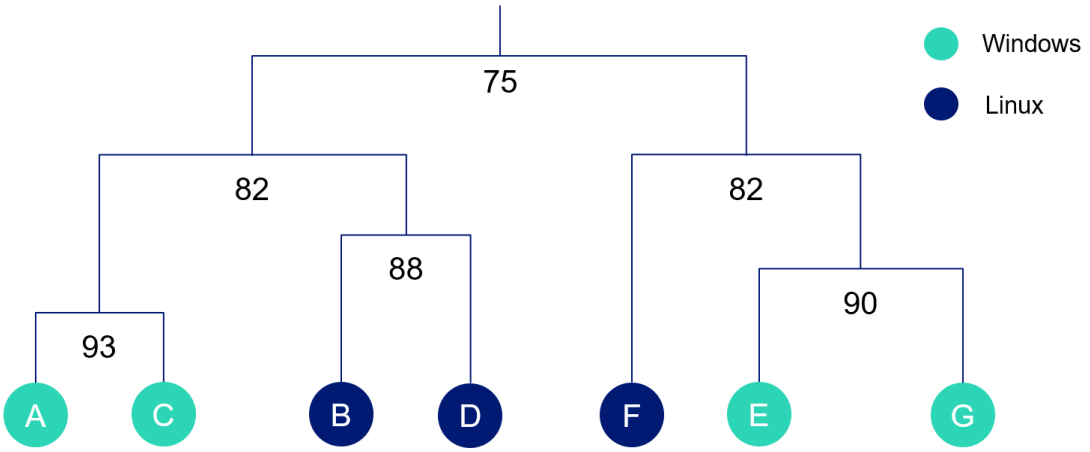


図 35 クラスタリングの過程

次に、各検体から抽出された関数の共通点と差異を確認します。図 36 は、各検体の main パッケージに含まれている関数名を列挙したものです。まず、Windows 環境を標的とする 4 検体（A, C, E, G）に含まれる関数に着目すると、観測時期が遅くなるにつれて関数の数も増加していることが分かります。追加された関数の名前から、攻撃者は検体の機能を追加していることが推測されます。

また、Windows 環境を標的とする検体と、Linux 環境を標的とする検体では、ほとんどの関数名が一致していますが、Linux 環境を標的とする検体のみ「getIP」という関数が存在していました。こうした差異が gimpfuzzy の類似度を変化させ、クラスタが結合する順番に影響を与えていると考えられます。

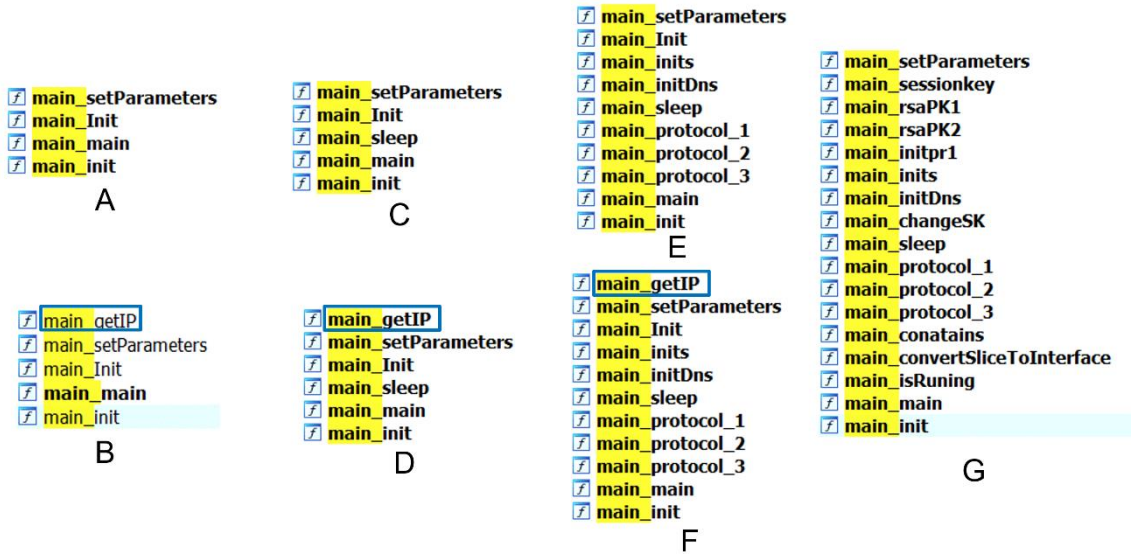


図 36 main パッケージの関数名

最後に、gimpfuzzy の計算において抽出される関数を用いて、検体間の共通点を確認します。7 検体から gimpfuzzy を計算する過程で、792 の関数名が抽出されました。表 4 は、その関数名がどの検体に含まれているかを算出しています。抽出された関数名全体の約 7 割は、すべての検体に含まれていました。このことから、gimpfuzzy ではこうした関数をマルウェアファミリ WellMess の特徴として捉えることで、他のマルウェアと区別して分類していることが分かります。また、標的とするプラットフォームが異なる検体や、機能が変更された検体であっても gimpfuzzy は検体を同一の特徴を持つ検体として正確に分類できます。

次に、ある特定の検体にのみ現れる関数に着目すると、標的とする環境ごと（Windows, Linux）に共通する関数、観測時期毎に共通する関数がそれぞれ確認されました。特に、関数の数を比較すると前者より後者の方が多くなっていることが分かります。これは、前述のクラスタリングの過程において、検体が標的とするプラットフォームによる差異よりも観測された時期の方が大きな影響をもたらすという内容を裏付けています。

表 4 検体から抽出された関数

	# func
Common	574
Windows (A, C, E, G)	3
Linux (B, D, F)	24
A, B	32
C – G	58
E – G	54
G	45
Others	22
Total	792

8. 課題と今後の展望

ここでは、gimpfuzzy を利用した分類における現在の課題を挙げます。

- フィルタ処理における情報の欠落

  gimpfuzzy ではハッシュ値を計算する際、抽出した関数のすべてをハッシュ関数の入力として利用するのではなく、一部の関数を除外する処理を行います。これは、多くの検体で利用される汎用的な関数を除外することで、より検体の特異性を表現したハッシュ値を生成するためです。しかし、これにより一部の検体では過剰に関数が削除されてしまい、検体固有の特徴が削除される可能性があります。この問題は gimphash においても議論されてきた課題です。現在の仕様は多数の Golang 製バイナリを分類、検証していく中で到達した実装であり、どの関数をフィルタ処理すべきか否かは gimphash, gimpfuzzy の根源的な課題となっています。今後、こうした指標による分類・探索を回避するようなテクニックが登場した場合、フィルタの実装を再考する必要が出てくる可能性があります。
- Fuzzy Hashing の制約

  gimpfuzzy では、Fuzzy Hashing として ssdeep を利用しています。ssdeep の制約として、入力のサイズが小さい場合には有意な結果が得られないとして正確な類似度が計算できなくなります。そのため、先の通り過剰に関数がフィルタされてしまった検体では、類似度の計算が行われず分類ができなくなってしまいます。
- 解析妨害されている検体

  今回の評価において扱った検体の中には、攻撃者によって解析妨害されており関数名の抽出に至らないものも確認されました。これらは AV 製品で Golang マルウェアの検知率が向上したため、さらなる検知回避テクニックを使用するようになったものと考えられます。観測した中には、独自の手法でパックされている検体や、関数名のほとんどが難読化されてしまっている検体が存在していました。攻撃者は解析妨害のためのツールを利用することで解析コストを上げたり、分類精度を下げたりすることが可能です。

また、今後の展望として以下の点が挙げられます。

- 大規模な Golang マルウェア分類への適用
  gimpfuzzy は高速にマルウェアの分類を行い、類似する検体をまとめることを可能とします。そのため、解析対象の検体に類似する検体を効率よく収集することで、解析に役立てることが可能です。また、その逆に検体すべてではなくクラスタごとに代表的な検体のみを解析することで、解析コストの削減につなげるといった使い方も考えられます。どちらの場合でも、これまで困難であった大規模な Golang マルウェアの分類を容易に実現し、効率の良い解析につなげることが可能となります。
- YARA モジュール化された gimpfuzzy の解析対象の拡大
  YARA と組み合わせることにより、gimpfuzzy は大規模な検体データベースに登録されている検体に対し、効率的な分類を行うことが可能です。今回公開した「go」「fuzzy」の 2 つの YARA モジュールは時間的制約のため、Windows で使用される PE のみを解析対象としています。今後、Mach-O や ELF といった Windows 以外のプラットフォームで使用される実行ファイルを順次解析対象に加えていくことで、クロスプラットフォーム検体の分類を効率化することが可能です。

9.おわりに

NTT セキュリティ・ジャパン株式会社の SOC では、インシデント発生の防止、インシデント発生時の早期発見のためのマルウェア解析やリサーチ活動を行っています。特に、Golang マルウェアに対して有効なアプローチについて、積極的なリサーチを行ってきました。

本稿では、近年増加し続けている Golang マルウェアについて、新たに提案したアプローチを実装し、その評価を行いました。

Golang 製のマルウェアは今後も日本の組織に対する攻撃でも引き続き利用されると考えられます。SOC では引き続き Golang 製のマルウェアについてリサーチを続けていくつもりです。

本稿で提案した gimpfuzzy を計算可能な YARA モジュールを Web サイトで公開していますので、ご活用いただければ幸いです。

10. 本レポートについて

レポート作成者

NTT セキュリティ・ジャパン株式会社

甘粕伸幸、澤部祐太、平尾早智澄、野村和也、小池倫太郎

履歴

2023 年 05 月 11 日（ver1.0）：初版公開

11. 参考文献

[1] mooncat-greenp, "Ghidra_GolangAnalyzerExtension", https://github.com/mooncat-greenpy/Ghidra_GolangAnalyzerExtension

[2] unixpickle, "gobfuscate", https://github.com/unixpickle/gobfuscate

[3] mooncat-greenp, "degobfuscate.py", https://github.com/mooncat-greenpy/Ghidra_GolangAnalyzerExtension/blob/master/ghidra_scripts/degobfuscate.py

[4] VirusTotal, “VirusTotal”, https://www.virustotal.com/

[5] abuse.ch, “MalwareBazaar”, https://bazaar.abuse.ch/

[6] Hatching, "Triage", https://hatching.io/triage/

[7] NextronSystems, "gimphash", https://github.com/NextronSystems/gimphash

[8] Mandiant, “Ready, Set, Go — Golang Internals and Symbol Recovery”, https://www.mandiant.com/resources/blog/golang-internals-symbol-recovery

[9] ssdeep Project, "ssdeep - Fuzzy hashing program", https://ssdeep-project.github.io/ssdeep/index.html

[10] Virus Total, "Yara - the pattern matching Swiss knife for malware researchers.", https://virustotal.github.io/yara/

[11] PaloAlto Unit 42, "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/

[12] devigned, "Veil", https://github.com/devigned/veil

[13] Sasaki, Yutaka. (2007). "The truth of the F-measure.", https://www.researchgate.net/publication/268185911_The_truth_of_the_F-measure

[14] Intezer, "A Storm is Brewing: IPStorm Now Has Linux Malware", https://intezer.com/blog/research/a-storm-is-brewing-ipstorm-now-has-linux-malware/

[15] JPCERT/CC, "Malware “WellMess” Targeting Linux and Windows", https://blogs.jpcert.or.jp/en/2018/07/malware-wellmes-9b78.html

[16] National Cyber Security Centre, "UK and allies expose Russian attacks on coronavirus vaccine development", https://www.ncsc.gov.uk/news/uk-and-allies-expose-russian-attacks-on-coronavirus-vaccine-development

12. 付録

- 検体のハッシュ値

	SHA-256
X	4f0add8eadb24a134b5cab6052920f576eec1bb39232c9548286a66883dcab82
Y	087f2ec8bbcee4091241e5ad30d449a1aecd0b9879338d072638c7d0ed6b30da
A	bec1981e422c1e01c14511d384a33c9bcc66456c1274bbbac073da825a3f537d
B	0b8e6a11adaa3df120ec15846bb966d674724b6b92eae34d63b665e0698e0193
C	d7e7182f498440945fc8351f0e82ad2d5844530ebdba39051d2205b730400381
D	7c39841ba409bce4c2c35437ecf043f22910984325c70b9530edf15d826147ee
E	8749c1495af4fd73ccfc84b32f56f5e78549d81feefb0c1d1c3475a74345f6a8
F	5ca4a9f6553fea64ad2c724bf71d0fac2b372f9e7ce2200814c98aac647172fb
G	4c8671411da91eb5967f408c2a6ff6baf25ff7c40c65ff45ee33b352a711bf9c