Malware Analyse

IM RAHMEN DER FORSCHUNGSWERKSTATT 1

- VON HENNING KRAUSE

HAW HAMBURG

Inhalt

Einführung

Statische Analyse

- “A Generic Approach to Automatic Deobfuscation of Executable Code”
- DREAM^{++}

Dynamische Analyse

- AMAL

Malware Images

Inhalt

Einführung

Statische Analyse

- “A Generic Approach to Automatic Deobfuscation of Executable Code”
- DREAM^{++}

Dynamische Analyse

- AMAL

Malware Images

Malicious Soft**ware** Allgemein

Viele Definitionen

- Schädlich, ohne Einverständnis des Nutzers, ohne Wissen des Nutzers

Arten

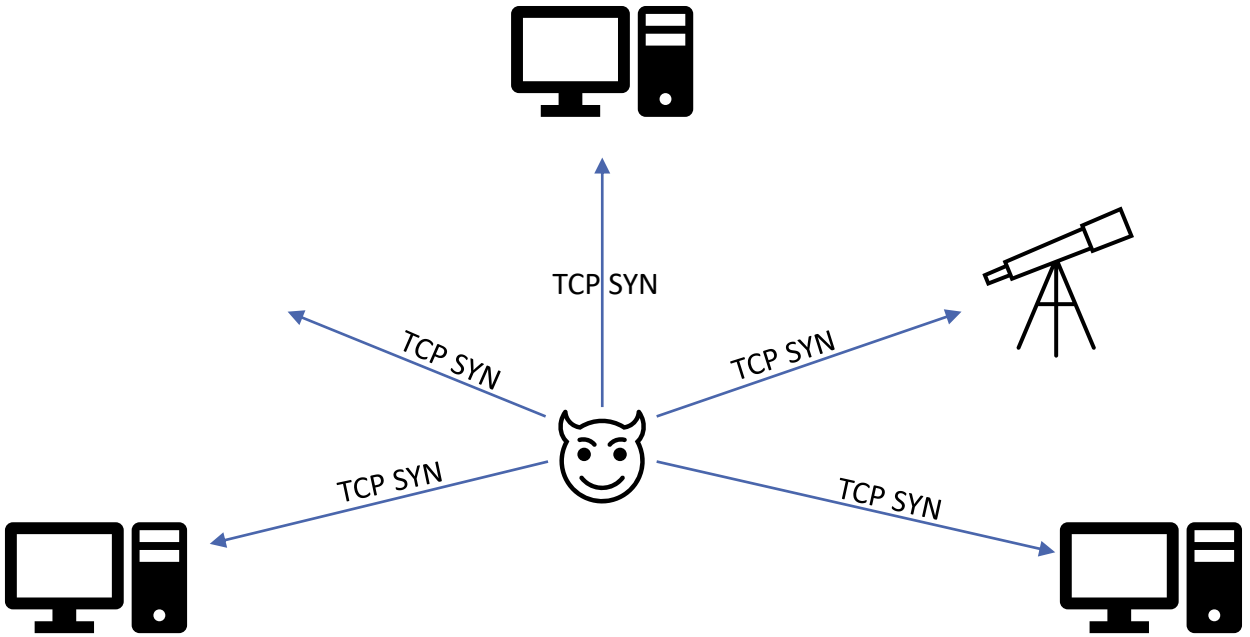
- Nach Typ: Trojaner, Würmer, Spyware, …
- Nach Verhalten: Informationsdiebstahl, Schwachstelle erzeugen, DoS, Kommandos ausführen

Kaspersky blockierte 33.412.568 einzigartige “malicious objects” 2020

Wer?

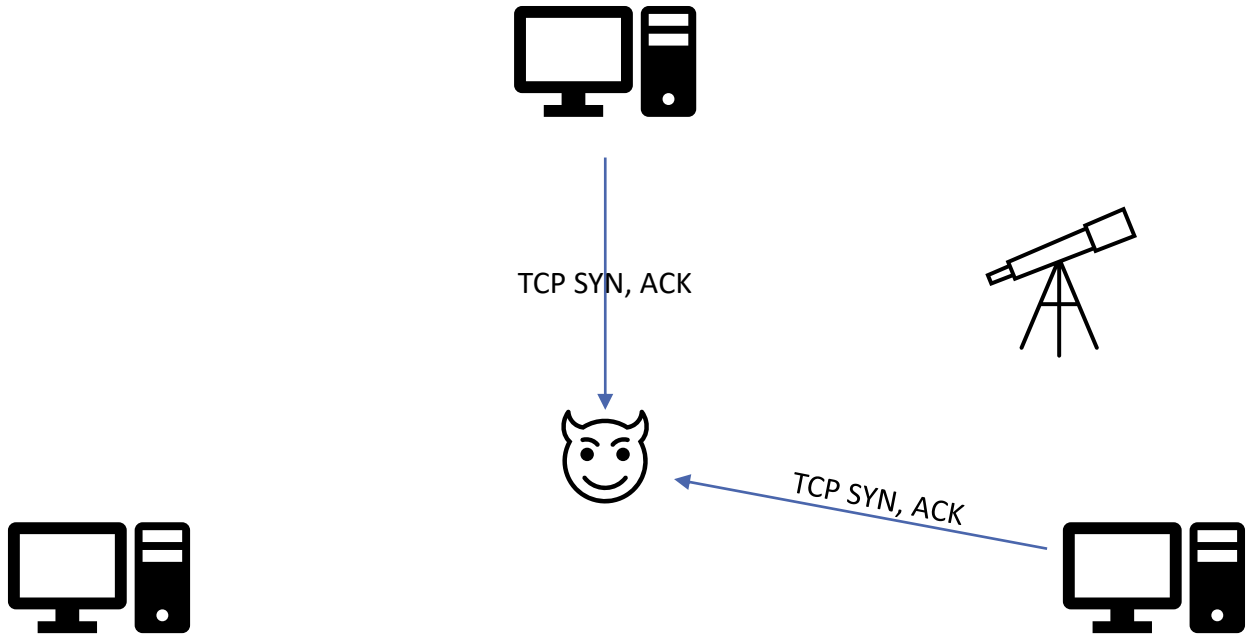
- Unklar – weit verfügbar: Toolkits zum Erstellen (z.B. Zeus, SpyEye)
- Hackerforen: „Malware as a service“

Woher kommt unsere Malware?



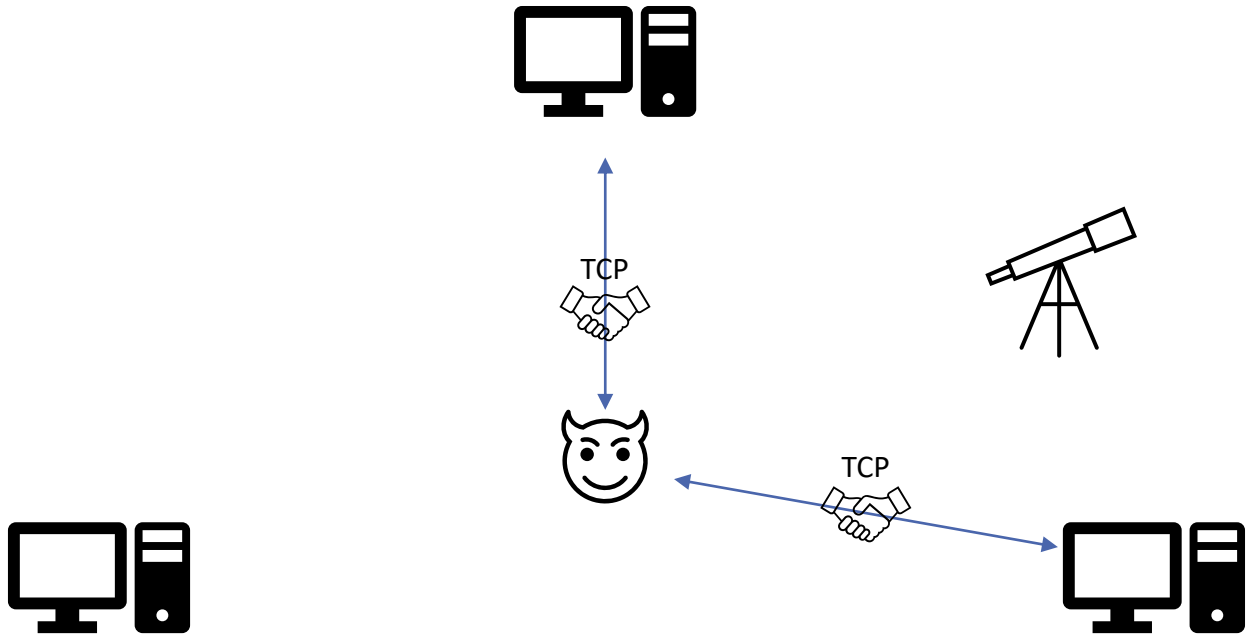
PHASE 1

Woher kommt unsere Malware?



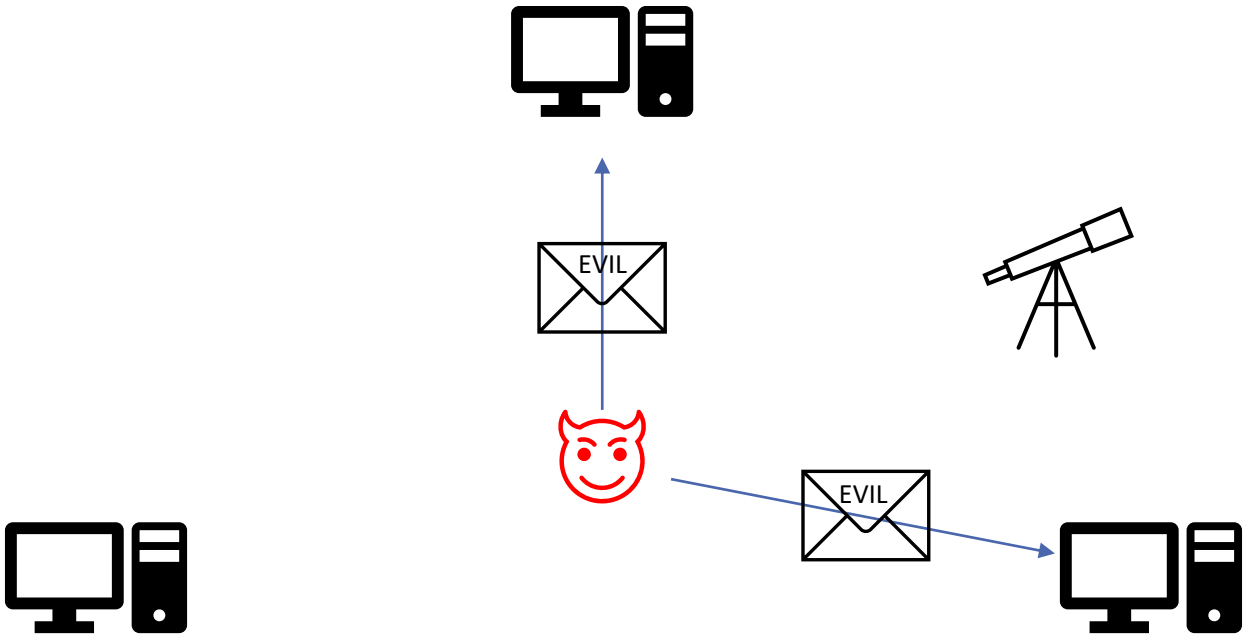
PHASE 1

Woher kommt unsere Malware?



PHASE 2

Woher kommt unsere Malware?



PHASE 2

Übersicht der Analysemethoden

Statisch

- Analysieren ohne Ausführung
- In Form von Binary oder Source Code
- Decompilation notwendig

Dynamisch

- Malware ausführen
- Verhalten analysieren während der Ausführung
- Umgebung zur Ausführung notwendig

Inhalt

Einführung

Statische Analyse

- “A Generic Approach to Automatic Deobfuscation of Executable Code”
- DREAM^{++}

Dynamische Analyse

- AMAL

Malware Images

Statische Analyse

Problem: Code Obfuscation

- Malware so schwer verständlich wie möglich machen → Ziel der Malware Autoren
- Codeabschnitte ersetzen durch semantisch identische, aber schwerer verständliche
  - Kontrollflusstransformationen
  - Data Flow Obfuscation
- Zusätzlichen Code hinzufügen, der keine Auswirkung auf Funktionsweise hat

Lösung: Deobfuscation

Generic Deobfusation

Ansatz: Semantik kann als Mapping von Input zu Output gesehen werden

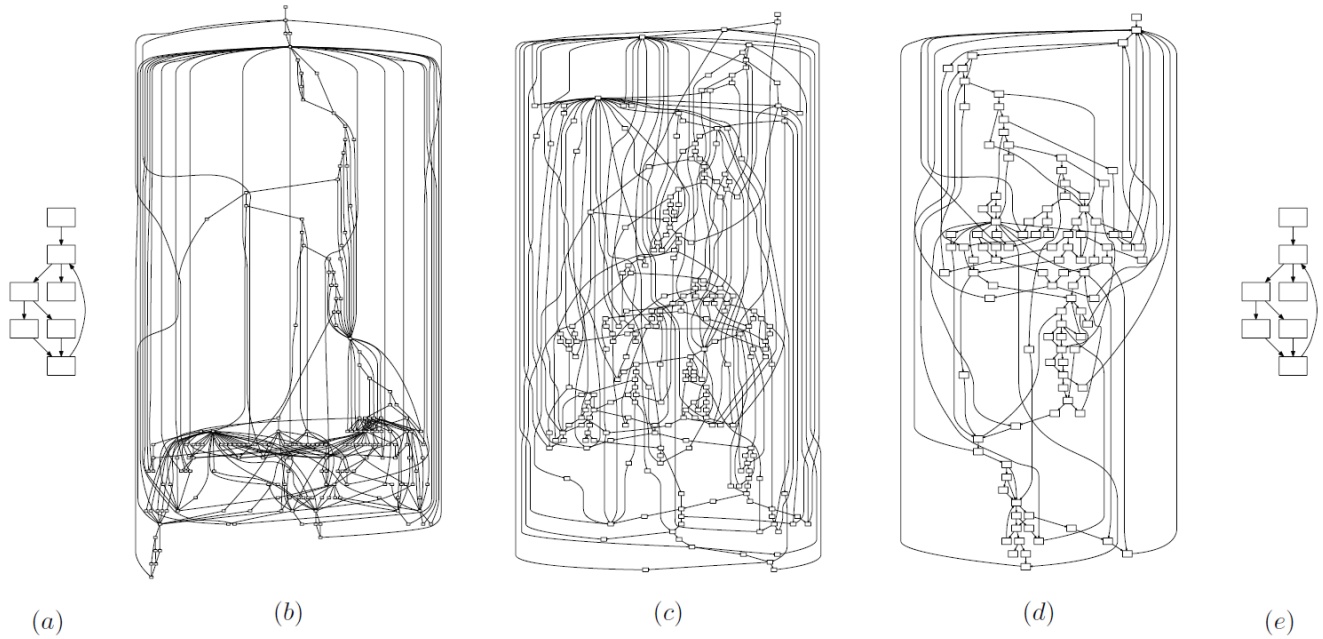
- Code vereinfachen, der dieses Mapping durchführt

Möglichst kein Annahmen über Art der Obfuscation machen

Prozess

- Input/Output identifizieren
- Forward taint propagation →Input durch Code folgen
- Code Vereinfachung → Semantikerhaltene Code Vereinfachungen
- Kontrollflussgraph erstellen → Genutzt um Kontrollfluss zu vereinfachen

Generic Deobfuscation



(a) (b) (c) (d) (e)

Key:

(a) Original program
(b) Obfuscated program
(c) Deobfuscation result: traditional byte-level taint analysis
(d) Deobfuscation result: bit-level analysis (taintedness information only)
(e) Deobfuscation result: enhanced bit-level analysis (taintedness + taint source information) (our algorithm)

Input Programm: Binäre Suche, Obfuscation durch ExeCryptor. Beispiel aus [1]

DREAM++

„Usability-optimised decompiler“

Basiert auf DREAM

- Ziel, diesen besser und Nutzerfreundlichkeit zu verbessern

Festgestellte Mängel:

- Komplexe Ausdrücke
  - Komplexe Logikausdrücke
  - Anzahl der Variablen
  - Pointerausdrücke
- Verworren Kontrollfluss
  - Doppelter Code
  - Komplexe Schleifenstrukturen
- Mangel an high-level Semantik

DREAM++

Was tun zur Vereinfachung? → 3 Kategorien

Ausdruckvereinfachung

- Z.B. Kongruenz Analyse → redundante Variablen entfernen

Kontrollflussvereinfachung

- Z.B. Schleifen vereinfachen (Compiler ändern durch Optimierung oft Struktur von Schleifen)

Semantik bewusste Namensgebung

- Z.B. Bedeutungsvolle Namen im Kontext: Zähler im Loop zu *i,j,…* umbennen

```
void *__cdecl sub_10006390(){
  __int32 v13; // eax@14
  int v14; // esi@15
  unsigned int v15; // ecx@15
  int v16; // edx@16
  char *v17; // edi@18
  bool v18; // zf@18
  unsigned int v19; // edx@18
  char v20; // dl@21
  char v23; // [sp+0h] [bp-338h]@1
  int v30; // [sp+30Ch] [bp-2Ch]@1
  __int32 v36; // [sp+324h] [bp-14h]@14
  int v37; // [sp+328h] [bp-10h]@1
  int i; // [sp+330h] [bp-8h]@1
  // [...]
  v30 = *"qwrtpsdfghjklzxcvbnm";
  v37 = *"eyuioa";
  // [...]
  v14 = 0;
  v15 = 3;
  if ( v13 > 0 )
  {
    v16 = 1 - &v23;
    for ( i = 1 - &v23; ; v16 = i )
    {
      v17 = &v23 + v14;
      v19 = (&v23 + v14 + v16) & 0x80000001;
      v18 = v19 == 0;
      if ( (v19 & 0x80000000) != 0 )
        v18 = ((v19 - 1) | 0xFFFFFFFE) == -1;
      v20 = v18 ? *(&v37 + dwSeed / v15 % 6)
                : *(&v30 + dwSeed / v15 % 0x14);
      ++v14;
      v15 += 2;
      *v17 = v20;
      if ( v14 >= v36 )
        break;
    }
  }
  // [...]
}
```

(a) Hex-Rays

```
LPVOID sub_10006390(){
  int v1 = *"qwrtpsdfghjklzxcvbnm";
  int v2 = *"eyuioa";
  // [...]
  int v18 = 0;
  int v19 = 3;
  if(num > 0){
    do{
      char * v20 = v18 + (&v3);
      int v21 = v18 + 1;
      int v22 = v21;
      int v23 = v21 & 0x80000001L;
      bool v24 = !v23;
      if(v23 < 0)
        v24 = !(((v23 - 1) | 0xfffffffeL) + 1);
      char v25;
      if(!v24)
        v25 = *(((dwSeed / v19) % 20) + (&v1));
      else
        v25 = *(((dwSeed / v19) % 6) + (&v2));
      v18++;
      v19 += 2;
      *v20 = v25;
    }while(v18 < num);
  }
  // [...]
}
```

(b) DREAM

```
LPVOID sub_10006390(){
  char * v1 = "qwrtpsdfghjklzxcvbnm";
  char * v2 = "eyuioa";
  // [...]
  int v13 = 3;
  for(int i = 0; i < num; i++){
    char v14 = i % 2 == 0 ? v1[(dwSeed / v13) % 20]
                          : v2[(dwSeed / v13) % 6];
    v13 += 2;
    v3[i] = v14;
  }
  // [...]
}
```

(c) DREAM++

DREAM++

Inhalt

Einführung

Statische Analyse

- “A Generic Approach to Automatic Deobfuscation of Executable Code”
- DREAM^{++}

Dynamische Analyse

- AMAL

Malware Images

Dynamische Analyse

Malware auf Testumgebung ausführen → Verhalten beobachten

Wichtig: Testumgebung darf nicht kompromittiert werden!

Framework Bestandteile:

- Malware sample
- Hardware und OS
  - Richtiger PC, VM, …
- Analysetool

Techniken:

- Funktionsaufrufanalyse
- Ausführungskontrolle → Debugging
- Informationsfluss Verfolgung
- Tracing → zurückgelassene Informationen analysieren
- Side-channel Analyse:

AMAL

“An operational and large-scale behaviour-based solution for malware analysis and classification”

2 Subsysteme

- AutoMal
- MaLabel

AMAL – AutoMal

VM genutzt als Umgebung zur Ausführung der Samples

Komponenten

- Sample submitter
  - Samples bereitstellen für AutoMal
  - Priorisierung möglich
- Controller
  - Strukturiert Haupprozess
  - Samples holen und einer freien VM bereitstellen
  - VM konfigurieren
  - Artefakte einsammeln (z.B. Registry, Netzwerk, Volatile Memory,…)
- Workers
  - Die tatsächlichen VMs
  - Unabhängig von Controller
- DB
  - Aufbewahrung der Artefakte

AMAL – MaLabel

Classification and Clustering

Basierend auf den Features, die durch AutoMal gesammelt wurden

Viele Algorithmen implementiert

- Z.B. SVM, decision trees, k-nearest-neighbor

Dementsprechend die Samples labeln

Inhalt

Einführung

Statische Analyse

- “A Generic Approach to Automatic Deobfuscation of Executable Code”
- DREAM^{++}

Dynamische Analyse

- AMAL

Malware Images

Malware Images

Prozess

- Binary kann als String aus 1 und 0 gesehen werden
- Diese als 8-Bit Vektor einlesen
- Transformieren in 2D Matrix und als Bild darstellen
- Die Höhe des Bildes ändert sich mit der Größe der Datei

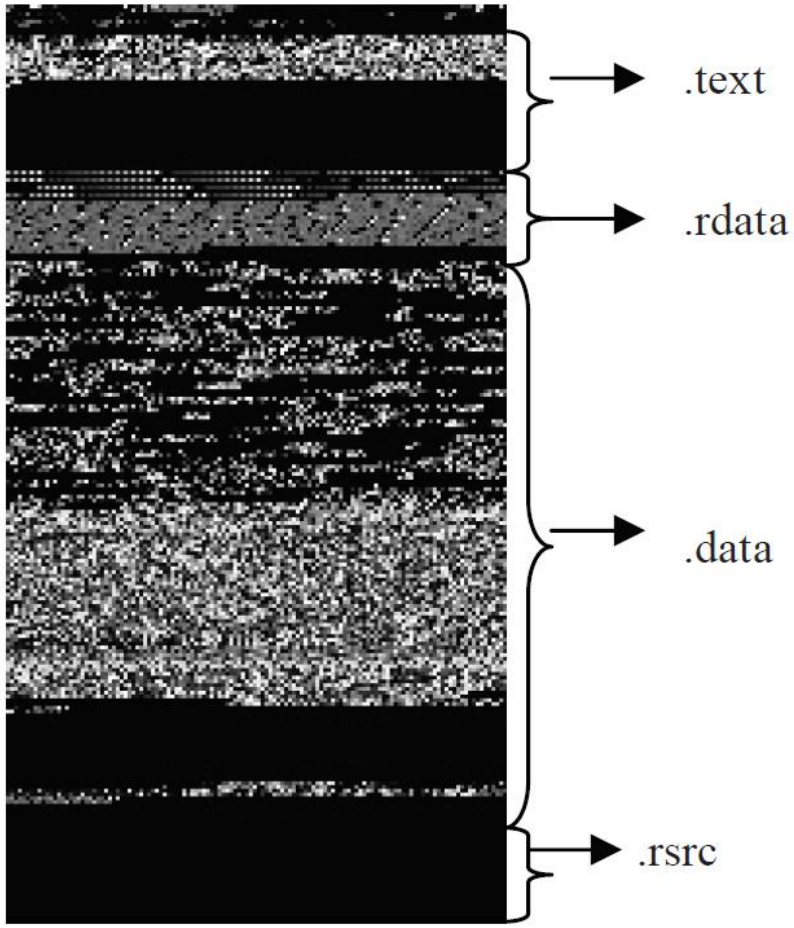


Fig. 2 Various Sections of Trojan: Dontovo.A

Malware Images

Malware Images

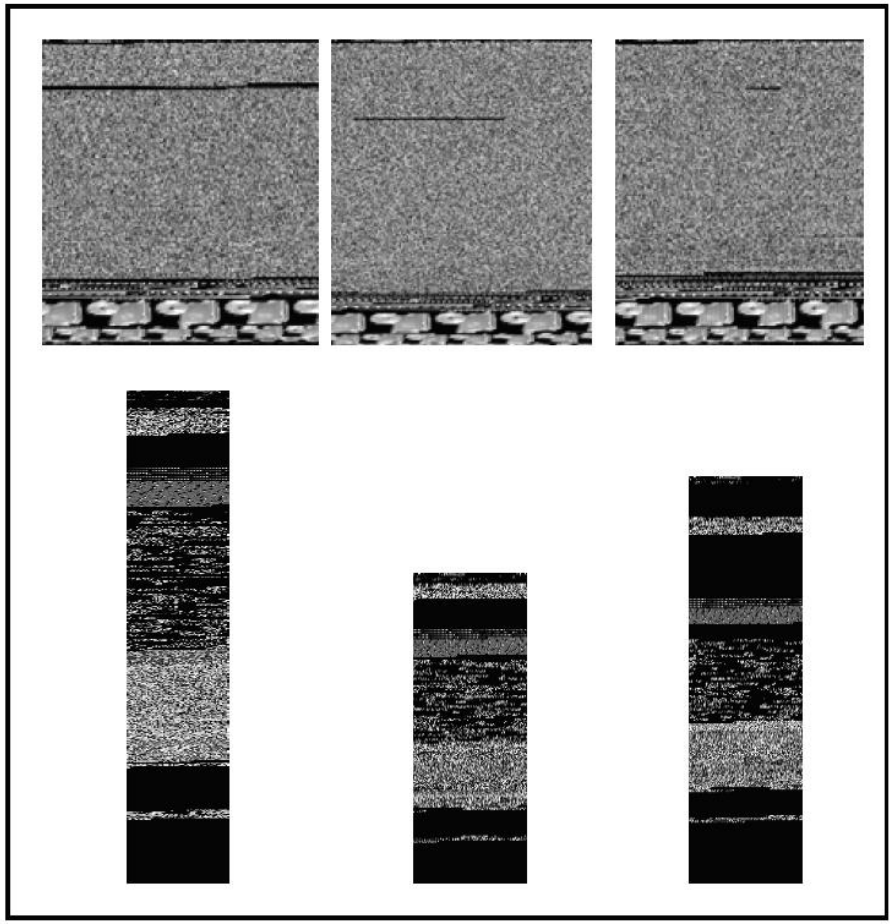


Fig. 3 The images in the first row are images of 3 instances of malware belonging to the family Fakerean [26] and those in the second row belong to the family Dontovo.A [26].

Schlussgedanken

Viele Ansätze

- Alle haben Vorteile und Nachteile

Erfolg hängt immer davon ab, wie sie angewendet werden

Als nächstes:

- Einen Ansatz auswählen
- Ggf. nach weiteren, sehr aktuellen Implementationen suchen
- Umsetzung

Vielen Dank für Eure Aufmerksamkeit!

FRAGEN?

Quellen

(1) B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In 2015 IEEE Symposium on Security and Privacy. IEEE, San Jose, CA, USA, 674–691. https://doi.org/10.1109/SP.2015.47

(2) K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. 2016. Helping Johnnyto Analyze Malware: A Usability-Optimized Decompiler and Malware AnalysisUser Study. In2016 IEEE Symposium on Security and Privacy (SP). IEEE ComputerSociety, San Jose, CA, USA, 158–177. https://doi.org/10.1109/SP.2016.18

(3) Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. 2015. AMAL: High-fidelity,behavior-based automated malware analysis and classification.Computers &Security52 (2015), 251–266. https://doi.org/10.1016/j.cose.2015.04.001

(4) L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. 2011. Malware Images:Visualization and Automatic Classification. InProceedings of the 8th InternationalSymposium on Visualization for Cyber Security(Pittsburgh, Pennsylvania, USA)(VizSec '11). Association for Computing Machinery, New York, NY, USA, Article4, 7 pages. https://doi.org/10.1145/2016904.2016908