

# TAITS - Industrie-Ransom 2.0: Evaluation der Anwendung von YARA zur Erkennung netzwerkbasierter Ransomware

Bernhard Birnbaum

**Zusammenfassung**—Im Zentrum dieses Projekts steht die Fragestellung, inwiefern Auswirkungen von netzwerkbasierter Ransomware m.H. des YARA Frameworks detektiert werden können. Weiterhin soll daraus die Eignung von YARA als Bestandteil von Next-Gen-Firewalls im industriellen Umfeld beurteilt werden.

**Index Terms**—IT-Security, Industrial Control Systems, Ransomware, YARA, Open Source



## 1 MOTIVATION

In der Industrie werden zur Steuerung von Anlagen und Prozessen sogenannte *Industrial Control Systems* (ICS) eingesetzt. Ein wesentlicher Bestandteil eines ICS ist ein Netzwerk, welches die verschiedenen Komponenten (Hosts) eines Industriesystems miteinander verbindet. Diese internen Netze sind immer wieder Ziel von Ransomware-Angriffen. Dadurch entstehen zum einen durch Erpressung von Lösegeldsummen finanzielle Schäden für Unternehmen, zum anderen auch Ausfälle von Dienstleistungen und Versorgung (besonders durch Angriffe auf kritische Infrastruktur) [1].

Um Malware und potentiell gefährliche Auffälligkeiten zu erkennen, nutzen Virens Scanner regelbasierte Detektionssysteme wie das **YARA Framework** [2]. Daraus ergibt sich die Fragestellung ob YARA auch dazu genutzt werden kann, Signaturen oder verdeckte Kommunikation im Netzwerkdatenstrom (speziell im industriellen Umfeld eines ICS) zu erkennen.

Die zugrundeliegende Aufgabenstellung [3] stammt aus der LV „IT-Security of Cyber-Physical Systems“ der AG AMSL. Dieses Thema wird gefördert durch das Projekt SMARTEST2 [4].

## 2 STAND DER TECHNIK

### 2.1 ICS Protokolle

Zur Koordination innerhalb des ICS wird ein Kommunikationsprotokoll (i.d.R. *Modbus* oder *OPCUA*) verwendet.

**Modbus** ist ein Protokoll welches nach dem Client-Server-Modell arbeitet und sich im Laufe der Zeit zu einem der meistgenutzten Protokolle im industriellen Umfeld entwickelt hat [5]. Da es für die interne Kommunikation zwischen einzelnen Rechnern in Industrie-Netzen entworfen wurde, welche nicht mit dem Internet verbunden sein sollten, unterstützt Modbus in seiner Urform (von 1979) keine Sicherheitsmaßnahmen wie z.B. Verschlüsselung. Trotz der Veröffentlichung einer sicheren Modbus-Variante im Jahr 2018 (m.H. von TLS) laufen in vielen Anlagen noch immer

unsichere Implementierungen, wodurch Modbus weiterhin anfällig für Ransomware-Angriffe ist [6].

Das **OPCUA**-Protokoll ist ein Standard für den Datenaustausch in Systemen der Industrie 4.0 [7]. Heutzutage gehört es zu den wichtigsten Kommunikationsstandards in diesem Umfeld. Im Gegensatz zu Modbus wird das Signieren sowie das Verschlüsseln von Nachrichten direkt unterstützt, wodurch die Integrität sowie Vertraulichkeit der ausgetauschten Daten sichergestellt werden kann.

### 2.2 YARA

Das YARA Framework ist ein von VirusTotal entwickeltes Werkzeug [8] zur Identifikation und Klassifikation von Malware. Dazu nutzt es textuelle sowie binäre Patterns, die m.H. von logischen Ausdrücken zu Regeln zusammengestellt werden können. **YARA-Regeln** [9] werden in einer eigens dafür entwickelten Beschreibungssprache verfasst.

Für einige Sachverhalte kann es sein, dass der Funktionsumfang von YARA nicht ausreichend ist. In diesem Fall ist vorgesehen, dass **YARA-Module** [10] eigenständig entwickelt und bei der Kompilierung von YARA eingebunden werden. Module ermöglichen die Implementierung von komplexeren Funktionalitäten, die dann in YARA-Regeln eingebunden und genutzt werden können.

### 2.3 Datensets

Ausgangspunkt dieser Arbeit stellen mehrere Netzwerk-mitschnitte aus verschiedenen Umgebungen dar. Ein Großteil der untersuchten Modbus-Mitschnitte stammt aus einem internen Repository der AG AMSL [11], einer Sammlung von PCAP-Dateien aus mehreren vorangegangenen Arbeiten [12] [13]. Weiterhin wurden Mitschnitte von Ransomware-Angriffen auf das OPCUA-Protokoll aus der Abschlussarbeit von Emirkan Toplu [14] untersucht, welche im Labor der AG AMSL erzeugt wurden.

### 3 KONZEPT

#### 3.1 Werkzeugauswahl

**YARA-Python** [15] ist die offizielle Python-Schnittstelle des YARA Frameworks, wodurch Regeln direkt m.H. von Python auf Daten angewandt werden können.

**Scapy** [16] ist ein interaktives Tool und eine Python-Bibliothek, wodurch u.a. der Netzwerkdatenstrom mitgelesen bzw. zuvor aufgezeichnete Netzwerkmitschnitte ausgelesen werden können. Dadurch wird das Handling von Netzwerkpaketen mit Python stark vereinfacht.

**Wireshark** [17] ist ein Werkzeug zur Analyse und Aufzeichnung von Netzwerkdatenströmen. Für diese Arbeit ist das Tool essentiell, da zur Ableitung von YARA-Regeln ein tiefer Einblick in die Pakete notwendig ist um spezifische Patterns, Werte und Offsets auszulesen.

Die Implementierung wurde m.H. von **Docker** [18] realisiert, um Kompatibilitätsprobleme mit der Python- bzw. Scapy-Version zu vermeiden. Da YARA für die Einbindung der Module vor der Installation kompiliert werden muss ist eine Virtualisierung sinnvoll, damit das Host-System unangetastet bleibt.

#### 3.2 Methodik

Das YARA Framework ist vorrangig auf die Analyse von schädlichen Dateien oder Prozessen ausgelegt. Außerdem ist das Betrachten von einzelne Netzwerkpaketen nicht immer ausreichend, um schädliches Verhalten von Ransomware erkennen zu können, da viele Angriffsvektoren komplexer sind. Aus diesen Gründen ist es notwendig einen Wrapper um das YARA Framework zu bauen, um den Netzwerkdatenstrom verarbeiten zu können. In Abb. 1 ist die im Folgenden beschriebene Methodik zur Aufbereitung des Netzwerkdatenstroms für die Betrachtung mit YARA schematisch dargestellt.

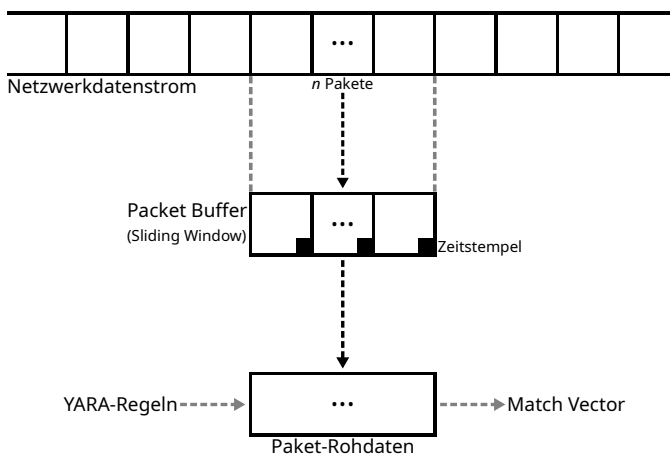


Abbildung 1. Darstellung der Aufbereitung des Netzwerkdatenstroms zur Anwendung von YARA-Regeln

Grundsätzlich sieht der in dieser Arbeit vorgestellte Ansatz einen **Sliding-Window-Mechanismus** vor (nachfolgend **Packet Buffer** genannt), um einen entsprechenden Kontext im Datenstrom zu schaffen und betrachten zu können. Somit können, unmittelbar nachdem ein neues Netzwerkpaket in das Fenster gerückt ist, auf die Rohdaten

der Pakete innerhalb des Puffers alle bereitgestellten YARA-Regeln angewendet werden.

Der Packet Buffer hat eine maximale Größe von  $n > 0$  Paketen, sodass zu jedem Zeitpunkt (nach anfänglicher Befüllung) die letzten  $n$  Pakete des laufenden Netzwerkdatenstroms im Fenster liegen. Die konkrete Größe von  $n$  wird maßgeblich von dem zu attributierenden Angriffsvektor, aber auch von der genutzten ICS-Umgebung bestimmt.

Das Konzept sieht zudem vor, hinter den Rohdaten jedes Netzwerkpakets einen Zeitstempel zu hinterlegen, da diese Information nicht zwingend in den Paketen selbst enthalten ist. Dies ist notwendig, um bestimmte Auswirkungen von Ransomware auf den Netzwerkdatenstrom validieren zu können, beispielsweise eine zeitliche Abweichung im Polling-Intervall eines Modbus-Systems.

#### 3.3 Untersuchte Auswirkungen von Ransomware-Angriffen auf ICS

Für diese Arbeit wurden einige Auswirkungen von Angriffen auf ICS ausgewählt, die im weitesten Sinne in Verbindung mit Ransomware stehen, aber nicht zwingend müssen. Folgende Sachverhalte wurden auf Handhabbarkeit mit YARA überprüft:

- **Flooding:**

Flooding-Angriffe sind eine Art von *Denial-of-Service* (DoS) Angriffen, um ein (Teil-)System gezielt zu überlasten. Abhängig von den im ICS genutzten Kommunikationsprotokollen können Flooding-Angriffe beispielsweise über Modbus (*Modbus-Query-Flooding*), aber auch über TCP (*TCP-SYN-Flooding*) erfolgen.

- **LSB-Stego Covert-Channel:**

Ransomware-Angriffe sind in den letzten Jahren zunehmend komplexer geworden, weshalb davon auszugehen ist, dass Angreifer in Zukunft auch fortgeschrittene Methoden zur verdeckten Kommunikation (z.B. C2-Kanäle) in Netzwerken nutzen werden [19]. Konkret können beispielsweise in Modbus-Registerwerten m.H. von *Least-significant-Bit-Steganographie* (LSB) unbemerkt Informationen ausgetauscht werden. Die Detektion solcher Covert-Channels ist i.d.R. relativ schwierig, da die Werte nur leicht verändert werden und deshalb normalerweise im Rahmen der erwarteten Größenordnung liegen.

- **Werteanomalien:**

Das Auslesen von den über die internen ICS-Protokolle übertragenen Werte zwischen den Teilsystemen kann als ein wichtiger Kontrollmechanismus gesehen werden. Sollte ein ICS-System beispielsweise bereits mit einer Ransomware infiziert oder Pakete durch einen *Man-in-the-Middle-Angriff* (MITM) manipuliert worden sein, können m.H. solcher Kontrollen auffällige oder schädliche Anweisungen im Besten Fall erkannt und die betreffenden Pakete entfernt werden.

- **Verschlüsselung:**

Ein Merkmal von Ransomware-Angriffen ist die Verschlüsselung von Daten. Werden Informationen innerhalb des Netzwerkdatenstroms von ICS verschlüsselt, kann dies zum Absturz von Clients führen [14]. Durch frühzeitige Erkennung und Aussortieren der betroffenen Pakete könnte dies verhindert werden.

- **MITM durch ARP-Spoofing:**

Bei *ARP-Spoofing* wird die MAC-Adresse von ARP-Paketen durch die eines anderen Gerätes ausgetauscht, was zu einer verfälschten Assoziation zwischen MAC- und IP-Adresse führt. Dadurch wird der Traffic an eine falsche IP-Adresse umgeleitet, was zur Manipulation von Werten genutzt werden kann. Die frühzeitige Erkennung ist deshalb wichtig, um das weitere Fortschreiten des Angriffs zu unterbinden.

- **Paketabwesenheit:**

Sollte ein Angreifer den Traffic innerhalb eines ICS umleiten oder einen DoS-Angriff ausführen, bleiben Antworten auf Anfragen aus oder verzögern sich. Das System sollte deshalb erkennen können, wenn erwartete Pakete (z.B. regelmäßige Modbus-Polling-Anfragen) ausbleiben.

In einer anfänglichen Untersuchung wie dieser können selbstverständlich nicht alle Angriffsvektoren bzw. Auswirkungen von Ransomware auf ICS geprüft werden, was Möglichkeiten für zukünftige Arbeiten bietet (siehe Abs. 6.2.1).

## 4 IMPLEMENTIERUNG

Die Python-Scapy-YARA-Toolchain wurde in einer virtualisierten Docker-Umgebung realisiert. Gründe dafür sind vor allem die Versionsunterschiede zwischen den Python-Versionen sowie die Tatsache, dass YARA für die Einbindung von eigenen Modulen selbst kompiliert werden muss. Um Komplikationen mit Abhängigkeiten und dem Host-System zu vermeiden, ist die Nutzung von Docker sinnvoll. Das zum Aufsetzen der Umgebung genutzte Dockerfile ist im Anhang A.1 eingebunden.

### 4.1 Wrapper mit Scapy und YARA-Python

Im Folgenden werden die wesentlichsten Schritte zur Implementierung des in Abs. 3.2 beschriebenen Konzepts erläutert (vollständiger Quelltext in Anhang B.1).

Um YARA-Regeln mit der YARA-Python-Schnittstelle nutzen zu können, müssen diese im Vorfeld erst kompiliert werden:

```
1 # Kompilierung aller YARA-Regel-Dateien
2 yara_rules = [yara.compile(str(rule_file)) for
    ↪ rule_file in YARA_FILES]
```

Code Listing 1. Kompilieren von Regeln mit YARA-Python

Für den Packet Buffer wurde eine Deque anstatt einer herkömmlichen Queue als Datenstruktur verwendet. Bei einer normalen Queue werden neue Werte ans Ende angehängt und von vorne entfernt. Für die Arbeit mit YARA-Regeln ist es allerdings sinnvoller, wenn das Verhalten genau umgekehrt ist, sodass neue Pakete an den Anfang kommen. So ist es später einfacher, auf das letzte bzw. neueste Match zuzugreifen:

```
1 # Initialisierung des Packet Buffers
2 packet_buffer_deque = deque([])
3 # Handling eines eintreffenden Pakets
4 def handle_packet(packet) -> None:
5     packet_buffer_deque.appendleft(packet)
6     if len(packet_buffer_deque) >
    ↪ PACKET_BUFFER_SIZE:
7         packet_buffer_deque.pop()
8     . . .
```

Code Listing 2. Implementierung des Packet Buffers

Da YARA-Regeln nicht auf eine Queue bzw. Deque angewendet werden können, müssen die Pakete im Packet Buffer vorher in die Paketrohdaten umgewandelt werden. Wie bereits in Abs. 3.2 motiviert, wird nach jedem Paket ein Zeitstempel eingefügt. Der Zeitstempel wird als 64-Bit Integer zwischen zwei Bytes (0xFF und 0xFE) eingefügt, damit er in YARA-Regeln leicht und eindeutig auslesbar ist. Abschließend können die Regeln angewandt und die Ergebnisse als Vektor gespeichert werden.

```
1 # Handling eines eintreffenden Pakets
2 def handle_packet(packet) -> None:
3     . . .
4     # Umwandlung in Rohdaten mit Zeitstempel
5     raw_data = b"".join([raw(packet) + b"\xff" +
    ↪ (round(packet.time * 1000000).to_bytes(8,
    ↪ "little")) + b"\xfe" for packet in
    ↪ packet_buffer_deque])
6     # Anwendung der kompilierten Regeln
7     match_vector = [rule.match(data=raw_data) for
    ↪ rule in yara_rules]
```

Code Listing 3. Einbettung der Zeitstempel und Anwendung der Regeln

## 4.2 YARA-Module

YARA-Module bieten die Möglichkeit, den Funktionsumfang von YARA zu erweitern. Diese müssen in der Programmiersprache C entwickelt und bei der Kompilierung von YARA mit eingebunden werden. Da C eine turing-vollständige Sprache ist, kann YARA somit alle berechenbaren Funktionen nutzen. Nichtsdestotrotz sollte darauf verzichtet werden, die Detektion vollständig in Module auszulagern, da dies der Idee von YARA widerspricht. Vielmehr sollten zur Detektion fehlende Metriken bereitgestellt werden, beispielsweise die Anzahl von Registerwerten in einem Modbus-Paket.

Zum Auslesen von Werten aus den Daten bietet YARA für einige Datentypen vordefinierte Funktionen an, darunter u.a. `int8(offset)`, `int16(offset)` und `int32(offset)`. Allerdings fehlen Funktionen für 64-Bit Integer sowie für Gleitkommazahlen, wie sie für einige der implementierten Regeln benötigt werden. Das Modul „numeric“ soll dementsprechend die fehlenden Typen ergänzen.

Zur Berechnung der Entropie gibt es bereits im YARA-

Modul „math“ die Funktion `entropy(offset, n)`, welche die Entropie über die `n` Bytes ab der Position `offset` berechnet. Um die Entropie von Daten zu errechnen, die an verschiedenen Positionen verteilt vorkommen, implementiert das Modul „numeric“ eine zusätzliche Funktion. Zusammenfassend werden folgende Funktionen zum Funktionsumfang von YARA ergänzt:

- `float32(offset)`: liest ab der übergebenen Position die nächsten 4 Bytes und interpretiert diese als 32-Bit Gleitkommazahl
- `float64(offset)`: liest ab der übergebenen Position die nächsten 8 Bytes und interpretiert diese als 64-Bit Gleitkommazahl
- `int64(offset)`: liest ab der übergebenen Position die nächsten 8 Bytes und interpretiert diese als 64-Bit Ganzzahl
- `distributed_entropy(p1, p2, p3, p4, p5)`: berechnet die Entropie von 16-Bit Integer über 5 verschiedene Positionen

Der vollständige Quelltext des „numeric“-Moduls ist in Anhang D.1 zu finden.

Tabelle 1

Auflistung der untersuchten Auswirkungen von Ransomware-Angriffen auf ein ICS mit den dazu implementierten YARA-Regeln zur Detektion

ID	Detektion von	Kurzbeschreibung	Packet Buffer	YARA-Regel	PCAP-Quelle
<b>Modbus</b>					
qf <sub>1</sub>	Query-Flooding I	Detektion durch mindestens 3 Vorkommen von Modbus-Query-Requests im Packet Buffer	$n = 3$	Listing 4 (C.1)	ICS-Dataset [11] CRITIS18 [12]
qf <sub>2</sub>	Query-Flooding II	Detektion durch Schwellwertunterschreitung der Zeitdifferenz zwischen 2 Modbus-Query-Requests im Packet Buffer	$n = 3$	Listing 5 (C.1)	ICS-Dataset [11] CRITIS18 [12]
lsb	LSB-Stego Covert-Channel	Detektion eines Covert-Channels in einem von 3 Modbus-Registern durch Entropieberechnung	$n = 100$	-	ICS-Dataset [11] Lemay [13]
<b>OPCUA</b>					
val <sub>1</sub>	Werteanomalie I	Abgleich von konkreten (Temperatur-)Werten in einzelnen OPCUA-Write-Requests	$n = 1$	Listing 6 (C.2)	Labor
val <sub>2</sub>	Werteanomalie II	Abgleich von konkreten (Temperatur-)Differenzen in 2 aufeinanderfolgenden OPCUA-Write-Requests	$n = 40$	Listing 7 (C.2)	Labor
val <sub>3</sub>	Werteanomalie III	Abgleich von Werten aus OPCUA-Paketen im Sign&Encrypt-Modus	-	-	Prosys OPC UA Simulation
enc <sub>1</sub>	Verschlüsselung I	Detektion von verschlüsselten Werten in OPCUA-Write-Requests durch Abgleich des ersten (d.h. des most-significant) Bytes	$n = 1$	Listing 8 (C.3)	Emirkan-BA [14]
enc <sub>2</sub>	Verschlüsselung II	Detektion von Verschlüsselung der SCID in OPCUA-Messages durch spontane Änderung der SCID	$n = 3$	Listing 9 (C.3)	Emirkan-BA [14]
<b>ARP</b>					
mitm	ARP-Spoofing	Detektion eines MITM-Angriffs m.H. von ARP-Spoofing durch Zählen der bekannten MAC-Adressen	$n = 1$	Listing 10 (C.4)	ICS-Dataset [11] CRITIS18 [12]
<b>allgemein</b>					
absc	Paket-abwesenheit	Detektion von Paketen, die wider Erwarten nicht eingetroffen sind	-	-	-

### 4.3 YARA-Regeln

#### 4.3.1 Aufbau von YARA-Regeln

Eine YARA-Regel wird mit dem Schlüsselwort `rule` eingeleitet, gefolgt von einem Namen bzw. Identifier. In einer Regel gibt es i.d.R. zwei Sektionen: Die `strings`-Sektion ist optional und definiert Patterns, die für die folgende Bedingung benötigt werden. Ein String wird im Kontext von YARA beginnend mit `$` definiert und kann entweder textuell, aber auch binär (hexadezimale Darstellung) sein.

Jede YARA-Regel muss zudem die Sektion `condition` implementieren, wo auch auf die zuvor definierten Strings zurückgegriffen werden kann (z.B. Positionen sowie Anzahl der Matches). Um eine Bedingung festzulegen steht eine wahrheitsfunktional-vollständige Menge von logischen Operatoren zur Verfügung (`and`, `or`, `not`, `==`, ...).

Details zu den einzelnen Sektionen sowie welche Operatoren jeweils verwendet werden können, ist der YARA-Dokumentation zu entnehmen [9].

#### 4.3.2 Implementierung der Regeln

In diesem Abschnitt werden die erfolgreich implementierten YARA-Regeln, sowie der jeweils zugrundeliegende Mechanismus beschrieben. Eine Auflistung aller untersuchter Auswirkungen von netzwerkbasierter Ransomware ist in Tab. 1 dargestellt. Die jeweils angegebenen Größen für den Packet Buffer sowie auch die weiteren Parameter sind stets auf die jeweils verwendete PCAP-Aufzeichnung bezogen. In der Praxis sind die notwendigen Größen zur Detektion vom konkreten ICS abhängig (d.h. wie viele Hosts sind im Netzwerk, wie sind die Polling-Intervalle gesetzt, etc.). Variierende Paketfelder wie z.B. Sequenznummern werden in YARA-Patterns durch „Gaps“ (`[*]`) ersetzt.

- **Regel `qf1`:**

Ein erster naiver Ansatz zur Detektion von Modbus-Query-Flooding-Angriffen mit YARA basiert auf dem Zählen der Vorkommen von  $i$  Modbus-Query-Paketen im Packet Buffer der Größe  $n = 3$ .

- Patterns (`strings`): In dieser Regel ist lediglich die Signatur von Modbus-Query-Paketen hinterlegt.
- Detektion (`condition`): Die Regel schlägt aus, sobald  $i \geq 3$  Modbus-Query-Pakete im Packet Buffer vorkommen.

In ICS, in denen der Netzwerktraffic eher von *Bursts* (d.h. geprägt von Lastspitzen) bestimmt ist, kann dieser Ansatz allerdings unpraktikabel werden.

- **Regel `qf2`:**

Eine alternative Möglichkeit zur Flooding-Detektion ist das Messen der Zeitdifferenz  $\Delta t$  zwischen den letzten zwei Vorkommen von Modbus-Queries im Packet Buffer der Größe  $n = 3$ . Dazu werden die Zeitstempel genutzt, welche nach dem Konzept in Abs. 3.2 hinter den Netzwerkpaketen abgelegt werden.

- Patterns (`strings`): Die Signatur von Modbus-Query-Paketen wird um `FF [8] FE` erweitert, um den 64-Bit Zeitstempel zu erfassen.
- Detektion (`condition`): Insofern mindestens zwei Modbus-Queries im Packet Buffer vorliegen, wird von beiden Zeitstempeln hinter diesen Paketen die Differenz  $\Delta t$  berechnet. Eine Detektion erfolgt, wenn  $\Delta t < 100000\mu s$  ist.

Der dieser Regel zugrundeliegende Mechanismus kann so auch zum Prüfen von Polling-Intervallen genutzt werden, dazu muss lediglich ein Intervall anstelle eines Schwellwerts genutzt werden.

- **Regel `val1`:**

Um Werteanomalien detektieren zu können müssen die betreffenden Werte ausgelesen und gültige Grenzen angegeben werden können. Bei den hier überprüften Werten handelt es sich um 32-Bit Floating-Point Zahlen, welche aus OPCUA-Write-Requests ausgelesen werden und mit einem oberen Schwellwert  $t$  abgeglichen werden. Dazu wird kein Kontext im Netzwerkdatenstrom benötigt, wodurch ein Packet Buffer der Größe  $n = 1$  ausreichend ist.

- Patterns (`strings`): Das hinterlegte Pattern dieser Regel beschreibt einen OPCUA-Write-Request.
- Detektion (`condition`): Wenn ein Write-Request vorkommt, wird der Wert gegen die obere Grenze von  $t = 50$  geprüft.

- **Regel `val2`:**

Für einige Sachverhalte, bei denen kein fixes Intervall angegeben werden kann, ist es sinnvoller die Änderungsrate  $\Delta t$  eines Wertes zwischen den letzten zwei OPCUA-Write-Requests zu betrachten. Um sicherzustellen, dass zwei aufeinanderfolgende Requests im Packet Buffer liegen, muss dieser groß genug sein (in diesem Fall  $n = 40$ ).

- Patterns (`strings`): Das hinterlegte Pattern dieser Regel beschreibt einen OPCUA-Write-Request.
- Detektion (`condition`): Die angegebene Bedingung prüft, ob  $\Delta t > 5$  ist.

- **Regel `enc1`:**

Durch die Verschlüsselung von Daten werden diese in pseudo-zufällige Bitfolgen umgeschrieben. Wenn beispielsweise eine Gleitkommazahl verschlüsselt wird hat dies zur Folge, dass sich (durch die Einteilung in Vorzeichen-Bit, Mantisse und Charakteristik) die Größenordnung vollständig verändert. Um Werte in der erwarteten Größenordnung abzubilden, wird (bei der untersuchten Aufzeichnung) das erste Byte nicht benötigt, da dieses Byte nur für extrem kleine bzw. extrem große Werte zur Verschiebung des Exponenten genutzt wird. Ein Packet Buffer der Größe  $n = 1$  ist ausreichend, um dieses Kriterium zu prüfen.

- Patterns (`strings`): Das hinterlegte Pattern dieser Regel beschreibt einen OPCUA-Write-Request.
- Detektion (`condition`): Insofern ein Write-Request im Packet-Buffer enthalten ist, wird geprüft ob das erste Byte des 32-Bit Floating-Wertes ungleich 0 ist.

- **Regel `enc2`:**

Die Verschlüsselung der SecureChannelID (SCID) in OPCUA-Paketen hat zur Folge, dass sich die SCID plötzlich verändert. Um Veränderung bei Werten aufeinanderfolgender Pakete zu detektieren, muss der Packet Buffer z.B.  $n = 3$  sein.

- Patterns (`strings`): Da alle OPCUA-Nachrichten ein SCID-Feld haben, wird als Pattern `MSGF` genutzt.
- Detektion (`condition`): Um eine Änderung der SCID zu erfassen, wird wieder die Differenz aufein-

anderfolgender SCIDs berechnet. Sollte die Differenz ungleich 0 sein, hat sie sich geändert.

- **Regel mitm:**

Bei einem MITM-Angriff über ARP-Spoofing werden gefälschte ARP-Pakete gesendet, um den Traffic an eine andere IP umzuleiten. Um diesen Angriffsvektor rechtzeitig zu erkennen, müssen alle korrekten MAC-Adressen des Systems bekannt sein. Mit einem Packet Buffer der Größe  $n = 1$  kann für einzelne Pakete entschieden werden, ob diese eine gefälschte MAC-Adresse beinhalten. In der Detektionsregel muss außerdem zwischen ARP-Requests und ARP-Replies unterschieden werden, da die Anzahl der beinhalteten MAC-Adressen jeweils verschieden ist.

- **Patterns (strings):** Zur Detektion werden die Patterns von ARP-Requests bzw. -Replies benötigt. Außerdem müssen alle bekannten MAC-Adressen definiert werden.
- **Detektion (condition):** Die Bedingung ist von der Art des ARP-Pakets abhängig. Bei Requests muss die Summe von bekannten MAC-Adressen genau 3 sein, bei Replies 4. Wird einer der Werte unterschritten, ist eine der MAC-Adressen im ARP-Paket unbekannt.

Unter anderem für diese Regel könnte es sinnvoll sein, Teile durch Meta-Programmierung dynamisch zu erzeugen (hier konkret eine Liste von bekannten MAC-Adressen), um die Anpassung der Regel an neue Systeme zu vereinfachen (siehe Abs. 6.2.2).

## 5 EVALUIERUNG

### 5.1 Detektionsfähigkeiten von YARA

Die im Rahmen dieser Untersuchung erzeugten Regeln zeigen, dass YARA durchaus schädliche Inhalte im Netzwerkdatenstrom, speziell auch Auswirkungen von Ransomware auf ICS, detektieren kann:

Für das Modbus-Protokoll wurden Query-Flooding-Angriffe mit zwei verschiedenen Mechanismen detektiert, wobei auch zeitbasierte Metriken realisiert werden konnten. Desweiteren können einzelne (OPCUA-)Paketfelder ausgelesen und inhaltlich geprüft werden. Zudem ist es ebenfalls möglich, Änderungsraten von Werten über mehrere Pakete hinweg zu betrachten. Durch Ransomware verschlüsselte Informationen können dann erkannt werden, wenn hinreichende Informationen über den Normalbetrieb vorliegen. Die ausgearbeiteten Regeln stellen dabei lediglich grundlegende Detektionsansätze mit YARA dar, die auch ähnlich auf weitere Angriffsvektoren bzw. Auswirkungen von Ransomware übertragen werden können.

Da die einzelnen Regeln allerdings oft verschiedene Parametrisierungen des Wrappers (d.h. Packet Buffer Größen) benötigen, können sie im Allgemeinen nicht in einer Instanz parallel geprüft werden. Desweiteren sind Patterns problematisch, die Sprünge variabler Länge beinhalten (z.B. [4-12]), insofern hinter diesen Sprüngen konkrete Werte ausgelesen werden sollen. Solche Patterns müssen für jede (Paket-)Länge einzeln definiert werden damit das angegebene Offset auf den korrekten Wert zeigt, was unter Umständen zu sehr vielen Patterns führt.

### 5.2 Grenzen des Ansatzes

#### 5.2.1 LSB-Stego Covert-Channel

Der untersuchte Covert-Channel basiert auf dem Modbus-Protokoll und bettet Informationen über LSB-Steganographie in einem von 3 Registerwerten von „ReadHoldingRegisters“-Paketen (RHR) ein. Mit einem Packet Buffer der Größe  $n = 100$  wird sichergestellt, dass mindestens 5 RHR-Pakete im Puffer liegen. Die Entropie wird anschließend für jedes Register einzeln über die 5 aufeinanderfolgenden Werte mit der Funktion `distributed_entropy` aus dem Modul „numeric“ (siehe Abs. 4.2) berechnet. Ein Ausschnitt des errechneten Verlaufs der Entropie über den Netzwerkdatenstrom wird in Abb. 2 dargestellt. Daraus ist erkennbar, dass die Entropie hauptsächlich zwischen 4.25 und 4.65 schwankt und der Einfluss des Stego-Kanals lediglich anteilig höher ist, allerdings nicht absolut. Dass die Entropie ab und zu kurzzeitig stark abfällt liegt am Verhalten des simulierten ICS und steht nicht in Verbindung mit dem Covert-Channel. Die durchschnittliche Entropien über den gesamten Verlauf sind als horizontale Linien in Abb. 2 eingezeichnet, wodurch der Stego-Kanal eindeutig dem Register 2 zuzuordnen ist.

Zusammenfassend kann festgehalten werden, dass die Entropieberechnung in YARA m.H. der Module relativ einfach möglich ist. Das Problem besteht vielmehr darin, dass die Berechnung der Entropie über nur 5 Werte nicht ausreichend für eine Detektion ist. Dazu müssten die Werte auf lange Sicht untersucht werden, was allerdings zu

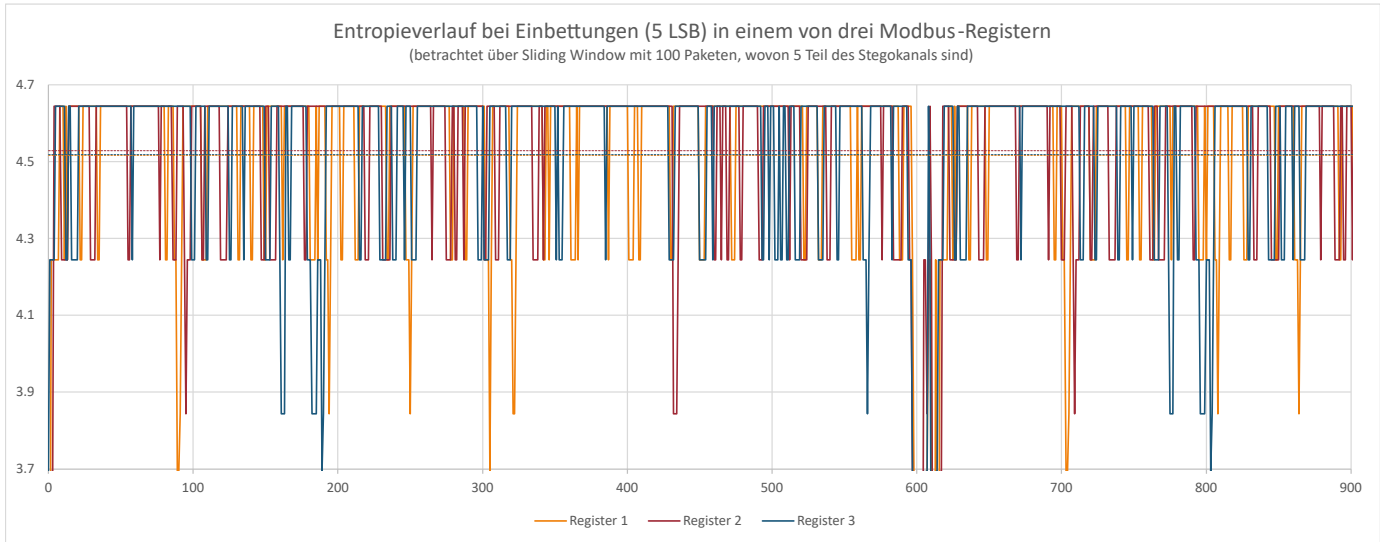


Abbildung 2. Ausschnitt des Entropieverlaufs bei LSB-Stego-Einbettungen in Modbus-Register

Funktionen mit extrem hohen Stelligkeiten führt. Ab einem gewissen Punkt wird dies unpraktikabel, theoretisch ist es allerdings möglich. Eventuell kann bei solchen Problemen die Anwendung von Meta-Programmierung aushelfen, um vielstellige Funktionen zu generieren (siehe Abs. 6.2.2).

### 5.2.2 Verschlüsselte Paketinhalte

YARA-Regeln basieren auf Pattern-Matching. Verschlüsselte Implementierung wie OPCUA im Sign&Encrypt-Modus verhindern, dass auf konkrete Inhalte zugegriffen werden kann. Im Warden-Szenario bzw. im Kontext einer Next-Gen-Firewall können allerdings durch Aufbrechen der Verschlüsselung trotzdem Paketinhalte betrachtet werden. Dementsprechend ist dies kein Grund von einer verschlüsselten Implementierung abzusehen, da OPCUA Sign&Encrypt nachweislich einen deutlich besseren Schutz vor Ransomware-Angriffen bietet [14].

### 5.2.3 Abwesenheit von Paketen

Der in dieser Arbeit vorgestellte Ansatz sieht vor, nach dem Eingang eines jeden Pakets in den Packet Buffer alle gegebenen YARA-Regeln zu evaluieren, um unmittelbar eine Aussage über potentiell schädliche Inhalte treffen zu können. Dadurch wird es allerdings unmöglich die Abwesenheit von Paketen direkt zu überprüfen, da immer erst nach dem Eingang eines Pakets evaluiert wird. Beispielsweise wäre eine Regel, die nach dem Überschreiten eines spezifizierten Zeitintervalls ausschlägt insofern ein erwartetes Paket nicht eingetroffen ist, mit dem in dieser Arbeit implementierten Wrapper für YARA nicht möglich.

Ein alternativer Ansatz könnte darin bestehen, YARA-Regeln in einem fest definierten Zeitabstand auf alle Pakete anzuwenden, die in diesem Zeitintervall eingetroffen sind (siehe Abs. 6.2.3).

## 6 ZUSAMMENFASSUNG UND AUSBLICK

### 6.1 Zusammenfassung der Ergebnisse

In dieser Arbeit wurde gezeigt, dass YARA definitiv einen Beitrag zur Erkennung von Ransomware in Netzwerken leisten kann. YARA bietet viele Möglichkeiten (darunter mächtiges Pattern-Matching, reguläre Ausdrücke, Bitoperationen sowie wahrheitsfunktionale Logik), um in Regeln auch komplexere Sachverhalte abbilden zu können. Eine besondere Stellung nimmt das Feature der C-Module ein, wodurch die Implementierung von Metriken ermöglicht wird, die den Standardfunktionsumfang von YARA übersteigen. So können beispielsweise Schnittstellen zu anderen Programmen realisiert und im allgemeinen turing-vollständige Funktionen berechnet werden.

Speziell für die Anwendung von YARA im Netzwerkdatenstrom müssen geeignete Puffer-Mechanismen implementiert werden. Die Anwendung von Regeln muss außerdem immer aktiv angestoßen werden, was entweder event-basiert (wie in dieser Arbeit) und/oder zeit-basiert erfolgen muss. Im Bereich von Next-Gen Firewalls können diese Voraussetzungen allerdings ohne Probleme geschaffen werden.

Die in dieser Arbeit identifizierten Grenzen sind vor allem dem recht einfachen Konzept geschuldet. Bei einzelnen Sachverhalten (siehe LSB-Stego Covert-Channel) kann es allerdings trotzdem passieren, dass einige Mechanismen unpraktikabel werden.

### 6.2 Ausblick für zukünftige Arbeiten

#### 6.2.1 Weitere Angriffsvektoren

Selbstverständlich können in einer anfänglichen Arbeit wie dieser nicht alle von Ransomware genutzten Angriffsvektoren untersucht werden. Dementsprechend ist es möglich in einer fortführenden Arbeit weitere Sachverhalte zu untersuchen. In den verwendeten Datensets sind beispielsweise Aufzeichnung von Metapreter Covert-Channels sowie weiteren ausgenutzten Exploits vorhanden, für die Regeln abgeleitet werden könnten.

### 6.2.2 Meta-Programmierung

Um Funktionen von besonders hohen Stelligkeiten implementieren zu können, ist die Generierung von Modul-Quelltext bzw. von YARA-Regeln eine zu betrachtende Möglichkeit. Durch Nutzen von Meta-Programmierung könnten YARA-Regeln insgesamt dynamischer gestaltet werden, da sie sich an das aktuelle System anpassen könnten. Exemplarisch könnten dadurch bekannte MAC-Adressen in die Regel mit dynamisch eingefügt werden.

### 6.2.3 Konzeptuelle Verbesserungen

Wie bereits in Abs. 5.2 begründet ist die Detektion von ausbleibenden Paketen mit dem in dieser Arbeit genutzten Ansatz nicht möglich. Dazu muss ein Mechanismus implementiert werden, welcher YARA-Regeln in zeitdiskreten Abständen automatisch anwendet. Dies führt allerdings zu weiteren Problemstellungen bei den Implementierungsdetails, weshalb in dieser Arbeit darauf verzichtet wurde. Beispielsweise ist das einfache Anwenden der Regeln in zeitlichen Abständen auf den Packet Buffer nicht ausreichend da es so passieren könnte, dass Pakete „übersehen“ werden, da sie das Fenster bereits verlassen haben. Dementsprechend könnte ein alternativer Ansatz darin bestehen, mehrere Sliding Windows parallel zu nutzen, die jeweils um ein zeitliches Offset verschoben agieren. Die Realisierung eines solchen Konzeptes ist nicht trivial, weshalb es in zukünftige Arbeiten ausgelagert wurde. Final ist wahrscheinlich ein hybrider Ansatz aus event-basierter bzw. zeit-basierter Regelanwendung am Besten geeignet, um ein hohes Maß an Flexibilität für Regel-Ableitungen zu ermöglichen.

## LITERATUR

- [1] "Enisa threat landscape for ransomware attacks." [Online]. Available: <https://www.enisa.europa.eu/publications/enisa-threat-landscape-for-ransomware-attacks/@download/fullReport>
- [2] "Yara - the pattern matching swiss knife for malware researchers." [Online]. Available: <https://virustotal.github.io/yara/>
- [3] "Aufgabenstellung 'it-security of cyber-physical systems'," 2023. [Online]. Available: <https://omen.cs.uni-magdeburg.de/itiamsl/deutsch/lehre/ws-23-24/it-security-of-cyber-physical-systems-its-cps.html>
- [4] "Smartest2 - evaluierung von verfahren zum testen der informationssicherheit in der nuklearen leittechnik durch smarte testfallgenerierung 2." [Online]. Available: <https://forschung-sachsen-anhalt.de/project/smartest-evaluierung-verfahren-zum-testen-24922>
- [5] "Modbus – kommunikationsprotokoll für die industrie." [Online]. Available: <https://www.kunbus.com/de/modbus>
- [6] K. Gebeshuber, "Modbus – angriffe im lokalen netzwerk," Jul. 2021. [Online]. Available: <https://www.informatik-aktuell.de/betrieb/sicherheit/modbus-angriffe-im-lokalen-netzwerk.html>
- [7] "Was ist opc ua?" [Online]. Available: <https://www.opc-router.de/was-ist-opc-ua/>
- [8] "Virustotal/yara - the pattern matching swiss knife." [Online]. Available: <https://github.com/VirusTotal/yara>
- [9] "Writing yara rules - yara 4.0.0 documentation." [Online]. Available: <https://yara.readthedocs.io/en/stable/writingrules.html>
- [10] "Writing your own modules - yara 4.0.0 documentation." [Online]. Available: <https://yara.readthedocs.io/en/stable/writingmodules.html>
- [11] "ics datasets," Sep. 2023. [Online]. Available: <https://gitti.cs.uni-magdeburg.de/klamshoeft/ics-datasets>
- [12] I. Frazão, P. H. Abreu, T. Cruz, H. Araújo, and P. Simões, "Denial of service attacks: Detecting the frailties of machine learning algorithms in the classification process," in *Critical Information Infrastructures Security*. Cham: Springer International Publishing, 2019, pp. 230–235. [Online]. Available: [https://doi.org/10.1007/978-3-030-05849-4\\_19](https://doi.org/10.1007/978-3-030-05849-4_19)
- [13] A. Lemay and J. M. Fernandez, "Providing scada network data sets for intrusion detection research," in *9th Workshop on Cyber Security Experimentation and Test (CSET 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: <https://www.usenix.org/conference/cset16/workshop-program/presentation/lemay>
- [14] E. Toplu, "Untersuchungen und evaluation der anwendbarkeit und auswirkungen von netzwerk ransomware im kontext von industrie 4.0," Jun. 2023.
- [15] "Virustotal/yara-python - the python interface for yara." [Online]. Available: <https://github.com/VirusTotal/yara-python>
- [16] "Scapy." [Online]. Available: <https://scapy.net/>
- [17] "Wireshark - go deep." [Online]. Available: <https://www.wireshark.org/>
- [18] "Docker: Accelerated container application development." [Online]. Available: <https://www.docker.com/>
- [19] A. Lemay, J. M. Fernandez, and S. Knight, "A modbus command and control channel," in *2016 Annual IEEE Systems Conference (SysCon)*, 2016, pp. 1–6. [Online]. Available: <https://doi.org/10.1109/SYSCON.2016.7490631>



## ANHANG A

### A.1 Python-Scapy-YARA-Toolchain: Dockerfile

```

1 FROM ubuntu:latest AS base
2     RUN apt update
3     RUN apt upgrade -y
4 FROM base AS python
5     RUN apt install python3 python3-pip git -y
6     RUN pip install ipython
7 FROM python AS scapy
8     RUN apt install libpcap-dev -y
9     RUN pip install scapy
10 FROM scapy AS yara
11     RUN git clone --recursive https://github.com/VirusTotal/yara-python
12     COPY ./modules_patch/numeric /yara-python/yara/libyara/modules/numeric
13     RUN echo "MODULE(numeric)" >> /yara-python/yara/libyara/modules/module_list
14     COPY ./modules_patch/Makefile.am /yara-python/yara/Makefile.am
15     RUN cd /yara-python && python3 setup.py build
16     RUN cd /yara-python && python3 setup.py install
17 FROM yara AS finish
18     RUN apt clean
19     WORKDIR /home/python-scapy-yara
20     COPY ./app/ .
21     ENTRYPOINT [ "python3", "-u", "./prototype.py" ]

```

## ANHANG B

### B.1 Implementierung der Methodik: prototype.py

```

1 import argparse
2 from collections import deque
3 from functools import reduce
4 import json
5 from pathlib import Path
6 import sys
7
8 from scapy.all import *
9 import yara
10
11 # argument parsing
12 parser = argparse.ArgumentParser(
13     prog="yara-packet-inspector",
14     description="applies yara rules to network data stream buffer"
15 )
16 parser.add_argument("yara_file", nargs="+", help="yara rule file(s)")
17 parser.add_argument("-pcap", type=str, help="pcap file to apply rules to; if none is given, real-time
18     ↪ network data stream will be used")
19 parser.add_argument("-pbs", "--packet-buffer-size", type=int, default=1, help="set packet buffer size on
20     ↪ which the rules are applied to")
21 args = parser.parse_args()
22
23 # parameterization
24 YARA_FILES = [Path(yara_file).resolve() for yara_file in args.yara_file]
25 if not reduce(lambda x, y: x and y, [yara_file.is_file() and yara_file.exists() for yara_file in YARA_FILES
26     ↪ ], True):
27     print(f"[!] ERROR: Failed to access one or more of the given yara rule files!")
28     sys.exit(1)
29
30 PCAP_FILE = None
31 if args.pcap is not None:
32     PCAP_FILE = Path(args.pcap).resolve()
33     if not PCAP_FILE.exists():
34         print(f"[!] ERROR: Failed to access given pcap file!")
35         sys.exit(1)
36
37 PACKET_BUFFER_SIZE = args.packet_buffer_size
38 PACKET_SNIFF_FILTER = "ip"
39
40 print(f"[!] Initializing with following parameterization:")
41 print(f"    YARA_FILES={([str(yf) for yf in YARA_FILES])}")
42 print(f"    MODE={'REALTIME_SNIFF' if PCAP_FILE is None else f'(PCAP_INSPECTION={PCAP_FILE})'}")
43 print(f"    PACKET_BUFFER_SIZE={PACKET_BUFFER_SIZE}")
44 print(f"    PACKET_SNIFF_FILTER={PACKET_SNIFF_FILTER}")
45
46 print(f"[!] Compiling {len(YARA_FILES)} YARA rule file(s): {', '.join([file.name for file in YARA_FILES])
47     ↪ }...")

```

```

44 yara_rules = [yara.compile(str(rule_file)) for rule_file in YARA_FILES]
45 print(f"[!] {len(yara_rules)} YARA rule files compiled.")
46
47 # global structure to temporarily store packets
48 packet_buffer_deque = deque([])
49
50 # global structure to log results
51 log_dict = {}
52
53 # only used when real-time sniffing mode is used
54 sniff_packet_index = 0
55
56 # collect and filter results
57 def fit_match_vector_to_dict(match_vector: list, index: int) -> None:
58     global log_dict, YARA_FILES
59     for i, rule_match in enumerate(match_vector):
60         # use yara rule file name as key to store related results
61         key_yara_file = YARA_FILES[i].name
62         if not key_yara_file in log_dict:
63             log_dict[key_yara_file] = {}
64
65         # check if any rule of yara rule file i matched
66         if len(rule_match) > 0:
67             # loop concrete matches of that rule file
68             for rule_match2 in rule_match:
69                 if "main" in rule_match2.tags:
70                     if not str(rule_match2) in log_dict[key_yara_file]:
71                         log_dict[key_yara_file][str(rule_match2)] = {
72                             "matches": 0,
73                             "packets": []
74                         }
75                         log_dict[key_yara_file][str(rule_match2)]["packets"].append(index)
76                         log_dict[key_yara_file][str(rule_match2)]["matches"] = len(log_dict[key_yara_file][str(
77                             ↪ rule_match2)]["packets"])
78
79 def handle_packet(packet, index = -1) -> None:
80     # append new packet to queue
81     packet_buffer_deque.appendleft(packet)
82
83     # remove oldest packet from queue if maximum size exceeded
84     if len(packet_buffer_deque) > PACKET_BUFFER_SIZE:
85         packet_buffer_deque.pop()
86
87     # convert queue to raw data
88     raw_data = b"".join([
89         raw(packet) + b"\xff" + (round(packet.time * 1000000).to_bytes(8, "little")) + b"\xfe"
90         for packet in packet_buffer_deque
91     ])
92
93     # match vector contains an entry for every yara rule, even if the rule has not been triggered
94     match_vector = [rule.match(data=raw_data) for rule in yara_rules]
95
96     # store rule matches in dict
97     if index == -1:
98         # sniffing mode case
99         global sniff_packet_index
100         sniff_packet_index += 1
101         fit_match_vector_to_dict(match_vector, sniff_packet_index)
102         print(f"{sniff_packet_index}: {match_vector}")
103     elif len(match_vector) > 0:
104         # pcap case
105         fit_match_vector_to_dict(match_vector, index + 1)
106
107 if PCAP_FILE is None:
108     print(f"[!] Sniffing...")
109     sniff(filter=PACKET_SNIFF_FILTER, prn=handle_packet)
110 else:
111     print(f"[!] Reading packets from pcap file '{PCAP_FILE}'...")
112     pcap_packets = rdpcap(str(PCAP_FILE))
113     print(f"[!] Applying YARA rules...")
114     [handle_packet(packet, i) for i, packet in enumerate(pcap_packets)]
115
116 print(f"[!] Done!")
117 print(f"[!] JSON={json.dumps(log_dict, indent=2)}")

```

## ANHANG C

### C.1 YARA-Regeln: Flooding

```

1 rule modbus_queryflooding_repetition : main
2 {
3     strings:
4         $modbus_query_request = { 00 80 F4 09 51 3B 00 0C 29 E6 14 0D 08 00 45 00 00 34 [2] 40 00 40 06 [2]
5         ↳ AC 1B E0 32 AC 1B E0 FA [2] 01 F6 [4] [4] 50 18 72 10 [2] 00 00 [2] 00 00 00 06 01 06 00 06 00 00 }
6     condition:
7         #modbus_query_request >= 3
8 }

```

Code Listing 4. YARA-Regel  $qf_1$

```

1 import "numeric"
2
3 rule modbus_queryflooding_timing : main
4 {
5     strings:
6         $modbus_query_request = { 00 80 F4 09 51 3B 00 0C 29 E6 14 0D 08 00 45 00 00 34 [2] 40 00 40 06 [2]
7         ↳ AC 1B E0 32 AC 1B E0 FA [2] 01 F6 [4] [4] 50 18 72 10 [2] 00 00 [2] 00 00 00 06 01 06 00 06 00 00
8         ↳ FF [8] FE }
9     condition:
10         #modbus_query_request >= 2 and (numeric.int64(@modbus_query_request[1] + 66) - numeric.int64(
11         ↳ @modbus_query_request[2] + 66)) < 100000
12 }

```

Code Listing 5. YARA-Regel  $qf_2$

### C.2 YARA-Regeln: Werteanomalien

```

1 import "numeric"
2
3 rule opcua_kochvorgang_xc50 : main
4 {
5     strings:
6         $opcua_writerequest = { 4D 53 47 46 [8] 01 00 00 00 [8] 01 00 A1 02 [46] FF FF FF FF 03 0A [4] 00
7         ↳ 00 00 00 }
8     condition:
9         #opcua_writerequest > 0 and numeric.float32(@opcua_writerequest[1] + 80) >= 50
10 }

```

Code Listing 6. YARA-Regel  $val_1$

```

1 import "numeric"
2
3 rule opcua_kochvorgang_diff5 : main
4 {
5     strings:
6         $opcua_writerequest = { 4D 53 47 46 [8] 01 00 00 00 [8] 01 00 A1 02 [46] FF FF FF FF 03 0A [4] 00
7         ↳ 00 00 00 }
8     condition:
9         #opcua_writerequest >= 2 and (numeric.float32(@opcua_writerequest[1] + 80) - numeric.float32(
10         ↳ @opcua_writerequest[2] + 80)) > 5
11 }

```

Code Listing 7. YARA-Regel  $val_2$

### C.3 YARA-Regeln: Ransom-Verschlüsselung

```

1 rule opcua_writevalue_encrypted : main
2 {
3     strings:
4         $opcua_writerequest = { 4D 53 47 46 58 00 00 00 [4] 01 00 00 00 [8] 01 00 A1 02 [46] FF FF FF FF 03
5         ↳ 0A [4] 00 00 00 00 }
6     condition:
7         #opcua_writerequest > 0 and (int8(@opcua_writerequest[1] + 80) != 0)
8 }

```

Code Listing 8. YARA-Regel  $enc_1$

```

1 rule opcua_securechannelid_encrypted : main
2 {
3     strings:
4         $opcua_msgf = { 4D 53 47 46 [8] 01 00 00 00 }
5     condition:
6         #opcua_msgf >= 2 and (uint32(@opcua_msgf[1] + 8) - uint32(@opcua_msgf[2] + 8)) != 0
7 }

```

Code Listing 9. YARA-Regel `enc2`

## C.4 YARA-Regeln: MITM-Angriff durch ARP-Spoofing

```

1 rule arp_request
2 {
3     strings:
4         $arp_req = { 08 06 00 01 08 00 06 04 00 01 [20] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 }
5         ⇨ 00 }
6     condition:
7         any of them
8 }
9 rule arp_reply
10 {
11     strings:
12         $arp_rep = { 08 06 00 01 08 00 06 04 00 02 [20] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 }
13         ⇨ 00 }
14     condition:
15         any of them
16 }
17 rule arp_mitm : main
18 {
19     strings:
20         $mac0 = { FF FF FF FF FF FF }
21         $mac1 = { 48 5B 39 64 40 79 }
22         $mac2 = { 00 80 F4 09 51 3B }
23         $mac3 = { 00 0C 29 9D 9E 9E }
24     condition:
25         (arp_request and (#mac0 + #mac1 + #mac2 + #mac3) < 3) or (arp_reply and (#mac0 + #mac1 + #mac2 +
26         ⇨ #mac3) < 4)
27 }

```

Code Listing 10. YARA-Regel `mitm`

## ANHANG D

### D.1 Implementierung des Moduls „numeric“: `numeric.c`

```

1 #include <yara/modules.h>
2 #include <inttypes.h>
3
4 #define MODULE_NAME numeric
5 // #define BE_VERBOSE
6
7 const uint8_t* block_data;
8 size_t block_size;
9
10 define_function(float32) {
11     int64_t offset = integer_argument(1);
12
13     #ifdef BE_VERBOSE
14     printf("[float32] offset = %" PRIu64 " , block size = %zd", offset, block_size);
15     #endif
16
17     // check if 4 bytes are available at given offset
18     if (block_size > offset + 4) {
19         // cast block data to float
20         float* reinterpreted_numeric = (float*)(block_data + offset);
21
22         #ifdef BE_VERBOSE
23         printf(" , float = %.6f\n", *reinterpreted_numeric);
24         #endif
25
26         // return
27         return_float(*reinterpreted_numeric);
28     }
29 }

```

```

28     } else {
29         printf("\n[float32] WARNING: Given offset exceeds block size, can not convert!\n
↪ Returned -1 may result in broken rules!\n");
30
31         return_float(-1);
32     }
33 }
34 define_function(float64) {
35     int64_t offset = integer_argument(1);
36
37     #ifdef BE_VERBOSE
38     printf("[float64] offset = %" PRIu64 " ", block_size = %zd", offset, block_size);
39     #endif
40
41     //check if 4 bytes are available at given offset
42     if (block_size > offset + 8) {
43         //cast block data to float
44         double* reinterpreted_numeric = (double*)(block_data + offset);
45
46         #ifdef BE_VERBOSE
47         printf(", double = %.6f\n", *reinterpreted_numeric);
48         #endif
49
50         //return
51         return_float(*reinterpreted_numeric);
52     } else {
53         printf("\n[float64] WARNING: Given offset exceeds block size, can not convert!\n
↪ Returned -1 may result in broken rules!\n");
54
55         return_float(-1);
56     }
57 }
58 define_function(int64) {
59     int64_t offset = integer_argument(1);
60
61     #ifdef BE_VERBOSE
62     printf("[int64] offset = %" PRIu64 " ", block_size = %zd", offset, block_size);
63     #endif
64
65     //check if 8 bytes are available at given offset
66     if (block_size > offset + 8) {
67         //cast block data to int
68         int64_t* reinterpreted_numeric = (int64_t*)(block_data + offset);
69
70         #ifdef BE_VERBOSE
71         printf(", int = %" PRIu64 "\n", *reinterpreted_numeric);
72         #endif
73
74         return_integer(*reinterpreted_numeric);
75     } else {
76         printf("\n[int64] WARNING: Given offset exceeds block size, can not convert!\n");
77
78         return_integer(YR_UNDEFINED);
79     }
80 }
81 define_function(print_hex) {
82     int64_t offset = integer_argument(1);
83     int64_t values = integer_argument(2);
84
85     //check if 8 bytes are available at given offset
86     if (block_size > offset + values) {
87         printf(">");
88         for(int i = 0; i < values; i++) {
89             printf(" %02X", *(block_data + offset + i));
90         }
91         printf("\n");
92
93         return_integer(0);
94     } else {
95         printf("[print_hex] WARNING: Given offset exceeds block size, can not print!\n");
96
97         return_integer(YR_UNDEFINED);
98     }
99 }
100 define_function(distributed_entropy) {
101     int64_t offset1 = integer_argument(1);
102     int64_t offset2 = integer_argument(2);

```

```

103 int64_t offset3 = integer_argument(3);
104 int64_t offset4 = integer_argument(4);
105 int64_t offset5 = integer_argument(5);
106
107 int64_t offsets[5] = {
108     offset1, offset2, offset3, offset4, offset5
109 };
110
111 // "histogram" to count occurrences of each byte
112 int byteCount[256] = {0};
113 for(int i = 0; i < 5; i++) {
114     uint8_t* ptr = block_data + offsets[i];
115     if (block_size > offsets[i] + 2) {
116         printf("> %02X %02X", *(ptr), *(ptr + 1));
117         byteCount[*ptr]++;
118         byteCount[*ptr + 1]++;
119     } else {
120         printf("[distributed_entropy] WARNING: Given offset exceeds block size, can not print!\n");
121     }
122     return_integer(YR_UNDEFINED);
123 }
124
125 // calculate entropy
126 float entropy = 0.0;
127 for (int i = 0; i < 256; i++) {
128     if (byteCount[i] > 0) {
129         float probability = (float)byteCount[i] / 5;
130         entropy -= probability * log2(probability);
131     }
132 }
133
134 return_float(entropy);
135 }
136
137 begin_declarations;
138 declare_function("float32", "i", "f", float32);
139 declare_function("float64", "i", "f", float64);
140 declare_function("int64", "i", "i", int64);
141 declare_function("print_hex", "ii", "i", print_hex);
142 declare_function("distributed_entropy", "iiii", "f", distributed_entropy);
143 end_declarations;
144
145 int module_initialize(YR_MODULE* module) {
146     return ERROR_SUCCESS;
147 }
148
149 int module_finalize(YR_MODULE* module) {
150     return ERROR_SUCCESS;
151 }
152
153 int module_load(YR_SCAN_CONTEXT* context, YR_OBJECT* module_object, void* module_data, size_t
    ↪ module_data_size) {
154     YR_MEMORY_BLOCK* block;
155
156     block = first_memory_block(context);
157     block_data = block->fetch_data(block);
158     block_size = block->size;
159
160     if (block_data != NULL) { }
161
162     return ERROR_SUCCESS;
163 }
164
165 int module_unload(YR_OBJECT* module_object) {
166     return ERROR_SUCCESS;
167 }
168
169 #undef MODULE_NAME

```