BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

GENERIC FLOW ANALYSIS IN COMPUTER NETWORKS

GENERICKÁ ANALÝZA TOKŮ V POČÍTAČOVÝCH SÍTÍCH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MARKÉTA JANČOVÁ**
AUTOR PRÁCE

SUPERVISOR **Doc. Dr. Ing. DUŠAN KOLÁŘ**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2019/2020

Master's Thesis Specification

Student: **Jančová Markéta, Bc.**
Programme: Information Technology Field of study: Information Systems
Title: **Generic Flow Analysis in Computer Networks**
Category: Networking
Assignment:

1. Study IEC 104 protocol to understand flow analysis (focus on limited information provided). Next, study possibilities on how to represent network flows in order to monitor correct behavior on network in real time - various automata, or their creation and usage, respectively.
2. Based on a given collection of network flow records, specify a possible way to automatically create and build suitable software apparatus for analysis of particular behavior in real time.
3. Develop automatic analysis of captured network flow containing correct behavior aiming at generator of analyzers, which will work in real time.
4. Test the developed software on a given collection of network flow records.
5. Evaluate your contribution and discuss possible further development of the software.

Recommended literature:

- Angulin, D.: Learning Regular Sets from Queries and Counterexamples, Information and Computation, 75, pp. 87-106, 1987.
- Carrasco, R. C., Oncina, J.: Learning Deterministic Regular Grammars from Stochastic Samples in Polynomial Time, Theoretical Informatics and Applications, ITA, Vol. 33, No1, 1999, pp. 1-19.
- Matoušek, P.: Popis a analýza protokolu IEC 104, technická zpráva, FIT VUT v Brně, 2017.
- According to supervisor's recommendation.

Requirements for the semestral defence:

- Items 1 and 2, elaboration of item 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Kolář Dušan, doc. Dr. Ing.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: June 3, 2020
Approval date: October 22, 2019

Abstract

This diploma thesis focuses on the automatic network traffic description using models of communication. The main aim is at *industrial control system* protocols, especially *IEC 60870-5-104*. We propose a method of characterizing the network state using both communication content and behavior in time viewpoints. These aspects are described using *finite state automata*, *prefix trees*, and recurrence analysis. In the second part of this thesis, we focus on the implementation of a program, which is capable to use the obtained model to verify network traffic in real-time.

Abstrakt

Tato práce se zabývá problematikou popisu síťového provozu pomocí automaticky vytvořeného modelu komunikace. Hlavním zaměřením jsou komunikace v *řídicích systémech*, které využívají speciální protokoly, jako je například *IEC 60870-5-104*. V této práci představujeme metodu charakteristiky síťového provozu z pohledu obsahu komunikace i chování v čase. Tato metoda k popisu využívá *deterministické konečné automaty*, *prefixové stromy* a analýzu opakovatelnosti. Ve druhé části této diplomové práce se zaměřujeme na implementaci programu, který je schopný na základě takového modelu komunikace verifikovat síťový provoz v reálném čase.

Keywords

IEC 60870-5-104, TCP, SCADA, industrial control systems, network traffic analysis, system model, generic analysis, finite automaton, timed automaton, prefix tree, pattern recognition, pattern matching, periodicity, recurrence

Klíčová slova

IEC 60870-5-104, TCP, SCADA, řídicí systémy, analýza síťového provozu, model systému, generická analýza, konečný automat, časovaný automat, prefixový strom, hledání vzorů, porovnávání vzorů, periodicita, opakovatelnost

Reference

JANČOVÁ, Markéta. *Generic Flow Analysis in Computer Networks.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Dr. Ing. Dušan Kolář

Rozšířený abstrakt

Analýza síťového provozu hraje důležitou roli v oblasti počítačové bezpečnosti. Díky ní můžeme sledovat chování jednotlivých zařízení v síti, monitorovat výkon, ale také předvídat budoucí komunikaci za účelem včasného odhalení nežádocího chování. Za tímto účelem bylo v minulosti navrženo mnoho metod se snahou poskytnout co nejlepší možný popis aktuálního stavu sítě. Ovšem sestavení modelu sítě, který přesně popisuje aktuální stav komunikace všech zařízení, je téměř nemožný úkol, protože v síti se přirozeně vyskytuje mnoho chyb. Za chyby považujeme například ztráty, zpoždění a duplicity paketů. K predikci budoucích stavů se proto využívají metody strojového učení. Tyto metody, a obzvláště ty z nich, které jsou založeny na učení s učitelem, jsou však velmi závislé na velikosti a kvalitě datové sady. Právě trénovací datová sada se tedy v mnoha případech stává klíčovou pro kvalitu výsledného modelu a je velmi důležité zajistit, aby v ní bylo obsaženo co nejvíce různých situací síťového provozu.

V této práci se zaměřujeme na metody učení bez učitele, obzvláště na ty, které jsou postaveny na konečných automatech. Naší snahou je vytvořit metodu, která není příliš závislá na kvalitě a velikosti datové sady. Navržená metoda by zároveň měla být schopna vytvořit co nejpřesnější model na základě informací dostupných ze samotného toku paketů, bez toho, aby jí byly poskytnuty dodatečné informace o prostředí, hostiteli nebo zařízeních, se kterými hostitel komunikuje. Za účelem implementace a následné verifikace řešení nám byla poskytnuta datová sada skládající se ze 688 979 paketů, které byly zachyceny v rozmezí 977.997794 sekund. V této datové sadě je zachycena komunikace prostřednictvím protokolu *IEC 60870-5-104* a je v ní nasimulován útok na zařízení.

Protokol *IEC 60870-5-104* je aplikační protokol nad transportním protokolem *TCP*, který se primárně využívá ke komunikaci mezi zařízeními v *řídicích systémech*. Protokol definuje 3 základní rámce paketů: *I-Type*, *U-Type* a *S-Type*. Tyto rámce se rozlišují na základě 7. a 8. bitu v prvním ze čtyř řídích polí v hlavičce paketu. *Typ I* má 8. bit nastaven na hodnotu 0 a slouží primárně k obousměrné výměně dat. *Typ U*, identifikován pomocí posledních bitů „01“, a *typ S*, s posledními dvěma bity „11“, jsou určeny k jednosměrné komunikaci. Účelem takových paketů je zejména zaslání potvrzení nebo řídicího příkazu druhému zařízení.

V první části této diplomové práce se zaměřujeme na identifikaci jednotlivých toků takové komunikace za účelem volby co nejvhodnější struktury modelu. Hlavním účelem je rozhodnout variabilitu kombinací informací v hlavičkách IEC 104 paketů. Na základě analýzy datové sady bylo zjištěno, že různorodost kombinací těchto informací je napříč všemi toky paketů velmi malá. Proto je výhodné neseparovat modely jednotlivých toků, ale vytvořit jediný komplexní model popisující toky mezi všemi zařízeními, a poté pro jednotlivé dvojice zařízení definovat nad tímto modelem konkrétní limity jejich komunikace. Byly proto vytvořeny třídy paketů, které shlukují pakety určitých rysů, přičemž každý paket náleží právě jedné této třídě. Tyto třídy označujeme jako *modely paketů*. Každý tok paketů lze poté generalizovat na řetězec modelů paketů. Taková reprezentace odhaluje velmi zřetelnou opakovatelnost podsekvencí modelů paketů. Tyto podsekvence označujeme za *vzory* komunikace. Model celé komunikace je poté postaven na identifikaci všech vzorů, které se vyskytují v datové sadě.

V druhé části této práce se zabýváme implementací programu, který je schopný nalézt všechny vzory v poskytnuté datové sadě. Kromě toho se také zabývá opakovatelností daných vzorů v rámci konkrétních datových toků. Zkoumá, zda neexistuje perioda, na základě které by se mohl předvídat další výskyt daného vzoru. Byly zkoumány dvě metody zjišťování periodicity. První z nich je založena čistě na statistice – ukázalo se však, že tento

přístup je velmi závislý na datové sadě, odhad je velmi nepřesný a každá odchylka velmi ovlivňuje výsledek. Statistické metody se tedy ukázaly být nevhodné pro tento případ. Druhá metoda byla proto postavena na diskretizaci času příchodu vzoru. Neuvažujeme tedy konkrétní čas výskytu, nýbrž počet výskytů v rámci časového úseku. V tomto případě byly zvoleny úseky o velikosti jedné sekundy. Takový sekundový interval označujeme za *segment*. Časové okno představuje poté jednu periodu a skládá se z několika segmentů. Například periodu 0.6 sekund můžeme vyjádřit nekonečnou sekvencí časových výskytů $t = \{0.0, 0.6, 1.2, 1.8, 2.4, 3.0, 3.6, 4.2, 4.8, 5.4, \dots\}$, uvažujeme-li však počet výskytů každou sekundu, můžeme sekvenci vyjádřit posloupností $w = \{2, 2, 1, 2, 2, 1, \dots\}$. V této sekvenci se evidentně opakuje podsekvence $\{2, 2, 1\}$, která představuje časové okno složené ze tří segmentů. Výstupem algoritmu je také *jistota* (anglicky *confidence*), s jakou bylo okno určeno. Pokud je tato jistota alespoň 0.5, tedy 50% shoda, považujeme vzor za periodický. Touto metodou byly v poskytnuté datové sadě odhaleny všechny opakující se vzory, pro krátké periody (0.6 sekund) s jistotou kolem 0.9, pro dlouhé periody (20 sekund) s jistotou v rozmezí 0.8 až 1.0.

Výsledná komunikace je tedy popsána jediným modelem, reprezentujícím všechny vzory napříč všemi toky komunikace. Tento model je reprezentován jako prefixový strom. Na model navazují *deterministické konečné automaty* popisující jednotlivé toky paketů, které jsou specifické pro každou dvojici zařízení. Tyto komunikace označujeme jako *oboustranný tok*.

Model komunikace je vygenerován jako modul programu, který má schopnost verifikovat síťový tok v reálném čase. Protože síťový provoz probíhá velmi rychle, tento program musí pracovat s co největší efektivitou a být schopen zpracovat desetitísíce paketů každou sekundu. Proto je nutné eliminovat počet časově náročných operací, jako je například alokace a dealokace paměti. Modul vygenerovaný v rámci analýzy paketů je tedy sestaven tak, aby postačil ve většině případů přímý přístup do paměti a také aby bylo možné pracovat s konstatní velikostí paměti.

Program provádějící verifikaci paketů se skládá ze dvou vláken. Hlavní vlákno přijímá tok paketů a uzpůsobuje tomuto toku interní stav proměnlivé části modelu komunikace, což je prováděno v podobě přechodů mezi stavy automatů reprezentujících jednotlivé oboustranné toky. Zároveň také zapisuje počty výskytů jednotlivých vzorů do kruhového seznamu o velikosti periody vypočítané během analýzy. Tento kruhový seznam je poté každou sekundu kontrolován druhým vláknem, které jej porovnává s očekávaným počtem výskytů. Pomocí vzorce založeném na Hammingově vzdálenosti poté vypočítává jistotu, s jakou se reálné výskyty shodují s očekáváním. Pokud komunikace dospěje do neznámého stavu nebo pokud se počet výskytů výrazně liší od očekávání, problém je nahlášen.

Oba programy byly testovány na obdržené datové sadě. První třetina byla využita na trénování modelu, na dalších dvou třetinách byl model otestován. Na základě testů byly postupně provedeny tři optimalizace, díky kterým je program schopný zotavení v případě, že komunikace je pozastavena a poté znovu započata, nebo také v situaci, kdy je v síti příliš velká latence a výskyty se začnou vychylovat ze segmentů.

Z výsledků vyplývá, že s pomocí kontroly periodicity jsme schopni odhalit velké množství chyb, protože každé nestandardní chování se projevilo na jistotě, se kterou se reálný počet výskytů shoduje s očekávaným. Velká část útoku je také reflektována ve výstupních souborech, do kterých jsou hlášeny nestandardní výskyty.

Generic Flow Analysis in Computer Networks

Declaration

I hereby declare that this Diploma thesis was prepared as an original work by the author under the supervision of Dušan Kolář, doc. Dr. Ing. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Markéta Jančová
May 31, 2020

Acknowledgements

I would like to express my sincere gratitude to my supervisor Mr. Dušan Kolář for the responsible guidance of my Master's thesis and related research; for his patience, motivation, and immense knowledge.

Contents

Chapter 1

Introduction

Network traffic analysis has a special role in current computer science. It helps us to predict the behavior of devices, monitor their performance, and reveal anomalous manners. Many techniques have been studied in order to develop a model providing as accurate system description as possible. However, the construction of a precise model is an almost impossible task because the network environment is very noisy and erroneous. Dealing with the noise is usually the biggest challenge in network traffic modeling because, without a sufficient dataset, it is impracticable to precisely determine the impact of the environment and specify exact patterns in network communication. Therefore, supervised learning techniques have become very popular in network traffic analysis for their capability of dealing with these aspects. However, these methods are very dependent on the quality of the given dataset. In this work, we focus on unsupervised learning using finite automata. Our algorithms are built to do not be too dependent on dataset size and quality. We focus on acquiring as much information as possible from a raw packet stream without any additional information about the host, connected devices, or the network itself.

The main goal of this diploma thesis is to develop an algorithm that automatically creates a model of network traffic, which can be later used for traffic flow verification. The described analysis specializes in *Supervisory Control and Data Acquisition (SCADA)* systems modeling with the main focus on *IEC 60870-5-104*, which is an application protocol over *TCP*, commonly used for master/slave communication in *industrial control systems*. The gathered information is then used to create a reduced packet representation taking into account the importance of particular header fields. The reduction of data attributes associates similar packets together, despite some of their fields may be distinct. Moreover, appropriate data reduction brings benefits to subsequent analysis because it significantly reduces both space and time complexity of model computation. In the construction phase, we first investigate related works and algorithms, and then we combine several techniques with the objective to create a model, which is as accurate as possible. Furthermore, we analyze recurrence intending to estimate periods in flows because, as ascertained from initial investigations, periodicity has an important role in these systems. The obtained model and additional information are generated as a module of software that evaluates network behavior based on the given specification in runtime. The system that verifies traffic in real-time is a program that is capable to accept received packet, adjust its internal state to the current network situation, and determine, whether the state is correct or not. If there is any uncertainty about the network situation, it is reported as a vulnerability. Using this software, we are able to obtain valuable error logs and recurrent behavior monitoring across the whole process. Finally, we test our solution aiming to determine whether it meets

requirements or not, discuss obtained results, and contributions of this work in furtherance of proposing improvements and future continuation.

1.1 Thesis Structure

This diploma thesis is further divided into 8 chapters. At first, we focus on the analysis of recent studies in the field of network flow monitoring and content examination. In Chapter 2, we study IEC 60870-5-104 protocol description and structure in pursuance of estimating its variety and investigate its behavior in a network. Chapter 3 discusses related works and algorithms intending to identify advantages and disadvantages of the mentioned approaches. Based on these observations, we investigate possibilities of generic model construction, which is applicable to any unknown dataset.

Based on related works and also own observations, in Chapter 4, we introduce the given dataset and focus on appropriate preprocessing, which includes data reduction. The principal purpose of this chapter is to find a uniform lossless data representation. Such a representation helps to associate similar packets together and also reduces both space and time complexity of large dataset processing. Processed data is used in Chapter 5 to describe the proof of concept of an algorithm that automatically recognizes a model in any dataset using *deterministic finite state automata* and *prefix trees*. Additionally, it analyzes flow patterns in order to reveal recurrent occurrences in the given packet stream and estimate their period. The network model is used in Chapter 6, which proposes a proof of concept of a system, which can verify traffic in real-time based on the model which is provided as a part of the system.

Finally; in Chapter 7, we compare the obtained results with expectations for the purpose of verifying capabilities and insufficiencies of the implemented solution. Based on the result evaluation, Chapter 8 summarizes our results and briefly introduce future work.

Chapter 2

IEC 60870-5-104 Protocol

Supervisory Control and Data Acquisition (SCADA) is a control system architecture playing a critical role in power system operation and communications. These systems are designed not only to gather, analyze, and store data, but also to transfer it to a central computer facility and display the information to the operator in graphical or textual representation, thereby, allowing the operator to monitor or control an entire system from a central location in real-time [1, 44].

International Electrotechnical Commission (IEC) has established *IEC 60870* standard defining telecontrol systems. This standard has 6 parts describing general principles, characteristics of interfaces, performance requirements, etc. *IEC 60870-5* is the fifth part, known as Transmission Protocols. It was developed by the *IEC Technical Committee 57*[1] and the main goal is describing a profile of communication between devices. Five documents specify the base of IEC 60870-5 including Transmission Frame Formats (IEC 60870-5-1), Data Link Transmission Services (IEC 60870-5-2), Security Extensions (IEC TS 60870-5-7) and more. The IEC Technical Committee 57 has also created companion standards. One of these standards is *IEC 60870-5-101* presenting protocol that provides a communication profile for sending basic telecontrol messages between a central telecontrol station and telecontrol outstations. The *IEC 60870-5-104* protocol is an extension combining the application layer of IEC 60870-5-101 and the transport functions provided by TCP/IP [15, 30, 31].

2.1 Protocol Description

IEC 60870-5-104 is a *Master/Slave* protocol, what means there is one device or process controlling other (one or more) connected devices or processes (Figure 2.1). Master and Slaves communicate in two alternative ways. Either Master controls the data traffic by *Polling* (Master invokes communication with Slaves itself by sending periodic queries) and Slave passively sends responses, this process is called *Unbalanced Transmission*, or every Slave station can immediately initiate a message transfer to Master itself, what is known as a *Balanced Transmission*.

IEC 60870-5-101/105 standards define a hierarchical architecture, where every connected system is either Master or Slave. In terms of these protocols, we use conventions *Controlled Station*, the monitored system (Slave), and *Controlling Station*, the system that performs control of other systems (Master).

[1] IEC Technical Committee 57 develops and maintains international standards used in planning, operation, and maintenance of power systems. http://tc57.iec.ch/index-tc57.html

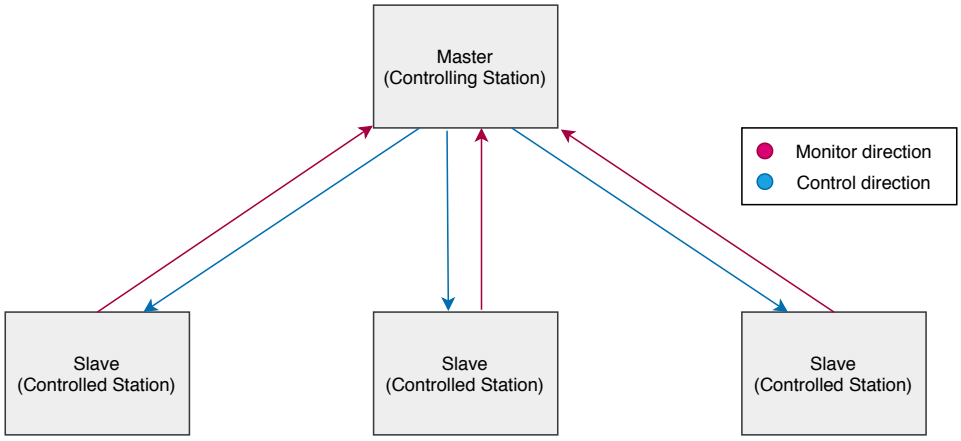


Figure 2.1: Controlling/Controlled stations and transmissions

Controlling and Controlled stations communicate in 3 types of direction:

- *Monitor Direction*, transmission is performed from Controlled to Controlling station,
- *Control Direction*, transmission is accomplished from Controlling to Controlled stations, and
- *Reversed Direction*, what is a combination of both principles enabling Controlling station to send commands and Controlled stations to send response/data (Figure 2.1) [30].

IEC 60870-5-101 defines addressing on two different levels: the *Link Address*, which uniquely identifies device, and a *Common Address of ASDU (COA)*, what in combination with information object address creates unique identification/address for each data element.

2.2 Protocol Header Parameters

IEC 60870-5-104 protocol header information is represented in *Application Protocol Data Unit (APDU)*, which is further divided into two parts. The first component is *Application Protocol Control Information (APCI)*, which starts with value 0x68, called *Start Byte*, followed by 1-byte *APDU Length* field, and four 1-byte *Control Fields (CF)*. The second part is *Application Service Data Units (ASDU)*, which is attached optionally. Thus, APDU contains either a 6-byte APCI or an APCI with *Application Service Data Units (ASDU)* (Figure 2.2).

The format of IEC 60870-5-104 APCI is slightly different from IEC 60870-5-101 APCI structure, which duplicates the APDU Length field and also repeats Start Byte. The main purpose of creating redundancy is the possible absence of a reliable transmission guarantee. IEC 60870-5-104 uses TCP, which provides a reliable transport guarantee, thus, the repetition of any field is dispensable. Each IEC 60870-5-104 APCI consists of a frame that appertains to one of three standard types. The type of frame determines the format of Control Fields.

2.2.1 APCI Control Fields

The APCI of IEC 60870-5-104 also contains four control fields. These fields differ according to the ACPI frame type. We identify three basic types of frames: I-format, U-format, and

S-format recognized by the last bit/2-bits of the first control field. Despite the size of APCI remains the same for all frame types, each frame type has a specific role and also determines whether an ASDU is attached or only single APCI is contained in APDU.

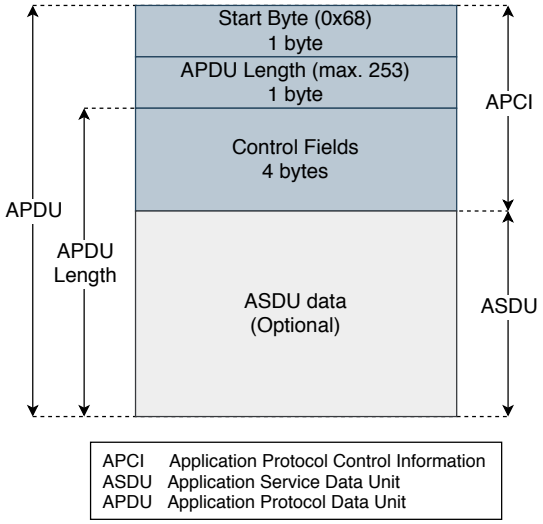


Figure 2.2: IEC 60870-5-104 APDU structure

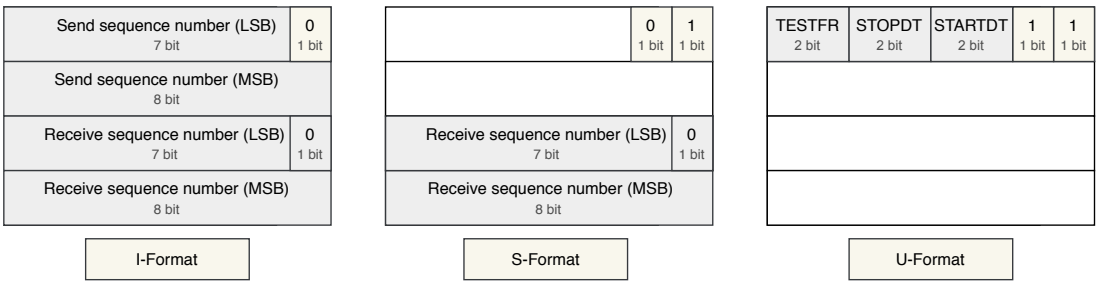


Figure 2.3: Comparison of I-format, S-format, and U-format APCI CF

Information Transfer Format (I-format)

The I-format header is identified by the 8th bit (the last bit of the first CF) set to value "0". It consists of a 15-bit *Send Sequence Number (SSN)* and also a 15-bit *Receiver Sequence Number (RSN)* (Figure 2.3). Both SSN and RSN are initially set to zero and sequentially incremented with every sent/received packet. The I-format APDU always contains an APCI and ASDU.

Whenever a transmitter sends a packet, it increases the SSN and stores packet's APDU to a buffer until it receives an acknowledgment from the receiver. The acknowledgment contains SSN of the received packet in the RSN field. When it sends more than one APDU and receives RSN equal to any of APDU's SSN, it is considered as a valid acknowledgment for all APDUs with lower SSN than received RSN.

When a station transmits a large batch of packets, it may lead to a situation where the buffer is overfilled by unacknowledged APDUs. To prevent overflow or acknowledge timeout, the receiving station sends a packet with the *S-format APCI* to signalize packet receipt.

Numbered Supervisory Functions (S-format)

The S-format APCI header is defined by value "01" in the 7th and 8th bit position (the last two bits of the first CF in the header). The other six bits of the first CF and also whole second CF are kept unused because, oppositely to the I-format, this header format does not contain a SSN (Figure 2.3). The I-format APDU always contains only one APCI.

The main and only purpose of this APCI is to signalize APDUs receipt to prevent buffer overfilling, which was described in the I-format section. Thus, RSN is a piece of essential information, it notifies the APDU transmitter that packets containing SSN in the received I-format APCI of lower or equal value than RSN in the sent S-format APCI were correctly received. On the contrary, the SSN in the S-format header is redundant because the S-format APCI is not intended to be acknowledged as it is acknowledgment itself.

Unnumbered Control Functions (U-format)

The U-format header is defined by the 7th and 8th bit both set to value "1". Contrarily to the I-format and S-format APCIs, the U-format APCI contains neither Sender sequence number or Receiver sequence number. On the other hand, the same as the S-format, the U-format APDU of this format always contains one APCI only.

In this case, the first CF has a special role. As shown in Figure 2.3, the first 6 bits are divided into three 2-bit fields: *TESTFR* (Test Frame Activation, Test Frame Confirmation), *STOPDT* (Stop Data Transfer Activation, Stop Data Transfer Confirmation), and *STARTDT* (Start Data Transfer Activation, Start Data Transfer Confirmation). Only one of these fields can be activated at a time, i.e., its value can be either set to "01" or "10" depending on the action, other fields are always set to "00".

2.2.2 ASDU Fields

The ASDU consists of a 6-byte *Data Unit Identifier* and the data itself. The data can be composed of up to 127 information objects. ASDU contains the following fields:

- *Type Identification (8 bits)* characterizes the whole ASDU content; it denotes the direction of the packet and also its purpose, i.e., whether the content is a piece of process information, system information, parameter, or a file transfer;
- *Structure Qualifier (1 bit)* is either "0", when a sequence of information objects is attached, or "1", when a single information object is included;
- *Number of Objects (7 bits)* defines the exact number of information objects in ASDU;
- *Test Bit (1 bit)* defines whether the ASDU was created during test conditions; "1" stands for "test", "0" means "no test";
- *Positive/Negative bit (1 bit)* indicates the positive or negative confirmation of an activation; the value is "0" when the confirmation is positive, otherwise "1";
- *Cause of Transmission (6 bits)* is used to provide an information that helps to interpret the message on the target device;
- *Originator Address (8 bits)* is used by controlling station to identify itself;
- *Common Address of ASDU (24 bits)* identifies all ojects contained within the ASDU. [30]

Chapter 3

Related Works and Studies

This chapter introduces existing studies and algorithms focused on network traffic analysis. Their concepts will be later used for automatic traffic model construction. We have identified three main approaches to generic network traffic modeling and analysis:

- inspection of packet header fields and also its content details in order to create an automaton that simulates the process of packet transmission between two monitored devices, which is later used to verify network traffic;
- observation of network traffic as an unknown system, which processes cannot be accurately specified, and construction of a supervised learning algorithm that improves over time; and
- measurement of the statistical aspects of a network flow such as time between two succeeding packet arrivals.

The following sections briefly introduce all three approaches and related algorithms. At last, we summarize their results and make conclusions about their usage for generic modeling and IEC 104 protocol traffic description.

3.1 Automata and Pattern Recognition

Pattern Recognition is a technique focused on a description or classification of measurements. It may be characterized as an information mapping, information reduction, or information labeling process. *Pattern Matching* is the act of checking an input sequence for the purpose of finding constituents of some template.

Pattern recognition in terms of packet stream analysis proceeds from an assumption that a substantial part of communication consists of recurrent, in many cases even periodic, events. This property utilized to construct "templates" describing some characteristics of these recurrent events. Considering a packet flow, a pattern usually represents the identity of communicating devices and other packet header parameters depending on a specific protocol, although, there are also studies taking into consideration other packet parameters such as packet size or the data content of the packet, what is usually referred to as *Deep Packet Inspection (DPI)* [35].

Using a characteristic summary, we can effectively describe an abundant amount of properties that contribute to efficient membership (pattern match) affirmation. Packet pattern matching/recognition is typically implemented using *Regular Expressions*, *Finite*

State Automata (FSM), and *(Prefix) Trees*, but also by less conventional techniques, such as *Spectral Analysis*.

3.1.1 Deterministic Finite State Automata

Definition 3.1.1 *Finite State Automaton (FA) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:*

1. *Q is a non-empty finite set of states;*
2. *Σ is a non-empty finite set of input symbols called alphabet;*
3. *δ is a state-transition function $\delta : Q \times \Sigma \to 2^Q$;*
4. *q_0 is an initial state, $q_0 \in Q$; and*
5. *F is a non-empty finite set of final states, $F \subseteq Q$ [46].*

Definition 3.1.2 *Deterministic Finite State Automaton (DFA) is a FA, where*

$$\forall q \in Q \ \forall a \in \Sigma : |\delta(q, a)| \leq 1;$$

contrarily, Non-deterministic Finite State Automaton (NFA) is a FA, where

$$\exists q \in Q \ \exists a \in \Sigma : |\delta(q, a)| > 1.$$

In previous studies, automata-based models (Definition 3.1.1) have been observed to be a very efficient solution for SCADA systems traffic modeling. However, dealing with high network error-rate and growing complexity, such as multi-periodic patterns, have appeared to be problematic due to their high sensibility [17].

In the simplest scenario, states of the FA model are representing moments between packet arrivals. Whenever a packet is received/transmitted, it is represented as a single transition. Therefore the transition function needs to be total, i.e., defined for every combination (q, a), $q \in Q$, $a \in \Sigma$; and defined exactly once for each pair (q, a). Thus, FA is usually required to be *deterministic* (Definition 3.1.2). Intuitively, such an approach provides a very clear overview of system behavior. Despite this property may be assumed to be an advantage, actually, it is also the most influential limitation of the before-mentioned systems. As described earlier, network traffic contains lots of anomalies, which are difficult to predict and/or describe, what constitutes the biggest challenge for systems built on exact match principles.

Niv & Goldenberg (2013) [17] have used an DFA to construct a model of communication based on *Modbus* [38, 21], what is an application protocol over TCP used for master/slave communication in industrial control systems. For the purpose of representing traffic by an DFA, they have made the following DFA adjustments:

1. There are not any final states – an input stream is considered as endless, therefore acceptance states are not needed;
2. every state transition is associated with an *action*;
3. the start state (q_0) is defined as a state corresponding to the first query that was recognized in the flow; and

4. a symbol in alphabet Σ is defined as a concatenation of several Modbus fields, i.e., each symbol is represented as a 33-bit key.

As mentioned above, the transition function is associated with a set of actions. Four basic types of actions were identified: *Normal* is an action identifying arrival of a known pattern element, *Retransmission* is an occurrence of a symbol identical to the previous symbol, *Miss* stands for an arrival of a known symbol but on unexpected position, and *Unknown* denotes an occurrence of a not yet identified symbol. The model is then automatically "learned" from captured traffic. Initially, the algorithm estimates pattern length 2, i.e., each flow consists of a single request and one response. In every following iteration, one symbol is processed, i.e., one step of adjusted DFA is performed, and the pattern is validated by Equation 3.1.

$$P = \frac{normals}{total} = \frac{normals}{normals + misses + retransmissions + unknowns} \tag{3.1}$$

The authors have also set a *threshold*, which is the lowest acceptable P rate. This limit is constructed of pattern lengths and components observation. If the result P is lower than the *threshold*, it means that too short pattern was selected. In this case, the algorithm continues with consequential iteration; otherwise, the algorithm stops. These patterns are then combined to a single model that is capable to verify traffic in run-time. The final model was validated using two datasets composed of packets obtained during long-drawn traffic capturing. Despite DFA models are very sensible, the model was successful in the network analysis. The "unrecognized packets" rate was less than 1%. Unfortunately, the authors did not test the algorithm in malicious or noisy environment.

Kleinmann & Wool (2016) [26] have followed the work of Niv & Goldenberg. They have observed that the quality of the DFA models is degrading by anomalies in the network during the unsupervised learning phase because they cause large expansion of the automaton. Thus, they have focused on improving the simple DFA model by creating additional steps preventing automaton growth. At first, an automaton is built for each pattern in the flow using the algorithm described by Niv & Goldenberg [17]. To deal with the size growth, they have created *Statechart* DFA that consists of many DFAs, one per cyclic pattern. These DFAs are combined together with a *DFA-selector* that de-multiplexes the incoming packets into sub-channels, represented by the DFAs. The authors improved the algorithm of pattern borders estimation using *Deterministic Time Markov Chain (DTMC)* and graph theory concepts. After building DTMC from the stream, cycles in the DTMC are detected and extracted, then each sub-graph is investigated in order to find *Euler cycle*, the test is done by evaluating simple condition that each of graph verticles has an in-degree equal to out-degree. If the sub-graph cycle condition is evaluated negatively, the sub-graph is "fixed" by adding a missing edge and/or dropping redundant edges. The last step of the algorithm is an estimation of the time between successive symbols in each of the cyclic patterns. For this purpose, each DFA retains timestamps of accepted symbols during the enforcement phase. This information helps the DFA-selector to decide which DFA to use in case of pattern symbols overlap as the time gap between successive symbols of the same pattern is usually very small. Experiments with the final model have shown that the Statechart DFA model has promising characteristics: it exhibits a low false-positive rate, it is effective in both time and space complexity, and it is scalable for multiplexed streams. However, it still does not completely deal with false-positive alarms.

3.1.2 Sequence Analysis

The previous section described automata as the models of the network state. Similar principles may be used in the static analysis that considers packet stream as a sequence of some pattern elements. With a compact representation of packet streams, for example, grouping described in Chapter 4, conventional pattern recognition algorithms might be an effective solution. These algorithms are based on the principle of determining the longest possible patterns in a given sequence. Typical pattern matching and recognition algorithms are based on regular expressions, prefix/suffix trees and finite automata [32, 34].

In the network traffic analysis and modeling, we do not usually search only for patterns themselves but we also research and analyze their relations, e.g., sequential order, for the purpose of investigating associations between pattern occurrences. Another interesting property of element sequences is periodicity of specific sub-sequences [9, 40]. Especially in industrial control systems, where most of the communication is recurrent, this information may be critical for further analysis as it may help to identify recurrent behavior and requests.

Despite this approach (with an appropriate algorithm) potentially mines all useful patterns, in the case of network traffic it cannot provide a complete model for complex communication using exact patterns because these algorithms do not consider anomalies or arrival times at all. Nevertheless, these principles can be used as a support for other algorithms to identify element's (in our case packet) context.

3.2 Probability Estimation

A network state is not a well defined notion. The same as most of the other realistic situations, it can be highly impacted by environmental aspects so the next state of the network is not securely predictable. As observers, we can evaluate this influence only from the receiver and/or sender viewpoints, which makes a precise analysis impossible. Hence we can consider these aspects as an impact of a random source on an "abstract variable" defining the current state of the network and instead of creating a deterministic model precisely predicting future behavior, we can observe events on end-stations and create a stochastic model with respect to a certain level of randomness.

3.2.1 Stochastic (Learning) Automata

As follow-up to Definition 3.1.1, we define *Stochastic Finite Automaton (SFA)* as [10]:

Definition 3.2.1 *Let M be a stochastic finite automaton defined as $M = (\Sigma, Q, P, s)$, where Σ is an alphabet, Q is a finite set of nodes, where $Q = q_1, q_1, \ldots, q_n$, s is the initial node, and P is a set of probability matrices $p_{ij}(a)$ giving the probability of transition from node q_i to node q_j, $q_i, q_j \in Q$, led by symbol "a", where $a \in \Sigma$. The probability $p(w)$ for string w to be generated by Σ is defined as follows:*

$$p(w) = \sum_{q_j \in Q} p_{1j}(w) p_{jf}, \tag{3.2}$$

where p_{jf} is the probability that state q_j is the final state for string w.

For each state, we can generalize the probability equation to:

$$p_{ij}(w) = \sum_{q_k \in Q} \sum_{a \in \Sigma} p_{ik}(wa^{-1}) p_{kj}(a), \tag{3.3}$$

where (wa^{-1}) represents prefix a of string w. Let's consider p_{if} to be the probability that the string ends in node q_i. For each state $q_i \in Q$, the transition function is limited by the following constraint:

$$p_{if} + \sum_{q_k \in Q} \sum_{a \in \Sigma} p_{ij}(a) = 1. \tag{3.4}$$

Language L is defined as a set of strings generated over alphabet Σ, formally $L \subseteq \Sigma^*$. Analogously, *Stochastic Languages* are specified as [11]:

Definition 3.2.2 *A Stochastic language L is defined by a probability density function over Σ^* giving the probability $p(w|L)$ that the string $w \in \Sigma^*$ appears in the language. The probability of any subset $X \subset \Sigma^*$ is given by*

$$p(X|L) = \sum_{x \in X} p(x|L), \tag{3.5}$$

and the identity of stochastic languages are interpreted as follows:

$$L_1 = L_2 \Leftrightarrow p(w|L_1) = p(w|L_2)\ \forall w \in \Sigma^*. \tag{3.6}$$

Considering the models described in Section 3.1.1, stochastic automata models might be one of the possible solutions to the sensitivity issue. These models introduce probability to the classical finite automata definition what removes the "precise behavior definition" problem, which led to the high false-positive rate of the final model. However, the randomness opens new issues in the network state modeling and requires a certain size and quality of the dataset to make the correct predictions. Thus, these automata are usually used in an adaptive form. In the adaptive learning strategies, the behavior of a system slightly improves every sampling period. The first idea of applying this concept on the basic automata was proposed by Mikhail Lvovich Tsetlin et. al in a series of papers describing the proof of concept. However, the term *Learning automata* was first introduced by Narendra & Thathachar (1974) [33]. In the stochastic learning automata theory, the probability distribution over automaton states is recursively updated to optimize some learning goals; initially, the distribution may not be known at all [36]. Due to the capability to adapt to changes, these automata are considered as suitable for unstable environment modeling.

However, the construction of the model that precisely predicts network behavior is a challenging task, which requires a large dataset of a certain quality. Moreover, the network state may be changing rapidly with significant fluctuation. Thus, usage of these automata is not the preferred technique for network state modeling. Nonetheless, their principles are used to build more complex probabilistic models, such as *Markov models*.

3.2.2 Markov Chains and Models

Markov Model is a stochastic model of a (randomly) changing system. It assumes that the future state of a system always depends only on the current state, past states are not considered. This characteristic is called *Markov Property*. According to the system characteristics, we distinguish 4 basic types of Markov models:

- *Markov Chain*, used for autonomous but fully observable systems;
- *Hidden Markov Model (HMM)*, when a system is autonomous and only partially observable;

- *Markov Decision Process*, in the case of a controlled and absolutely observable system;
- *Partially observable Markov Decision Process*, when a system is not well-observable but controlled.

Network state is usually treated as an autonomous system, hence both Markov Chains and HMMs are widely used for modelling, since the observability is disputable, depending on the position of the observer.

A *Markov Chain* refers to a sequence of random variables and changes between them, appointed as the probability of a transition occurrence. Let's suppose we have a set of states $\{s_1, s_2, \ldots, s_n\}$ in a chain that is currently in state s_i, then it may move to s_j with probability p_{ij} or remain in the state s_i with probability p_{ii}. Transition probabilities are commonly represented as a matrix, where rows/columns represent states and values denote probabilities between them. Considering P to be the transition matrix, and u be the probability array representing the starting distribution, then the probability that the chain is in the state s_i after n transitions is the array $u^n = uP^n$. Formally, a Markov chain is defined as (Q, A, π), where $Q = \{q_1, q_1, \ldots, q_N\}$ is a set of N states, $A = \{a_{11}, a_{12}, \ldots, a_{n1}, \ldots, a_{nn}\}$ is a transition probability matrix, where $\sum_{j=1}^{n} a_{ij} = 1$, and $\pi = \{\pi_1, \pi_2, \ldots, \pi_N\}$ is an initial probability distribution over states, $\sum_{i=1}^{n} \pi_i = 1$ [23, 18].

Contrarily to the Markov Chain, a *Hidden Markov Model (HMM)* represents a system which states cannot be observed. However, we know the output of the system, represented as a sequence of observations, and the probability distribution over specific observations for each state. Based on output stream order and the probability distribution, we can make "a guess" about possible sequence of states producing such an output. Formally, a HMM is specified as a tuple (Q, A, O, B, π), where $Q = \{q_1, q_1, \ldots, q_N\}$ is a set of N states, $A = \{a_{11}, a_{12}, \ldots, a_{n1}, \ldots, a_{nn}\}$ is a transition probability matrix, where $\sum_{j=1}^{n} a_{ij} = 1$, $O = \{o_1, o_2, \ldots, o_T\}$ is a sequence of T observations, each one is drawn from a vocabulary $V = \{v_1, v_2, \ldots, v_V\}$, $B = b_i(o_t)$ is a sequence of observation likelihoods, also called emission probabilities, expressing the probability of an observation o_t being generated from the state i, and $\pi = \{\pi_1, \pi_2, \ldots, \pi_N\}$ is an initial probability distribution over states, $\sum_{i=1}^{n} \pi_i = 1$ [23].

Markov models have become a popular technique for network traffic modeling due to their ability to represent a system which is not well-observable [7, 29], and also to identify anomalies, such as end-to-end loss process [39].

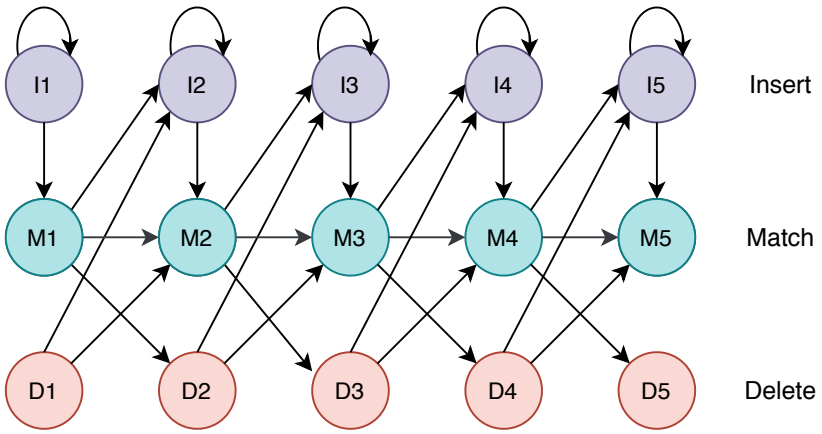


Figure 3.1: HMM Profiles

Wrigh et al. (2004) [43] have focused on identifying patterns in encrypted traffic. Previous studies have shown that examination of packet size and/or time analysis are very successful techniques for inferring encrypted communication over SSL/SSH. Therefore their goal was to create a general model using *HMM Profiles* with a capability to deduce information also from other forms of encrypted networks without specialization.

HMM Profiles are left-right HMM built around a central chain, creating groups of states (denoted by numbers in Figure 3.1). The original idea of HMM profiles comes from biology, S. Forrest (1997) [16] has adapted this idea to usage in computer security. Originally, HMM profiles consist of the following components:

- *Match* state, capturing the essential behavior that is being analyzed;
- *Insert* state, representing an unexpected insertion in known pattern; and
- *Delete* state, allowing some pattern elements to be omitted.

The authors have modified this model by splitting *match* states to two separate states to represent packet direction and interpreted state group as following: *match* states represent valid packets, *insert* states denote duplicates, and *delete* states stand for lost packets. They have used this structure to build two models, one focuses on the packet size, the other emphasizes packet *inter-arrival times*. Initially, all packets are supposed to be in every position equally likely, then 400 flows are randomly selected and used to re-estimate the initial parameters of the model. To classify a sequence, the probability of being generated for each model is computed and assigned to a *label* of the model that generates the sequence with the highest probability. Then, a *confusion matrix* is generated, illustrating the results of each classifier. Both models reached over 60% of correct classifications, for example, classification of AIM protocol was successful in more than 80% cases, on the other hand, models did not reach good results in classifying SSH and Telnet.

3.3 Arrival Time Analysis

This section focuses on studies where time is the principal property. Information about time can be implicitly available, explicitly recorded, or originally not available at all. According to this data property, we can divide time records into the following categories:

- *Event Sequence*, if time is explicitly recorded and together with data we can put records in a sequence of pairs of the form $\{(t_1, o_1), (t_2, o_2), \ldots, (t_n, o_n)\}$;
- *Irregular Time Series*, when time information is provided but it is inaccurate, the time does not play special role in such records (usually roughly measured in number of hours/days);
- *Point Sequence*, if the time series, denoted as an ordered collection of occurrences $\{t_1, t_2, \ldots, t_n\}$, are the only measured property and no further details are stored;
- *Periodic Point Sequence*, when a Point Sequence has period p with time tolerance δ and every point occurs repeatedly in every $p \pm \delta$ time units;
- *Value Sequence*, in a case where only values, which can be denoted as set $\{o_1, o_2, \ldots, o_n\}$ called *symbol sequence*, are provided and time is not recorded at all [41].

In experiments focused on gathering description from time characteristics, *periodicity* has a special place. It can be defined as a sequence of same/similar observations occurring in (more or less) regular intervals. Although, in a noisy environment, periodicity may be difficult to detect because of the following aspects:

- *Non-persistence*, despite the periodical behavior can be observed, it may be only a temporal state (dependent on another event);
- *Impreciseness*, although time information is provided, it can be inaccurate due to many aspects, e.g., network latency, overloading, and packet loss;
- *Magnitude*, the scope of possible interval values can be from milliseconds to days;
- *Quantity*, above all in high period streams, low amount of samples can be provided;
- *Noisiness*, because packet flows representing two samples of the same occurrences may significantly vary, it is important to include mechanism providing tolerance of variety [28].

In pursuit of avoiding impact of these aspects, Ma & Hellerstein (2001) [28] have introduced term *p-pattern*, representing intermittent periodic occurrences. P-pattern is defined in the following Definition (3.3.1), where w is a predefined time window demarcating temporal association; D is a sequence of all events; A is a set of all event types and A_1 is a subset $A_1 \subset A$. S_1 is a set of events, where every occurrence belongs to a specific event type A_1; δ is a predefined time tolerance of period length for specific events in S_1; p is a period length; and *minsup* is a minimum support (i.e., the minimum number of occurrences) for p-pattern:

Definition 3.3.1 *A set of event types $A_1 \subset A$ is partially periodic temporal association (p-pattern) with parameters p, δ, w, and minsup, if the number of qualified instances of A_1 in D exceeds a support threshold minsup. A qualified instance $S_1 \subset D$ satisfies the following two conditions:*
(C1) The set of event types of events in S_1 is equal to event types A_1 and there is a t such that for all $e_i \in S_1$, $t \leq t_i \leq t + w$.
(C2) The point sequences for each event type in S_1 occur partially periodically with the parameters p and δ.

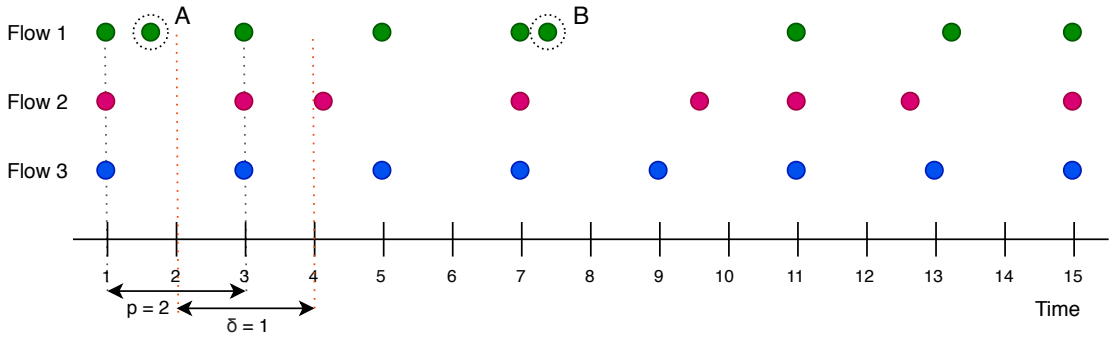


Figure 3.2: Example of periodic behavior in packet flows

Figure 3.2 shows 3 packet streams in a timeline. Each dot represents one packet arrival. Supposing time window $p = 2 \pm 1$ and restriction to a single arrival in the window, we can observe that *Flow 3* is indisputably periodic. On the other hand, in *Flow 2*, there are some diverging arrivals from the expected moments of occurrences. Despite there are inter-arrival time dissimilarities, the deviation is always lower than δ, therefore this flow can be also considered as periodic. Considering *Flow 1*, divergence became higher than δ in two occurrences (marked as "A" and "B"), so this stream cannot be taken as periodic.

The time segmentation approach helps to deal with situations when periodic behavior is not constant but appears only in particular time intervals (e.g. it is invoked by another event). *On-segments* are the parts when we expect periodic behavior, contrarily *off-segments* are the parts when we do not expect any occurrence, if present, we consider it as a noise.

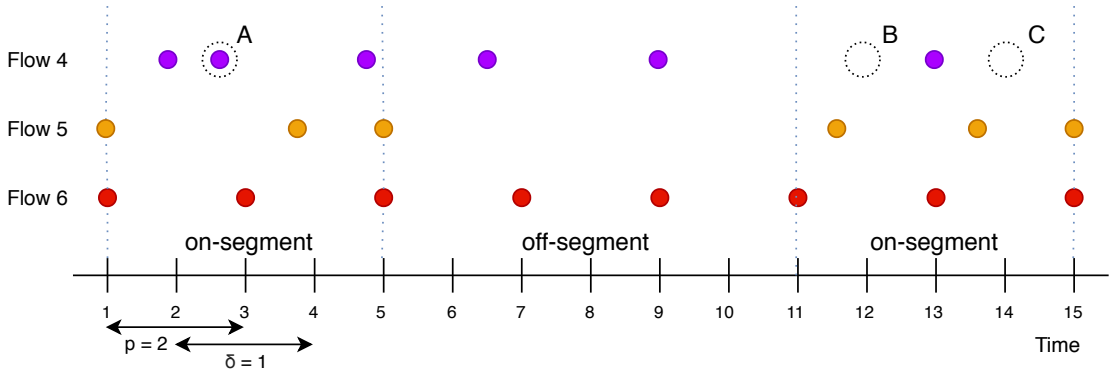


Figure 3.3: Example of periodic behavior in segments

Figure 3.3 shows 3 packets streams in a timeline, taking into account a time window with period $p = 2 \pm 1$ and restriction to a single packet in the window, and also with respect to segmentation, which includes on-segments of length 4 interspersed with the off-segment of length 6. Despite *Flow 6* is fully periodic in all moments, only intervals <1,5> and <11,15> are taken as valid because events in interval $(5, 11)$ happened inside off-segment, so they are recognized as noise. *Flow 5* is also periodic, although there are arrivals with deviation higher than δ, they are found only outside on-segments. Oppositely *Flow 4* is periodic inside off-segments only, otherwise it violates periodicity restrictions (denoted as "A", "B", and "C").

As mentioned earlier, analyses based purely on time series analyses have proved to be surprisingly successful. In the following two sections, we discuss two of the most widely used techniques for behavior estimation without any investigation of the packet content.

3.3.1 Estimation Using Statistics

Since periodicity in a network stream is a well-known characteristic of packet flows, there have been done many studies focused on flow analysis based on packet arrival or *inter-arrival* times, defined as:

Definition 3.3.2 *Let* $S = \{S_1, S_2, ..., S_n\}$ *be a set of timestamps denoting sequence of packet arrival times. Inter-arrival time* $t_{kl} = |S_k - S_l|$ *is a time between arrivals* S_k *and* S_l*;* $S_k, S_l \in S$*;* $k \neq l$*.*

Hubbali & Goyal (2013) [22] have studied network summarization techniques in order to detect network anomalies. They have observed that periodic behavior exhibit very low *variance* (σ^2) and *standard deviation* (σ), which is given as:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}, \tag{3.7}$$

where N is the total number of items and $\overline{x}$ is *average*, computed as:

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i. \tag{3.8}$$

On the other hand, in (near) random streams we can observe very high variance. Hubbali & Goyal have also introduced term *DiffTime*. Considering $P_1, P_2, \ldots, P_N$ as a series of packets exchanged between host H and other devices addressed as $IP_1, IP_2, \ldots, IP_M$, $M \leq N$, over a period of time, there may be one or more single-directional flows represented as $F_1^{IP_K}, F_2^{IP_K}, \ldots, F_J^{IP_K}$, $J \leq N - M + 1$, with $t_1^{IP_K}, t_2^{IP_K}, \ldots, t_J^{IP_K}$ representing timestamps of the first packets of these flows. DiffTime is the difference $t_{l+1}^{IP_K} - t_l^{IP_K}$, denoting inter-arrival time of the first packets from two distinct successive flows. Communication is determined to be periodic if the DiffTime standard deviation is below a predefined threshold. Every (single-directional) network traffic between two hosts can be defined as *FlowSummary*, denoted as $SrcIP, DstIP, LS, SS, M, t_1$, where $SrcIP$ and $DstIP$ are IP addresses of endpoints, LS is a linear sum of DiffTime $\sum_{l=1}^{M}(t_{l+1}^{IP_K} - t_l^{IP_K})$, SS is a squared sum of DiffTime $\sum_{l=1}^{M}(t_{l+1}^{IP_K} - t_l^{IP_K})^2$, M is the number of flows during the period, and t is a timestamp of the first packet in the last flow. FlowSummary represents all flows with avoidance of keeping packet or inter-arrival times details and provides essential details to calculate standard deviation at any time. When the DiffTime standard deviation is lower than the estimated threshold, communication between two devices is considered as a periodic one in a particular direction. Although the algorithm has proved to be a very effective method of periodicity detection, it reveals periods only in a purely periodic single-directional stream. The algorithm does not consider protocol at all, what is the biggest issue for complex multi-periodic communication, as well as it does not consider any stream noise.

3.3.2 Spectral Analysis and Fourier Transformation

Packet arrival times can be taken as a series of indexed data points, called *Time Series*. A time serie is defined by its 3 main components:

- A trend component represented as a long-term pattern of time series;
- a periodic component, denoted by a repeating pattern of a certain period and shape;
- a random component, symbolizing an impact of uncontrollable variation on demand, expressed by non-periodic patterns.

In time series analysis, the observed part is the periodic component. Two types of periodic time series were analyzed: *Seasonal*, which means an interval of recurrence cannot be precisely defined and is not constant, and *Cyclical*, which stands for patterns that are

repeating in a constant time [37]. Time series analysis regards event occurrences as a discrete signal. Thus, several well-established signal processing tools and techniques such as *Fourier Transform* and *Autocorrelation* can be used to analyze periodicity [5].

Fourier Transform (FT) identifies underlying periodic patterns by transforming a signal into frequency domain. It decomposes the original signal in a linear combination of complex sinusoids called *Fourier Series*. The main frequencies within the signal are corresponding to *amplitude*, a measure of its change over a single period, and *phase*, a relative variable describing how far along the wave is in its cycle, of these sinusoids. The phase and amplitude are recorded in *Fourier Coefficients*. *Discrete Fourier Transorm (DFT)* then helps to obtain corresponding *Periodogram* of the discrete signal, which is an estimate of the spectral density. The periodogram is provided by the squared length of each Fourier coefficient, which gives information about the strength at a given frequency. Finding the power at each frequency provides us a piece of information about the dominant frequency. The main disadvantage of this method is that the accuracy of the discovered periods deteriorates for large periods [42, 4].

Autocorrelation is a method to measure a similarity between values in a signal by correlating the time series with itself. It is defined by *Autocolerration Function*. Unlike Fourier transform, the autocorrelation function estimates periods precisely for both short and long time intervals. To determine the dominant period significance threshold needs to be specified, which is difficult to set automatically [24, 37].

Kleinmann & Wool (2016) [25] have followed up their previous work, described in Section 3.1.1. They have suggested an extension of the Statechart model to make the analysis more accurate. The authors have observed that the major reason for the false alarms is an inaccuracy of the algorithm that splits channel to sub-channels (what determines patterns in the stream). This inaccuracy is caused by the strict combinatorial requirements that characterize Euler cycles. Therefore, a new construction algorithm based on spectral analysis was proposed. The new algorithm treats the captured trace of SCADA packets as a binary signal, where "1" indicates the presence of the packet at a particular time. Then, they calculated the Fourier transform for the signal and its periodogram to identify dominant periods. Each dominant period corresponds to a cyclic pattern in the packet stream. After transforming results back to the time domain, each symbol in the trace can be associated with one of the dominant periods, sub-sequences of symbol streams associated with a period are treated as sub-channels so finally, a DFA is created for each sub-channel. By combining these DFAs, the authors have created full Statechart, similarly as in Section 3.1.1. Proposed Statechart improvement has consistently outperformed the previous combinatorial Statecharts. During stress-testing, the algorithm exhibited reduced false-positive rate (0.16%) for synthetic dataset and 0% false-positive rate on production traffic.

3.3.3 Timed Automata

Timed Automata (TA) are a compromise between naive FA model and time analysis. These automata have the ability to consider the time aspect of a system behavior. They were proposed by Alur & Dill (1994) [3] to model systems where time is an important parameter, such as real-time systems.

In the original form, every TA has one or more *Clocks*, which are variables representing real-time. Clocks are initially set to zero and their value constantly and monotonically increments. The main purpose of including such a variable is to define constraints over

Timed automata	Supported features					
	Hierarch. structures	Number of clocks	Absolute time	Relative time	Non-determinism	Value changes within a state
Basic TA	×	multiple	✓	×	×	×
Hierarchical TA	✓	multiple	✓	×	×	×
R-T Statechart	✓	multiple	✓	✓	✓	×
Scenario TA	×	multiple	✓	×	×	×
R-T automata	×	single	×	✓	×	×
Prob. TA	×	multiple	✓	×	✓	×
Prob. hybrid TA	×	multiple	✓	partially	✓	✓

Table 3.1: Comparison of timed automata based models. Data retrieved from [27].

transitions to limit the transition function. These time constraints are explained in Definition 3.3.3.

Definition 3.3.3 *For a set X of clock variables, a set $\Phi(X)$ of clock constraints δ is defined inductively by*

$$\delta := x \leq c \mid c \leq x \mid \neg\delta \mid \delta_1 \wedge \delta_2,$$

where x is a clock in X and c is a rational number.

Formally, we define *Timed Automata* as:

Definition 3.3.4 *Timed Automaton is a tuple (Σ, Q, q_0, T, C) where:*

- *Σ is a finite set of symbols (actions),*
- *Q is a finite set of states,*
- *C is a finite set of clocks,*
- *$q_0 \subseteq Q$ is a finite set of start locations, and*
- *$T \subseteq Q \times \Sigma \times 2^C \times \Phi(C) \times Q$ is a finite set of transitions. An edge $(q, a, \lambda, \delta, q')$ is a transition from the state q to state q' reading symbol "a". $\lambda \subseteq C$ is a set of clocks that will be reset by the transition and δ is a clock constraint over C.*

The authors have defined *timed word* as a pair (σ, τ), where $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ is a finite word over alphabet Σ and τ is a time sequence. On the basis of previous definition, they have described a *timed language* as a set of timed words over alphabet Σ. Timed languages are usually defined with a set of constraints over their timed words. For example, a language with constraint allowing all transitions after the time has passed constant z may be characterized as $L = \{(\sigma, \tau) | \forall i.((\tau_i > z))\}$.

From the automaton definition, we can intuitively deduce that TA, oppositely to the classical FA, does not define any final states, thus, computation of the automaton is an infinite sequence. A computation is called *recurring* iff it contains infinitely many configurations in which the location counter has the value 1. A problem of deciding the automaton recurrence has been proven to be *NP-complete* (proved [20]) [3].

The concept of timed automata has been used to extend existing models to be able to express behavior in time. Kumar & Niggemann & Jasperneite (2009) [27] have observed their characteristics and advantages in network modeling, some of their conclusions are

shown in Table 3.1. Compared to the standard finite automata, expressiveness has greatly increased in all discussed models. In most cases, apart from the basic timed automata, a modeling power is also stronger than the basic finite automata.

3.4 Evaluation of Approaches

Previous sections have introduced several algorithms and techniques used to build a model of network traffic. In the first section, we have introduced algorithms based on deterministic finite state automata. Although DFA have the ability to model a behavior of a well-defined system, dealing with traffic noise is usually a challenging task as we are not able to describe it precisely. The noise led to false-positive alarms in both mentioned algorithms, naive DFA and Statechart.

Algorithms based on probabilistic models, such as stochastic automata and Markov models, countenance the fact that the network state cannot be well-defined as we can observe it only from the point of view of the sender and/or receiver. To deal with this fact, these algorithms use probability estimation in order to predict system (network) behavior. Although these algorithms are adaptive and evince required characteristics, they are mostly based on supervised learning, which requires a learning phase and convenient dataset that demonstrates various situations. In our case, we perform unsupervised learning on any unknown dataset, which might be very small, containing a few packets.

At last, we have introduced algorithms based on statistics, which mostly include time series analysis. Time was observed to be critical information in periodic streams such as traffic between devices in industrial control systems, where the communication is mostly recurrent. However, we consider models based on time analysis inconvenient, because in our case, the traffic is multi-periodic, i.e., there are many periodic patterns over time, which we need to differ. One of the possible solutions are timed automata because of their capability to represent real-time systems.

Chapter 4

Data Preprocessing

This chapter introduces our dataset and methods of processing that were applied to make this data more suitable for further analyses. In our case, the dataset consists of 688 979 packets; and initially, there is not any information provided about its content. To process data, we require a consistent form enabling to match some data pieces together and find associations between them. To reach a uniform data representation, we perform data preprocessing to make the data form and volume more convenient for further analyses. Preprocessing in terms of packet streams focuses mainly on data filtering and reduction in order to reduce data volume and also remove some needless data attributes. The following paragraphs briefly introduce the most common data reduction strategies [2, 19, 45].

Dimensionality Reduction (Figure 4.1) is a technique of suppressing redundant and non-critical attributes in order to select only the most critical parameters and get rid of attribute redundancy. This strategy includes techniques such as:

- *Feature Selection*, a process of selecting the smallest subset of attributes with the least possible information loss.
- *Attribute Construction*, which is based on merging related attributes together and creating new "summary" attributes;
- *Feature Extraction*, a method of searching descriptors in source data, it usually includes mapping to a new (smaller) set of attributes;

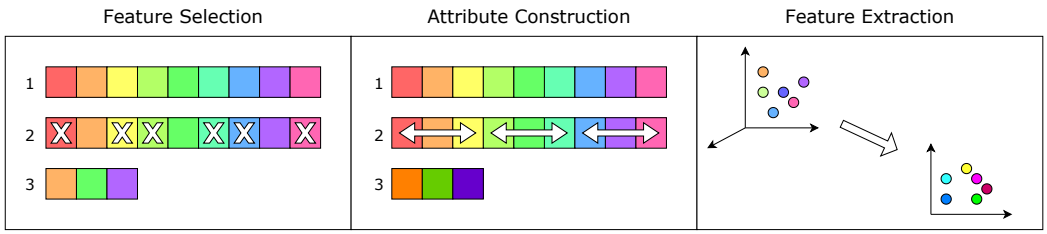


Figure 4.1: Dimensionality reduction strategies

Data Compression is a process of modifying data to optimize its storage size. It commonly includes data encoding, converting bit structures, and removing repetitive elements and symbols. Depending on the storage size reduction and selected algorithm, the process is lossless or lossy.

Data Cube Aggregation (Figure 4.2) is a process where aggregation operations are applied to the data in the construction of a data cube, which stores multidimensional aggregated information.

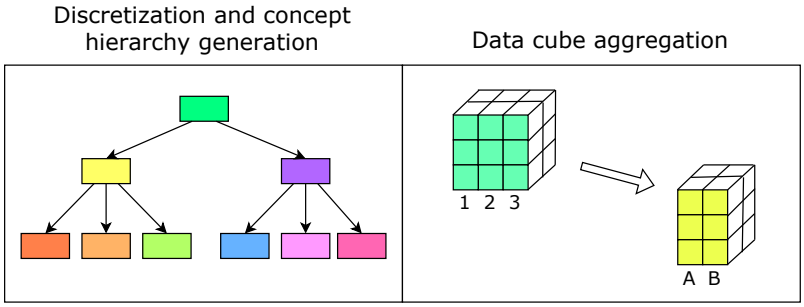


Figure 4.2: *Data cube aggregation* and *discretization and concept hierarchy generation* reduction strategies

Numerosity Reduction reduces the data volume by replacing the original form of data by a smaller form of representation. It can be either *parametric* or *non-parametric*. Parametric methods, for example, *regression*, typically use some model to estimate new data, thus only parameters need to be stored. Non-parametric methods, such as *clustering*, are used to store a reduced representation of data.

Discretization and concept hierarchy generation (Figure 4.2) is used to reduce and simplify continuous values using discretization. These techniques can be categorized based on the type of discretization (supervised or unsupervised) and its direction (top-down or bottom-up).

Sampling (Figure 4.3) is a process of selecting a subset of data in order to create a complete (or as accurate as possible) representation of dataset with fewer records. This technique is usually based on selecting a random or *stratified* sample, which guarantees even distribution.

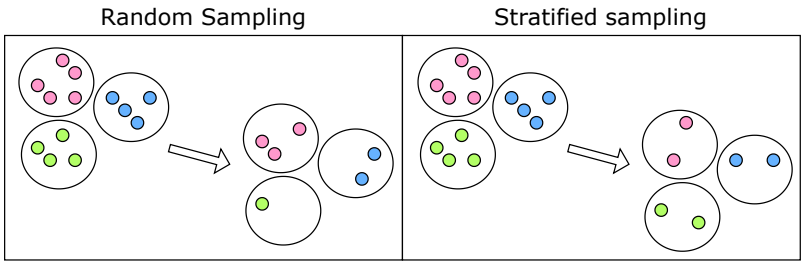


Figure 4.3: Sampling reduction strategies

4.1 Reduction of Packet Stream Data

Network traffic has a high error rate that complicates the analysis process. Packet streams commonly include anomalies, such as a single packet failure, or even larger scale issues, e.g., multiple packet losses caused by network overload. Moreover, it needs to be taken into consideration that packet stream consists of different protocol types and we need the ability to recognize packets, which should be considered during analysis and monitoring.

At first, we need to filter out packets where the IEC-104 non-membership is evident. In our case, we do analysis based on L7 protocol information (considering *ISO/OSI model*) and previous layers are used only for addressing purposes. Hence, despite TCP Acknowledgement is considered as a part of the response to an I-format packet frame, it is considered to be a noise. The first analysis is focused on determining header control field variety with the main intention of identifying all possible combinations for packet aggregation. We use a dataset consisting of 688 979 packets that were sent/received during 977.997794 seconds of capturing. Dataset does not contain only IEC 104 packets but also a plentiful amount of TCP operations (over 30% of packets), DNS, NBNS, and ARP packets.

4.1.1 Header Fields Variety

The first step in data volume reduction is focused on the header field variety investigation. The main goal of this analysis is to determine, whether there are some evident patterns in packet sequences. By "pattern" we understand a set of succeeding packets with some specific values in protocol headers. We search for patterns that appear repetitively.

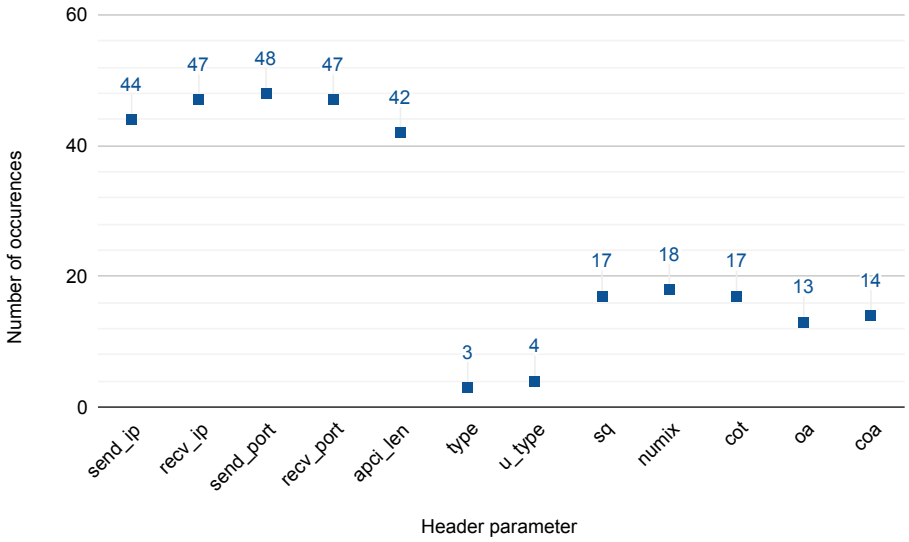


Figure 4.4: Variability of header parameters

Figure 4.4 shows the number of different values in every field of the IEC 104 protocol. Analysis has shown that the dataset consists of 44 unique senders and 47 receivers which communicate (or try to communicate) from 48 (senders) and 47 (receivers) different ports. Surprisingly, the variety of protocol-specific fields (described in Section 2.2.2) is rather small even for fields without predefined value groups.

According to this observation, dimensionality reduction appears to be a profitable approach. Generalization of some fields by grouping them together without information loss may significantly reduce data dimensionality. Because variety seems to be low, header fields `type`, `u_type`, `sq`, `numix`, `cot`, `oa`, and `coa` were grouped together by exact field match. This approach creates 16 different abstract models of IEC 104 packets. These models consist of 1 S-format, 4 U-format and 11 I-format frames. Packet data content is not considered at all in described models.

Table 4.1 shows the field specification of the identified packet model groups in the dataset. Every group is assigned to a single number from a successive number sequence. Every packet processing corresponds to one of the following scenarios:

1. The packet corresponds to some known model, its timestamp and identification (sender and addressee) are recorded. Then, the packet is thrown and replaced by the group identifier of the matching model.

2. The packet does not correspond to any known model, a new model, which conforms packet fields, is created. The successive number from the sequence counter is assigned to the new model and the counter is incremented. Then, the process described in (1) is performed.

3. The packet contains an invalid field value (a value that is outside defined field scope) and/or some essential protocol field values are missing. In this case, the packet is considered as unrelated (not IEC 104) and thrown away.

Using this algorithm, the data amount is substantially reduced. Instead of storing 17 fields for every I-format packet, only 2 attributes for each packet are stored in a global table. This approach brings profit in both space and time complexity of processing algorithms described in the following chapters because both memory usage and number of comparisons are significantly reduced.

Group ID	First occurence	Type	UType	SQ	Numix	COT	OA	COA
0	1	0		100	1	6	0	1
1	3	0		100	1	7	0	1
2	4	1						
3	5	0		1	2	20	0	1
4	8	0		11	9	20	0	1
5	9	0		100	1	10	0	1
6	102	0		1	4	20	0	1
7	103	0		11	15	20	0	1
8	190	3	16					
9	192	3	32					
10	23406	0		100	1	6	0	3
11	23407	0		100	1	7	0	3
12	336912	0		45	1	6	0	1
13	336914	0		45	1	47	0	1
14	360622	3	1					
15	360520	3	2					

Table 4.1: Packet Summary Groups

4.2 Flow Extraction

Packet stream can be considered as a *bi-directional flow*. Thus, we implicitly do not differ sender and receiver of a packet but contemplate the packet as a part of a flow identified by two endpoints, between which the packet is transmitted. Although exact identification

of sender/receiver does not play a critical role in initial flow analysis, this information is kept as one of the packet summary model parameters to be able to separate individual flows during final model construction.

Whenever a new packet is received, its TCP header is extracted in order to record `sender_ip`, `receiver_ip`, `sender_port`, and `receiver_port`. Based on these four parameters, a string called "identity key" is created to represent the bi-directional flow. Such an identification is then stored in a map, which associates these identity keys with specific packet flows. After the received packet is linked to a specific packet summary model (described in Section 4.1), its Group ID and timestamp are stored in a list-represented packet stream associated with the particular identity key.

An output of the described algorithm is a set of endpoint pairs and packet streams between them. Keeping sent and received packets in a single stream helps to evaluate associations of sub-flows inside flow such as recognition of requests and related responses. We also keep the timestamp of every sent/received packet intending to track time gaps between succeeding packets.

4.3 Behavior Observations

A deep investigation of packet streams led to several observations of protocol behavior.

As mentioned in Section 4.1, protocol fields combination variety is rather small so grouping fields together into "packet models" is a profiting technique. After splitting traffic into separate streams based on source and target endpoints (Section 4.2), there are evident patterns in communication, which indicate periods in packet streams. Despite periodic patterns contain a large amount of "noise" (inserted packets, missing packets, interchanged position of packets...), two patterns are never fused; however, they may be sent right after the other. Besides periodic patterns, there are also noticeable occurrences of asynchronous events, such as asynchronous reset.

Chapter 5

Implementation of Automatic Model Recognition

This Chapter focuses on the implementation of a program able to describe network traffic and represent the description as a model of communication. Proposed algorithm is based on the combination of both existing solutions and own observations. Henceforward we suppose packet streams in the form defined in Chapter 4: IEC 104 packet header fields are summarized as a pair of packet group ID and timestamp. Such a packet representation is then assigned to a specific couple of endpoints, identified from the packet TCP header, which creates a single packet sequence for each identified bi-directional packet stream.

The further analysis focuses on the examination of specific bi-directional streams, which are from now on considered separately (identified by endpoints). We aim on the identification of patterns in flows in order to find associations between them. Such an approach enables generalization of a stream, using which we can determine packet correctness from both content and position viewpoints.

5.1 Splitting Flows to Sub-flows

The first stage in the bi-directional stream examination is the separation of independent parts of the flow. We consider a part of the packet stream as "independent" if it is surrounded by significantly larger packet arrival delays than all delays inside the part. Such a part of a stream can be considered as a separate piece of communication, which is not directly dependent on surrounding packet exchanges. We define several naming conventions that will be used in this chapter (Figure 5.1):

- A *Flow* denotes a packet sequence between two particular endpoints, i.e., consists of packets transmitted through the same bi-directional channel.
- A *Sub-flow* denotes a part of a flow consisting of succeeding packets (their inter-arrival times are very low). Each flow consists of one or many sub-flows.
- An *Inter-arrival time inside sub-flow* denotes a moment between the arrival of directly succeeding packets (of the same sub-flow), i.e., inter-arrival of packets inside the same sub-flow.

- An *Inter-arrival time between sub-flows* denotes a moment occurring between two sub-flows, i.e., a situation when one sub-flow has already ended but the other has not started yet.
- A *Sub-flow pattern* is a template of sub-flow. The number of patterns is always lower than the number of sub-flows. Such a template defines sub-flow as a sequence of packet models.

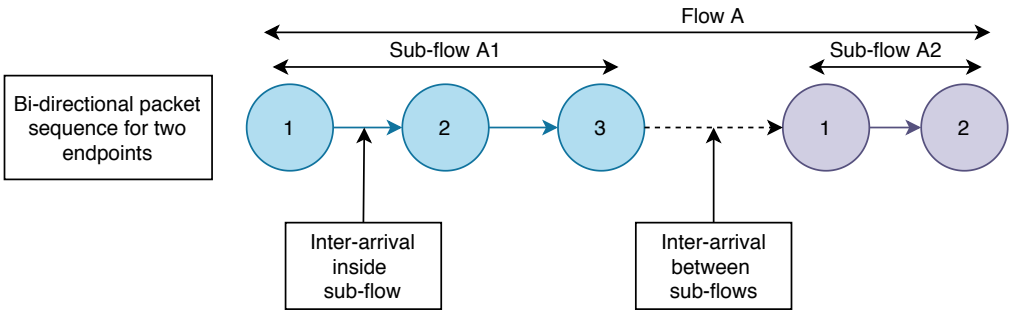


Figure 5.1: Naming conventions definition

We also consider a *threshold* to be the maximal inter-arrival time inside a single sub-flow. Whenever an inter-arrival time is higher than the threshold, the succeeding packet is considered as the start of a new sub-flow. In further analysis, we proceed from the following observations that will be considered as facts:

- Considering $(f_1, f_2, \ldots, f_n)$ to be a sequence of n sub-flows denoted by k patterns, $k \leq n$, and f_i, f_j are two subsequent sub-flows, where $j = i+1$, and $(p_{i1}, p_{i2}, \ldots, p_{im})$, $(p_{j1}, p_{j2}, \ldots, p_{jn})$ are sequences of packets inside sub-flows f_i and f_j, respectively. In valid recurring sub-flows, the inter-arrival time $|p_{im} - p_{j1}|$ is always higher than any inter-arrival time of two subsequent packets inside sub-flow f_i or f_j.
- In every periodic flow, the period of recurrent sub-flow is always higher than average inter-arrival time inside all sub-flows of the flow.

Our approach is concentrating on sub-flows that can be securely demarcated; i.e., the inter-arrival times of preceding and succeeding sub-flows are "high enough" to be considered as distinct. Despite inter-arrival times variety is large and we cannot precisely determine the threshold, we can estimate a very low threshold with the purpose to separate the most obvious independent sub-flows with a high certitude. Despite this approach does not delimit all sub-flows, many of them might still be united, it provides certain information about possible patterns.

To select an appropriate minimal threshold, we have chosen a method of finding a convenient midpoint value. We proceed from the assumption that the difference between succeeding packets inside sub-flow and between sub-flows is significant. A midpoint value of such time series should be impacted enough by the high inter-arrival times so the midpoint is securely higher than directly succeeding packets. There are several basic techniques of midpoint estimation:

- *Mean* (defined in Section 3.3.1) computation is impacted by all inter-arrival values. For the given dataset (described in Chapter 4), average value is around 120000μs.

- *Median* is the middle value in a sorted time series. If the number of arrival times is even, we compute it as an average of two middle values. For the given dataset, the median is around 7000μs.
- *Mode* is the most frequent value in given sequence, it can be roughly estimated by an equation $(average - mode) = 3 \times (average - median)$. In our case, we do not consider mode at all because arrival times are continuous, absolutely same inter-arrival times are a very rare occurrence.
- *Midrange* is a difference between the lowest and highest inter-arrival times. For the given dataset, it is around 2000000μs.
- *Interquartile Mean* is a mean that takes into consideration values between the first and fourth quartile, in our case 18000μs.

For the initial estimation, we have selected a *mean* of all inter-arrival times as a threshold because it evinced the best accuracy from all naive approaches. *Median* and *mode* are unsuitable for streams consisting of long patterns because most of the inter-arrival times in such streams are very low. On the other hand, *midrange* considers the highest and lowest inter-arrival times which causes a selection of abnormalities. *Interquartile/bounded mean* cuts off a substantial part of the lowest and highest inter-arrival times what might be also problematic for short patterns containing only a few packets (all inter-arrival times between two sub-flows might be excluded). Therefore, we consider *mean* as the most suitable naive approach.

After setting the threshold, every flow is processed in order to detect packet inter-arrival times higher than the estimated threshold, these packets are marked as the start of a new sub-flow. The output of the described algorithm is an ordered group of short packet sequences (called sub-flows) for each pair of communicating devices. Although, some sub-flows might still be attached together due to low inter-arrival time.

The algorithm described above was applied to the given dataset (described in Chapter 4) in order to verify correctness of our suppositions. We can observe that the inter-arrival time of two packets inside the same sub-flow is usually in the scope of several microseconds. On the other hand, the inter-arrival time between two sub-flows matching the same pattern is usually higher than 0.5 second, which is approximately 100 000 times more. Approximately, we separate a sub-flow after finding a packet which arrived after longer time than 0.27 second, what securely keeps all inter-arrival times inside one sub-flow united. Although, recognition of such inter-arrival time differences is getting complicated by the fact that despite the communication is mostly purely periodic, there are usually many recurrent events at a time (with different periods), which complicates the recognition of periodic sub-flows with a long period. For these periods, there is a high probability that the most of their occurrences arrive right after/before different sub-flow (typically with smaller period). This issue causes a situation, where most of the pattern occurrences are attached to different sub-flow. Hence the knowledge of patterns in flow is required for splitting flow to sub-flows with a certain level of confidence.

5.2 Complete Traffic Model

After splitting streams as described above, every bi-directional flow is represented as a set of sub-flows. These sub-flows can be generalized into patterns, where each pattern represents

a unique sub-flow. Because packet models describe only packet header fields, without considering any endpoint-specific information, patterns are very similar across all bi-directional flows. Thus, instead of describing communication separately for each bi-directional flow, we have selected an approach of describing the whole network environment uniformly. Such a complete model consists of all possible patterns across all streams.

To describe the whole network environment, we use a special tree structure, called *Prefix Tree*. A prefix tree, which is also known as a *trie*, is a special type of a *search tree*. It was invented by Briandais (1959) [13]. The main idea of creating this structure was to extend a binary tree structure with the capability of string representation. To reach this property, every node represents exactly one character (in a compressed version of the structure may represent a string). In such a tree, nodes are not binary but contain maximally one outgoing edge for each possible character that can follow the character in the specific position the node represents. The degree of any node is always lower or equal to the size of the tree alphabet [8]. The root is considered as a special node, which stands for an empty string. Traversal of a trie creates a string, called *tree prefix*, consisting of all passed nodes. Any two distinct traversals always produce different prefixes.

Using a prefix tree, we can define the complete model of monitored communications. The complete model is gradually constructed by matching sub-flows of all flows with a temporal general tree representation. A temporal tree representation contains initially only the root node. With every new pattern, the size of the tree progressively enlarges. For each sub-flow in every flow, we perform a tree traversal intending to find the longest prefix of the tree matching with some prefix of the pattern that is verified. If the longest prefix does not correspond to the whole pattern that is under verification, the temporal tree is elongated to cover the sub-flow the pattern represents. The temporal tree structure is final when its prefixes cover all possible sub-flows.

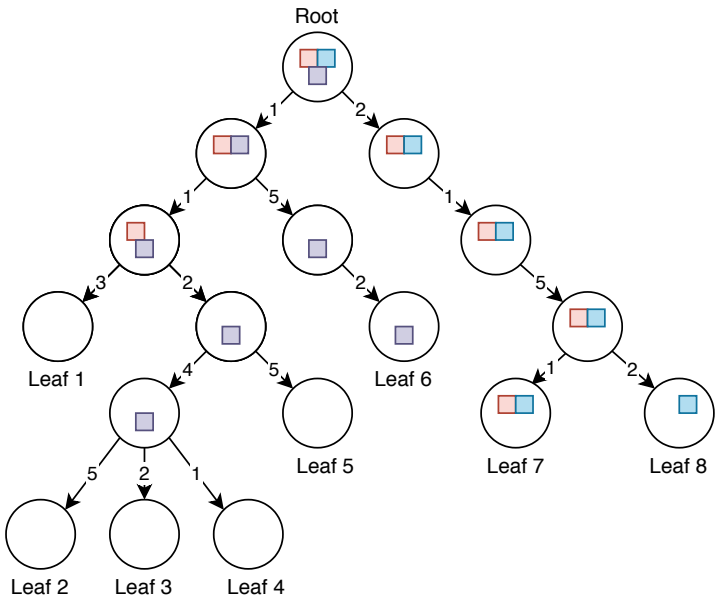


Figure 5.2: Prefix tree representing generic model

The main advantage of such a structure is the effectiveness of inserting patterns and pattern presence verification. Moreover, every pattern prefix is stored only once for all existing patterns. Intuitively, in such a tree, some patterns become a part of a longer pattern. For

example, a pattern representing a sequence of packet models “1225” is absorbed by pattern “122572” and represented as the longer pattern without any reference to the shorter pattern existence. Therefore the complete model represents only the longest possible sequences by their prefixes. Specific flow is then represented as a “partial sub-tree” of the prefix tree and sub-flow is denoted as a sequence of nodes which begins from the tree root and ends in any child node (not necessarily a leaf). Figure 5.2 visualizes the described architecture, colored squares represent packet sub-flows.

A *Tree pattern* is any part of a tree consisting of directly succeeding nodes. An example of tree patterns is shown in Figure 5.3. Thus, considering a flow as a set of general prefix tree traversals, where each traversal represents one valid sub-flow; these traversals can be combined together into a single pattern of the general prefix tree. Applying this approach to all monitored flows, all valid communications can be denoted as a subset of all tree patterns.

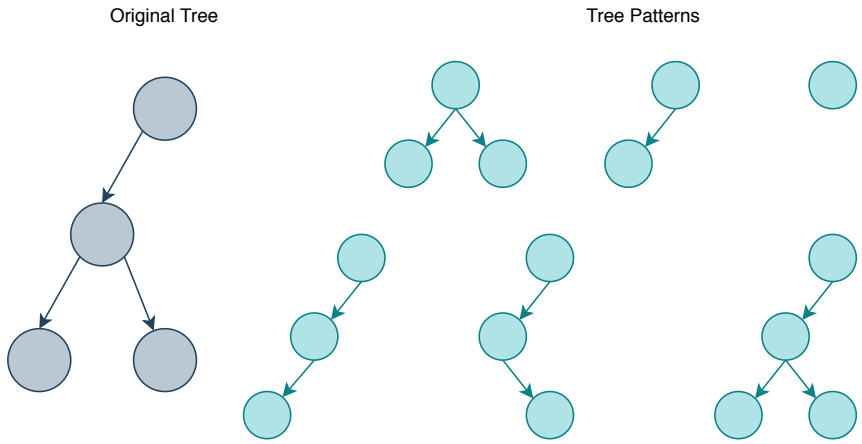


Figure 5.3: Example of a tree and its patterns

Using the described algorithm, we have obtained a structure representing all stream patterns across the whole environment. However, as mentioned above, there are short valid patterns hidden inside longer patterns. Furthermore, some patterns are valid for one bi-directional flow but invalid for the other. Thus, in the following section, we focus on restricting the set of expected patterns for each bi-directional flow and also specify patterns, which are hidden inside a prefix of others.

5.3 Bi-directional Flow Analysis

After the complete model is constructed, flows are analyzed separately for each pair of end-points. Considering a group of all patterns that are valid for a particular bi-directional flow, our main goal is to create an automaton, which accepts all these patterns and nothing else. Such an automaton represents a complete communication of the specific bi-directional flow. Figure 5.4 visualizes the connection of the general tree and a flow automaton; note that this automaton is defined for each bi-directional flow but general prefix tree is one for the whole environment.

For further analysis, we consider a traffic between two devices to be one pattern of the general traffic model. Let’s suppose T to be a general traffic model tree described in the previous section. This tree can be delineated as a set of all possible patterns of

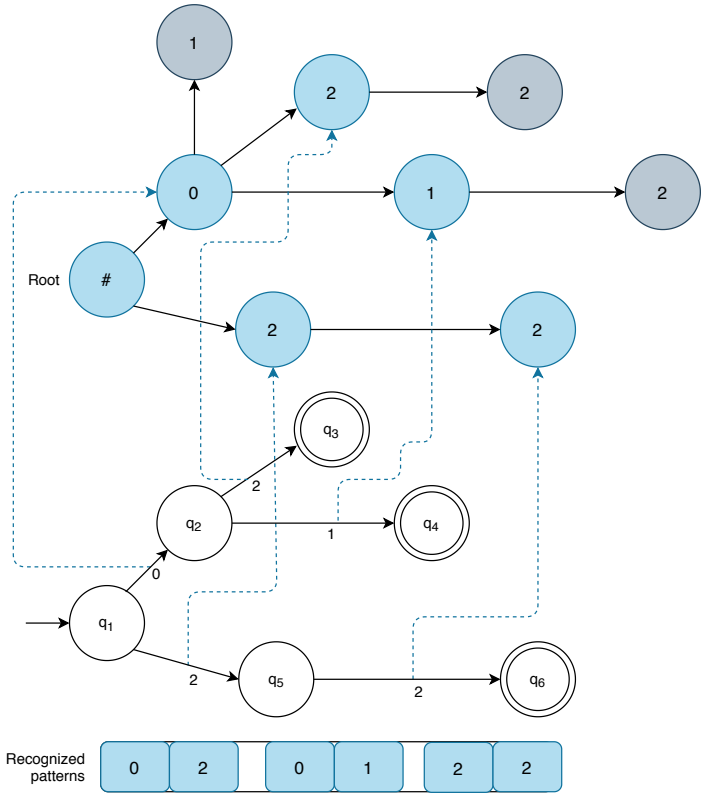


Figure 5.4: Mapping of flow automaton to the general prefix tree model

this tree, denoted as $T = \{t_1, t_2, t_3, \ldots, t_l\}$. $T_p \subseteq T$ is a subset including only patterns that correspond to at least one complete flow. Each flow pattern in T_p contains a set of subsequent nodes, where each node has a single attribute g_i of a value equal to the group ID of the packet that the node represents. Let's consider $A = \{A_1, A_2, \ldots, A_m\}$ to be a set of deterministic automata, where each automaton represents one tree pattern from the set T_p. An automaton representing a pattern of a prefix tree accepts all valid prefixes that can be denoted by the pattern. These automata correspond with the general tree model as following: Every automaton A_i, $1 \leq i \leq m$, is defined as a tuple $(Q_i, \Sigma_i, \delta_i, q_{i0}, F_i)$ where symbols correspond to Definition 3.1.2 with an addition that transition function is defined with a limitation that every state $q \neq q_0$ has a unique rule containing the state on the right-hand side, i.e., every state excluding start states has always single incoming transition. Components of these automata are specified as:

- Q_i is a set of states corresponding to the inter-arrival moments, i.e., edges of the general tree model.
- Σ_i corresponds to group numbers of packets contained in the sub-flow.
- q_{i0} is a state representing a moment between any two sub-flows.
- F_i is a set of final states, $F_i \subseteq Q_i$, that represents the situation occurring after the last pattern element arrived.

A language generated by the described automaton is evidently finite, by definition it does not contain any cycles.

Such a structure provides us a complete model of the flow. As we mentioned at the start of this Section, a flow model expressed as an automaton generating patterns is mapped to the general tree model (each state transition points at a particular tree node). This property helps us to compare distinct flows in order to find a pattern applicable to both of these flows. Furthermore, using this representation, we can perform a basic verification of packet position correctness. Henceforth, we focus on verifying other communication aspects, such as recurrence and time analysis. Up to now, we were not limited by the fact that two occurrences of the same pattern might be a little bit distinct. Currently, the only issue it causes is a potential model growth. In the following analysis, we focus on finding associations between flows. However, such an analysis is significantly impacted by every deviation, for example, swapped packet order. Thus, in the following section, we focus on the correction of the most evident errors in streams to minimize the impact of minor deviations.

5.3.1 Noise Correction

In this section, we focus on the detection of invalid patterns. As an "invalid pattern", we consider any pattern that:

- consists of two sub-flows which should have been separated,
- has swapped packet order,
- has duplicated packet, or
- misses packet.

At first, we construct all flow patterns by creating a set of unique sub-flows (or generating all sentences of the automaton). Then, we assign arrival times to these patterns. For example, considering pattern "1322" which occurs 3 times in the flow, we merge these occurrences together and assign their timestamps to the pattern, so we obtain pattern:

$$1 : \{ts_{11}, ts_{12}, ts_{13}\}; 3 : \{ts_{21}, ts_{22}, ts_{23}\}; 2 : \{ts_{31}, ts_{32}, ts_{33}\}; 2 : \{ts_{41}, ts_{42}, ts_{43}\}.$$

After forming unique patterns, the attached sub-flows are detected by searching for their components that are isolated. To separate a pattern, we need to find two distinct patterns, which together can construct the joined pattern. Furthermore, we require that the separated form should have more occurrences than the joined form. This scenario is visualized in Figure 5.5. The 4th row consists of two previous patterns, the 1st row is a prefix ("C"), the second row is a suffix ("D"). Similarly, the 5th row contains pattern from the 1st row ("B"), appended to the pattern from the 3rd row ("E"). After a fusion is detected, timestamps from the joined pattern are attached to timestamps of the isolated patterns.

In the third stage, we reduce swapped packets. Incorrect packet order is usually caused by network delay or packet loss. This situation is visualized as "A" in Figure 5.5. The 3rd row is similar to the 4th row; however, two packets are swapped. The matching algorithm should respect these deviations and consider given patterns as the same.

Detection of corrupted patterns is a challenging task because a strict algorithm does not recognize pattern swaps, on the other hand, a lenient algorithm may match unrelated patterns. Hence, we have built an algorithm that accepts single failure in a *window*. We define the window as a dynamic group of packets of a floating size which starts at a moment when the first error occurs, i.e., the first position index where non-matching packets are

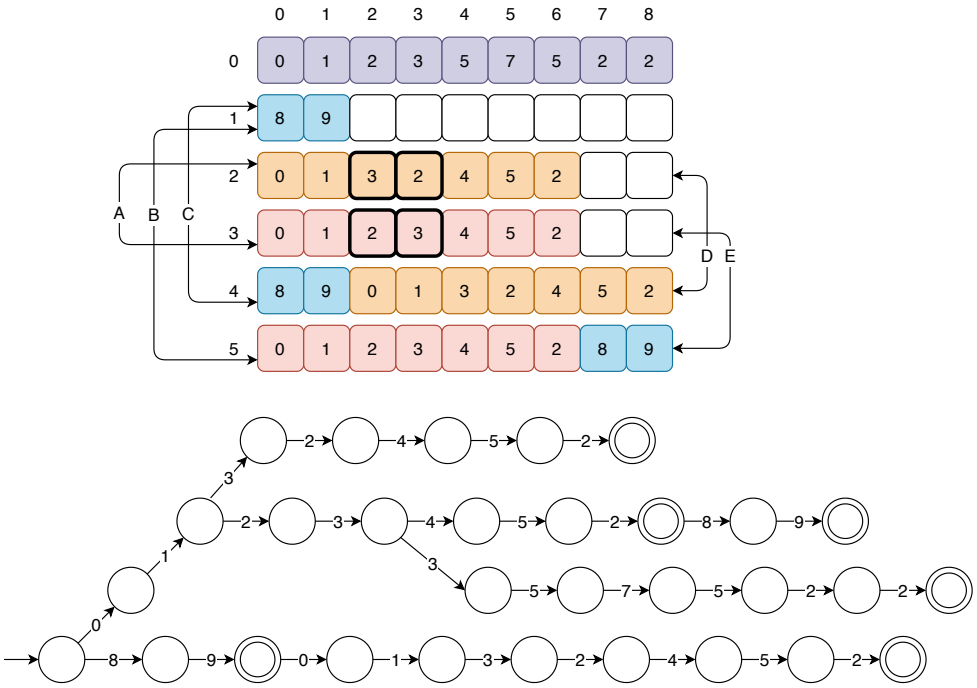


Figure 5.5: Resolution of noise and deviations

found, and ends at a moment when patterns are equal. Algorithm has two input parameters: *inserting pattern* and *original pattern*. The *inserting pattern* is a pattern that is considered as the noisy one and the *original pattern* is the target pattern form. When the first mismatch is found, it is considered as an inserted packet in the *inserting pattern* and its value is saved until this element is found in the *original pattern*. Intuitively, this algorithm needs to be executed for all non-matching pattern pairs twice, so each of these patterns is tested as both "inserting", what reveals potential packet insertion, and "original", what reveals packet deletions.

If two patterns are considered as matching in one way or the other, timestamps of the pattern with lesser occurrences are joined to timestamps of the pattern of higher occurrence. Figure 5.6 visualizes 4 scenarios of such a comparison. Multiple failures are accepted in the case, where there is only one "held failure" at a time, which is shown in scenarios 2 and 3. Scenario 4 holds packet 4. At a point when 2 is expected but 3 is given and 4 is still not released, the pattern is considered as "too noisy" so patterns are evaluated as mismatching. This algorithm naturally respects also a single missing element and a single inserted element. Detailed steps of described algorithm are provided in Algorithm 1.

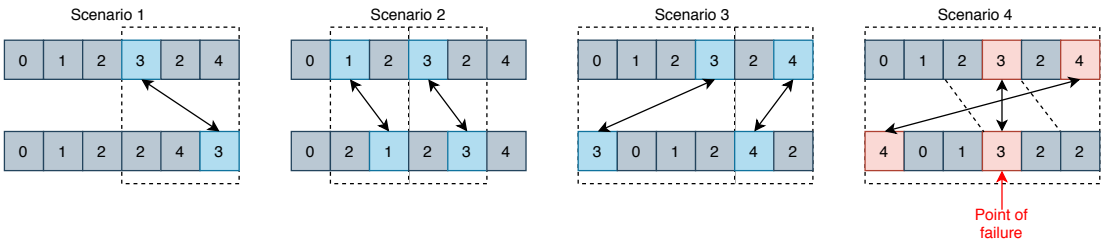


Figure 5.6: Pattern swap in a window

```
Data: inserting and original
Result: True when patterns are matching, otherwise False
if abs(inserting.size-original.size) > 1 then
    return False;
end
while not the last index of longer pattern do
    if current original is equal to held element then
        move to following index in original;
        release held element;
    else
        take both current elements;
        if current elements not matching then
            if we already hold something then
                return False;
            else
                hold current inserting element;
                move to following element in inserting;
            end
        else
            move to following element in both inserting and original;
        end
    end
end
```

Algorithm 1: Single failure in a window algorithm pseudocode

5.3.2 Periodicity Estimation

Finally, we estimate the periodicity of extracted patterns. Previous steps joined some groups of patterns in order to narrow time gaps of missing segments in an arrival time series. In the final part, we calculate approximate inter-arrival times between pattern occurrences in flows, which later helps to detect recurrent behavior and ideally also estimate a relevant period.

In a perfectly periodic stream, all sub-flow inter-arrival times should be identical. However, due to latency and other aspects, there might be deviations that need to be respected. In related studies, we have discussed algorithm FlowSummary (Section 3.3.1), which computes standard deviation and compares it to the threshold to determine whether it is low enough to be periodic. The authors set the threshold to 10 seconds. In our case, we have observed two recurring patterns and estimated their periods to 0.5 and 2 seconds. We consider the analysis based on the strict threshold as inconvenient for generic modeling, because initially, we do not have any information about periods in packet streams.

To provide as accurate result as possible, we have implemented two algorithms. The first one is inspired by FlowSummary, thus, completely constructed on stream statistics. The second one is built on *Dynamic Time Warping (DTW)* [6] with an adjustment of using *Hamming distance*. In the following sections, we discuss both these approaches.

FlowSummary-based Algorithm

We have established a measure that determines periodicity using available general information obtained from the given dataset. The main goal is to avoid manual selection or a requirement of additional information. Before the algorithm is performed, the following preprocessing steps are required to be performed, in order to guarantee consistency of input data:

1. For each sub-flow f_i in flow F where $i > 1$, we calculate the difference in arrival timestamps between the last packet of f_{i-1} and the first packet of f_i.
2. For each f_i we set arrival time of the first packet to zero.

Then, the sequence of inter-arrival times between sub-flows is sorted into an ascending order what reveals the minimal and maximal time it took between two pattern occurrences in the given flow. Although it needs to be taken into account that large periods might be impacted by unmatched patterns, i.e., pattern occurrences that were too noisy to be considered the same as the original pattern.

Following processes proceed from the assumption that in periodic streams, the number of mentioned "too noisy sub-flows" is minimal. These occurrences should be surely lesser than 20% of the longest inter-arrivals, otherwise we consider the given flow as a recurrent without established period. Compliantly with previous assumptions, it is expected that the middle 60% of inter-arrival times are nearly the same or "very similar". Therefore, both 20% of the largest and the smallest intervals are filtered out.

The generalization of defining "very similar" inter-arrival times is a challenging task. Intuitively, this measure cannot be no matter how dependent on the inter-arrival value itself. The reason is simple, if we supposed acceptable deviation as a proportion of average value, the result would be very high acceptable deviations for large periods. On the other hand, small periods would have unattainably low maximal deviation requirements. In reality, there should not be any difference in acceptable delay for both small and large periods because this measure is not dependent on the period length but on other aspects, such as network state, load, device condition, etc. However, these conditions are hardly observable/unobservable in a short continuous packet stream without any initial knowledge or estimation.

For initial analysis, we have selected the mean of middle 60% of inter-arrival times as expected period. The acceptable deviation value is set to the duration of the longest sub-flow in the particular flow. This approach proceeds from the assumption that involved devices invoke communication based on an internal clock, which is expected to be accurate. The delay is mostly caused by one of the following four aspects:

- internal processing on stations,
- network latency,
- packet losses and retransmissions, or
- waiting for the finish of the previous communication.

In the common state, the fourth aspect has the biggest impact on packet sub-flow delay. Supposing multi-periodic flow where distinct periodic patterns have different periods, sooner or later, some periods will overlap the others. In the worst scenario, the longer pattern

precedes the shorter, which needs to wait until the whole packet exchange passes off. This process might critically impact the waiting exchange process. These assumptions led to the approach of selecting the longest duration of the stream of directly succeeding packets (i.e., the longest sub-flow duration). This duration is expected to be "high enough" to cover common latency and delay. Moreover, despite this measure is different for each bi-directional flow, it is the same for all sub-flows in the particular flow, i.e., it does not depend on the period length.

DTW-based Approach

As we have already discussed earlier in this Section, statistical methods are too dependent of the training dataset and every deviation is reflected in the result. To avoid this issue, current periodicity mining studies usually assume the input to be a sequence of symbols instead of time series [12]. For example, let's consider a communication between two devices where A, B, and C denote different packet sequences. Let's suppose a time window $t = \{1, 2, \ldots, 6\}$, where sequence A occurs at times $t_A = \{1, 4, 6\}$, B at $t_B = \{2\}$ and C at $t_C = \{3, 5\}$. We can represent such a situation as a string "ABCACA". This is usually called a *symbolic sequence*. Intuitively, such a representation is suitable for purely periodic sequences where we expect events to happen in a fixed order. When we search for a periodic event in a set of non-periodic events, representation of a stream should not mix these events together, otherwise, the periodicity detection might be impossible. Thus, the timestamp stream $t = t_1 t_2 t_3 \ldots t_n$ can be represented as a binary sequence, where $t_i = 1$ denotes "event occurred at t_i", contrarily, $t_y = 0$ denotes "event did not occur at t_y". *Hamming distance* is one of the most popular algorithms for comparing symbol-represented streams. It compares sequences position-wise symbol by symbol; the output of this algorithm is the number of non-matching symbols.

Elfeky et al. [14] (2005) have focused on finding periodicity in noisy streams, i.e., they assumed insertions, deletions, and swaps to be a part of a packet sequence, what is usually not considered in previous algorithms based on Hamming distance. Let's consider strings $A = adbcde$ and $B = abcde$; comparing these sequences position-wise, the number of mismatches would be 5, on the other hand, considering symbol "d" to be an insertion, the number of mismatches would be 1. Such an approach is called *time warping (TW)* [6]. TW is usually computed dynamically, thus, it is also referred to as *Dynamic Time Warping (DTW)*. Considering $X = [x_0, x_1, \ldots, x_n]$ and $Y = [y_0, y_1, \ldots, y_n]$ to be two sequences of symbols and $\tilde{X} = [x_1, \ldots, x_n]$ and $\tilde{Y} = [y_1, \ldots, y_n]$ are X and Y after the first symbol is removed; we can define DTW as

$$DTW(X, Y) = d(x_0, y_0) + min \begin{cases} DTW(X, \tilde{Y}) \\ DTW(\tilde{X}, Y) \\ DTW(\tilde{X}, \tilde{Y}) \end{cases} \tag{5.1}$$

where $d(x_i, y_j)$ is the distance between the ith symbol in sequence X and the jth symbol in sequence Y. The distance is determined as "0" when symbols match and "1" when symbols do not match. To compare two strings, a matrix is created, where columns denote symbols of the first string and rows denote symbols of the second string. Cell (i, j) then contains the result of comparison of symbols x_i and y_j. A *Warping Path* of a matrix $n \times n$ is the path between cell $(0, 0)$ and $(n-1, n-1)$. A *Warping Cost* is then a sum of all cells crossed by the particular warping path. In this algorithm, we search for the minimum warping path, which is the total DTW distance (Figure 5.7).

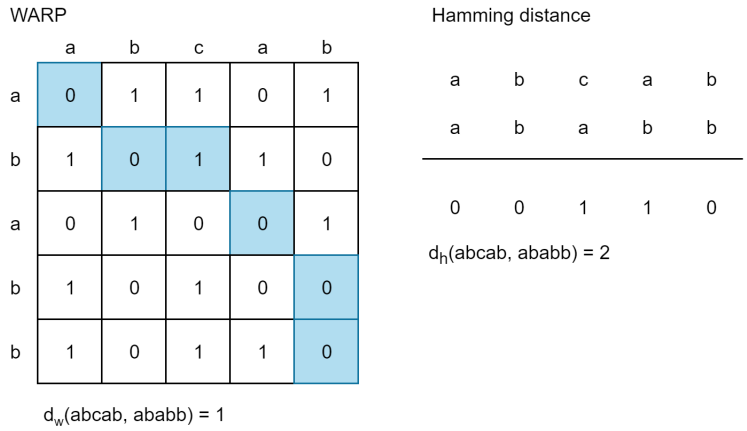


Figure 5.7: Comparison between Hamming distance and WARP

Elfeky et al. have used the time warping algorithm to construct a new algorithm called *The Warping for Periodicity*, which has the main aim to detect periodicity in symbolic sequences. They proceed from the assumption that when we cut off p symbols from a periodic string with period p, the shortened string will be matching to the initial non-contracted string. For example, let's consider string $T = abcabcabc$. Obviously, this string is purely periodic. When we start cutting off symbols from the beginning of this string, earlier or later, we will get to the second period of the string, which is detected by a very low difference rate in comparison with the complete string. The authors define *perfect period* as a period, after which all symbols match with the non-contracted string version. Correspondingly, a period, where several mismatches are detected is referred to as *candidate period*. To determine string similarity, they use a measure called *confidence*, calculated as

$$c = \frac{n - p - DTW(T, T^{(p)})}{n - p} \tag{5.2}$$

where n is the string length and p is the number of removed symbols.

In our case, we combine both Hamming distance and WARP principles to construct an algorithm that automatically detects periodicity in any packet stream. As far as two distinct patterns in a packet stream never depend on each other, we need to perform periodicity detection for each pattern separately, i.e., we consider only one event in the system, therefore, instead of considering symbols to be (different) events, we consider symbols to be the number of occurrences of a single event in a time segment of a specific length. The time segment length is set to 1 second and it is represented as the number of particular occurrences during this period. For example, let's suppose event X occurred at times $t_X = \{0.2, 0.5, 0.7, 1.2, 2.8, 2.9, 4.1\}[s]$, considering time window that starts at 0s and ends at 5s, time series t_X can be represented as sequence a "31201". Intuitively, symbols in this representation are ordinal. Hence, in contrast to nominal representations discussed earlier, this representation not only determines whether symbols are different but also concludes how much. A distance between two symbols is calculated as

$$d(x, y) = \frac{|x - y|}{w_{max}} \tag{5.3}$$

where w_{max} is the maximum of occurrences in a single segment across all segments. This capability ensures that two nearly the same windows, such as $A = 8788$ and $B = 9678$, are

considered as similar. On the other hand, segments $C = 1011$ and $D = 1100$ are considered as different. Note that in the first case, A and B have very low Hamming distance on every index because the unit of distance is 1/9; contrarily, in the case of C and D, the unit of distance is 1, thus, these sequences significantly differ. We have combined this approach with the principle of the Warping for periodicity algorithm proposed by Elfeky et al. At first, we represent time series as described above, and then we remove the first element from the sequence. Both strings are then aligned to start at the same index and compared to obtain *error rate*. The error rate is characterized as

$$err = \sum_{i=0}^{n} d(T_i, T_i^{(p)}) \tag{5.4}$$

where n is the length of the shortened string, T denotes input string and $T^{(p)}$ stands for its left-shifted version by p symbols.

We also consider that the value of information about 0 occurrences is different than about (at least) 1 occurrence. The information that 1 event was expected and 1 occurred is intuitively much more valuable than the information that 0 events were expected and 0 occurred. Hence a situation where both indexes have 0 are not considered as a "match" but are excluded from the calculation. Thanks to this approach, a comparison between "0100" and "0001" is not considered as a similarity.

Based on described assumptions, the confidence calculation was adjusted to

$$c_e = \frac{n - p - err - z}{n - p - z} \tag{5.5}$$

where z denotes the number of matching zeros in given strings, and n, p correspond to symbols in DTW definition.

Comparison of Algorithms

In previous Sections, we have introduced two algorithms estimating periodicity in streams.

The FlowSummary-based algorithm focuses on statistical aspects of periodic streams and proceeds from the assumption that in a purely periodic stream, inter-arrival times between the same pattern occurrences are nearly identical in all cases. However, using this algorithm, some not negligible differences are difficult to predict. For one stream, there might be a deviation up to ±10% of its period; but for the other, there may be a deviation up to ±25% of the period. This estimation is even more complicated if we consider environmental aspects. We may respect deviation ±10% but in real-time analysis, there may be deviation ±13% due to temporal latency. Moreover, after trying different approaches of estimation, we have reached maximally around 70% of correctly guessed periods. Despite we were successful in 93% of short period estimations, in the case of very long periods, only 47% of calculations were correct. Long periods are influenced more because up to 40% of the boundary values are removed, which significantly reduces the dataset.

Then, we have introduced a customized WARP algorithm. This algorithm aims not to be too dependent on minor deviations and more focused on the number of occurrences during a particular time window. In this case, we are not dependent on statistics at all, we include all values to calculation, and treat all streams the same. Moreover, using this approach, we are able to appropriately identify all periods in the given dataset. Based on

these arguments, we assume the WARP-based approach to be the more convenient solution for our case.

5.4 Model Generation

The final stage of model construction is a generation of convenient representation of all valuable information that can be provided to the software that verifies traffic based on the given model in runtime. The main requirement for this functionality is to provide a complete model, which can be used without any additional processing needed not to slow down the target device. Thus, the target model is provided as a C file, which consists of related information represented as constants and initialized structures/arrays. The model is split into three separated components: *packet models*, *general model*, and *bi-directional flows models*.

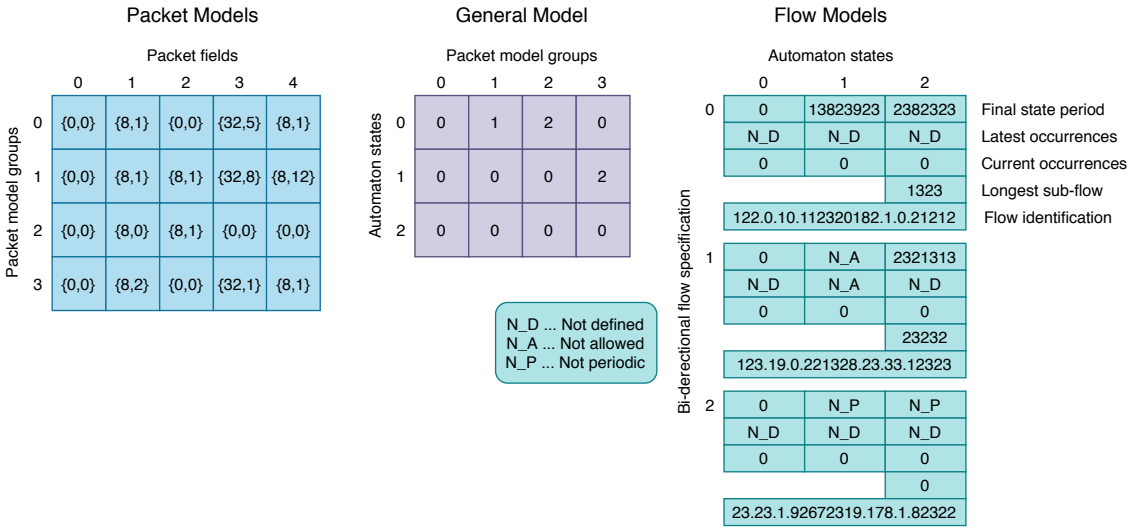


Figure 5.8: Model components

The first part represents packet fields and model identification that stands for a specified combination of values (referred to as "packet group ID"). To keep this model as generic as possible, every packet header field consists of two parameters: the number of bits n and the value v. The combination of these parameters creates pairs $\{n_i,v_i\}$. If the field value is considered as unimportant, it is denoted by the pair {0,0}, which signalizes that the field value should be skipped during parsing. Despite the field length is applicable for all values in a given column, we represent it separately. The added value of such an approach is that specific IEC 104 types use different fields, for example, U-type keeps all protocol-related fields empty. This representation was selected in order to distinguish different interpretations/usages of the same fields.

This way represented packet fields are merged into a sequence, what creates a matrix where rows denote packet model identifiers and columns denote specific field values. Intuitively, the number of rows is equal to the number of packets that vary from others and the number of columns is equal to the number of all captured packet fields. Whole structure is represented as "Packet Models" in Figure 5.8. The matrix is accompanied by several convenient constants, what facilitates process of going through the matrix.

The second part of the model represents the general model described in Section 5.2. The described model is transformed to a transition function of automaton, represented as a matrix where columns denote symbols and rows denote states. In cells, there are target states of specific transitions. We call this component "General model", which is visualized in the middle of Figure 5.8.

Finally, the third part of the model represents specific packet streams. These streams are the only non-constant part of the model as they are created to reflect current state of the network. Each flow is represented by 5 records:

1. flow endpoints identification,
2. final state period,
3. latest occurrences,
4. current occurrences, and
5. the longest sub-flow duration.

The first item uniquely identifies both communicating devices by their IPs and ports. Items 2, 3, and 4 are represented as arrays of 64-bit timestamps. Their length is the same as the number of states, i.e., the number of rows in the general model.

The *Final state period* is an array, which denotes accessibility and periodicity of states, i.e., whether the general model's state is a member of the particular model patterns. Items of this array can take a value from one of the following groups:

- "0" when the state is not final;
- "N_A" when the state is not allowed, i.e., it is final but not a member of any pattern in the flow;
- "N_P" when the state is allowed but not periodic, i.e., it can occur in any time;
- period in microseconds when the final state is periodic.

The *Latest occurrences* element is an array, which is initially completely set to generic values:

- "N_A" when the state is not the final state of allowed periodic pattern or
- "N_D" when the state is an allowed periodic final state.

Contrarily to the Final state period element, these values are changeable in time, so they can represent the current state of the network. Their main purpose is to trace periodic occurrences and guarantee that once they start occurring, they continue. At the same time, it helps to track that a recurrent event does not occur more often than expected.

The last element is an array of "current occurrences". These values are all set to zero because, initially, we suppose network with 0 received packets. When the first packet is received, it is set to the value of the current timestamp. Succeeding packets do not refresh this time, they only move the value to other states; at the end, we can compare the initial value in the case of long delay somewhere during a packet exchange. This component is shown as the "Flow model" in Figure 5.8. Intuitively, this array might have been merged with the *Latest occurrences* array, as they both represent the current state. However, we

need to take into consideration the situation when patterns overlap, i.e., a pattern might be a part of the other. Therefore, we cannot consider the first reached accepting state as the completion of a pattern.

These components are merged together into a single C module which represents all information obtained during analysis. We refer to this module in the following chapters while dealing with verification of network traffic in real-time.

Chapter 6

Implementation of Runtime Traffic Analyzer

This chapter describes the construction of software that has the capability to verify traffic in real-time. This program proceeds from the model described in the previous chapter. The generated model is compiled as a part of the runtime analyzer, which dramatically lowers the time of initial processing and also prevents the need to allocate memory dynamically. Usage of constant memory size in runtime very positively affects both performance and security.

The analyzer consists of two threads, each of them has a specific role. The main thread reads packets from *standard input (stdin)* in CSV format, where every line represents a single packet and columns represent its header fields. Based on the acquired information, it verifies packet correctness in terms of position in the packet stream. Oppositely, the secondary thread verifies communication from the statistical point of view, e.g., when the traffic is expected to be periodic, this thread verifies its periodicity confidence. These threads are implemented separately to maximize the performance of packet processing and to create a precise timing of periodicity checking, which should happen in a strictly defined period to ensure a required amount of data for confidence calculation was gathered. The most reliable way of reaching this property is to create a dedicated thread with the only competency to check statistical correctness and to produce details of time analysis. Both threads together verify communication from the time perspective, content correctness, and a specific packet occurrence with respect to a specific context/state. In the following sections, we introduce the technical details of these threads and their connection.

For further description, we consider the model to be a part of the runtime analyzer. Thus, both threads can access data generated in the training phase and have information about expected communication and its occurrence frequency. Generated module consists of a *model part* and *runtime part* (Figure 6.1). In the model part, there are only constant data, used for behavior verification. Oppositely, in the runtime part, there are mostly dynamic data reflecting the current network state. Although data content is mentioned to be changing, its size is stable. New values are never added; whenever the state is changed, existing values are updated. The internal state always reflects the current situation; there are not any historical records (however, past states are certainly mirrored in the current state). Besides performance benefits, this property also ensures stability from the point of view of memory usage and prevents increasing memory consumption.

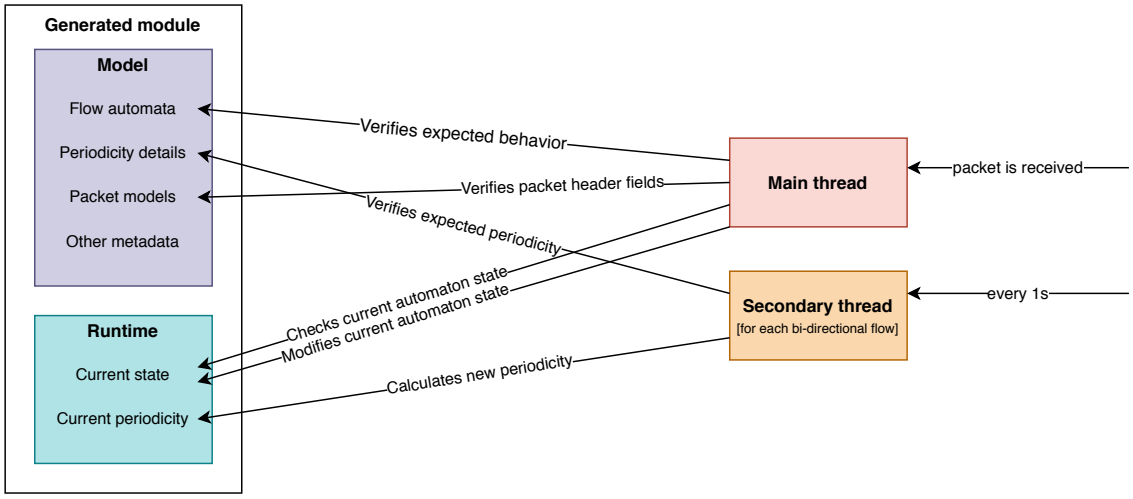


Figure 6.1: Model components usage by threads

6.1 The Main Traffic Analyzer Thread

The main thread uses packet models to verify fields of packets coming to *stdin*. Moreover, it employs the general traffic model and relevant flow automata to verify packet correctness in a given context. When the execution starts, all automata are set to the initial state and windows for periodicity checking are empty. These structures represent the internal state of the program, which simultaneously reflects the network state. The following paragraphs briefly introduce actions performed by the main thread, i.e., processes executed when a packet arrives to *stdin*.

In the first phase, after a packet is received, the analyzer parses it to obtain details from the packet header and matches these details with a known model. If it is not able to associate the packet with any known model, it throws the packet away and logs this event as an error. Otherwise, it continues to the second stage.

In the second stage, happening only in the case where the packet was successfully matched with a known model in the first phase; the packet is associated with a particular bi-directional flow. Comparisons are performed using IP header information. Again, if the packet cannot be matched with any known flow, it is thrown away and a corresponding error is logged.

In the third processing phase, the main goal is to retrieve the current state of the automaton appertaining to the specific flow and perform a move to the state corresponding with the flow situation. The target state is obtained using a reference to the general prefix tree and a knowledge about the arrived packet model. Based on these details, we can find the correct transition using direct memory access. However, a confident (and correct) decision of the following move might be an impossible task. Figure 6.2 visualizes state uncertainty issue. Let's suppose a network communication to be in the state $B1$. The automaton on the left shows the situation where the state is not final. When we receive a packet of model "2", intuitively, we should move to state $C1$. However, in a real situation, there are 4 possible scenarios:

1. Packet "2" is valid and succeeds current state; thus, the automaton state should change to $C1$.

2. Packet “2” is an insertion; hence automaton should not move anywhere and log this occurrence.

3. Packet “2” should be preceded by other packets, which were lost; consequently, there was a packet deletion before. In this case, automaton should be reset and the occurrence reported.

4. Packet “2” is the start of new communication and current communication was ended; therefore the automaton state should be changed to $B2$.

The situation gets even more complicated when $B1$ is a final state. This scenario is shown in the right automaton (Figure 6.2). In addition to possible situations mentioned above, the current state might be final, thus, we should accept the current sequence. But, how can we be sure that the sequence will not continue by “2” or “23” as there are another two accepting states? Because automaton always reflects only current network status, we are not capable to certainly determine current or future states even for the most simple automata. Hence, in the following paragraphs, we describe an approach of reflecting the current network state, which respects a certain level of uncertainty.

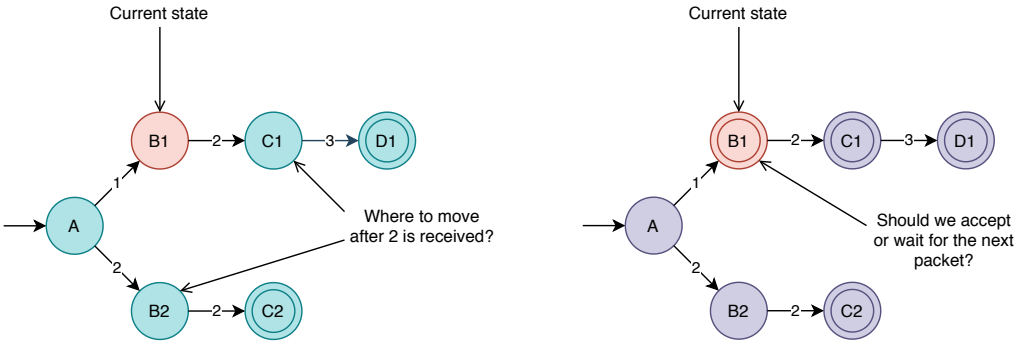


Figure 6.2: State uncertainty

6.1.1 Dealing with Network State Uncertainty

To keep a track of network events properly, we need to perform an estimation of the current state. Considering scenario in Figure 6.2, the situation after receiving packet “2” is visualised in Figure 6.3, where:

- errors are represented as a transition to state A, which denotes the network state before any (valid) packet is received;
- acceptance of “1” followed by the start of a new communication with packet “2” is represented in state $B2$; and
- communication continuance is denoted in $C1$.

Based on these behavior observations, we can define an important property of automata representing the current network state in the runtime part of network model (shown in Figure 6.1): The current automaton state is unknown; however, there is a set of states, in which it can be with the same probability. This set of states is called *candidate states*.

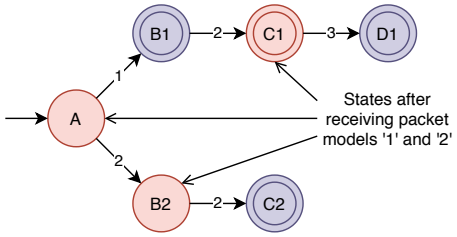


Figure 6.3: State uncertainty after “12” sequence is received

From given examples, we can deduce that whenever a packet, which can be used for a transition from the initial state, is received, it creates a new candidate state. Although, there is always maximally one new candidate state for each packet arrival because automaton is deterministic. Thus, after n packets are received, there is a maximum of n candidate states, in which the network can be with the same probability. This principle can be used to define another automaton property: A set of candidate states has a maximum of k members, where k is the length of the input stream.

With such an approach, every time, after a new packet is received, a move is performed for all automaton candidate states. Intuitively, it causes the generation of many “blind” candidates that get stuck after a few moves, because there is not any transition for the received packet. This situation cannot be allowed for a valid sequence, thus, after a candidate state gets stuck, it is removed from the candidate set, however, with the condition that the stuck candidate is not the most distant candidate from the initial state. The distance condition was introduced because of the requirement to have all packets as a part of the accepted stream and, intuitively, there is always only one candidate state including all packets, which is the most distant one. Hence, stuck candidates can be safely removed as far as the most distant one can continue. For example, considering the situation in Figure 6.3, if a packet of model “3” was received, $C1$ still could continue to $D1$ but A and $B2$ would be stuck. In this case, if the following packet after “3” was “1” or “2”, we could securely accept the $D1$ candidate and start a new stream by a move from A to $B1$ or $B2$. Such an approach is simply applicable for all valid sequences. However, non-acceptance of the most distant candidate does not necessarily mean a communication failure.

As mentioned in Chapter 5, one stream may come right after the other one and, as a consequence, they could be considered as one. Such a situation is displayed in Figure 6.4. This automaton does not accept the “1233” sequence, however, it does accept “12” and “33” sequences separately. After “123” is received, the most distant candidate is able to continue with “4” only, but “3” is received. Despite this situation is correct, the most distant candidate is stuck. Albeit $C2$ is in a valid end state, this stream does not cover all packets in sequence, hence the automaton cannot accept.

One of the possible resolutions of such a situation is displayed in Figure 6.5. Whenever any flow passes a final state, it sets a mark on it and continues normally. When the most distant candidate gets stuck, runtime analyzer looks behind and searches for passed final states. If there are final states that can together cover the whole packet stream, the sequence can be accepted. In Figure 6.5, “1233” can be folded up using $C1$ and $C2$.

6.1.2 Preparing Statistical Data

Besides flow content verification, another responsibility of this thread is to prepare data for processing by the secondary thread. In Section 5.3.2, we have introduced an algorithm that

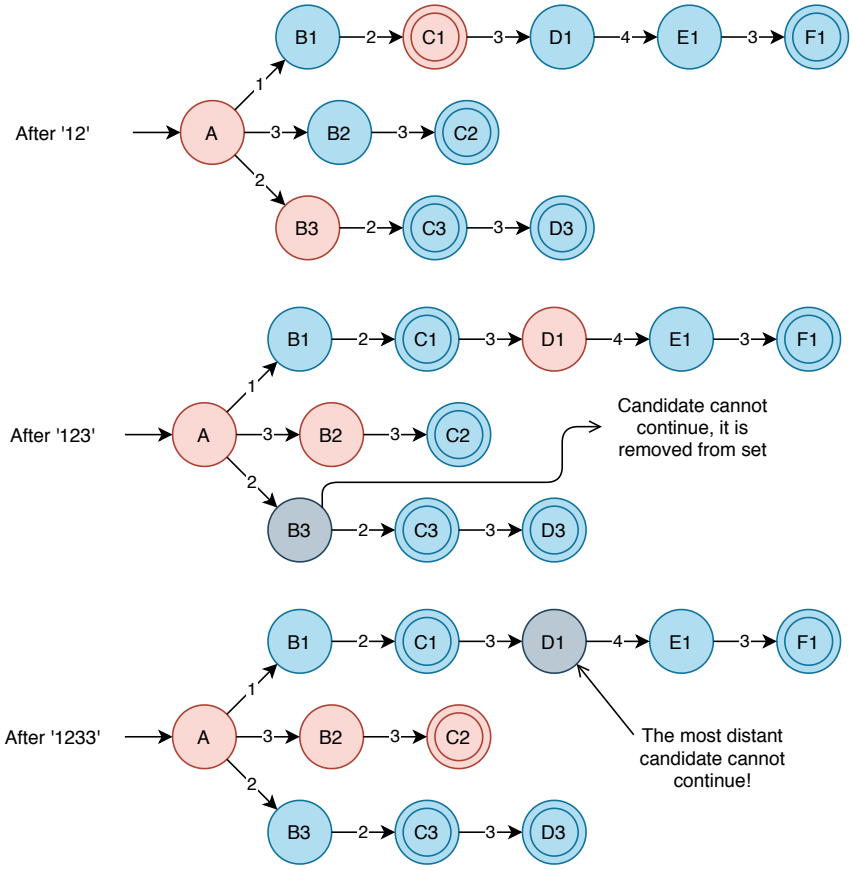


Figure 6.4: State uncertainty after “1233” is received

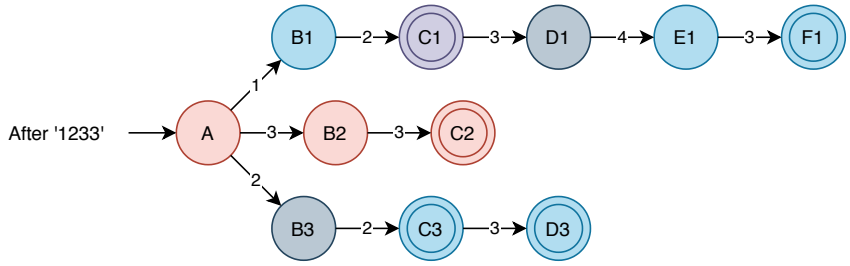


Figure 6.5: Resolution of state uncertainty after “1233” is received

produces an array of 1-second segments. Every segment carries two values – the number of packets that passed through during this time and the confidence of this distribution. In the runtime phase, these windows of segments are used to verify the number of occurrences by constructing these windows again and aligning them with their expected form. Albeit this is the competency of the second thread, the main thread collects data during packet processing and records the number of occurrences to segments.

In the beginning, all segments are set to zero. Then, whenever a packet sequence is accepted and assigned to some specific pattern, the main thread increments the value in the corresponding segment, which represents the specific pattern occurrence. Intuitively, the number of these segments constantly increments with passing execution time, which is unacceptable in the view of the fact that we require constant memory allocation. To

achieve this property, these segments are represented as a *circular linked list* of length equal to the expected period. Segment index is then calculated using a simple equation:

$$i_j = t_{sec} \ mod \ p_j \tag{6.1}$$

where i_j is an index of jth segment window, t_{sec} is current time in seconds from the start of program execution, and p_j is the period of jth segment window.

6.2 Periodicity Verifying Thread

The secondary thread has a single competency – it wakes up every second, aligns the expected window to the actual number of occurrences, and calculates a difference in the same way as described in Section 5.3.2, respectively. The difference is then used to calculate a *cumulative confidence*, which summarizes confidences through all past states. Cumulative confidence is calculated using Equations 6.2, 6.3, and 6.4, where x denotes a segment value (which is the number of occurrences during a particular time period), i is the current index, z is the number of segments with value 0 considering all previous indexes, *dif* is the distance between real and expected number of occurrences in the segment, and c is the confidence.

$$x_{i-1} = i - z_{i-1} \tag{6.2}$$

$$x_i = i + 1 - z_i \tag{6.3}$$

$$c_i = \frac{(c_{i-1} * x_{i-1}) + (z_{i-1} - z_i) + (1.0 - dif_i)}{x_i} \tag{6.4}$$

Comparing ith segment of expected occurrences window and real occurrences window, Equation 6.3 denotes the number of segments that were non-zero at least in one of these two cases and Equation 6.2 denotes the same for previous time segment. For example, in Figure 6.6, there are 2 zeros for $i = 6$ – on the second and the fourth position (we consider only segments, where 0 is in both windows). Hence we set $z = 2$ and $x_i = 5$. Generally, z_i stands for the number of segments that were zero in both windows at the same time, where for z_i we count segments $1 \leq j \leq i$. Intuitively, the difference between x_i and x_{i-1} is either 0 or 1. We can calculate cumulative confidence 6.4 based on 6.3, 6.2, previous confidence c_{i-1}, and dif_i, denoting the distance between ith number of occurrences in comparison with ith expectation, which is in interval <0;1>. We initially consider $c_0 = 1$ and $x_0 = 1$.

This process is repeated for each periodic pattern. Oppositely to the window construction process described in Section 5.3.2 (where the aim is to reach the highest confidence), the goal is to have the same confidence as the expectation, measured during training phase. However, in some cases, where the confidence estimation is around 0.5 (i.e., it was created with a 50% window mismatch rate), it might seem to be a piece of information with very low value. Let's suppose a set of partially periodic occurrences that can be split to segments $\{1, 6, 3\}$ and estimated window $\{3, 3, 6\}$; mismatches correspond to $\{0.3, 0.5, 0.5\}$, thus, confidence correspond to 0.63. As far as the model consists of the estimations only and real occurrences are not provided in any form, the runtime analyzer receives very inaccurate information. Obviously, we cannot consider such pattern as purely periodic. However, even inaccurate partial periodicity provides certain information, based on which we can build rough recurrence expectation. When a pattern is considered random, we can hardly create limitations on the pattern occurrence. The only possibility how to keep track of these occurrences is to count the number of these events in a large period and expect more or less

the same. However, even very rough information about occurrences in 1s segment may be valuable, e.g., when 6 is expected with 0.5 confidence in three segments in a row but 20 is received in all these segments, we can quickly reveal odd behavior.

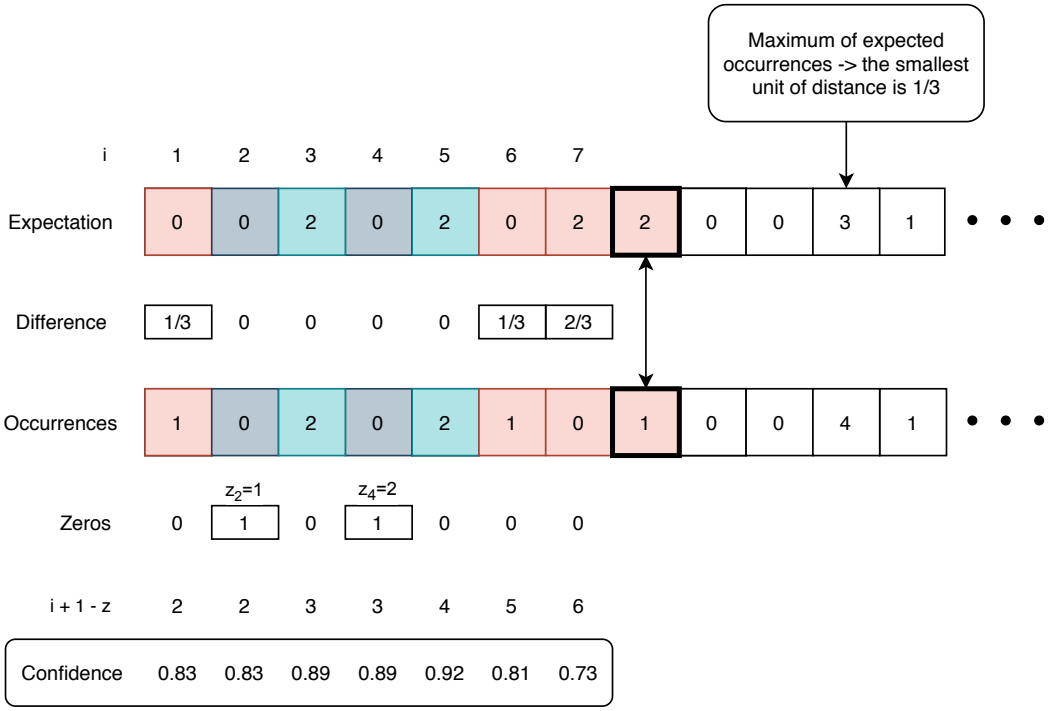


Figure 6.6: Cumulative window comparison

6.3 Handling the Moment of Connection

Until now, all discussed algorithms supposed that monitored traffic starts when the analyzer is already connected and listens to the interface. Albeit, in real-time analysis, the analyzer can be connected to the network at any moment. Thus, the analyzer, especially the secondary thread, needs the capability to familiarize itself with the environment as quickly as possible.

In the case of the main thread, the only inconsistency that might occur is the situation where the analyzer connects during some packet exchange stream, which would be normally represented as a pattern occurrence but starting packets were not appropriately captured. Hence the thread is unsure about the pattern start symbols. We can resolve this issue using an initial error tolerance. For a certain time, if the main thread cannot map an incoming packet to any known packet type, the packet is thrown away. However, this tolerance is not applicable to failures of type “unknown device” or “unexpected packet type”, these errors are never tolerable.

For the secondary thread, this situation is more troublesome. The thread always expects a flow starts at the segment of index 0 in a specific window. However, in runtime, the analyzer may be connected in any state of communication. As mentioned in this section, this thread aligns windows to calculate the difference. In such a situation, there might be a false positive alarm of mismatch because of incorrect window alignment. Thus, when the thread starts, at first it tries to find the current point for each periodic pattern. It

waits until the main thread writes to the penultimate index in the window and then tries to align the window with expectation. It searches for the most exact match by calculating the best possible offset, i.e., the value by which the window has to be moved to provide the best result. Considering two windows of length 4, $A = \{1, 2, 3, 4\}$ and $B = \{5, 6, 7, 8\}$, we align them as following:

- $\{1, 2, 3, 4\}$ to $\{5, 6, 7, 8\}$ with offset 0,
- $\{1, 2, 3, 4\}$ to $\{6, 7, 8, 5\}$ with offset 1,
- $\{1, 2, 3, 4\}$ to $\{7, 8, 5, 6\}$ with offset 2, and
- $\{1, 2, 3, 4\}$ to $\{8, 5, 6, 7\}$ with offset 3.

For each pair of sets, we calculate total difference using Equation 5.3, which is applied to values on the same indexes. Then, we search for the minimum variance. The offset associated with the most similar pair is then considered as the starting phase for the particular pattern. Using this offset, all values in the window of expected occurrences are circularly moved by this offset.

6.4 Prevention of Log Overflow

In previous sections, we have described methods of anomalous manners detection. Introduced techniques react to every unexpected occurrence, which may be problematic in the case where an error state persists. For example, if a periodic communication starts to occur randomly, sooner or later, there will be a report of periodicity confidence drop. After the drop, the confidence is reset to 1.0 and measured again. If the communication does not resume its standard period, it will cause another drop, and so forth. Another example is when an unallowed device connects to the network. Without any adjustments, every packet sent by this device would be reported as a vulnerability. Furthermore, there can be cases of false-positive reports because of packets that were not correctly captured by the monitoring device. Thus, in the following sections, we focus on minimizing the number of redundant reports.

6.4.1 Report Curtailment

To prevent excessive vulnerability reporting, we introduce an interval, during which logging is interrupted. This interval is specified separately for each error type as there are different measurements for packet errors, confidence drops, unexpected devices, etc. Also, the interval start/end is different for each bi-directional flow. For example, if we set the curtailment value for confidence drops to 24 hours and all devices shut down at some point; during the next 24-hour interval, there will be one report for each bi-directional flow. The curtailment interval start depends on the point when periodicity confidence decreases to a critically low value, thus, it may be different for each of these flows.

The curtailment only sets logging restrictions. When an error occurs, it is evaluated whether the error type is not paused for the given flow at the particular point in time. If a report curtailment is in progress, reporting is skipped but analysis continues in a standard way. Otherwise, the error is logged, and a curtailment record is created for the bi-directional flow and specific error type.

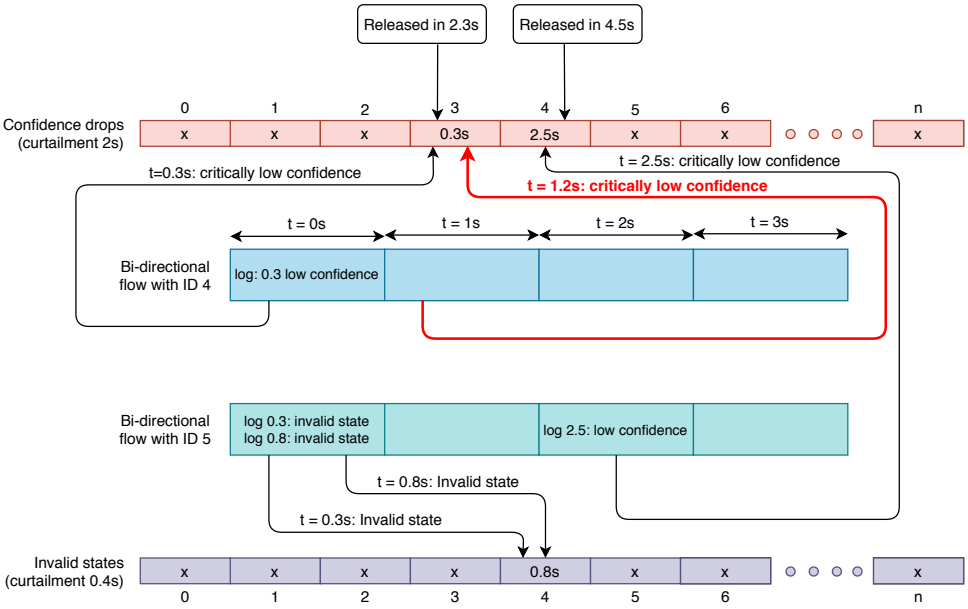


Figure 6.7: Curtailment adjustment

The whole process is displayed in Figure 6.7. The red array represents records of "low confidence error" cutback, the purple array stands for "invalid communication state error" pause records. Indexes in both arrays represent the identification of known bi-directional flows. Curtailment is set to 2 seconds for confidence drops and 0.4 seconds for invalid states. The blue array represents bi-directional flow with identification *4*, the green array is bi-directional flow *5*. In the case of the blue flow, there are two periodicity confidence errors. The first one occurs at 0.3 seconds and the other at 1.2 seconds. Without any adjustments, there would be 2 reports. However, after the first report, a curtailment record is created with the current timestamp. When the second drop occurs at 1.2 seconds, before it is logged, it is checked whether the flow reporting is paused. As we can see, there is a report curtailment in progress, which will be released at 2.3 seconds, thus, reporting is skipped. Contrarily, in the case of green flow, there were 2 invalid state occurrences during the first second. However, they were delayed enough so both are reported and curtailment is set to the time of the second error.

The situation gets complicated in the case of error type *unallowed flow*. Other errors can be associated with specific bi-directional flows, so they are easy to monitor. In this case, we miss the flow reference. Hence, we need a mechanism of handling unknown flow communication recurrence. The method is based on keeping records of new bi-directional flows to track related reports. Therefore, there is a special buffer, which contains two kinds of information: unknown device identification and time of the error. Whenever an error of type "unallowed flow" occurs, related flow identification is searched in the buffer of unknown identifications. If a reference is found in the buffer, its timestamp is checked in order to detect possible expiration. If the record is not expired, reporting is skipped. Otherwise, the record is invalidated, an error is logged and a new record is inserted to the end of the buffer, respectively. To avoid memory consumption by a growing number of unallowed flows, the buffer size is limited to 256 flow records. After it is filled, records are circularly rewritten (the oldest first).

As mentioned earlier in this section, all curtailment intervals are configurable for specific errors. The configuration is performed using program arguments so they are fully optional. Details are described in Appendix A.

6.4.2 Single Packet Deletion Acceptance

In this section, we focus on handling single deletion errors, so it is possible to skip reporting of this error kind. This error is handled using a look-ahead method. Using the generated traffic model, we have a reference to all "potentially future" automaton states. Hence we can look for all states following the succeeding states of the current state. Intuitively, every captured packet can be considered as a packet after a potential deletion. To avoid superfluous processing, this detection is performed only in the case where an error is revealed and it is about being reported.

Deletion correction always considers packet miss inside sub-flow, i.e., excludes the first state. Every communication is required to be started appropriately. Let's consider a stream, which currently consists of n packets, thus, it has already performed n automaton state moves, which we refer to as "is at *nth* automaton level". Packet $n+1$ is already received but using it the stream cannot pass to any following state. Instead of logging this situation, we look into all possible states at $(n+1)st$ automaton level. If any of these states have a transition with the received $(n+1)st$ packet, we move to this state and perform the transition instead of logging an error. Thus, we skip one state and $(n+1)st$ packet becomes $(n+2)nd$.

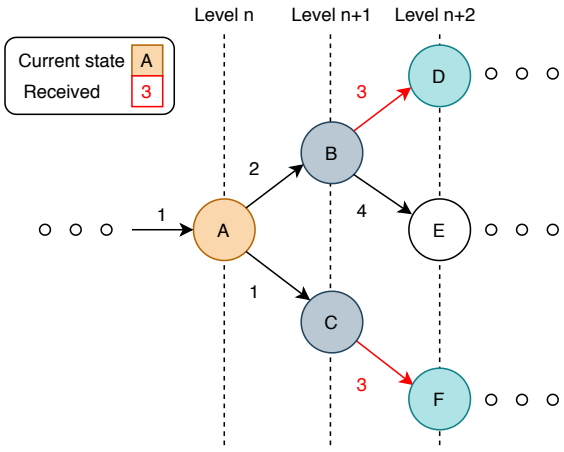


Figure 6.8: Following state uncertainty

The situation is more complicated when there is more than one possible state. It causes that one *candidate state* splits to two candidates that are at the same level. This scenario is visualized in Figure 6.8, where we move from state A to both D and F states. As we mentioned in Section 6.1.1, we always require maximally one candidate on a specific level. This situation is problematic for the most distant candidate representing the whole packet stream. Having two candidates there, we cannot correctly reveal the situation where the correct most distant candidate accepts/fails. To resolve this problem, we select one of these candidates and consider it the most distant one, despite there are multiple "the most distant" candidates. The selection is performed based on state ID. Every state has its own ID, which is based on its position in the automaton. The ID of a state at automaton level

n is always higher than ID of all states on lower levels. Correspondingly, all states on levels higher than n have higher ID than nth level. States at the same level have sequential IDs, the highest is selected and considered as the longest flow. Other processes are kept unchanged. Whenever the most distant candidate cannot continue, other candidates are searched in order to match the whole packet sequence.

Chapter 7

Verification and Consequent Adjustments

This chapter focuses on the verification of the proposed solution. There are three main demands on the analyzer functionality:

- monitoring of "standard" traffic with some common deviations, such as temporarily higher latency, without reporting serious issues;
- revealing an unknown communication or unknown devices in the network; and
- warranting permanent frequency of the communication between known devices.

To demonstrate required behavior, we have used obtained dataset of simulated attack (Section 4.1) and also a custom-developed generator of IEC 104 packet headers that allow customizing streams from many points of view (packet losses, latency, periodicity, pattern length, number of patterns, etc.) so we can demonstrate some specific situations and incidents (details in Appendix B).

7.1 Analysis of the Simulated Attack

At first, we focus on the verification of the model, generated during the training process. The generated model is compared to data representation in Wireshark[1] to roughly estimate expected representation. Then, we verify whether the analyzer would recognize malicious manners based on the generated model. To construct the model, we use the first third of the dataset, which includes 200 000 packets captured during 258 seconds of monitoring.

7.1.1 Generated Model

After running the model recognition onto the CSV file with limitation to 200 000 lines, we have received .c and .h modules (their structure is described in Section 5.4). These files contain a definition of communication inside 37 bi-directional flows, what indicates that there are 38 devices in the network. During runtime analysis, these 38 addresses are the only allowed as senders and receivers of IEC 104 packets.

[1] https://www.wireshark.org/

Evidently, in all bi-directional flows, there are 2 major periodic patterns, the first one repeats every 0.6 seconds, the other repeats once per 20 seconds. In all 37 cases, the analyzer has recognized window `{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}`, what exactly represents a window of pattern that occurs every 20 seconds. In the second case, we can observe a much shorter window with a higher frequency:

- in 31 cases, the window is represented as `{2,2,2,1,2,2}`,
- in three cases as `{2,2,2,1,1,2}`,
- in two cases as `{2,2,1,1,2,2}`, and
- in one case as `{2,2,2,1,2,1}`.

Obviously, all these cases, more or less, match the estimated 0.6s recurrence. However, in one case, devices do not communicate in this way at all.

Besides the period window, there is another important attribute: a confidence, guaranteeing the window matched real occurrences during the training phase with a certain accuracy. In the case of the pattern with 20s period, the count of samples in the training dataset is small (statistically less than 10 in all cases). Nevertheless the lowest recognition confidence is 0.55, contrarily, the highest confidence is 1.0, and the median is 0.8. In the case of the pattern with the shorter period, the highest confidence is 0.92, oppositely, the lowest is 0.88, and the median is 0.91, which is much more consistent than the pattern with the longer period. All recognized devices and their confidences for both periodic patterns are summarized in Appendix C.

7.2 Runtime Traffic Verification Results

To demonstrate real-time environment, we use a custom replay utility that is capable to read packets from CSV file and redirects them to standard input with a respect to arrival times, what simulates real traffic. Because we use the same dataset for both training and testing, we do not focus only on the result itself but also on the trend after runtime analyzer processes packets, on which its traffic model has been trained.

After the whole file is processed, there is a series of failures in the log file. The first log occurred at 467 seconds from the beginning of monitoring, reporting unknown packet arrival, followed by many reports of packets from non-permitted devices. Also, from this time, we can observe a sudden drop of periodicity confidence in the case of the more frequent pattern. Figure 7.1 displays the periodicity confidence during runtime. Axis x denotes the number of seconds from the start of monitoring (the model was trained on data from 0s to 268s) and y denotes confidence of the more frequent pattern (implicitly 1.0). A complete overview of all monitored flows is shown in Appendix D.1.

Besides the periodicity drop, observable in the case of all streams, there are two noticeable anomalies. The first one is the red curve in Figure D.1, the second one is the pink curve in Figure D.3. These two flows exhibit significantly odd behavior compared to the other 35 streams. In most cases, the confidence drop was around 10 percent of confidence. In the case of the mentioned two flows, the confidence drop is immense, causes the value approximates to zero. We can observe that in both cases, at around 800 seconds from the beginning of monitoring, the confidence is suddenly shifted to 1.0, what is caused by the periodicity checking thread, which has detected very low confidence, reported it, and

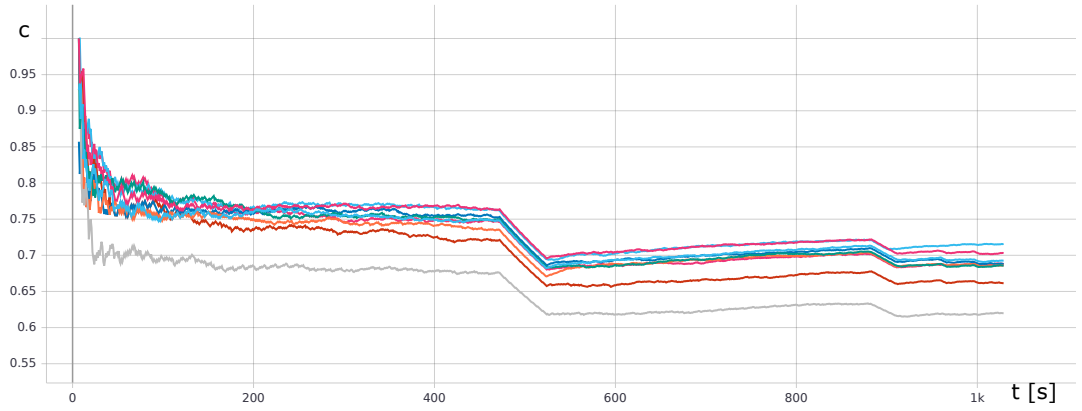


Figure 7.1: Pattern 106 for devices with ID 10-19

reset the value, respectively. However, the confidence trend seems to be forthwith dropping again after the reset.

Further investigation of the mentioned anomalies showed that at around 450 seconds from the analysis start, these two flows stopped all communication of the specific pattern. It follows that the drop is not caused by differences in periodicity but by a total absence of the pattern from the point when some unexpected communications were reported; whereas the impact on other devices is also significant but evidently temporal. After this anomaly, in most cases, there is a slight recovery during the next 200 seconds, disturbed by another, this time milder, confidence drop.

Although the source of confidence drop is unknown, it probably does relate to the error in the log file described at the beginning of this section. Despite real-time confidence is in general smaller than expected values (specified in Appendix C), after a certain amount of time, it gains stability and oscillates around 0.77. Even after the outage, there is a noticeable convergence towards this value.

These observations have proven that for short periods, there is not continuous groundless degradation of confidence and the algorithm handles a certain level of error rate in the time aspect of the communication.

However, the situation is different when it comes to streams with a very long period. In the field of confidence, our current approach takes into account only moments (i.e., 1s segments), where at least one of expectation/reality is non-zero; because "both zero" situations are considered as beneficially inferior. Hence considering a 1-minute interval (which is represented as a 60-segment window), if the noise is absent, there are exactly 3 segments taken into account and others are evaluated as non-valuable. Such an approach may be problematic in noisy streams because every minor deviation dramatically lowers total confidence. Moreover, there is not any guarantee of period stability. There might be some external aspects (or imprecision may accumulate over time) affecting the period length. In the current approach, such a situation would cause occurrences that do not fit with the expected moments of these events, which causes false-positive alarms.

Figure 7.2 shows confidence for the pattern with long period (20s). Complete graphs are included in Appendix D.2. Likewise the previous case, sudden confidence grows are caused by a reset, occurring when a critically low confidence is detected. In this case, we can notice a slower confidence drop after a pattern is reset than in the previous case; however, in this case, there are observable drops even in the first part of monitoring. Albeit a moment, in which pattern may come to be considered as fitting to period, is set to one-second segment;

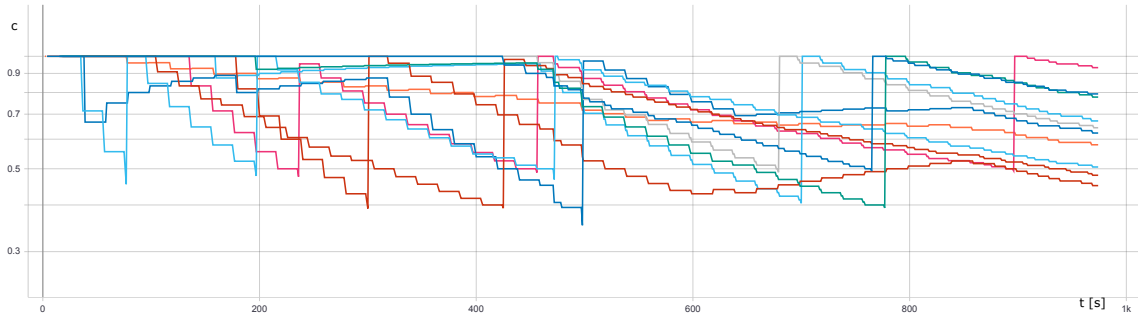


Figure 7.2: Pattern 9 for devices with ID 10-19

obviously, it is not large enough and the trend keeps stagnating during the whole process, which is unacceptable. Notwithstanding making segments larger might seem to be the correct approach, it may only be a temporal solution. Let's consider a situation, where we have a periodic pattern with period 5s in a noisy environment, where streams are gradually getting delayed. If every stream has approximately 0.005-second delay from the preceding one, after 1000 seconds, the pattern occurrence does not come in the 5th segments but one segment later, thus, the 200th occurrence has hypothetically period 6 seconds. Despite the following 199 occurrences have period 5 seconds again, window segments do not match anymore. Such situation is displayed in Figure 7.3. If we resolve such a situation by extending the segment range, sooner or later, this issue appears again, independently on the period length. In the following sections, we focus on narrowing this inconsistency in order to avoid false-positive alarms.

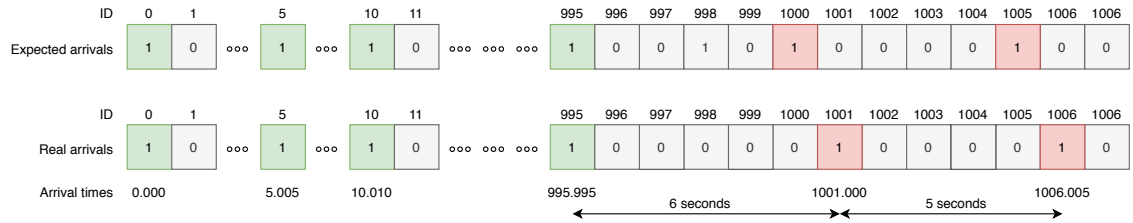


Figure 7.3: Enlarged period

7.2.1 Dealing with Low Frequency

This section discusses a possible approach to handle streams occurring with a long period. The resolution of this issue needs to be generic and work for both long and short periods, as determination of the boundary between "short" and "long" is in general very vague.

One of the possible solutions may be making segments of expected occurrences slightly indistinct, however, assuredly limited. We apply this approach to windows where the number of occurrences is lower than half of the window length. The main principle is very simple: whenever a pattern appears next to an expectation, with a maximum distance of one segment, it is accepted as a valid pattern presence. Using this principle, the window of expected arrivals behaves as following:

1. Whenever an occurrence appears at index i and it is expected at this index, it counts as a valid occurrence;

2. when a pattern arrives at index i but it is not expected at indexes i, $i-1$, and neither $i+1$, it is considered as a mismatch;

3. if an occurrence appears at index i but it is expected at index $i-1$, it is considered as a valid occurrence; however, the previous mismatch (at time $i-1$) has already affected confidence, thus, the difference is -1 to narrow the previous mismatch;

4. whenever a pattern comes at index i but it is presumed to appear at index $i+1$, it counts as a match but oppositely to the previous case, the difference is 0.

In the 3rd and 4th situation, this principle might cause an acceptance of duplicate values. For example, let's consider a situation where we expect an occurrence only at index 5, however, we receive a pattern at $i = 4$, so we consider it a match. One second later, we receive another occurrence of the same pattern, thus, we count it as a match again. The same may happen at $i = 6$; this time, we, even more, increase the confidence by $dif = -1$. Finally, after double duplicate, the difference is -1, which may critically distort total confidence. The problem is resolved using the following 3 adjustments:

- whenever a non-matching situation is accepted using one of two techniques mentioned above, we (circularly) move the window of expected occurrences to match given situation;

- we keep track of the previous situation by clearing index $i-1$ instead of the current one, what is performed at the end of every periodicity checking epoch;

- a match of non-matching situation does not consider only index i in real occurrences and $i \pm 1$ in expectations but we compare these indexes in both windows. To apply the discussed algorithm, we require either Equation 7.1 or 7.2 to be evaluated as true, considering e is a value in the window of expected occurrences and a is a value in real occurrences.

$$diff(e_i, a_{i-1}) + diff(e_{i-1}, a_i) = 0 \tag{7.1}$$

$$diff(e_i, a_{i+1}) + diff(e_{i+1}, a_i) = 0 \tag{7.2}$$

Intuitively, such an approach causes that the model part is no more static; however, the window of expected arrivals dynamically moves and adjusts to the current network state.

Figure 7.4 (and related Appendix D.3) depicts the situation after described algorithm is applied. Comparably to Figure 7.1, we can observe a rapid confidence drop beginning at 450 seconds from the monitoring outset. In this case, it is more significant because until this time, there were much fewer occurrences than in the previous case; thus, every failure impacts confidence much more. Contrarily to the situation before applying adjustments, Figure D.5 is the only one, where we can observe total confidence drop before 450 seconds. Albeit we can observe decreases also in Figures D.6 and D.8, they are only temporal and the confidence has raised again by itself without any interventions. In the following section, we focus on the investigation of these confidence drops. At first, we need to determine the source of this event to verify the analyzer's behavior is correct. Then, we focus on the investigation of this issue.

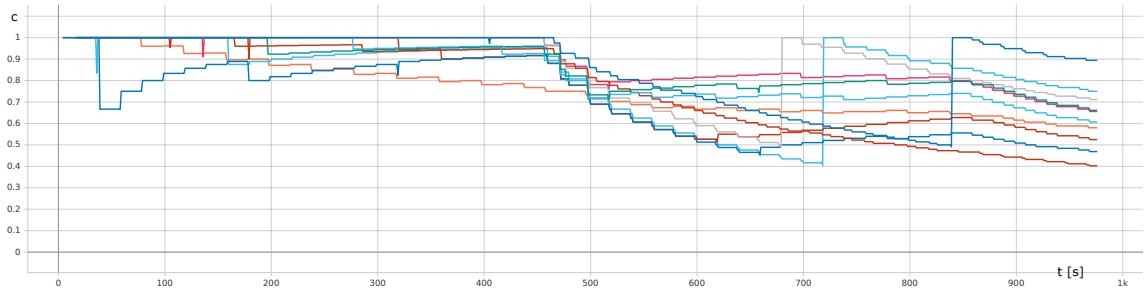


Figure 7.4: Pattern 9 for devices with ID 10-19

7.2.2 Investigation of Unexpected Confidence Drops

In this section, we focus on the investigation of confidence drops, displayed in confidence timelines, and describe specific cases of these events.

At first, we focus on the issue in the pattern with the smaller period, depicted in Figures D.1 and D.3. There are two flows unable to recover after the confidence drop, we identify them as *stream 10* (the red one in D.1) and *stream 27* (the purple one in D.3), which refers their identity described in Appendix C. A deeper investigation of pattern occurrences has shown that at a certain point, these devices completely stop the communication flow. However, an examination of the associated log file in detail has revealed that there are pertained errors, referring unknown channels between these two couples of devices. This fact implies that these devices continued communication but used different ports, which are not allowed by the model. If only IP addresses are considered as identification and ports are not taken into account at all, statistics of these flows are narrowed to precisely correspond to the confidence timeline of other streams.

In the case of the longer period (referred to as *pattern 9*), the situation is different. After the confidence drop at approximately 450 seconds from the monitoring outset, the confidence recovery seems to be problematic for most of the devices. A closer investigation has shown that at a certain point in time, all devices had stopped to communicate for approximately one minute. After the recovery, they started to communicate again; however, the first two occurrences of this pattern are sent twice. In our case, these duplicates are not the mandatory impact decreasing periodicity; actually, the main issue is that they do not fit the expected period. Despite the future flow is theoretically valid, it is moved by a couple of segments from the expectation.

Based on these observations, we assume that in all cases, devices completely stopped standard communication for around one minute. In the case of the short period, proper recovery is not an issue, because all segments of expected occurrences are the same (1-2 arrivals expected every second). Thus, it is not dependent on the point where communication starts or stops and continues. The situation is more problematic for long periods because it is not possible to estimate the reason why the expected occurrence has not arrived. Therefore, we cannot even guess the periodic occurrence arrived earlier or later than expected because of reset and that we should adapt our expectations to the situation.

7.2.3 Dealing with Runtime Resets

This section proposes a possible resolution of handling the moment when communication is renewed after device reconnection. Described practices are based on the approach introduced in Section 6.3.

As we only track the history of communication between two devices using a confidence measure, there is not any possibility of how to precisely pursue past specific matches/mismatches in streams. Hence at a certain point of traffic monitoring, we can guess device availability from two parameters: total confidence of the stream and current expectation/reality difference. However, from this information, we cannot precisely determine when the potential reset occurred. Also, we should take into account that outages are not the only reason for mismatching confidence windows, e.g., there might be an asynchronous event impacting the period, which also leads to mismatch. Thus, instead of focusing on the reason causing mismatch, we concentrate on an opportune correction.

As introduced earlier, the periodicity checking thread has the capability to detect critically low confidence and perform an appropriate reaction. This competency may be also used to narrow gaps in confidence checking. After a confidence counter is reset, all checking activities of the given pattern are suspended for an amount of time, which is equal to the number of segments in the confidence window. After this time expires, we consider all occurrences that appeared during this period. Based on this information, we narrow the time in the window of expected arrivals in the same way as at the beginning of monitoring (Section 6.3). We only perform a segment move, the content itself is not changed, inasmuch as periodic events are expected to keep the same period even after resets.

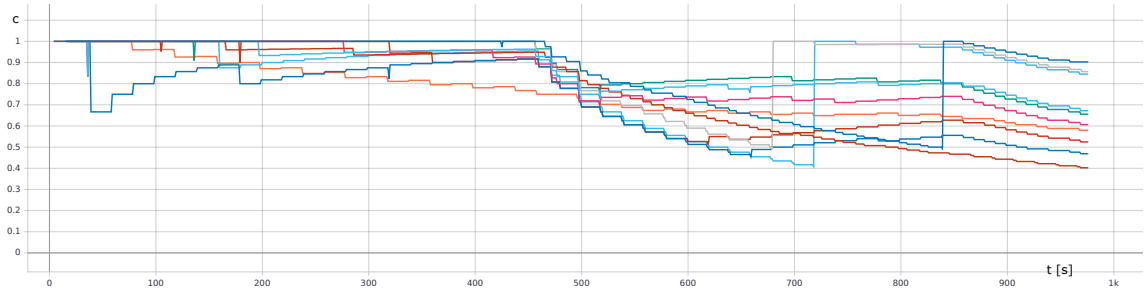


Figure 7.5: Pattern 9 for devices with ID 10-19

Figure 7.5 shows the result of applying all proposed adjustments. Correspondingly, Appendix Section D.4 delineates results for all captured streams. Despite there are streams that are gradually decreasing because of losing track of expected occurrences, after the reset, all expectations in terms of periodicity were appropriately aligned with the present state. We can observe the pink and blue streams in Figure 7.5, which dropped between 600 and 800 seconds from the start of monitoring. After the reset, these flows continue with very stable confidence, before the global slight confidence drop at about 850 seconds, what is presumed behavior.

7.2.4 Examination of Log File

In this section, we concentrate on particular events, which were appraised as potentially malicious and reported to the log file. As we already discussed earlier, there are reports warning about unknown communications, which are unaccepted by the model automaton. This suspicion of malicious behavior has arisen from downtime, during which all communications stopped and started packet exchange again from a different port. Furthermore, we can observe reports about unallowed states of particular bi-directional flows. To preclude acceptance of ensuing packet exchange, the automaton is reset to the initial state and an exception is logged.

Demonstration on Real Flow

To verify correct behavior, we have tracked the existing packet stream and logged all received packets and related errors. Results are compared to captured packets in *Wireshark*.

We have used *stream 19* (number refers its identification in Appendix C) for the behavior verification. The selected stream exhibits behavior representing the majority of allowed bidirectional flows. An example of standard log output, proving the communication fits automaton language, is shown in Figure 7.6.

```
[DEBUG] 825.446433000: (STREAM 19): Received packet 0.
[DEBUG] 825.447159000: (STREAM 19): Received TCP ACK.
[DEBUG] 825.447247000: (STREAM 19): Received packet 1.
[DEBUG] 825.447898000: (STREAM 19): Received packet 2.
[DEBUG] 825.449144000: (STREAM 19): Received packet 6.
[DEBUG] 825.456187000: (STREAM 19): Received packet 7.
[DEBUG] 825.456361000: (STREAM 19): Received packet 5.
[DEBUG] 825.457059000: (STREAM 19): Received TCP ACK.
[DEBUG] 825.468056000: (STREAM 19): Received packet 2.
```

Figure 7.6: Standard communication in *stream 19* (in log file)

Apart from presumed behavior, we can observe several warnings about malicious manners. During standard packet exchange, we can observe several errors (Figure 7.7) notifying about automaton halt. This situation is also reflected in Wireshark (Figure 7.8), where we can see two incongruous packets. As we consider only application (L7) information from the packet header, the position of reported warning in both sources slightly differs. In our case, instead of verifying sequence numbers in TCP header, we investigate the continuation of IEC 104 headers, where the error is reflected at a different point. Nevertheless, it has led to errors correlating with inconsistencies in TCP fields.

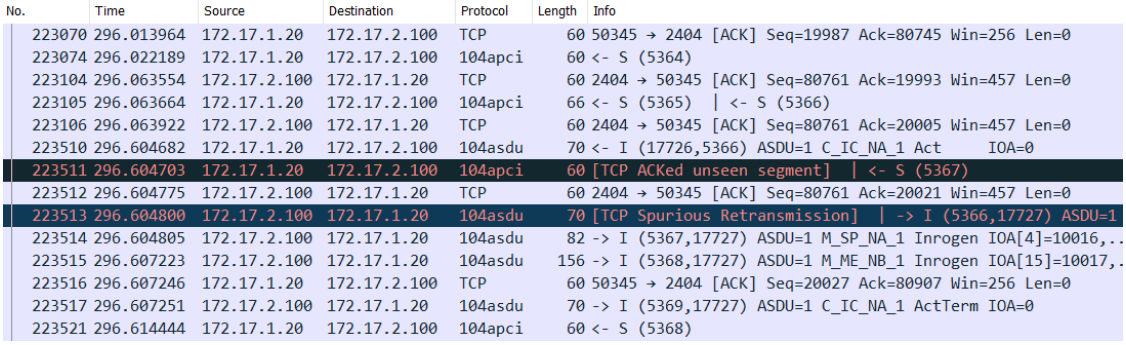
```
[DEBUG] 296.013964000: (STREAM 19): Received TCP ACK.
[DEBUG] 296.022189000: (STREAM 19): Received packet 2.
[DEBUG] 296.063554000: (STREAM 19): Received TCP ACK.
[DEBUG] 296.063664000: (STREAM 19): Received TCP ACK.
[DEBUG] 296.063922000: (STREAM 19): Received TCP ACK.
[DEBUG] 296.604682000: (STREAM 19): Received packet 0.
[DEBUG] 296.604703000: (STREAM 19): Received packet 2.
[DEBUG] 296.604775000: (STREAM 19): Received TCP ACK.
[DEBUG] 296.604800000: (STREAM 19): Received packet 1.
[DEBUG] 296.604805000: (STREAM 19): Received packet 6.
[DEBUG] 296.607223000: (STREAM 19): Received packet 7.
[WARN] 296.607223000: (STREAM 19, PACKET 7): Unknown communication state.
[DEBUG] 296.607246000: (STREAM 19): Received TCP ACK.
[DEBUG] 296.607251000: (STREAM 19): Received packet 5.
[WARN] 296.607251000: (STREAM 19, PACKET 5): Unknown communication state.
[DEBUG] 296.614444000: (STREAM 19): Received packet 2.
```

Figure 7.7: Standard communication of *stream 19* (in log file)

However, there are also uncommon situations, which are visible in Wireshark but are not reported by our analyzer. The source of this incongruity is in the training phase. These errors have been observed during the training process, thus, they are considered as a valid part of communication.

Failures are getting more frequent between 830 and 870 seconds from the monitoring outset. During this period, every pattern occurrence ends as a failure, which explains the second confidence drop, observable in Figure 7.1.

From the time when the confidence starts decreasing, there are logs about unknown automaton state. As depicted in Figure 7.9, none of the pattern occurrences is accepted



No.	Time	Source	Destination	Protocol	Length	Info
223070	296.013964	172.17.1.20	172.17.2.100	TCP	60	50345 → 2404 [ACK] Seq=19987 Ack=80745 Win=256 Len=0
223074	296.022189	172.17.1.20	172.17.2.100	104apci	60	<- S (5364)
223104	296.063554	172.17.2.100	172.17.1.20	TCP	60	2404 → 50345 [ACK] Seq=80761 Ack=19993 Win=457 Len=0
223105	296.063664	172.17.1.20	172.17.2.100	104apci	66	<- S (5365) | <- S (5366)
223106	296.063922	172.17.2.100	172.17.1.20	TCP	60	2404 → 50345 [ACK] Seq=80761 Ack=20005 Win=457 Len=0
223510	296.604682	172.17.1.20	172.17.2.100	104asdu	70	<- I (17726,5366) ASDU=1 C_IC_NA_1 Act IOA=0
223511	296.604703	172.17.1.20	172.17.2.100	104apci	60	[TCP ACKed unseen segment] | <- S (5367)
223512	296.604775	172.17.2.100	172.17.1.20	TCP	60	2404 → 50345 [ACK] Seq=80761 Ack=20021 Win=457 Len=0
223513	296.604800	172.17.2.100	172.17.1.20	104asdu	70	[TCP Spurious Retransmission] | -> I (5366,17727) ASDU=1
223514	296.604805	172.17.2.100	172.17.1.20	104asdu	82	-> I (5367,17727) ASDU=1 M_SP_NA_1 Inrogen IOA[4]=10016,..
223515	296.607223	172.17.2.100	172.17.1.20	104asdu	156	-> I (5368,17727) ASDU=1 M_ME_NB_1 Inrogen IOA[15]=10017,.
223516	296.607246	172.17.1.20	172.17.2.100	TCP	60	50345 → 2404 [ACK] Seq=20027 Ack=80907 Win=256 Len=0
223517	296.607251	172.17.2.100	172.17.1.20	104asdu	70	-> I (5369,17727) ASDU=1 C_IC_NA_1 ActTerm IOA=0
223521	296.614444	172.17.1.20	172.17.2.100	104apci	60	<- S (5368)

Figure 7.8: Pattern 9 for devices with ID 10-19 (in Wireshark)

by automaton because of duplicated IEC 104 packets; what precisely corresponds with confidence trend at this point. There are also analogous records in Wireshark packet stream (Figure 7.10). Equivalently to the previous pictured situation, there are errors reported on the L4 (ISO/OSI) layer, which are undetectable from our position. Thus, these failures are detected at the moment, when they are reflected in the application layer.

```
[DEBUG] 841.015643000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.015673000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.077562000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.077576000: (STREAM 19): Received packet 1.
[DEBUG] 841.077588000: (STREAM 19): Received packet 6.
[DEBUG] 841.077598000: (STREAM 19): Received packet 7.
[WARN] 841.077598000: (STREAM 19, PACKET 7): Unknown communication state.
[DEBUG] 841.077611000: (STREAM 19): Received packet 5.
[WARN] 841.077611000: (STREAM 19, PACKET 5): Unknown communication state.
[DEBUG] 841.077621000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.077629000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.120876000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.120888000: (STREAM 19): Received packet 1.
[DEBUG] 841.120901000: (STREAM 19): Received packet 6.
[DEBUG] 841.120916000: (STREAM 19): Received packet 7.
[WARN] 841.120916000: (STREAM 19, PACKET 7): Unknown communication state.
[DEBUG] 841.120930000: (STREAM 19): Received packet 5.
[WARN] 841.120930000: (STREAM 19, PACKET 5): Unknown communication state.
[DEBUG] 841.120943000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.120954000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.153254000: (STREAM 19): Received packet 0.
[DEBUG] 841.153265000: (STREAM 19): Received packet 2.
[DEBUG] 841.153277000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.153284000: (STREAM 19): Received packet 2.
[DEBUG] 841.163469000: (STREAM 19): Received TCP ACK.
[DEBUG] 841.192714000: (STREAM 19): Received TCP ACK.
```

Figure 7.9: Standard communication in *stream 19* (in log file)

7.3 Testing Using Own Generated Data

We have implemented a program, which is capable to generate packet streams in a configured format (Appendix B). Constructing data artificially allows customizing its content to demonstrate individual situations in network traffic. Henceforth, all discussed datasets are created using this generator.

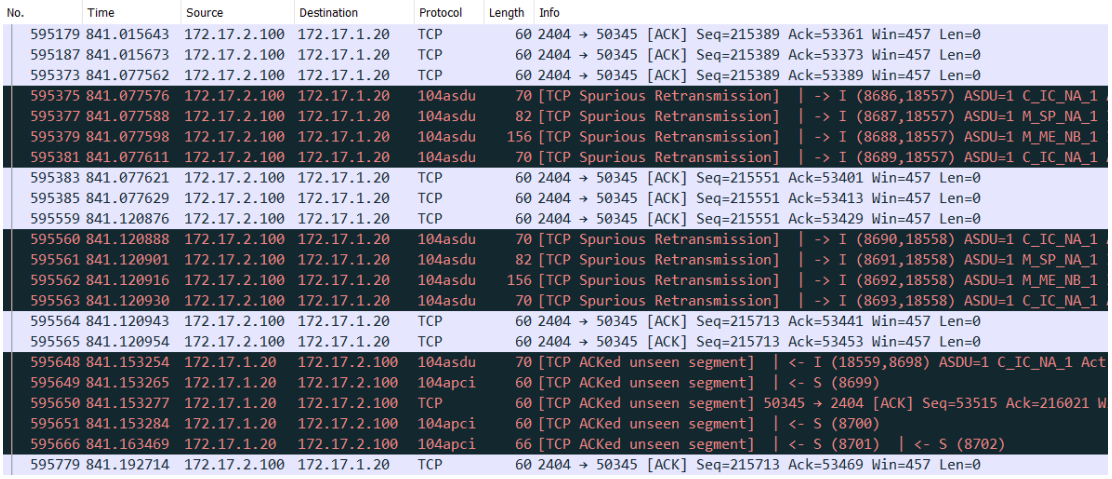


Figure 7.10: Pattern 9 for devices with ID 10-19 (in Wireshark)

7.3.1 Purely Periodic Stream

At first, we have generated a stream, which contains 3 bi-directional flows and 3 periodic patterns, each device communicate using a different pattern. Simulated traffic lasts 50 seconds and consists of:

1. 200× pattern of length 4 with period 0.5 seconds, referred to as *pattern 1*;
2. 100× pattern of length 2 with period 1 second, called *pattern 2*; and
3. 2× pattern of length 3 with period 50 seconds, referred to as *pattern 3*.

Using these patterns, we aim to demonstrate undisturbed traffic, which consists of expected packets only. Thus, the application should not report any alarms.

The training phase has produced a model which includes two periodic patterns. As there are only two occurrences of *pattern 3*, it was correctly excluded from the periodicity estimation. Generally, at least 5 occurrences are required to deliberate about pattern period. Thus, the last pattern is considered as valid, however, random. For *pattern 1*, the periodicity is estimated to window $\{2, 2, 2, 2, 2, 2\}$ and confidence 0.955 due to some segments, where belongs only single packet. In the case of *pattern 2*, the window is estimated to $\{1, 1, 1, 1, 1, 1\}$ with confidence 1.0.

This model was used as a base for the runtime traffic analyzer. For reference, we have used both training and testing sets as inputs. In the case of the training set, we reached final confidence 0.980 for *pattern 1* and 1.0 for *pattern 2*. Using testing set, we have reached 0.980 for *pattern 1* and 0.990 for *pattern 2*; thus, results are very similar to the modelled confidence. In both of these tests, there were not any reports about malicious manners.

7.3.2 Denial of Service Attack

In this test, we have used the same dataset as in Section 7.3.1 with a modification. At 97 seconds from the monitoring begin, we have multiplied traffic to 100 pattern occurrences instead of 3.

Because the traffic stream is not invalidated and the communication still corresponds to expected patterns, there is a single exception in logs after test execution. The error

is related to unexpectedly low confidence of the periodicity checking and ensuing reset. This event is clearly visible in overall confidence statistics, depicted in Figure 7.11. After an abrupt number of occurrences is noticed, confidence significantly drops within a single second at around 98 seconds from the monitoring outset.

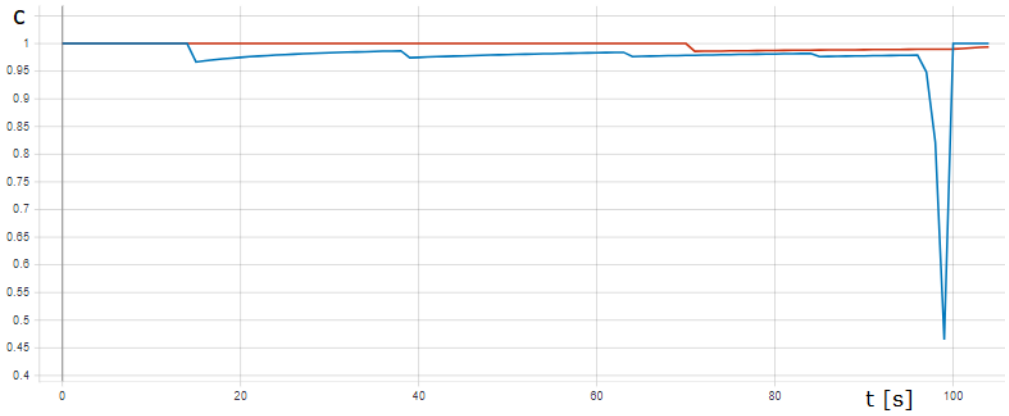


Figure 7.11: Denial of service reflected on periodicity confidence.

7.3.3 Single Packet Failures

This test demonstrates individual packet failures separately. In these tests, we do not consider periodicity confidence monitoring as a source of failure reveal. Despite it might play a critical role in potential misbehavior reveal, we contemplate log files only. We focus on the following situations:

1. communication initialization is duplicated,
2. communication is not initialized,
3. valid packet appears on unexpected place,
4. invalid packet arrives,
5. unknown device connects to network and starts to communicate in a valid way.

These individual failures were applied to the generated dataset, described in Section 7.3.1, and executed separately in the standard way.

In both examples (1) and (2) of failures during initialization, the error was correctly reported. In the case of (1), after the second initialization is received, the first one is thrown away, the event is logged, and the program continues with the second initialization, respectively. In the case (2), all subsequent packets are refused and logged before we capture a packet, which may be the start of a new pattern. In the case of (3), it strongly depends on a specific situation. In our case, there appeared a packet that might be the start of a new communication (despite it was not). Then, if the stream is currently in the state of automaton acceptance, it is ended in a standard way, and a new communication is commenced, respectively. Otherwise, the pattern is considered as corrupted and the unexpected arrival is logged. In the case of (4) and (5), packets were thrown away immediately at the beginning of their processing and reported.

7.4 Results Evaluation

In this chapter, we have discussed the capabilities of the final real-time packet analyzer. We have focused on both periodicity checking and records in the log file. In the first phase, we have revealed an issue caused by periodicity instability. The fluctuation has two major reasons:

1. One non-fitting occurrence, e.g., an arrival with high latency, may critically impact subsequent flow part; and

2. in the field of periodicity checking, bi-directional flows are often unable to recover after device downtime.

Based on these observations, we have introduced two adjustments to deal with environmental aspects and unexpected situations, which significantly improved the statistics of monitoring, so it correctly reflects the current network state.

Thereafter, we have investigated associated log files and compared particular records with captured packets in Wireshark to evaluate whether revealed anomalies correspond to real flaws. Deep investigation has shown that the analyzer is capable to detect failures that have not occurred during the training phase. In the case of unrecognized failures; arising from a situation where an automaton representing flow is halted and unable to perform a move, the analyzer is competent to detect a wide range of errors. However, only the application layer is considered during analysis. All failures reflected on this layer are detected but there might be potentially unrecognized failures, which are only reflected on lower layers.

Chapter 8

Conclusions and Future Work

In this work, we have focused on the investigation of network traffic between two devices in industrial control systems. Our main goal was to construct a program, which is able to automatically recognize a model of network communication. This model should grant the competency to verify traffic between particular devices in real-time.

In the first part, we have investigated a given dataset and protocol description with the following observations:

- Variety of packet header attributes is rather small even for continuous values.
- Flows between specific devices contain observable patterns, which indicate periods in flows.
- Network traffic is noisy and interspersed with a significant amount of packets unrelated to IEC 104 communication. Almost 40% of captured data consists of TCP signalization packets and other protocols (ARP, DNS, NBNS...).

Based on the first observation, we have decided to reduce the number of considered fields using the *feature selection* technique, which is based on the analysis of packet field variety. These sets of attributes are aggregated, which creates a group of "packet models" representing all packets in the flow. This technique reduces all IEC 104 fields to a single number representing a model identifier. Using this algorithm, 16 unique models have been found for the given dataset. Matching each packet to one of these models reveals obvious patterns in traffic between connected devices.

In the second part of the thesis, we have studied related works that were focused on network traffic modeling with the main focus on automata-based approaches. We have discussed many possible approaches based on automata, such as deterministic finite state automata, statecharts, stochastic automata, timed automata, but also other widely used techniques, such as Fourier transform and Markov models.

Based on related works results and also own observations, we have proposed an algorithm with the ability to automatically detect model in an unknown dataset. This algorithm consists of three main stages. At first, a single prefix tree is created to represent the whole network traffic. The purpose of creating such a model is to be able to track similarities across streams between different devices. Then, specialized models using deterministic finite automata are constructed to represent specific sub-flows that are mapped to the general model nodes. At last, we have discussed the suitability of different approaches in terms of periodicity estimation. We have appraised algorithms independent on statistics to be more convenient for periodicity determination. Information gathered during these steps is used to

construct a group of matrices and structures, which contain both constant and changeable parts, describing presumptive network behavior. Such a representation is generated as a C module, which provides a core for runtime traffic verification.

Afterward, we have implemented a program capable to verify traffic in real-time. This software consists of two separate threads. The main thread has the ability to

1. read captured packets;
2. verify their sender/receiver and assign the packet to a specific bi-directional flow;
3. confirm the packet belongs to some known group of packets (referred to as *packet models*), and generalize packet identity to the identification of associated packet model; and
4. adjust the current state of related automaton to reflect the state of the network after the packet is received.

Whilst the main thread is adjusting its internal state, it provides a piece of valuable information for the secondary thread, which has the competency to verify time aspects of the flow. This thread uses gathered information in combination with the estimated recurrence from the traffic model and uses this data to compute periodicity confidence. This confidence expresses the trend of communication recurrence and whether it meets expectations or not.

At last, we have collected results by executing the analyzer on the given dataset and discussed the contribution of our outputs. We have realized that periodicity checking is inaccurate for long periods, which caused immense confidence deviations, leading to false-positive alarms. To narrow this inaccuracy, we have implemented two adjustments, dealing with environmental aspects and helping analyzer to recover after communication downtime. Results have revealed that a combination of time aspect checking and reported anomalies provides a decent overview of the network state.

8.1 Future Work

We are planning to extend current behavior by the capability to precisely determine the reason for individual packet errors, as we are currently competent to only reveal a generic inconsistency in packet stream. Furthermore, current periodicity checking algorithms have already proven their benefits from many aspects; however, their appropriate recognition strongly depends on the quality of the training dataset. Thus, we will focus on making our solution more resistant to deviations during the training phase. Moreover, similar algorithms may be then used to deal with deviations during real-time monitoring to be able to recognize malicious manners earlier than after a critical confidence drop.

Bibliography

[1] ABB. *MicroSCADA Pro SYS 600 9.3 IEC 60870-5-104 Slave Protocol*. 2012.

[2] Alasadi, S. A. and Bhaya, W. S. Review of data preprocessing techniques in data mining. *Journal of Engineering and Applied Sciences*. 2017, vol. 12, no. 16, p. 4102–4107.

[3] Alur, R. and Dill, D. L. A theory of timed automata. *Theoretical computer science*. Elsevier. 1994, vol. 126, no. 2, p. 183–235.

[4] Anand, D. A. *Foundations-II: Periodicity Detection, Time-series Correlation, Burst Detection*. The LS3 Research Center, 2014.

[5] Barbosa, R. R. R., Sadre, R. and Pras, A. A first look into SCADA network traffic. In: IEEE. *2012 IEEE Network Operations and Management Symposium*. 2012, p. 518–521.

[6] Berndt, D. J. and Clifford, J. Using dynamic time warping to find patterns in time series. In: Seattle, WA. *KDD workshop*. 1994, vol. 10, no. 16, p. 359–370.

[7] Boujbel, R. Channel classification with hidden Markov models in mobile networks. *Informatik 2014*. Gesellschaft für Informatik eV. 2014.

[8] Brass, P. *Advanced data structures*. Cambridge University Press Cambridge, 2008. 336–356 p. ISBN 0521880378.

[9] Breslauer, D. and Galil, Z. Real-Time Streaming String-Matching. *ACM Trans. Algorithms*. 2014, vol. 10, no. 4.

[10] Carrasco, R. C. and Oncina, J. Learning stochastic regular grammars by means of a state merging method. In: Springer. *International Colloquium on Grammatical Inference*. 1994, p. 139–152.

[11] Carrasco, R. C. and Oncina, J. Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO-Theoretical Informatics and Applications*. EDP Sciences. 1999, vol. 33, no. 1, p. 1–19.

[12] Chu, W. W. Data mining and knowledge discovery for Big Data. *Studies in Big Data*. Springer. 2014, vol. 1.

[13] De La Briandais, R. File searching using variable length keys. In: ACM. *Papers presented at the the March 3-5, 1959, western joint computer conference*. 1959, p. 295–298.

[14] Elfeky, M. G., Aref, W. G. and Elmagarmid, A. K. WARP: time warping for periodicity detection. In: IEEE. *Fifth IEEE International Conference on Data Mining (ICDM'05)*. 2005, p. 8–pp.

[15] Equipment, T. Systems — Part 5-104: Transmission Protocols — Network Access for IEC 60870-5-101 Using Standard Transport Profiles. *IEC Standard*. 2006, vol. 60870.

[16] Forrest, S., Hofmeyr, S. A. and Somayaji, A. Computer immunology. *Communications of the ACM*. Citeseer. 1997, vol. 40, no. 10, p. 88–96.

[17] Goldenberg, N. and Wool, A. Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems. *International Journal of Critical Infrastructure Protection*. Elsevier. 2013, vol. 6, no. 2, p. 63–75.

[18] Grinstead, C. M. and Snell, J. L. *Introduction to probability*. 1st ed. American Mathematical Society, 1997. ISBN 9780821894149.

[19] Han, J., Pei, J. and Kamber, M. *Data mining: concepts and techniques*. Elsevier, 2011. ISBN 9780123814791.

[20] Harel, D., Pnueli, A. and Stavi, J. Propositional dynamic logic of nonregular programs. *Journal of Computer and System Sciences*. Elsevier. 1983, vol. 26, no. 2, p. 222–243.

[21] Honeywell, spol. s.r.o. *Modbus Messaging Implementation Guide*. 2002.

[22] Hubballi, N. and Goyal, D. Flowsummary: Summarizing network flows for communication periodicity detection. In: Springer. *International Conference on Pattern Recognition and Machine Intelligence*. 2013, p. 695–700.

[23] Jurafsky, D. and Martin, J. H. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall, Pearson Education International, 2009.

[24] Karvounas, G., Oikonomidis, I. and Argyros, A. A. Localizing Periodicity in Time Series and Videos. In: *BMVC*. 2016, vol. 2.

[25] Kleinmann, A. and Wool, A. Automatic construction of statechart-based anomaly detection models for multi-threaded scada via spectral analysis. In: ACM. *Proceedings of the 2nd ACM Workshop on Cyber-Physical Systems Security and Privacy*. 2016, p. 1–12.

[26] Kleinmann, A. and Wool, A. Automatic construction of statechart-based anomaly detection models for multi-threaded industrial control systems. *ACM Transactions on Intelligent Systems and Technology (TIST)*. ACM. 2017, vol. 8, no. 4, p. 55.

[27] Kumar, B., Niggemann, O. and Jasperneite, J. Timed automata for modeling network traffic. In: *Machine Learning in Real-Time Applications (MLRTA 09) (in conjunction with 32nd Annual Conference on Artificial Intelligence (KI 2009))*. 2009.

[28] Ma, S. and Hellerstein, J. L. Mining partially periodic event patterns with unknown periods. In: IEEE. *Proceedings 17th International Conference on Data Engineering*. 2001, p. 205–214.

[29] Maia, J. E. B. and Holanda Filho, R. Internet traffic classification using a Hidden Markov model. In: IEEE. *2010 10th International Conference on Hybrid Intelligent Systems*. 2010, p. 37–42.

[30] Matoušek, P. Description and analysis of IEC 104 Protocol. *Faculty of Information Technology, Brno University o Technology, Tech. Rep.* 2017.

[31] Maynard, P., McLaughlin, K. and Haberler, B. Towards Understanding Man-in-the-middle Attacks on IEC 60870-5-104 SCADA Networks. In: *ICS-CSR*. 2014.

[32] Melichar, B., Holub, J. and Polcar, J. *Text searching algorithms*. 2005. Available at: http://stringology.org/athens/TextSearchingAlgorithms (visited on 11/26/2019).

[33] Narendra, K. S. and Thathachar, M. A. Learning automata-a survey. *IEEE Transactions on systems, man, and cybernetics*. IEEE. 1974, no. 4, p. 323–334.

[34] Neuburger, S. Pattern matching algorithms: An overview. Department of Computer Science, The Graduate Center, CUNY. 2009.

[35] Pietrzyk, M. *Methods and Algorithms for Network Traffic Classification*. 2011. Dissertation. PhD thesis, Telecom Paris Tech, 2011. 77, 87.

[36] Poznyak, A. S. and Najim, K. *Learning automata and stochastic optimization*. Springer, 1997. ISBN 978-3-540-76154-9.

[37] Puech, T., Boussard, M., D'Amato, A. and Millerand, G. A fully automated periodicity detection in time series. In: Springer. *International Workshop on Advanced Analysis and Learning on Temporal Data*. 2019, p. 43–54.

[38] Ronešová, A. Přehled protokolu MODBUS. 2005.

[39] Salamatian, K. and Vaton, S. Hidden Markov modeling for network communication channels. In: ACM. *ACM SIGMETRICS Performance Evaluation Review*. 2001, vol. 29, no. 1, p. 92–101.

[40] Sirisha, G., Shashi, M. and Raju, G. P. Periodic pattern mining–algorithms and applications. *Global Journal of Computer Science and Technology*. 2014.

[41] Van Splunder, J. Periodicity detection in network traffic. *Technical Report, Mathematisch Instituut Universiteit Leiden*. 2015.

[42] Vlachos, M., Yu, P. and Castelli, V. On periodicity detection and structural periodic similarity. In: SIAM. *Proceedings of the 2005 SIAM international conference on data mining*. 2005, p. 449–460.

[43] Wright, C., Monrose, F. and Masson, G. M. HMM profiles for network traffic classification. In: ACM. *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*. 2004, p. 9–15.

[44] Yang, Y., McLaughlin, K., Sezer, S., Yuan, Y. and Huang, W. Stateful intrusion detection for IEC 60870-5-104 SCADA security. In: IEEE. *2014 IEEE PES General Meeting| Conference & Exposition*. 2014, p. 1–5. ISBN 978-1-4799-6415-4.

[45] Zendulka, J. *Získávání Znalostí z Databází: Studijní opora.* Faculty of Information Technology, Brno University of Technology, 2009.

[46] Češka, M., Vojnar, T., Smrčka, A. and Rogalewitz, A. *Teoretická Informatika: Studijní opora.* Faculty of Information Technology, Brno University of Technology, 2018.

Appendix A

Real-time Analyzer Configuration

Runtime analyzer can be configured using 5 command line options and a *help option* (`-h`). Option `-o` sets the error reporting format. It can be either "LOG" or "JSON". The "JSON" selection formats reports to JSON objects. Every object consists of 4 attributes:

1. *type*, which is "DEVICE", "MODEL", "CONFIDENCE", or "STATE" depending on the error source;
2. *time*, when the error occurred;
3. *devices*, containing identification of bi-directional flow; and
4. *message*, providing error description.

Contrarily, the "LOG" format creates reports in a plain text form, represented as

```
{time}: [ERROR:{type}] [DEVICES:{devices}] {message}
```

where value placeholders represent the same values as JSON attributes above.

Additionally, we can configure curtailment intervals for each error type. There are 4 configuration options. Each of these options expects a value which has to be numeric. This value represents a time in microseconds, for which subsequent errors are paused. The following options stand for specific error types:

- `-d` stands for unknown device reports,
- `-c` represents confidence drops,
- `-m` denotes packet model assignment failures, and
- `-s` symbolizes communication state errors.

Appendix B

Packet Stream Generator

To verify Analyzer behavior in specific situations and environments, we have implemented a packet generator with the capability to produce customized CSV packet streams based on selected header patterns, in our case, used to create a pure IEC 104 stream.

Program **stream-generator-1.0.jar** produces two CSV files, one used for training, the other for testing. These sets can optionally contain more or less the same or distinct patterns. Implicitly, even files generated from identical stream patterns are slightly different in packet arrival timestamps, thus, the order of particular packets always subtly differs. Generated content is managed using program arguments; it accepts an optional number of arguments, each of them has one of the four forms:

- **p** denotes a periodic pattern, generated into both training and testing set;
- **n** stands for a non-periodic pattern, produced to both training and testing set;
- **tp** symbolizes a periodic pattern, which is added to testing set only; and
- **tn** indicates a non-periodic pattern, attached to testing set only.

Arguments need to correspond a pattern associated with the argument type. Every type has a specific pattern form:

- `"p|{1}|{2}|{4}|{3}|{5}|{6}"`,
- `"n|{1}|{2}|{3}"`,
- `"tp|{1}|{2}|{4}|{3}|{5}|{6}"`,
- `"tn|{1}|{2}|{3}"`.

These patterns can be represented as regular expressions, their meaning is described in Table B.1. Patterns are generated without any dependencies between them, hence, they are combined together in the target file and two patterns of distinct communication flows may overlap. Training set is generated into **test_raw** folder, testing set is generated into **test_noised** folder. Both of them are named **generated_sequence_{identity}.csv**, were *identity* is a random hash, uniquely distinguishing generated packet streams.

ID	Regular expression	Minimum	Maximum	Description
1	\d+	1	LONG_MAX	The number of streams corresponding to this pattern.
2	\d+	1	LONG_MAX	The number of packets in this pattern.
3	\d+.\d+	0.0	1.0	Repetition of packets in the stream (i.e. one packet arrives multiple times in one stream).
4	\d+	1	LONG_MAX	Period length in micro-seconds.
5	\d+.\d+	0.0	1.0	The probability of 'getting lost' for each pattern occurrence in the stream.
6	\d+.\d+	0.0	1.0	The probability that a pattern occurrence arrives twice.

Table B.1: Meaning of symbols in argument patterns.

B.1 Example of Generated Streams

In this section, we demonstrate an execution of described packet stream generator. We have selected arguments `"p|4|3|500000|0.2|0.0|0.0" "p|2|2|1000000|1.0|0.0|0.0"`. These arguments generated 2 bi-directional flows, one contains 3 packets with period 0.5s, the other consists of 2 packets with period 1s. The result is shown in Figure B.1.

```
0.511630000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000003,0x00000001,141,147,134,36,41207
0.511910000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000002,0x00000000,109,189,113,221,31249
0.511960000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000000,0x00000002,30,50,149,155,27114
1.006400000,105.195.55.63,50.151.111.12,256,55,,,,,0x00000000,0x00000002,185,250,50,252,5651
1.007940000,105.195.55.63,50.151.111.12,256,55,,,,,0x00000000,0x00000002,185,250,50,252,5651
1.018580000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000003,0x00000001,141,147,134,36,41207
1.019250000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000002,0x00000000,109,189,113,221,31249
1.019470000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000000,0x00000002,30,50,149,155,27114
1.576380000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000003,0x00000001,141,147,134,36,41207
1.576590000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000002,0x00000000,109,189,113,221,31249
1.578280000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000000,0x00000002,30,50,149,155,27114
2.011220000,105.195.55.63,50.151.111.12,256,55,,,,,0x00000000,0x00000002,185,250,50,252,5651
2.012430000,105.195.55.63,50.151.111.12,256,55,,,,,0x00000000,0x00000002,185,250,50,252,5651
2.086940000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000003,0x00000001,141,147,134,36,41207
2.087050000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000002,0x00000000,109,189,113,221,31249
2.088380000,182.207.233.229,157.220.216.147,221,50,,,,,0x00000000,0x00000002,30,50,149,155,27114
```

Figure B.1: Generated (training) CSV packet stream

Appendix C

Recognized Bi-directional Flows

Table C.1 shows an identification of devices that were recognized in the training phase. Additionally, every record is associated with the confidence of its recognized period. In this work, we refer to these devices only as their ID.

ID	IP 1	IP 2	20s period confidence	0.5s period confidence
1	172.17.1.20	172.17.2.116	0.8182	0.9070
2		172.17.2.114	0.8182	0.9186
3		172.17.2.102	1.0000	0.9109
4		172.17.2.109	0.8000	0.8760
5		172.17.2.8	0.8333	Not present
6		172.17.2.134	1.0000	0.8992
7		172.17.2.101	1.0000	0.9225
8		172.17.2.125	1.0000	0.9109
9		172.17.2.104	1.0000	0.9070
10		172.17.2.132	0.8000	0.9047
11		172.17.2.136	1.0000	0.8953
12		172.17.2.127	0.8000	0.9124
13		172.17.2.123	1.0000	0.9147
14		172.17.2.118	0.8000	0.8992
15		172.17.2.106	1.0000	0.9186
16		172.17.2.129	0.7778	0.9241
17		172.17.2.103	0.8000	0.9222
18		172.17.2.111	0.8182	0.9070
19		172.17.2.120	1.0000	0.9086
20		172.17.2.100	0.8889	0.8953
21		172.17.2.108	1.0000	0.8876
22		172.17.2.107	1.0000	0.8953
23		172.17.2.115	1.0000	0.9031
24		172.17.2.124	1.0000	0.9128
25		172.17.2.110	0.6364	0.8876
26		172.17.2.133	0.8000	0.9008
27		172.17.2.131	1.0000	0.9202
28		172.17.2.105	1.0000	0.9070
29		172.17.2.135	1.0000	0.9047
30		172.17.2.117	0.6364	0.9109
31		172.17.2.113	0.8000	0.9167
32		172.17.2.126	0.8000	0.9070
33		172.17.2.122	1.0000	0.9128
34		172.17.2.119	0.7500	0.9186
35		172.17.2.128	0.5556	0.9222
36		172.17.2.130	0.8182	0.8930
37		172.17.2.121	1.0000	0.9031
38		172.17.2.112	0.7778	0.9031

Table C.1: Recognized devices and their communication periodicity.

Appendix D

Monitoring Results

In this Appendix Chapter, we present complete results of monitoring, described in Chapter 7. This chapter is further divided into four sections. Section D.1 shows the results of tracking a pattern with period 0.6 seconds, which we refer to as *pattern 106*, which relates to its identification during pattern analysis.

In the second phase, we focus on a pattern with a period of 20 seconds, called *pattern 9*. As discussed in Chapter 7, confident pattern tracking is problematic due to environmental impacts, thus, it is divided into 3 parts, further describing applied processes in order to reach results corresponding to reality. Section D.2 shows complete results of monitoring without any adjustments, which revealed the impact of the environment erroneous on analysis results. Section D.3 displays the same measurements with adjustments allowing infinitesimal deviations in packet arrivals, based on which window slightly adapts to the network situation. Finally, we introduce the result of monitoring with all applied adjustments. All measurements were performed using the same model and identical condition, so we can evaluate the contribution of applied enhancements.

D.1 Pattern 106

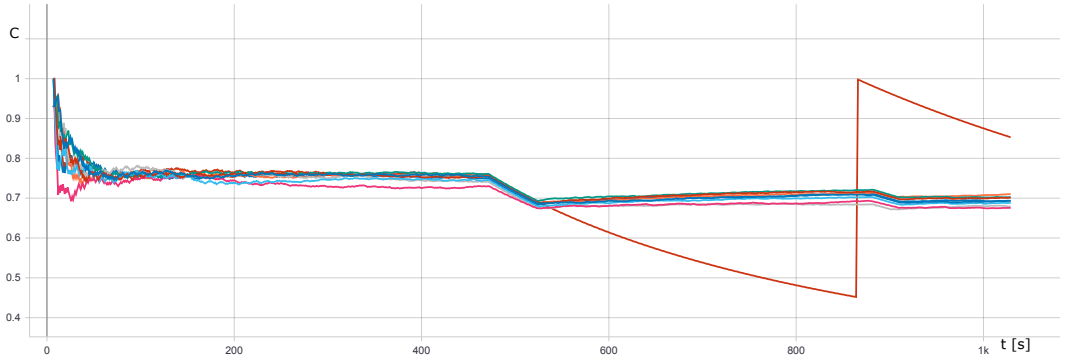


Figure D.1: Pattern 106 for devices with ID 1-9.

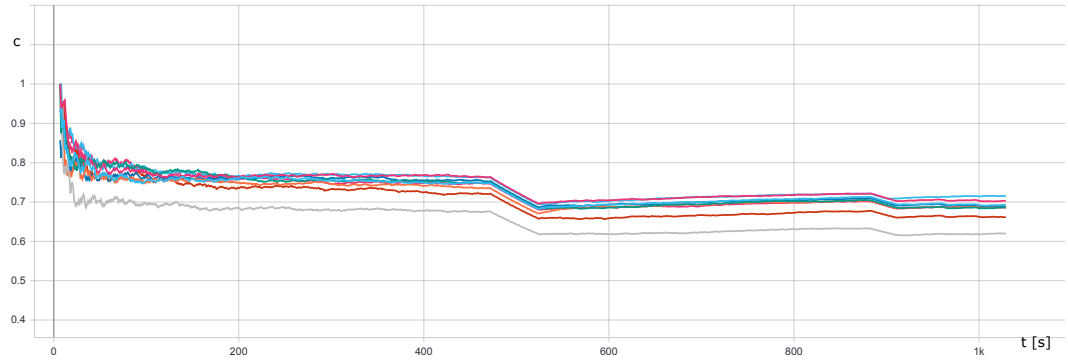


Figure D.2: Pattern 106 for devices with ID 10-19.

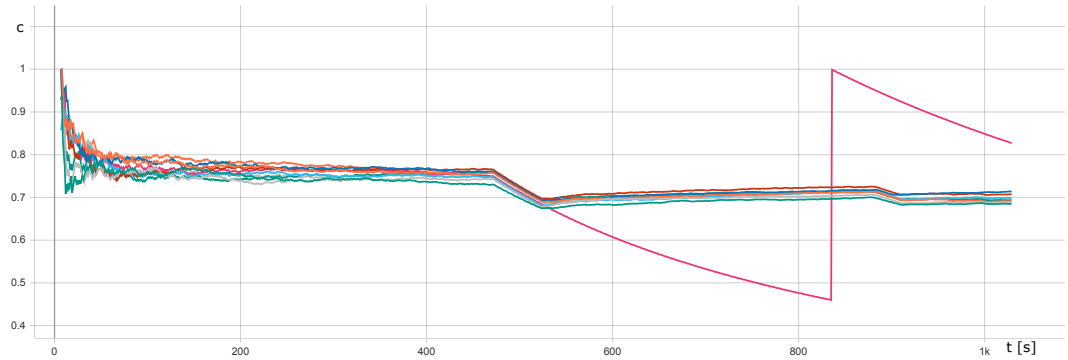


Figure D.3: Pattern 106 for devices with ID 20-29.

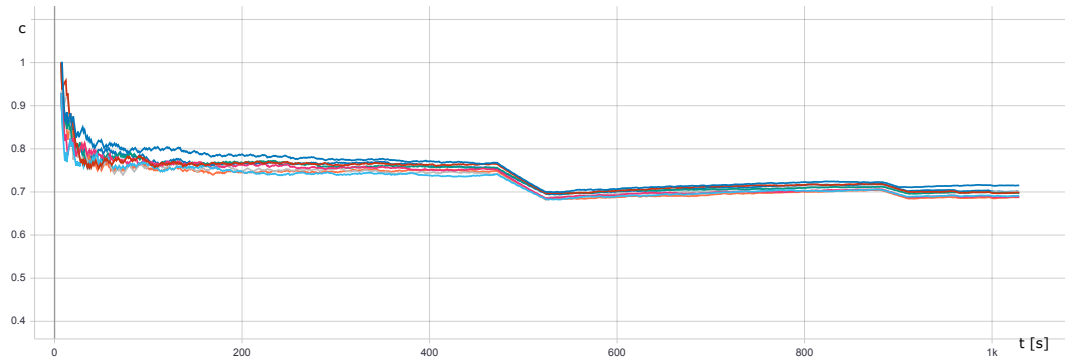


Figure D.4: Pattern 106 for devices with ID 30-38.

D.2 Pattern 9 without Adjustments

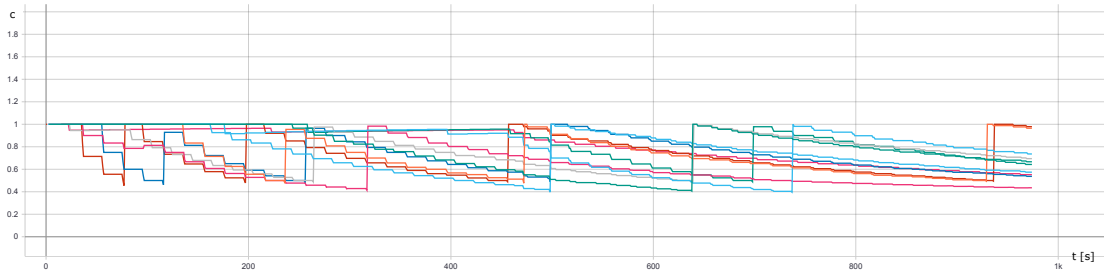


Figure D.5: Pattern 9 for devices with ID 1-9.

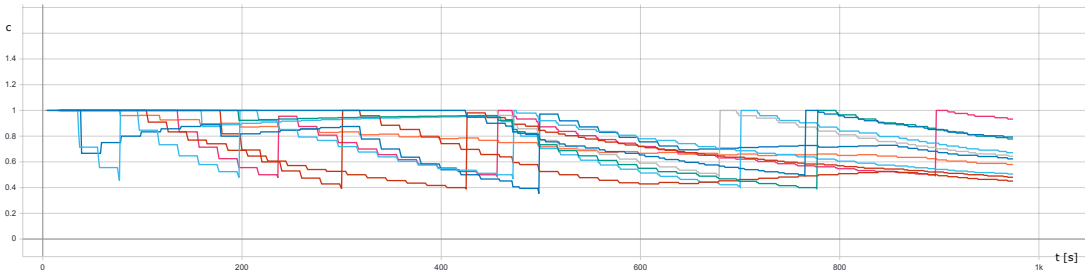


Figure D.6: Pattern 9 for devices with ID 10-19.

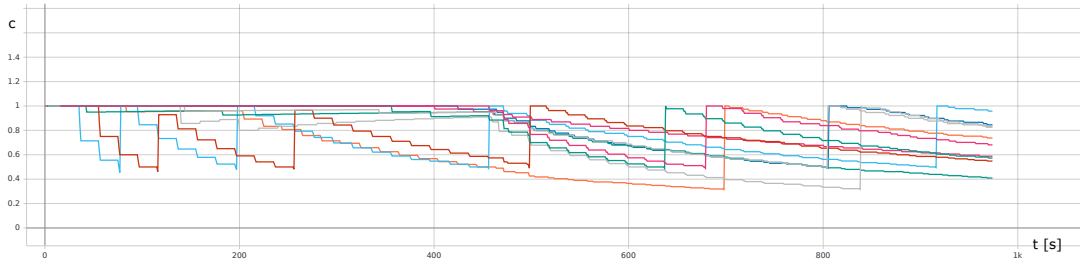


Figure D.7: Pattern 9 for devices with ID 20-29.

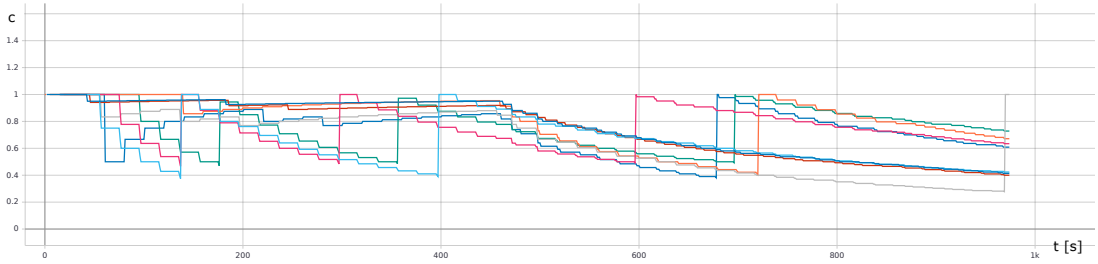


Figure D.8: Pattern 9 for devices with ID 30-38.

D.3 Pattern 9 with Sliding Window

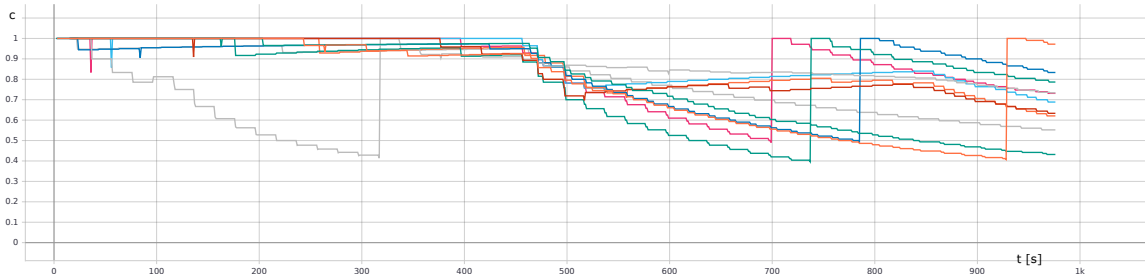


Figure D.9: Pattern 9 sliding window for devices with ID 1-9.

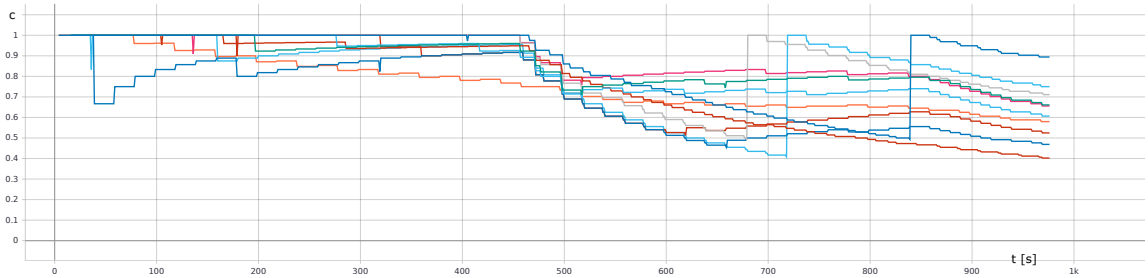


Figure D.10: Pattern 9 sliding window for devices with ID 10-19.

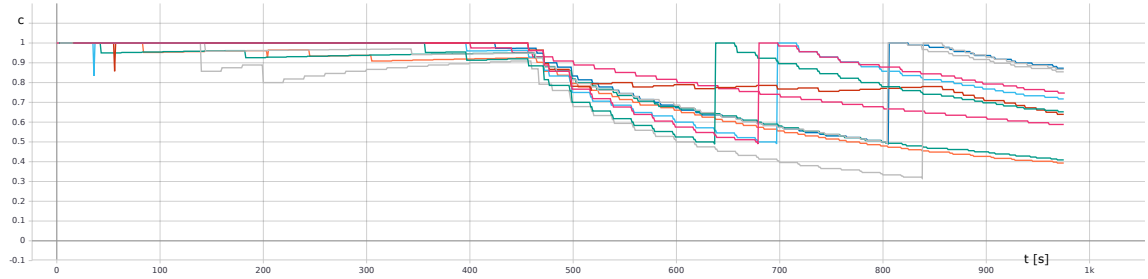


Figure D.11: Pattern 9 sliding window for devices with ID 20-29.

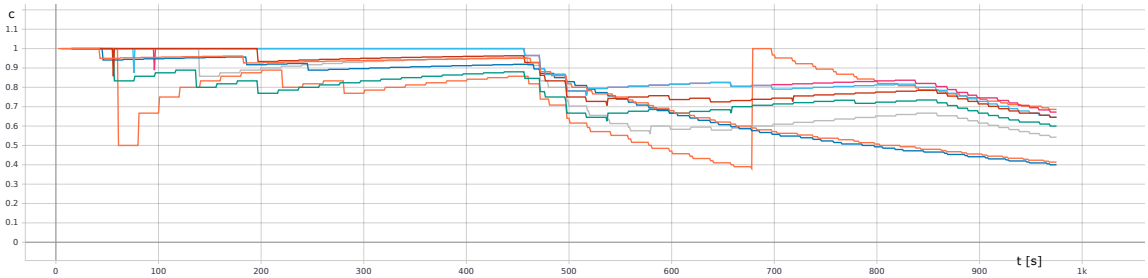


Figure D.12: Pattern 9 sliding window for devices with ID 30-38.

D.4 Pattern 9 with Sliding Window and Periodicity Recovery

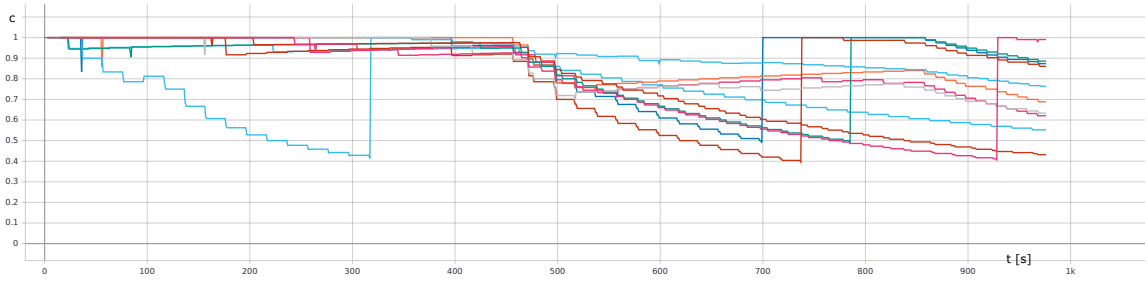


Figure D.13: Pattern 9 periodicity recovery for devices with ID 1-9.

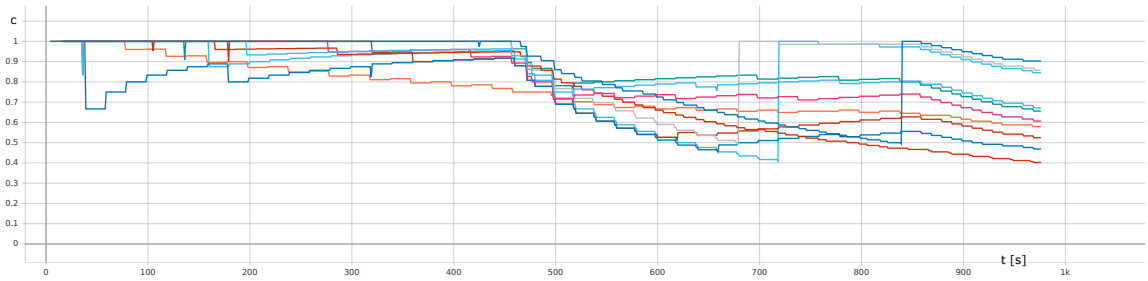


Figure D.14: Pattern 9 periodicity recovery for devices with ID 10-19.

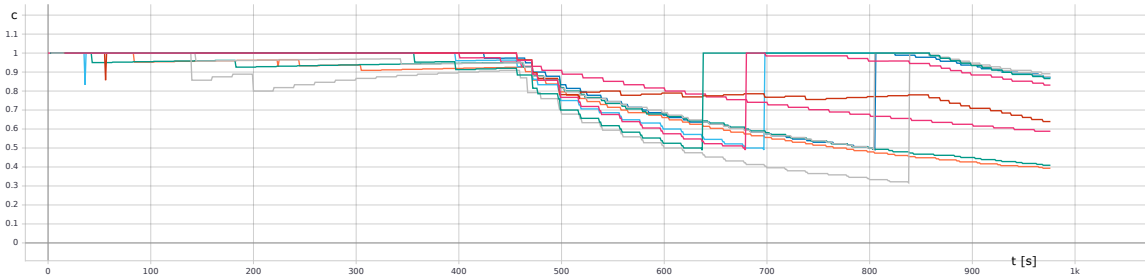


Figure D.15: Pattern 9 periodicity recovery for devices with ID 20-29.

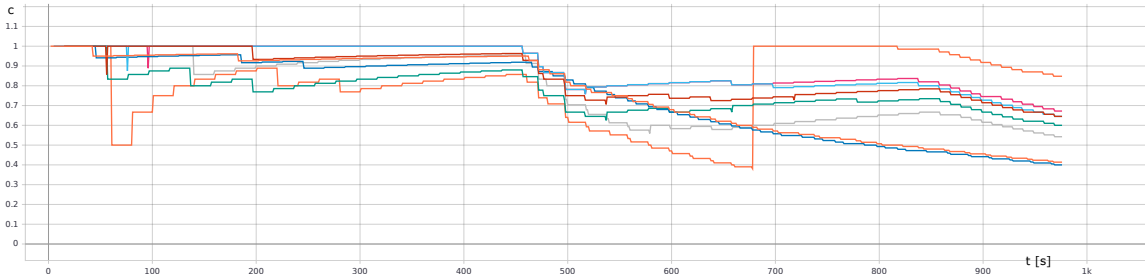


Figure D.16: Pattern 9 periodicity recovery for devices with ID 30-38.

Appendix E

CD Content

The attached CD contains the following items:

- **src** folder, which contains the source files,
- **doxygen** folder, which contains documentation of the source code,
- **output_examples**, a folder containing examples of analysis results (confidence reports and failures in both LOG and JSON format), these results are a part of reports collected during execution on the dataset with the simulated attack, which was discussed in this work,
- the text of this thesis in **DT_xjanco06.pdf**,
- **text** folder, which contains LaTeX source files,
- **README** file, describing both training and testing phases and usage of both implemented programs,
- **LICENSE** file, providing license details,
- **TEST_CASES.txt** file, describing a short step-by-step tutorial to test the functionality in real-time.