FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Diagnosis of traffic of ICS protocols
Student:	Bc. Peter Páleník
Supervisor:	Ing. Tomáš Čejka, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Information Security
Validity:	Until the end of summer semester 2020/21

Instructions

Study the area of Industrial Control System (ICS) protocols that are used for communication of devices in industrial environments.
Study the selected ICS protocols (such as CoAP, or IEC protocols) in detail, and focus on error states that can be identified in the network traffic.
According to the analysis of specification documents and traffic samples of the selected protocols, design decision trees for evaluation of network traffic to detect error states in communication.
Implement an extension of the DISTANCE diagnosis system that was developed by the CESNET association and Flowmon Networks; all resources will be provided by the supervisor.
Evaluate the performance of the developed extension using the provided datasets.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 4, 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práca

Diagnostika ICS protokolov

Bc. Peter Páleník

Katedra informační bezpečnosti
Vedúci práce: Ing. Tomáš Čejka, Ph.D.

28. mája 2020

Poďakovanie

Ďakujem svojmu vedúcemu Ing. Tomášovi Čejkovi, Ph.D. za pomoc s prácou, poskytnutie potrebných materiálov a ľudský prístup. Ďalej ďakujem Ing. Martinovi Holkovičovi, autorovi nástroja Distance, za rady pri tvorbe diagnostických pravidiel. Ďakujem Ing. Dominikovi Soukupovi a Ing. Jiřímu Havránkovi, ktorí pracovali na protokole IEC 104 a MQTT a s ktorými som spolupracoval za dobrú tímovú prácu a vzájomné rady. Na záver sa chcem poďakovať mojej manželke Anne Páleníkovej za podporu pri tvorbe tejto práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. V súlade s ustanovením § 46 odst. 6 tohoto zákona týmto udeľujem bezvýhradné oprávnenie (licenciu) k užívaniu tejto mojej práce, a to vrátane všetkých počítačových programov ktoré sú jej súčasťou alebo prílohou a tiež všetkej ich dokumentácie (ďalej len „Dielo“), a to všetkým osobám, ktoré si prajú Dielo užívať. Tieto osoby sú oprávnené Dielo používať akýmkoľvek spôsobom, ktorý neznižuje hodnotu Diela (vrátane komerčného využitia). Toto oprávnenie je časovo, územne a množstevne neobmedzené.

V Prahe 28. mája 2020 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na túto prácu

Páleník, Peter. *Diagnostika ICS protokolov*. Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Práca sa zaoberá diagnostikou ICS a IoT protokolov, konkrétne CoAP a IEC 61850, pomocou diagnostického nástroja Distance. Obsahuje analýzu spomenutých protokolov, detailné rozobratie možných chybových stavov a popis bezpečnostných hrozieb. Následne popisuje diagnostický nástroj Distance a spôsob tvorby jeho pravidiel – konfiguračných súborov. Ťažiskom je detailná štúdia protokolov a ich možných chybových stavov. Z tejto analýzy práca následne odvodzuje diagnostické pravidlá, ktoré čo možno najpresnejšie popisujú chovanie protokolu a snažia sa odhaliť možné konfiguračné chyby alebo bezpečnostné incidenty. Výsledkom práce sú konfiguračné súbory nástroja Distance pre protokoly CoAP a IEC 61850. Tieto súbory boli otestované na vlastnoručne vygenerovaných a aj ostrých dátových tokoch.

Klíčová slova ICS, IoT, CoAP, IEC 61850, MMS, GOOSE, Distance

Abstract

This thesis deals with diagnostics of ICS and IoT protocols, specifically CoAP and IEC 61850, using diagnostic engine Distance. It contains an analysis of the aforementioned protocols, analysis of possible error states, and description

of security risks. Subsequently, it describes diagnostic engine Distance and the process of creating its rules – configuration files. The core of this work is a detailed study of the protocols, and their possible error states. From this analysis, it subsequently derives diagnostic rules, which describe the protocol as precisely as they can, and they also try to discover possible misconfiguration problems or security incidents. The product of this thesis are Distance configuration files for protocols CoAP and IEC 61850. These files were tested on self-generated and production network traffic.

Keywords ICS, IoT, CoAP, IEC 61850, MMS, GOOSE, Distance

Obsah

Úvod

Automatizácia v sfére priemyslu je dnes samozrejmosťou. Každý priemyselný objekt má veľké množstvo rôznych senzorov (teplota, vlhkosť atď.), detektorov (požiar, dážď atď.), prepínačov, ističov, motorov, ventilov. Každá veľká budova potrebuje ventilačný systém, ktorý musí bezpodmienečne dodávať čerstvý vzduch a regulovať teplotu a vlhkosť v budove.

Rovnako ani výrobné procesy sa už v dnešnej dobe nezaobídu bez automatizácie. Automobilky potrebujú koordinovať dopravné pásy, robotov, žeriavy a zároveň mať neustále pod kontrolou bezpečnosť svojich zamestnancov. Elektrárne zasa potrebujú zaisťovať nepretržitú reguláciu energie v podobe vody alebo palivových tyčí, mať prehľad o vyrobenej energií a tiež monitorovať teplotu v jadre, či výšku vodnej hladiny v priehrade.

Automatizovať tieto procesy je nutnosť, avšak existuje viacero spôsobov. Jeden spôsob je mať na každý systém dedikovanú kabeláž privedenú do kontrolnej miestnosti, a privedenú k správnej kontrolke či prepínaču. Lepší spôsob je však využiť silu počítačových sietí, ktoré umožňujú pomocou jedného média – väčšinou káblu ale veľmi často aj bezdrôtového signálu – preniesť rôzne typy správ. Nie je tak nutné mať dedikovanú kabeláž ťahanú od koncových zariadení do kontrolnej miestnosti, stačí kabeláž koncentrovať do zariadení s podporou sieťovej komunikácie a následne prenášať namerané hodnoty či príkazy pomocou sieťovej architektúry s podporou štandardných sieťových protokolov. Ďalšou výhodou je, že kontrolná miestnosť môže byť len virtuálna – napríklad desktopová aplikácia a s pomocou internetu môže byť dokonca úplne „off-site“, teda kľudne aj tisíce kilometrov od ovládaných fyzických zariadení.

Spomínané priemyselné riešenia, nazývané Industrial Control Systems (ICS), sa prelínajú aj s „filozofiou“ Internet of Things (IoT), ktorú možno chápať ako internetom či inou sieťou prepojené zariadenia. Tieto zariadenia sú vybavené procesorom a sieťovým rozhraním na komunikáciu a ich firmware umožňuje napríklad ovládanie zariadenia alebo odosielanie dát cez sieť. Medzi takéto zariadenia môžu tiež patriť aj spomínané priemyselné senzory, ventily

atď. Avšak táto kategória zahŕňa omnoho viac. Patria do nej autá, IP kamery, drony, mobily, hodinky, ale zariadenia z odvetvia chytrej domácnosti (TV, routre, chytré žiarovky, robotické vysávače atď.).

V sfére priemyslu, konkrétne v odvetví energetiky sa často používa štandard IEC 61850, ktorý popisuje viacero protokolov umožňujúcich komunikáciu medzi zariadeniami.

V oblasti IoT sa zasa okrem iných používa protokol CoAP, ktorý je zjednodušene povedané binárnym HTTP nad UDP.

ICS protokoly často prepájajú fyzický svet s tým kybernetickým. To však znamená, že ak niečo prestane fungovať virtuálne, môže niečo prestať fungovať aj fyzicky. To môže mať veľmi vážne následky ak sa jedná napríklad o ovládanie palivových tyčí v jadrovom reaktore alebo o systém pre detekciu požiaru. Preto je ich správne fungovanie nutné pre zachovanie fyzickej bezpečnosti. Chyba v konfigurácií zariadení, v sieťovej infraštruktúre alebo kybernetický útok môžu spôsobiť veľmi vážne fyzické následky.

Tu prichádza do hry diagnostický nástroj Distance. Jedná sa o engine, ktorý pomocou rozhodovacích pravidiel modeluje stavy sieťových protokolov a kontroluje ich hlavičky/atribúty/obsah a na základe toho generuje udalosti (events) popisujúce povšimnutiahodné chovanie sieťového toku daného protokolu. Táto práca sa zaoberá práve tvorbou takýchto pravidiel v podobe YAML konfiguračných súborov.

V práci sú najprv vysvetlené oblasti ICS a IoT, následne je preberaná špecifikácia ICS61850 a CoAP, analýza bezpečnostných hrozieb, popis nástroja Distance a napokon sa prechádza k samotnej implementácií diagnostických pravidiel a k ich testovaniu.

V analýze aj implementácií som sa detailnejšie zameral na protokol CoAP, ktorého špecifikácia má okolo 100 strán v porovnaní s IEC 61850 kde sa jedná o viac než 1000 strán, čo by bolo časovo nezvládnuteľné detailne preštudovať.

Kapitola 1

Cieľ práce

Hlavným cieľom práce je naštudovať protokoly štandardu IEC 61850 a protokol CoAP a následne vytvoriť pravidlá (konfiguračné súbory) pre diagnostický nástroj Distance Predpokladá sa, že mnohé problémy a bezpečnostné hrozby sa prejavia v sieťovom toku. Preto s táto práca zameriava aj na dôkladnú štúdiu a analýzu protokolov, ich chybových stavov a možných rizík, s cieľom nájsť indikátory potenciálnych problémov. Nemôže chýbať ani validácia vytvorených diagnostických pravidiel na testovacích a ostrých dátach.

Kapitola 2

Analýza a návrh

Aby bolo možné vytvoriť diagnostické pravidlá, vyžaduje si to najprv dôkladnú analýzu špecifikácií protokolov a tiež preštudovanie a pochopenie nástroja Distance. Práve na to som sa zameral v tejto kapitole.

2.1 ICS a IoT

Tieto dva pojmy zahŕňajú rôzne sféry, avšak majú veľa spoločného. V tejto sekcií sa na ne pozrieme hlbšie, popíšeme spoločné črty a rozdiely.

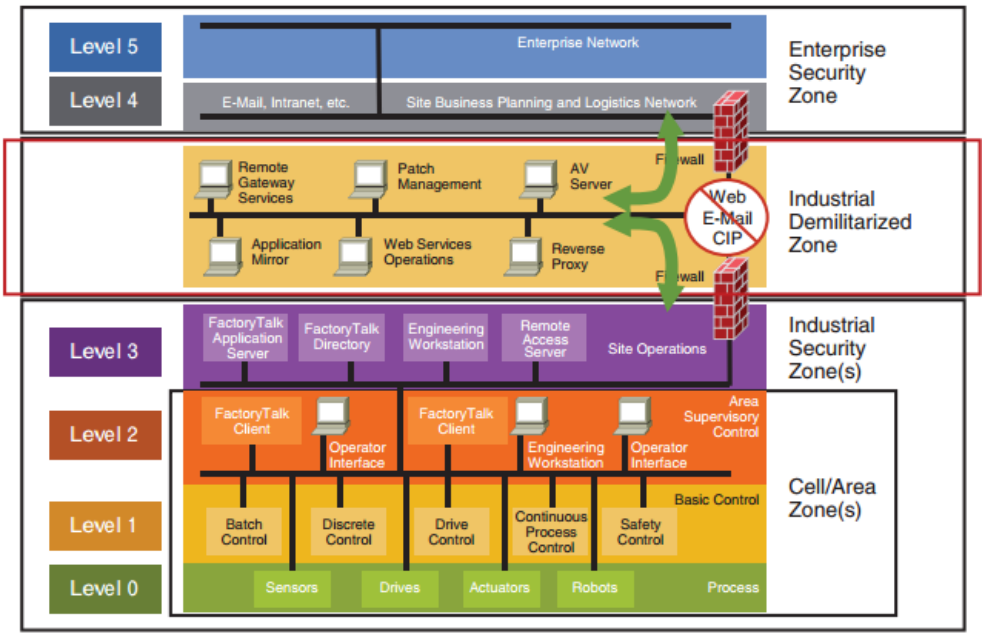
2.1.1 ICS

Industrial control system je pomerne všeobecný pojem zahŕňajúci automatizáciu priemyselných procesov pomocou množiny rôznych technológií. Výrobné procesy sú v dnešnej dobe nesmierne komplikované a ich automatizácia je nevyhnutná. Napriek tomu že implementácia takéhoto systému môže byť veľmi nákladná, v konečnom dôsledku to umožní vyrábať so zvýšenou efektivitou, a tak je návratnosť vysoká.

Komplexita systému môže byť rôzna. Od jedného kontroleru riadiaceho jednoduchý proces až po celosvetovo distribuovaný hierarchický systém prepojený rôznymi typmi sietí používajúci rôzne technológie a postavený na vysokej redundancií, vďaka čomu je schopný čeliť výpadkom.

Spomenutá redundancia je často nutná pre zabezpečenie nepretržitého chodu a odolnosti voči výpadkom. Často môžme byť svedkami duplikovaných kontrolerov, sieťových spojení či iných systémov. V niektorých prípadoch fungujú oba systémy súčasne, v iných len jeden a v prípade výpadku ten druhý systém prevezme kontrolu, často do jednotiek milisekúnd.

Rýchla odozva je vo veľa prípadoch nutná. To sa prejavuje aj na sieťových protokoloch ktorými systémy komunikujú. Protokoly majú v sebe špecifikovaný čas, do ktorého musí prísť odpoveď a následne chovanie, keď odpoveď nepríde. Rovnako aj kontrolery musia byť tomu prispôsobené. Nie je reálne,



Obr. 2.1: Purdue architektúra ICS. Zdroj: [1]

aby na nich bežal klasický operačný systém ako v klasických počítačoch či serveroch. Linux ani Windows nie sú real-time systémy, takže nezaručujú vykonanie nejakej rutiny v danom čase. Je preto často nutné v kontroleroch používať špecializovaný real-time operačný systém garantujúci dostatočne rýchlu odozvu.

Pozrime sa teraz na častý príklad architektúry ICS na obrázku 2.1 podľa [1]. Väčšinou sa architektúra riadi takzvaným *Purdue* modelom, ktorý rozdeľuje systémy na 3 zóny a 6 vrstiev.

- Enterprise – Jedná sa o štandardnú podnikovú sieť, ktorá funguje aj v iných odvetviach. Niekedy sa nad touto zónou ešte môže vyskytovať DMZ smerom do internetu, ak má firma verejne dostupné systémy.
  - Úroveň 5 – Enterprise network – Súčasťou tejto siete sú klasické pracovné stanice. Môže sa jednať aj o celosvetovú firemnú sieť, kde sú prepojené všetky závody či pobočky firmy.
  - Úroveň 4 – Site business and logistics – Súčasťou tejto vrstvy sú všetky podporné systémy zabezpečujúce business a výrobu ako emailové servery, súborové servery, či administratívne rozhranie k nižším vrstvám.
- Industrial Demilitarized zone – Slúži na logické oddelenie firemnej siete od lokálnej výrobnej siete. Súčasťou sú firewally do oboch susedných vrstiev a systémy, ktoré tieto dve vrstvy nepriamo prepájajú. V prípade

bezpečnostného incidentu je možné tieto zóny úplne oddeliť a tak zamedziť prístupu útočníka k výrobnému procesu bez prerušenia prevádzky. Výrobná zóna totiž dokáže naďalej autonómne pracovať, iba nie je možné posielať nové príkazy.

- Manufacturing (industrial) zone – Jedná sa o zónu samotnej výroby. Sú v nej teda všetky zariadenia, ktoré sa priamo podieľajú na výrobe, systémy ktoré ich ovládajú, systémy ktoré ich kontrolujú a tiež siete ktoré ich prepájajú. Takýchto zón môže mať jedna firma viac — napríklad každý závod má svoju vlastnú výrobnú zónu a zvyčajne aj DMZ.
    - Úroveň 3 – Site operations – Na tejto úrovni sa nachádza miestne operačné stredisko, to znamená že sútu terminály umožňujúce operátorom ovládať a monitorovať výrobu v tejto zóne.
    - Úroveň 2 – Area supervisory control – Podobné ako 3. úroveň, avšak viac špecializované na konkrétnu časť výroby alebo konkrétny systém.
    - Úroveň 1 – Basic control – Táto úroveň zabezpečuje základné ovládanie prvkov v nižšej úrovni. Nachádzajú sa tu PLC (Programmable Logic Controller), frekvenčné meniče alebo PID (proportional–integral–derivative) regulátory.
    - Úroveň 0 – Process – Tu sa nachádzajú samotné ovládané prvky ako motory, ventily a monitorovacie prvky – senzory. Táto úroveň sa stará o samotný výrobný proces.

Vidíme, že vrstvenosť systému umožňuje veľkú mieru modularity na vrstvách a zároveň tu máme zavedený princíp secure-by-design vďaka požadovanej DMZ. Samozrejme, DMZ nie je samospasná a je dobré aplikovať aj iné bezpečnostné princípy ako monitoring, autentizácia a autorizácia, šifrovanie a vykonávať bezpečnostný audit.

2.1.2 IoT

Internet of Things je pomerne abstraktný pojem popisujúci „dotiahnutie“ siete či internetu do jednoduchých či komplikovanejších zariadení — vecí. Tieto zariadenia sa tak stávajú „smart“ zariadeniami — rozširuje sa ich funkcionalita a pridáva sa im schopnosť „rozmýšľať“. Samotné zariadenia často obsahujú len jednoduchý mikroprocesor, avšak „rozmýšľanie“ môže byť delegované do *Cloudu*. Táto filozofia umožňuje prepájať „hlúpe“ zariadenia typu žiarovka, zámok či termostat a dáva užívateľovi ich na diaľku ovládať alebo automatizovať ich správanie. [2]

IoT architektúra býva často jednoduchšia než architektúra ICS, avšak môže byť aj oveľa zložitejšia a to aj vďaka tomu, že sa jedná o tak všeobecný pojem. Najčastejšou a asi najjednoduchšou architektúrou je: $IoT zariadenie \rightleftarrows Cloud \rightleftarrows Klient$.

- IoT zariadenie – Tu si môžme predstaviť každé chytré zariadenie, ktoré dokáže sieťovo komunikovať a byť ovládané klientom. Senzory, žiarovky, spotrebiče a pod.

- Cloud – Tu sa nachádza serverová časť systému a prepája klienta so samotnými zariadeniami. Väčšina logiky systému sa nachádza práve tu.

- Klient – Je to časť systému, ktorá umožňuje monitorovať, ovládať a spravovať zariadenia. Najčastejšie je klientom mobilná aplikácia.

Niekedy sa ešte medzi cloudom a zariadeniami nachádza IoT brána (gateway), ktorá spravuje a prepája zariadenia ktoré nepodporujú priame pripojenie na klasickú sieť, ale používajú nízkoenergetickú rádiovú komunikáciu pomocou ZigBee alebo BLE (Bluetooth Low Energy). Vďaka tomu nemusí každé zariadenie podporovať protokol Ethernet a znižuje s tak komplexita a cena zariadení.

V niektorých prípadoch môže klient komunikovať priamo so zariadeniami či bránou a nemusí ísť cez cloud. Výhodou je efektívnosť komunikácie po lokálnej sieti, avšak nevýhodou je že klient aj zariadenie nemôžu komunikovať ak nie sú na rovnakej sieti.

Sféry, kde sa často využíva IoT je hlavne oblasť chytrej domácnosti, ale aj ďalšie, a to napríklad transport, automatizácia miest, medicína či monitorovanie životného prostredia.

Často sa používajú „lightweight“ protokoly a technológie, pretože IoT zariadenia majú často obmedzený výpočtový výkona a tiež sú často napájané z obmedzeného zdroja energie (batéria či solárny panel). Stretávame sa s protokolmi MQTT či CoAP a technológiami LPWAN (Low Power WAN), ZigBee, BLE.

2.1.3 Zhrnutie

Na záver môžme zhrnúť, že ICS sa týka oblasti priemyslu a IoT skôr oblasti chytrej domácnosti. IoT je však tak rozšíriteľný koncept, že sa v dnešnej dobe často aplikuje aj v priemysle. Takže by sa dalo povedať, že ICS a IoT majú spoločnú podmnožinu.

2.2 Štúdia protokolov

Bolo nutné vykonať detailnú štúdiu protokolov, aby bolo možné vytvoriť pre ne diagnostické pravidlá. V tejto sekcií sa pozrieme na štandard IEC 61850 z oblasti ICS a protokol CoAP z oblasti IoT.

2.2.1 IEC 61850

IEC 61850 je medzinárodný štandard pre komunikáciu systémov elektrických staníc. Štandard sa skladá z 10 častí a má okolo 1200 strán.

Čo je z pohľadu tejto práce najdôležitejšie, že definuje protokoly, pomocou ktorých môžu komunikovať takzvané IED (Intelligent Electronic Device).

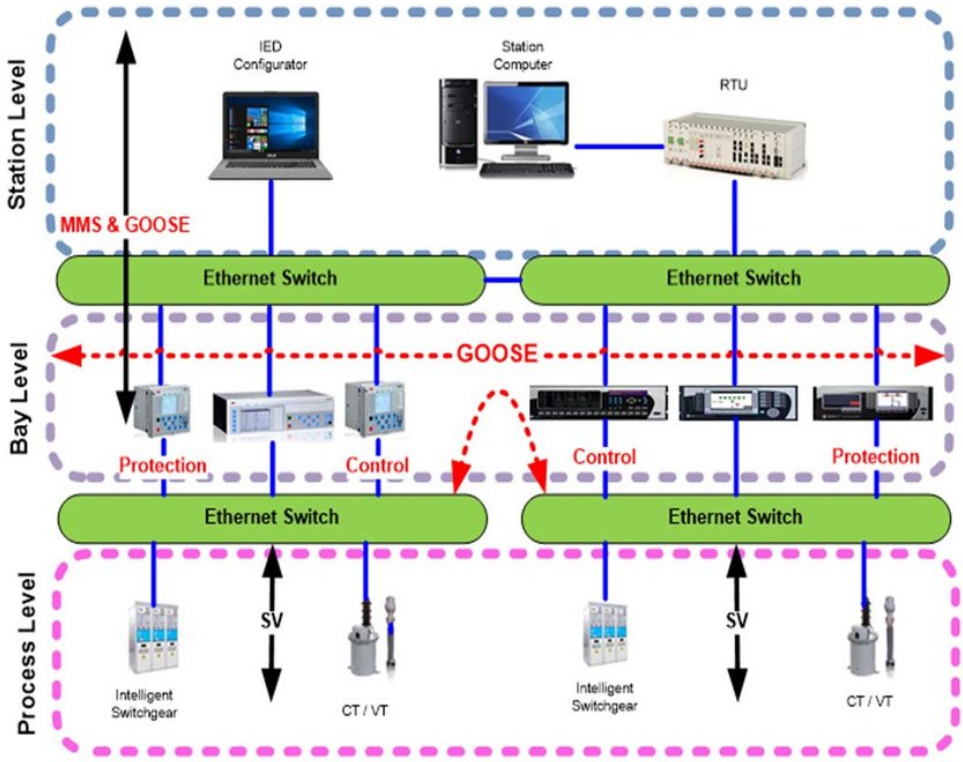
Okrem protokolov štandard zavádza aj dátový model a jeho mapovanie do protokolov, štandardizuje prístup k zariadeniam, ich službám a ich dátam, zavádza konfiguračný jazyk zariadení a logické oddelenie služieb zariadení od používaných protokolov. [3]

Motivácia pre vznik Automatizácia elektrických staníc je rovnako nutná, ako automatizácia v akejkoľvek inej sfére priemyslu. Obzvlášť v sfére výroby a distribúcie elektrickej energie je nutné na zmeny reagovať automatizovane a v reálnom čase. Preto sa v tejto oblasti používalo mnoho technológií a protokolov. Problémom však je, že mnohé z nich sú proprietárne a každý výrobca môže používať iný protokol. Preto vznikol štandard IEC 61850, aby sa zjednotil komunikačný model medzi systémami od rôznych výrobcov a zaistila kompatibilita IED zariadení.

Protokoly Protokolmi, ktoré štandard definuje sú MMS, GOOSE a SVM, ktoré preberieme. Najdôležitejším protokolom pre tento štandard však je Ethernet, na ktorom je postavená celá architektúra. Vďaka Ethernetu, nad ktorým funguje protokol GOOSE a SVM môžu IED na lokálnej sieti komunikovať obrovskou rýchlosťou s nízkou latenciou, čím je zaistená dostatočne rýchla odozva vyžadovaná štandardom. MMS zasa pracuje nad protokolmi vyššej úrovne, ktoré zaisťujú spoľahlivosť prenosu.

Architektúra Architektúra elektrickej stanice podľa IEC 61850 je zobrazená na obrázku 2.2. Je rozdelená na 3 úrovne, ktoré sa podobajú prvým 3 vrstvám Purdue architektúry zobrazenej na obrázku 2.1. Medzi nimi sa nachádzajú siete, zvané zbernica (bus).

- Station level – Na tejto úrovni sa nachádzajú takzvané HMI (Human Machine Interface) zariadenia a systémy prepájajúce stanicu s ostatými stanicami či dispečingom.
- Station bus – Sieť, ktorá prepája tieto dve vrstvy.
- Bay level – Na tejto úrovni sú zariadenia ktoré priamo kontrolujú a ovládajú zariadenia na nižšej úrovni.
- Process bus – Sieť, ktorá prepája tieto dve vrstvy.
- Process level – Tu sa nachádzajú samotné výkonné a detekčné zariadenia ako inteligentné rozvádzače, transformátory a senzory.

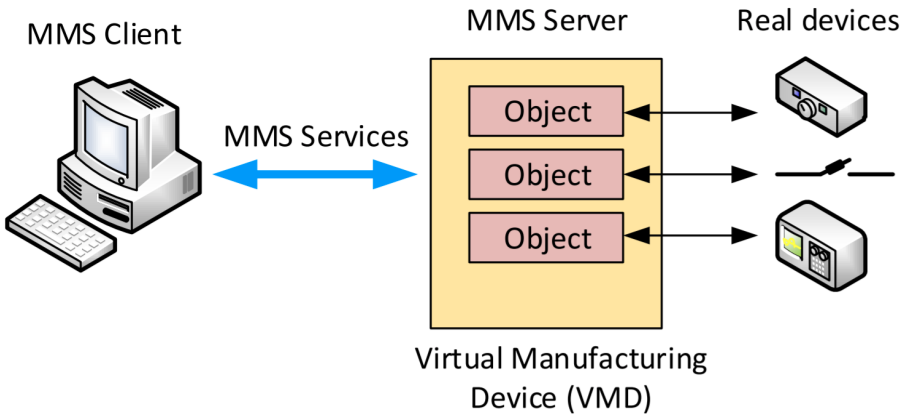


Obr. 2.2: IEC 61850 Architektúra elektrickej stanice. Zdroj: [4]

MMS

MMS, čo znamená Manufacturing Message Specification je protokol ktorý umožňuje spoľahlivú komunikáciu medzi klientom a serverom, ktoré si prostredníctvom neho môžu vymieňať procesné dáta, zisťovať informácie o zariadení a volať definované operácie. MMS presne definuje správy ktoré môžu byť posielané a objekty ktoré musia existovať na každom zariadení a na ktorých môžu byť vykonávané štandardné operácie ako čítanie a zápis [5].

MMS definuje takzvané VMD (Virtual Manufacturing Device), ktoré vytvára abstrakciu nad reálnymi zariadeniami. MMS server je zaradenie alebo aplikácia, obsahujúca toto VMD a jeho objekty, ku ktorým môže klient pristupovať. MMS klient pošle na server operáciu ktorú chce vykonať nad daným objektom a server mu vráti odpoveď. Tento model je zobrazený na obrázku 2.3.



Obr. 2.3: Klient-server architektúra MMS. Zdroj: [5]

MMS používa objektovo orientovaný prístup – nad dátami definuje operácie [5].

MMS protokol definuje dva typy komunikácie – potvrdzovanú a nepotvrdzovanú [5]. Potvrdzovaná služba vyžaduje odpoveď, ktorá bude buď úspech alebo chyba. Dotaz a odpoveď sú navzájom korelované pomocou položky `invokeID`, ktorá je 32 bitové číslo.

Nepotvrdzovaná služba je volaná bez vrátenia odpovede. Slúži na informovanie klienta o zmene či udalosti na serveri.

Enkapsulácia MMS Protokol MMS je pomerne hlboko enkapsulovaný do iných protokolov. Z obrázku 2.4 vidíme, že sú netradične obsadené všetky vrstvy OSI modelu a na niektorých je dokonca viac protokolov.

Layer	PDU	Protocols
Application (L7)	APDU	Manufacturing Message Specification (MMS): ISO 9506
		Association Control Service Element (ACSE): ISO/IEC 8650/X.227
Presentation (L6)	PPDU	OSI Connection Oriented Presentation ISO 8823/X.226
Session (L5)	SPDU	OSI Connection Oriented Session: ISO 8327/X.225
Transport (L4)	TPDU	Connection-Oriented Transport Protocol: ISO/IEC 8073/X.224
		ISO Transport over TCP (TPKT): RFC 1006
		Transmission Control Protocol (TCP): RFC 793
Network (L3)	NPDU	Internet Protocol (IP): RFC 791
Data Link (L2)	Data Frame	Ethernet: ISO/IEC 8802-3
Physical (L1)	Bits	

Obr. 2.4: Protokol MMS so všetkými jeho protokolmi na nižších vrstvách. Zdroj: [5]

Cieľom však nie je vysvetliť tieto protokoly. Stačí vedeť, že rôzne typy MMS správ sa v týchto protokoloch prejavia rôznym spôsobom. Zároveň protokoly riešia aj segmentáciu správ, inicializáciu spojenia a kódovanie správ. [5]

MMS správa nesie v tele takzvané PDU (Protocol Data Unit), ktoré môže byť rôznych typov na základe toho, o aký typ správy sa jedná.

Priebeh komunikácie Na začiatku je nutné otvoriť spojenie. Najprv je nutné naviazať spojenie na nižších protokoloch. Následne klient pošle `initiateRequest-PDU`, ktoré je pokus o iniciáciu MMS komunikácie. Server odpovedá `initiateResponse-PDU`, čím je spojenie vytvorené. Účelom týchto správ je aj dohoda na detailoch spojenia a podporovaných službách. Príklad takejto inicializácie vidíme v na obrázku 2.5, kde vidíme naľavo request a napravo response. Zaujímavým je pole `Supported Services`, ktoré nesie informáciu o tom aké operácie daná strana podporuje. Ak server nemôže vytvoriť spojenie, odpovie správou s `initiateError-PDU`.

Po inicializácií prebieha komunikácia pomocou správ typu `confirmed-RequestPDU` a `confirmed-ResponsePDU`, ktoré sa k sebe mapujú pomocou spomínaného `invokeID`. Request vždy nesie v poli `confirmedServiceRequest` názov služby – teda operácie ktorá má byť vykonaná. Na začiatku si klient vypýta zoznam objektov ku ktorým možno pristupovať a následne nad nimi vykonáva operácie. [5]

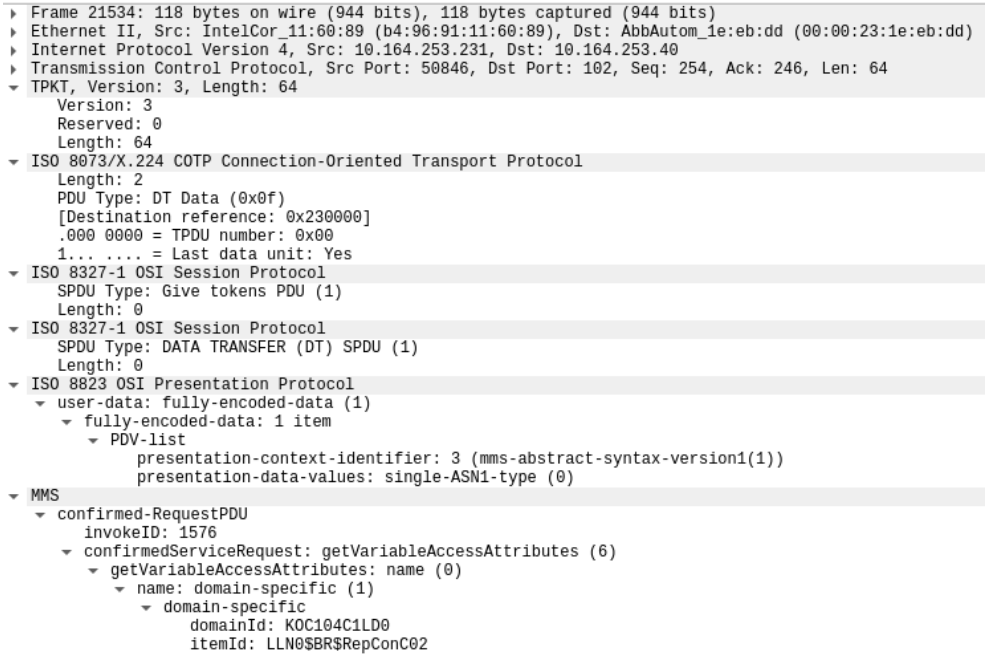
Okrem potvrdzovaných správ sa v tejto fáze posielajú aj nepotvrdzované, avšak iba zo serveru na klienta. Tento typ správ umožňuje klienta notifikovať a tak klient nemusí napríklad periodicky kontrolovať nejakú hodnotu. Existujú 3 typy týchto správ: `UnsolicitedStatus`, `InformationReport` a `EventNotification`. [5]

Na záver musí byť komunikácia ukončená, aby sa korektne uvolnili zdroje. Keď chce klient ukončiť spojenie, pošle `concludeRequest-PDU` na ktoré server odpovedá `concludeResponse-PDU` v prípade úspechu alebo `concludeError-PDU` v prípade neúspechu. V prípade vrátenia chyby ostáva spojenie otvorené. [5]

Na obrázku 2.6 vidíme príklad MMS requestu z programu Wireshark. Môžme si všimnúť hlbokú enkapsuláciu protokolu MMS do protokolov nižších vrstiev.

parameters	initRequest	initResponse
localDetailCalling	64000	32000
MaxServOutstandingCalling	10	10
MaxServOutsendingCalled	10	8
DataStructureNesting	5	5
Version	1	1
ConformanceBlock options	str1	str1
	str2	str2
	vnam	vnam
	vlis	vlis
Supported Services	status	status
		getNameList
	identify	identify
		read
		write
		getVariableAccessAttributes
		getNameVariableAttributes
		getDomainAttributes
		getCapabilityList
	obtainFile	
	fileOpen	fileOpen
	fileRead	fileRead
	fileClose	fileClose
	fileDelete	fileDelete
	fileDIrectory	fileDirectory
	informationReport	informationReport
	conclude	conclude
		cancel

Obr. 2.5: Príklad inicializácie spojenia. Zdroj: [5]



Obr. 2.6: Príklad MMS requestu z Wiresharku.

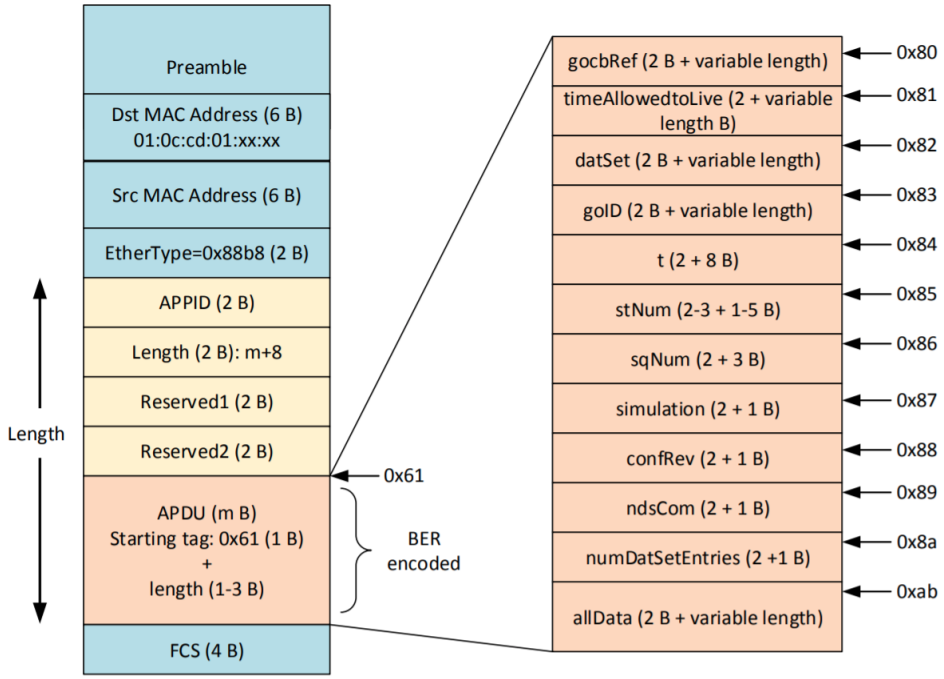
GOOSE

GOOSE, čo znamená Generic Object Oriented Substation Event, je protokol ktorý implementuje prenos časovo kritických udalostí medzi zariadeniami. Príkladom môže byť prekročenie povoleného napätia, čo môže vyžadovať prerušenie obvodu v záujme ochrany.

Médiom na prenos správ je lokálna sieť, konkrétne prepínaný Ethernet, čo redukuje hustotou kabeláže nutnú k prepojeniu zariadení. Protokol pracuje priamo nad vrstvou Ethernetu, takže je schopný pracovať len v rámci jedného sieťového segmentu. Protokol je založený na modeli publisher-subscriber, kde publisher – zdroj dát posiela dáta na istú multicastovú adresu a jeden či viac subscriberov sa k tejto adrese prihlási a počúvajú publikované správy. Výhodou umiestnenia protokolu GOOSE nad linkovú vrstvu je malá réžia (žiadne zbytočné protokoly transportnej vrstvy) a tiež nízka odozva (napríklad úplne obchádza ARP) v rámci sieťového segmentu prepínaného Ethernetu.

Správy protokolu GOOSE sú periodicky posielané ako keep-alive mechanizmus. Zároveň to znamená, že ak s pripojí nové zriadenie, ihneď má aktuálne informácie a nemusí čakať na zmenu. Ak nenastala žiadna zmena, je iba inkrementovaná položka *sqNum*. [5]

Správy sú posielané na multicastovú MAC adresu s prefixom `01:0c:cd:01` definovaným štandardom. Na tejto adrese môžu počúvať viacerí subscriberi.

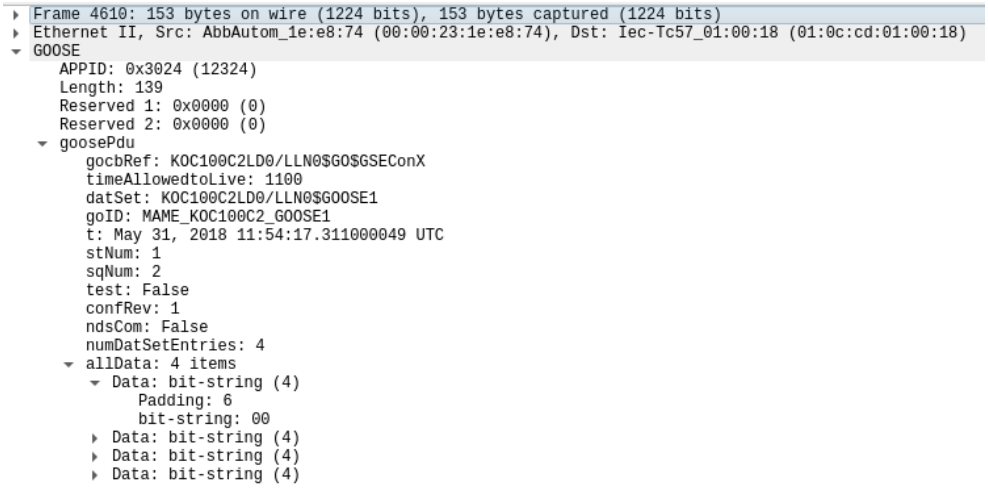


Obr. 2.7: Schéma GOOSE správy v rámci Ethernetového rámca. Zdroj: [5]

Pozrime sa teraz na najdôležitejšie položky definované štandardom [6]. Niektoré položky, ktoré nie sú relevantné pre diagnostiku preskakujem.

- APPID – identifikátor aplikácie
- Length – dĺžka správy
- Reserved 1 a 2 – Tieto položky sú použitá zabezpečená verzia protokolu definovaná v štandarde IEC 62351-6.
- sqNum – Sekvenčné číslo GOOSE správy, ktoré je inkrementované s každou GOOSE správou.
- stNum – Stavové číslo, ktoré je inkrementované vždy pri zmene dát.
- t – Čas kedy bola hodnota stNum inkrementovná.
- numDataSetEntries – Počet posielaných dátových položiek.
- allData – Zoznam užívateľsky definovaných dátových položiek.

Na obrázku 2.8 vidíme príklad GOOSE správy zobrazenej v programe Wireshark.



Obr. 2.8: Príklad GOOSE správy z Wiresharku.

SMV

SMV, čo značí Sampled Measured Values, je protokol, ktorého úlohou je v reálnom čase posielať namerané (nasamplované) hodnoty fyzikálnych veličín ako napríklad napätie či prúd z detekčných zariadení do kontrolných zariadení. Tento protokol funguje na rovnakom princípe ako GOOSE – nad Ethernetom a na modeli publisher-subscriber. Cieľom protokolu je doručiť informácie o nameraných hodnotách čo najrýchlejšie a bez zbytočnej réžie.

Bohužiaľ, pre protokol SMV sme s vedúcim práce nezohnali potrebné testovacie dáta. Preto bol tento protokol na základe dohody s vedúcim práce vynechaný ako v analytickej tak aj v praktickej časti.

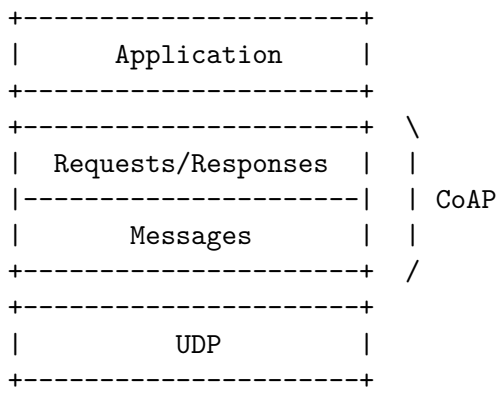
2.2.2 CoAP

Táto sekcia čerpá z RFC7252 [7]. CoAP, čo v skratke znamená Constrained Application Protocol, je protokol pre zariadenia s obmedzeným výkonom. Motivácia pre tento protokol bola požiadavka týchto obmedzených zariadení pristupovať k internetovým zdrojom pomocou REST architektúry.

CoAP implementuje podmnožinu HTTP protokolu, avšak v podobe, ktorá je optimalizovaná pre obmedzené zariadenia a M2M (Machine To Machine) komunikáciu.

Hlavné vlastnosti protokolu sú:

- Pracuje nad protokolom UDP s možnosťou spoľahlivého prenosu.
- Asynchrónny prenos správ.
- Nízka réžia a náročnosť na parsovanie.



Obr. 2.9: Vrstvy protokolu CoAP. Zdroj: [7]

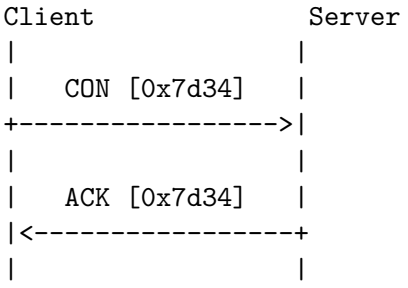
- Podpora URI a Content-type.
- Podpora cachovania a proxy.
- Mapovanie na HTTP a z HTTP – Umožňuje prístup cez proxy k HTTP zdrojom pomocou CoAP a CoAP zdrojom z HTTP.
- Podpora DTLS (Datagram Transport Layer Security) čo umožňuje šifrovaný prenos.

CoAP je založený na klient server modeli, kde klient posiela požiadavku (request) a server posiela odpoveď (response). Avšak povaha M2M komunikácie spôsobuje, že oba komunikujúce CoAP endpointy sa sa správajú ako klient aj ako server.

Základná architektúra protokolu CoAP je zachytená na obrázku 2.9. Protokol funguje nad UDP, nad ním stavia vrstvu správ a nad nimi vrstvu dotazov a odpovedí.

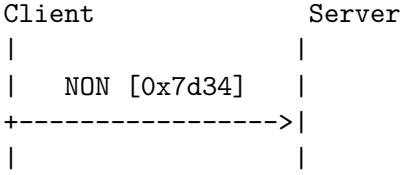
Model správ Správy sú posielané pomocou UDP datagramov. Sú podporované 2 typy prenosu – spoľahlivý a nespoľahlivý. Existujú 4 typy správ Confirmable (CON, potvrdzovaná), Non-Confirmable (NON, nepotvrdzovaná), Acknowledgement (ACK, potvrdenie) a Reset (RST). Každá správa obsahuje 16 bitové *Message ID*, ktoré umožňuje korelovať správu s jej potvrdením. Spoľahlivý prenos sa realizuje pomocou CON správy, na ktorú príde ACK odpoveď s rovnakým Message ID – obrázok 2.11. V prípade nedoručenia ACK v istom čase je správa opakovane odoslaná s rovnakým Message ID.

V prípade že adresát nedokáže z nejakého dôvodu správu spracovať, odošle naspäť Reset správu. Dôvodom môže byť strata kontextu na spracovanie správy pri reštarte zariadenia. Reset správa sa tiež môže použiť v spojení s prázdnou správou ako „CoAP ping“.



Obr. 2.10: Spoľahlivý prenos správy. Zdroj: [7]

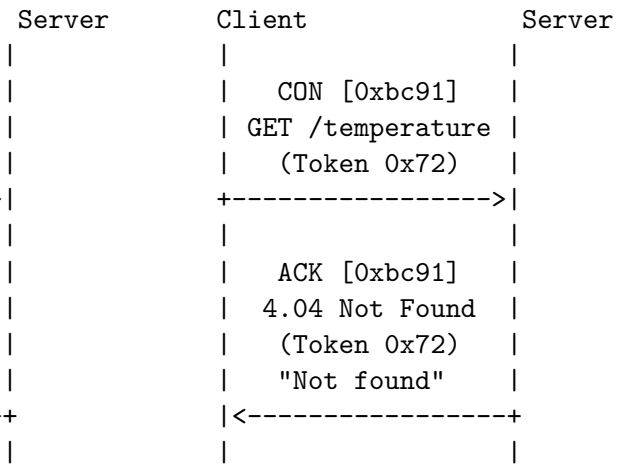
Nespoľahlivý prenos prebieha pomocou správy NON, na ktorú sa neočakáva potvrdenie – obrázok 2.11.



Obr. 2.11: Nespoľahlivý prenos správy. Zdroj: [7]

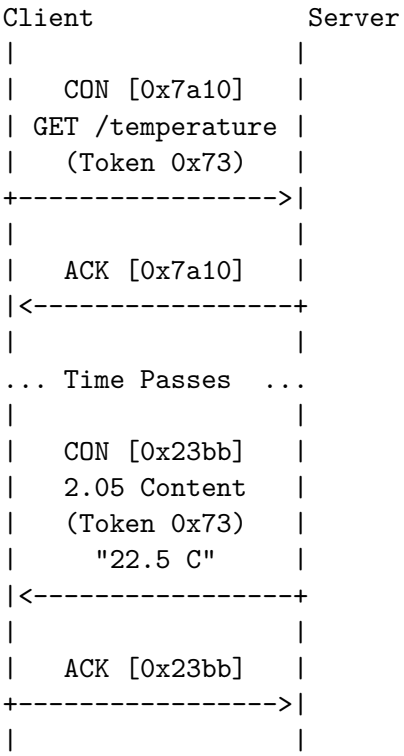
Model dotazov a odpovedí CoAP posiela request podobne ako HTTP, pomocou metódy GET, POST, PUT alebo DELETE. Každý request nesie v tele *Token*, ktorý je jeho identifikátor a nie je to to isté ako Message ID. Odpoveď následne obsahuje rovnaký Token.

V prípade spoľahlivého prenosu, ak je schopný server okamžite odpovedať, pošle odpoveď priamo v ACK správe – to sa nazýva *Piggybacked* odpoveď – obrázok 2.12.



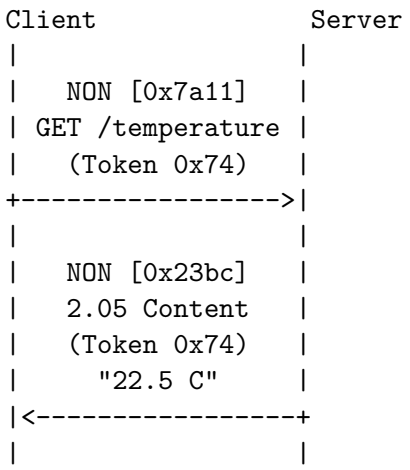
Obr. 2.12: Dve CON správy s Piggybacked odpoveďou. Zdroj: [7]

Ak nie je schopný hneď odpovedať, odošle najprv potvrdenie a následne pošle odpoveď ako zvlášť CON správu – takzvaná *Separátna odpoveď* – obrázok 2.13. Vidíme, že Token je pri odpovedi rovnaký, avšak Message ID je rôzne, pretože sa jedná o samostatnú správu.



Obr. 2.13: CON správa so separátnou odpoveďou. Zdroj: [7]

V prípade nespoľahlivého prenosu príde odpoveď ako NON správa – obrázok 2.14.



Obr. 2.14: NON správa s NON odpoveďou. Zdroj: [7]

Request môže ale nemusí obsahovať URI, ktoré identifikuje zdroj rovnako ako v HTTP. Formát URI vyzerá nasledovne:

```
"coap:" "//" host [ ":" port ] path [ "?" query ]
```

V prípade CoAP nad DTLS je používaná schéma `coaps:`. URI je dopredu rozparsované klientom a v správe sú nesené časti.

Request môže obsahovať *Options*, ktoré sú ekvivalent HTTP hlavičiek. Práve v týchto Options sú nesené časti URI.

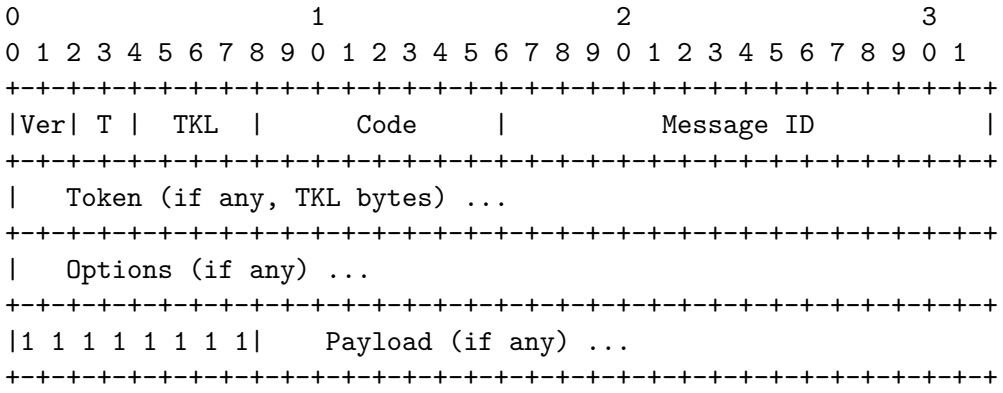
Options ďalej môžu obsahovať informáciu o type obsahu v správe a očakávanom type obsahu odpovede.

Response obsahuje podobne ako v HTTP kód, ktorý hovorí o úspešnosti requestu. Kódy majú rovnaký význam ako ekvivalentné HTTP kódy. Máme tu 3 kategórie:

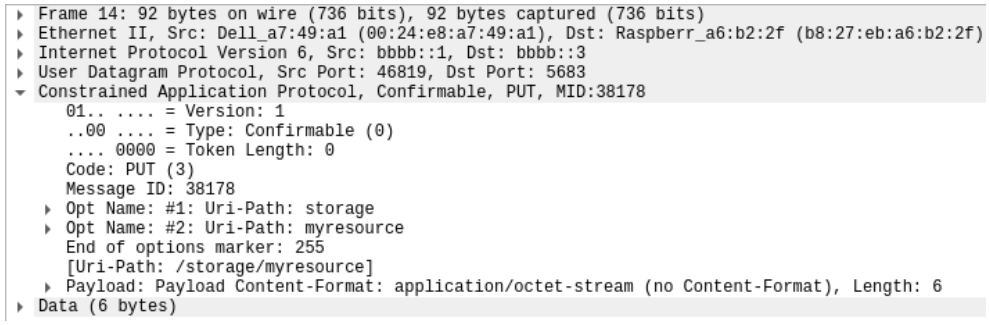
- 2.XX – Úspech
- 4.XX – Chyba klienta
- 5.XX – Chyba serveru

Ostatné kódy sú rezervované.

Formát správy Formát CoAP správy je zobrazený na obrázku 2.15 a príklad CoAP správy z programu Wireshark na obrázku 2.16.



Obr. 2.15: Formát správy. Zdroj: [7]



Obr. 2.16: Príklad CoAP requestu z Wiresharku. Vidíme PUT request na `/storage/myresource` s telom typu `application/octet-stream`. Jedná sa o potvrdzovanú správu a nie je použitý token.

Vysvetlime si teraz čo ktoré pole znamená.

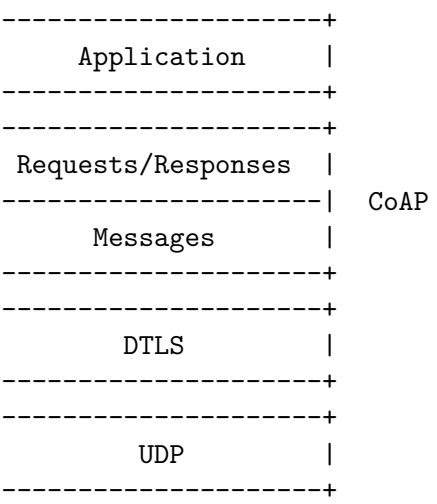
- Ver – verzia protokolu.
- T – Typ správy – CON, NON, ACK, RST.
- TKL – Dĺžka Tokenu. Ak je 0, správa je bez tokenu.
- Code – 8 bitové číslo, ke prvé 3 bity sú takzvaná *Class* a zvyšných 5 je *Description*. zapisuje sa ako c.dd. Class 0 znamená request, 2, 4, 5 sú response a ostatné sú rezervované. V prípade requestu má kód význam CoAP metódy, v response je to odpoveďový kód.
- Message ID – ID správy určené pre detekciu duplikátov a koreláciu správ s potvrdeniami.
- Token – ID requestu.
- Options – Môže ich byť 0 - N nesú dodatočné informácie o správe.

- Payload – Telo správy.

Bezpečnosť Protokol CoAP umožňuje šifrovanie dátového toku pomocou protokolu DTLS, ktorý funguje podobne ako TLS, ale nad UDP. Podporuje 4 režimy:

- NoSec – Bez DTLS. Dáta sú prenášané v plaintexte.
- PreSharedKey – Šifrované pomocou zdieľaného kľúča.
- RawPublicKey – Šifrované asymetricky pomocou páru verejného a súkromného kľúča, ktorý je overený Out-Of-Band.
- Certificate – Šifrované asymetricky pomocou X.509 certifikátu overeného nejakou dôvernou autoritou.

Prvý režim je indikovaný `coap:` schémou v URI, zvyšné tri používajú schému `coaps:`. Použitie DTLS vidíme na obrázku 2.17.



Obr. 2.17: Vrstvy protokolu s použitím DTLS. Zdroj: [7]

Ďalšie funkcie Významnou funkciou je proxying a caching, čo umožňuje urýchlenie komunikácie. Patrí sem aj CoAP → HTTP proxying, ktorý umožňuje CoAP uzlom pristupovať k HTTP zdrojom cez proxy. Tento režim je jednoduchý na realizáciu, pretože CoAP je podmnožinou HTTP. Podporovaný je aj opačný režim, teda HTTP → CoAP proxying, avšak nepodporuje všetky funkcie HTTP a v niektorých prípadoch môže byť vrátený chybový kód `501 Not Implemented`.

CoAP tiež podporuje posielanie multicastových správ, avšak len v nešifrovanej podobe. Server sa vždy môže rozhodnúť na takúto správu neodpovedať, samozrejme to záleží na tom čo vyžaduje aplikácia. Server na správu nemá

odpovedať hneď, ale v náhodne v istom časovom rozsahu zvanom *Leisure*. Odpoveď má byť posielaná unicastovo na zdrojovú adresu.

Zhrnutie CoAP je „lightweight“ IoT protokol používajúci podmnožinu HTTP operácií avšak komprimovanú z textovej do binárnej podoby, čím znižuje potrebnú réžiu na prenos správ. Tiež znižuje réžiu nahradením protokolu TCP protokolom UDP a definuje vlastný, jednoduchší, mechanizmus zaručeného prenosu správ. Preto je CoAP vhodný pre zariadenia z obmedzenými zdrojmi.

2.2.3 Porovnanie

Hlavným rozdielom medzi CoAP a IEC 61850 je, že CoAP je konkrétny protokol, avšak IEC 61850 je štandard definujúci resp. používajúci viacero protokolov. CoAP a protokoly štandardu IEC 61850 sú úplne odlišné. CoAP je orientovaný na komunikáciu s nízkou réžiou a optimalizovaný pre obmedzené zariadenia a obmedzené siete. Naopak protokoly IEC 61850 sú orientované na zariadenia pracujúce na elektrickej stanici, kde sa predpokladá neobmedzený prísun energie a tak na zariadenia nie sú kladené výkonnostné obmedzenia. To sa odzrkadľuje aj na protokoloch, napríklad GOOSE posiela správy redundantne. MMS zasa používa TCP protokol a dodatočné vrstvy nad ním, čo zvyšuje réžiu a v prípade obmedzených sietí by to nebolo žiadúce. V prípade elektrickej stanice však možno použiť kľudne 10G Ethernet cez optické vlákno, takže zariadenia môžu chrliť takmer neobmedzené množstvo správ.

Protokoly GOOSE a SMV sa od CoAP líšia značne tým, že sú jednosmerné, teda je tam nejaký publisher ktorý posiela správy a subscriber, ktorý ich počúva. CoAP je postavený na princípe request/response, čím sa podobá na MMS, ktorý tiež používa tento princíp, avšak nie je to vyžadované. Znamená to, že tieto protokoly podporujú obojsmernú komunikáciu.

Bohužiaľ odlišnosti ktoré protokoly majú sú oveľa väčšie než spoločné črty, a tak nie je možné zovšeobecniť ich diagnostiku.

2.3 Analýza bezpečnostných hrozieb

2.3.1 Kybernetické hrozby

Pri analýze kybernetických hrozieb som sa do väčšej hĺbky zameral na protokol CoAP. IEC 61850 je rozobratý tiež, avšak nie až tak detailne hlavne z dôvodu spomínaného rozsahu špecifikácie (cez 1000 strán).

CoAP

U tohto protokolu oceňujem, že autori pri návrhu mysleli na bezpečnosť. Napriek tomu je však zdôrazňujú, že je možné zneužiť slabiny protokolu hlavne

pri chybnom použití. V špecifikácií pre tento protokol [7] je priamo sekcia venujúca sa bezpečnosti, z ktorej som primárne čerpal, a tiež z [8].

Parsovanie protokolu a spracovávanie URI CoAP sa snaží obmedziť priestor pre útok zjednodušením logiky nutnej pre parsovanie protokolu a parsovanie URI. Dôvodom je zvýšené riziko implementačnej chyby pri vyššej komplexnosti protokolu.

Spracovanie URI je ponechané na klienta, čo znamená že server dostane už len časti URI a nemusí ho parsovať sám, čo opäť znižuje pravdepodobnosť zraniteľnosti v kóde serveru.

Proxy a cachovanie Proxy je z definície man-in-the-middle, teda nutne narušuje šifrovanie v prípade DTLS. Sú teda rizikom pre Confidentiality (utajenosť) a Integrity (integritu).

Problémom môže byť aj cachovanie, kde sa dáta posielané protokolom ukladajú a hrozí ich modifikácia v mieste uloženia.

Spoofing IP adresy Keďže sa jedná o protokol fungujúci nad UDP, nie je vyžadované prvotné vytvorenie spojenia — handshake. Preto je tu riziko možného spoofingu zdrojovej IP adresy, teda podvrhnutia zdrojovej IP adresy. Tým pádom v útočník môže:

- Poslať falošnú RST správu ako odpoveď na CON alebo NON správu, čím „prehluší“ legitímnu odpoveď.
- Poslať ACK správu ako odpoveď na CON správu, čím sa „deaktivuje“ mechanizmus retransmisie správ v prípade straty správy.
- Poslať úplne falošnú odpoveď s podvrhnutými Options a telom správy. To môže mať rôzny dôvod, napríklad cache poisoning či pokus o útok typu buffer overflow na klientskú aplikáciu.
- Poslať falošný multicastový request s podvrhnutou zdrojovou adresov uzlu, na ktorý chce vykonať DoS útok. Tým spôsobí, že všetky zariadenia počúvajúce na danej multicastovej adrese zašlú odpoveď na adresu obete.

Týmto útokom sa dá zabrániť v prípade použitia DTLS. Ak útočník nemá možnosť čítať správy, môže útkom zabrániť použite dostatočne kvalitne náhodného Tokenu a Message ID. Message ID je však iba 16 bitová hodnota, čo je možné v niektorých prípadoch prejsť hrubou silou.

Riziko amplifikácie CoAP server v prípade použitia štandardných metód GET, PUT, POST, DELETE vždy zašle odpoveď na dotaz. Táto odpoveď môže byť oveľa väčšia než dotaz. Útočník teda môže zneužiť CoAP server na

to, aby z malého dotazu vytvoril veľkú odpoveď, čo nazývame amplifikácia. To môže byť zneužité na útok typu DoS, čo bude popísané nižšie.

CoAP siete však často nie sú schopné preniesť dostatočný dátový tok, aby mohli byť zdrojom útoku, avšak o to ľahším cieľom sa môžu stať ak útočník sieť zahltí.

Je na programátorovi, aby zaistil nízke amplifikačné pomery v prípade neautentizovaných requestov. CoAP server je schopný zaistiť malý amplifikačný pomer v prípade použitia blokového prenosu popísaného v [9].

Ďalším problémom môže byť schopnosť protokolu používať multicast, čo môže byť veľkého objemu odpovedí na jediný dotaz. CoAP sa proti tomu bráni randomizáciou času odpovede v prípade multicastu. Rovnako by mal byť endpoint odpovedajúci na multicast schopný nejakým spôsobom overiť autenticitu správy a tiež by mal byť dostatočne dobrý dôvod na použitie multicastu.

Medziprotokolové útoky Spoofing zdrojovej adresy (a portu) je možné zneužiť aj na útok na iný port či protokol. Útočník pošle na server správu z podvrhnutým zdrojovým portom a adresou obete. Následne server odpovie na túto adresu a port. Obeť obdrží datagram na danom porte, na ktorom beží služba, ktorá následne dáta interpretuje podľa svojich pravidiel.

Tento typ útoku je možné zneužiť na obídenie firewallu, ak je zakázaná priama komunikácia útočníka s obeťou.

CoAP server môže byť tiež obeťou medziprotokolového útoku, kedy je na vygenerovanie odpovede použitý iný UDP protokol, napríklad DNS.

Denial of Service DoS môže byť na CoAP uzol vykonaný viacerými už popísanými spôsobmi. Ak zariadenie používa nejakú nízkovýkonovú sieť, je možné preťažiť samotnú sieť a tak zamedziť zariadeniu komunikovať.

Rovnako je možné preťažiť zariadenie samotné, napríklad spracovaním veľkého množstva paketov na sieťovom rozhraní, keďže majú často obmedzený výkon. Ďalší spôsob môže byť posielať na zariadenie veľké množstvo requestov ktorých spracovanie má v sebe veľa aplikačnej logiky.

Zariadenia často fungujú len na batérií, takže môže byť cieľom útočníka zvýšiť príkon zariadenia jeho zaťažením, čo spôsobí rýchlejšie vybitie batérie.

Riziká obmedzených zariadení CoAP často používajú zariadenia s obmedzenými zdrojmi. Často nemusia byť vybavené kvalitným zdrojom entropie. V takom prípade by zariadenie nemalo generovať kľúče samo, ale mali by byť externe vygenerované a do zariadenia nahrané pri výrobe či inštalácií.

Ďalšou hrozbou súvisiacou s obmedzeným výkonom sú časové útoky. Rozdielny čas výpočtu sa totiž prejaví oveľa viac pri nižšom výkone. Preto je nutné použiť robustné funkcie pri implementácií kryptografických operácií.

Rizikom môže byť aj nízka ochrana zariadenia voči *tamperingu*, teda nedovoleného zásahu do hardwaru zariadenia napríklad s cieľom extrahovať sú-

kromné kľúče. Je preto vhodné použiť pre každé zariadenie iný súkromný kľúč, aby pri kompromitácií jedného neboli kompromitované všetky.

IEC 61850

Štandard IEC 61850 bohužiaľ nerieši detailne bezpečnosť. Tú doplňuje až štandard IEC 62351, ktorý napríklad pridáva do MMS protokolu požiadavku na autentizáciu a použitie TLS. Od GOOSE sa zasa vyžaduje použitie VLAN (Virtual LAN), teda segregáciu zariadení ktoré spolu nemajú komunikovať. [10]

Odpočúvanie Keďže žiaden z týchto protokolov nevyužíva šifrovanie, je jednoduché pre útočníka, ktorý sa dostane do rovnakého sieťového segmentu, odpočúvať sieťový tok. Tieto dáta môže následne útočník použiť na iné nekalé účely, napríklad na ďalší útok.

Spoofing a injekcia GOOSE správ Tento útok spočíva v počúvaní GOOSE správ v sieťovom segmente a následné generovanie podvrhnutých správ s položkami ako napríklad `sqNum` odvodenými z legitímnych odpočutých správ. Takýto útok je takmer nedetekovateľný, pretože protokol nepodporuje žiadnu autentizáciu zdroja správy. Správa sa preto javí úplne legitímne a útočník teda môže vykonávať všetky operácie, ktoré zariadenia podporujú. Tento typ útoku si vyžaduje priamy prístup do sieťového segmentu v ktorom sú GOOSE komunikujúce zariadenia.

Denial of Service Jedná sa o klasický typ útoku, kedy útočník nejakým spôsobom vyradí systém trvalo alebo dočasne z prevádzky. Existuje viacero spôsobov, napríklad môže útočník zahltiť zariadenie veľkým množstvom správ, ktoré sieťové rozhranie síce stihne preniesť, ale aplikačné nároky na spracovanie protokolu sú tak vysoké, že systém nemá dostatočné prostriedky na spracovanie legitímnych správ.

Ďalším spôsobom môže byť pokus o zahltenie samotného sieťového rozhrania. To môže útočník dosiahnuť napríklad amplifikáciu sieťového toku napríklad pomocou broadcastového ICMP echo — Smurf attack. Ďalší spôsob môže byť amplifikácia pomocou spoofingu requestu nejakej UDP služby. Tieto útoky však často vyžadujú nesprávne nakonfigurované zariadenia/protokoly, ktoré umožnia ich zneužitie.

Zhrnutie Všetky tieto útoky si väčšinou vyžadujú priamy prístup do siete elektrickej stanice. Buď sa môže jednať o fyzický prístup, kedy má útočník takmer neobmedzené možnosti, alebo o vzdialený prístup napríklad kompromitovaním nadradených systémov a následným prienikom hlbšie do siete samotnej stanice.

2.3.2 Fyzické hrozby

Keďže ICS a IoT sú tak hlboko previazané s reálnym svetom a fyzickými a fyzikálnymi procesmi, následky útoku či chybnej konfigurácie môžu byť aj životohrozujúce.

Vo všeobecnosti pri ICS môžu byť následky rôzne, čo záleží od konkrétneho odvetvia V prípade elektrickej stanice môže byť následkom napríklad vypnutie dodávky elektrickej energie, preťaženie siete či ohrozenie zamestnancov.

V prípade IoT to opäť závisí od odvetvia. Veľmi kritická môže byť oblasť automobility, kde môže byť v dôsledku ovládnutia systémov automobilu ohrozený vodič auta ale aj iní účastníci premávky.

Zhrnutie

Popísané kybernetické hrozby sú len špičkou ľadovca hrozieb v tomto odvetví. Každá z týchto hrozieb sa totiž môže prejaviť aj vo fyzickom svete, čo vyplýva z povahy a využitia ICS a IoT.

2.4 Projekt Distance

Projekt Distance je spoločný projekt združenia CESNET a Flowmon Networks, ktorý obsahuje nástroje na analýzu PCAP súborov s cieľom odhaliť problémy v konfigurácií alebo komunikácií. Nástroje sú iba skripty s textovým výstupom, avšak po pridaní grafického rozhrania majú slúžiť administrátorom pri odhaľovaní spomínaných problémov. Podľa dokumentácie [11] sa projekt Distance skladá z nasledujúcich častí:

- **Ingestor** Agreguje dáta z PCAP súborov do lokálnej databázy.
- **Capinfo** Poskytuje štatistiky o IP flow dátach (viac o flows napríklad tu [12]), ktoré do DB zapísal ingestor. Tu je príklad výstupu:

```
Top 10 Src IP Addr ordered by flows:
Duration     Src IP Addr  App  Flows  Packets   Bytes  pps    bps  bpp Seg. lost  Retr.
1192.834  94.180.151.203       62794   219715  25.9 M  184 182294  123         2     21
 330.110    9.209.28.173       27864    47943   2.2 M  145  55769   48         0      0
 304.257   255.93.216.44       22769   102496   7.6 M  336 210205   77         0      0
1190.594  125.248.33.146       17271    41942   5.7 M   35  40438  143         0      0
 341.892   138.5.122.251       12253    75925  39.2 M  222 962768  541         0      0
 310.211  130.195.23.210       11742    46928   3.2 M  151  86940   71         0      0
 304.257   255.93.216.43       11383    56943   4.5 M  187 123968   82         0      0
 304.894   219.182.16.57       11209    44784   2.0 M  146  54640   46         0      0
1068.361      3.15.99.52        9000    16962   3.1 M   15  24415  192         0      0
 172.102  11.121.123.165        7176     7176  330096   41  15344   46         0      0
1147.332               *      109183   202523  13.1 M  176  96116   68         8     56
```

- **Tcpinfo** Analyzuje TCP streamy z databázy a poskytuje štatistiky o nich. Tu je príklad výstupu:

```
TCP@192.168.1.101:4248-205.178.187.11:80#0
    origin: CLIENT
        connection_status: CLOSED_BY_RST
        client_connection_setup_time: 0.224 ms
```

```
    total_segments: 5
    total_bytes: 382
    lost_segments_pct: 0.00 %
    lost_segments_abs: 0
    lost_bytes_pct: 0.00 %
    lost_bytes_abs: 0
    flow_rxmt_avg_delay: nan ms
    not_captured_segments_pct: 0.00 %
    not_captured_segments_abs: 0
    not_captured_bytes_pct: 0.00 %
    not_captured_bytes_abs: 0
origin: SERVER
    connection_status: HALF_OPENED
    server_connection_setup_time: 33.729 ms
    tls_setup_time: nan ms
    data_transfer_time: 1.734 ms
    server_response_time: 44.666 ms
    round_trip_time: 0.864 ms
    total_segments: 5
    total_bytes: 2633
    lost_segments_pct: 0.00 %
    lost_segments_abs: 0
    lost_bytes_pct: 0.00 %
    lost_bytes_abs: 0
    flow_rxmt_avg_delay: nan ms
    not_captured_segments_pct: 0.00 %
    not_captured_segments_abs: 0
    not_captured_bytes_pct: 0.00 %
    not_captured_bytes_abs: 0
```


- **Diagnostics** Odhaľuje problémy v sieťovej komunikácií alebo zlej konfigurácií zariadení na základe sieťových dát z PCAP súborov Toto je modul, ktorým sa zaoberá a rozširuje ho táto práca. [11]

Keďže modulom Ingestor, Tcpinfo a Capinfo sa v tejto práci nevenujem, ak budem hovoriť o engine/nástroji Distance, budem mať konkrétne na mysli modul Diagnostics. Ďalej v tejto sekcií preberieme detailne práve tento modul.

Doposiaľ diagnostický modul riešil iba klasické protokoly používané v IP sieťach ako napríklad HTTP, SMTP, ICMP, či DNS. Táto práca k nim ešte pridáva protokoly zo sféry ICS a IoT.

2.4.1 Fungovanie

Diagnostika funguje na základe pravidiel, ktoré sú špecifické pre každý protokol. Pravidlá sú písané v jednoduchom deklaratívnom jazyku. Jedná sa o kombináciu wireshark filtrov, YAML formátu a Python kódu. Pravidlá sa snažia popisovať štandardné chovanie sieťového protokolu a tak v prípade nezrovnalosti odhaliť chybu v komunikácií či konfigurácií. Pravidlá fungujú v dvoch úrovniach. Prvá vrstva, zvaná facts, z PCAPu extrahuje dáta, ktoré sú dôležité pre rozhodovanie v druhej úrovni – takzvané fakty. Druhá vrstva, zvaná tree, je popis konečného automatu, ktorý na základe faktov prechádza medzi stavmi, a tak kontroluje chovanie protokolu.

2.4.2 Prvá vrstva

Ako bolo spomenuté, táto vrstva pravidiel sa snaží extrahovať podstatné fakty z protokolu. Extrakcia faktov prebieha pomocou vylepšeného Wiresharkového

filtra, ktorý sa vykoná nad PCAPom. Dokumentácia [13] popisuje pravidlo tejto úrovne nasledovne:

```
- rule:
    id: RULE_NAME
    facts:
        - FACT_NAME_1: FACT_FILTER_1
        - ...
        - FACT_NAME_N: FACT_FILTER_N
    params:
        - PARAM_NAME_1
        - ...
        - PARAM_NAME_N
    asserts:
        - CONDITION_1
        - ...
        - CONDITION_N
```

- id – Identifikátor daného pravidla, pomocou ktorého sa na pravidlo odkazuje
- facts (nepovinné) – Zoznam filtrov, pomocou ktorých sa extrahujú potrebné dáta. Každý fakt `FACT_NAME_*` bude vo výsledku obsahovať všetky rámce spĺňajúce podmienky.
- params (nepovinné) – Zoznam parametrov pre `facts` alebo `asserts` predávaných pravidlu z vyššej úrovne.
- asserts (nepovinné) – Zoznam doplňujúcich podmienok, ktoré musia všetky byť splnené.

Príklad pravidla z protokolu CoAP:

```
- rule:
    id: has_response
    facts:
        - response: udp.srcport == request[udp.dstport]
&& udp.stream == request[udp.stream]
    params:
        - request
    asserts:
        - response[coap.type] != 3
        - response[coap.token_len] == 0
|| request[coap.token] == response[coap.token]
        - response[frame.number] > request[frame.number]
        - response[coap.code] >= 32
```

Toto pravidlo má za úlohu k `requestu` – požiadavku nájsť `response` – odpoveď. Časť `udp.srcport == request[udp.dstport] && udp.stream == request[udp.stream]` hovorí o tom, že response má prísť z UDP portu, na ktorý bol adresovaný request a ďalej že má byť súčasťou rovnakého UDP streamu.

Parameter `request` prichádza z vyššej úrovne – bol už predtým identifikovaný a teraz k nemu hľadáme odpoveď.

Časť `asserts` obsahuje viacero pravidiel, napríklad `response[coap.type] >= 32` hovorí, že sa má jednať o správu typu `response` – má kód aspoň 32.

2.4.3 Druhá vrstva

Druhá vrstva popisuje logiku samotného protokolu. Robí to pomocou pravidiel, ktoré tentokrát reprezentujú stavy. Medzi týmito stavmi sú prechody.

Dokumentácia [13] popisuje pravidlo tejto úrovne nasledovne:

```
- rule:
    id: NAME
    note: DESCRIPTION
    query: L1_RULE
    query_params:
     - PARAMS
    type: RULE_TYPE
    success:
        code: |
            PYTHON_CODE
        state: NEXT_STATE
    fail:
        code: |
            PYTHON_CODE
        state: NEXT_STATE
```

Vysvetlenie:

- id – Identifikátor stavu, pomocou ktorého sa na stav odkazuje.
- note (nepovinné) – Popis pravidla.
- query – Odkaz na pravidlo prvej úrovne, ktoré tvorí podmienku pre prechod do iného stavu.
- query_params (nepovinné) – Definícia parametrov s ich hodnotami, ktoré majú byť predané do prvej úrovne.
- type (nepovinné) – Typ pravidla. Bude vysvetlené neskôr v sekcií2.4.3.2.

- success (nepovinné) – Vetva, ktorá má byť vykonaná v prípade, že aspoň rámec splňujúci podmienku bol nájdený.
- fail (nepovinné) – Vetva, ktorá má byť vykonaná v prípade, že žiaden rámec splňujúci podmienku nebol nájdený. Pozor, nemusí to vždy znamenať chybu. Napríklad ak je v podmienke pravidlo, ktoré vyhľadá všetky chybové rámce a nastane vetva fail, znamená to že nebol nájdený žiaden chybový rámec, teda žiadna chyba nenastala.
- final (nepovinné) – Vetva, ktorá má byť vykonaná po skončení iterácií pravidla typu `foreach`, čo bude vysvetlené neskôr v sekcií 2.4.3.2.
- code (nepovinné) – Blok Python kódu, ktorý má byť v danej vetve vykonaný.
- state (nepovinné) – Nasledujúci stav, do ktorého treba prejsť.

Príklad pravidla z protokolu CoAP:

```
- rule:
    id: response
    query: has_response
    success:
      state: confirmable_request
      code: |
        save("response", _fact["response"])
        debug("Response received")
        save("message", load("request")) # for is_confirmable rule
    fail:
      state: confirmable_request
      code: |
        debug("response not received")
```

Tento stav kontroluje podmienku, či k danému `requestu` existuje `response`. Toto vyhodnotenie sa vykoná pomocou pravidla prvej úrovne `has_response`, ktoré bolo už popísané. V prípade že odpoveď existuje, prejde sa do vetvy success, v prípade neexistencie do vetvy fail. Vidíme, že v oboch vetvách sa prechádza do rovnakého nasledujúceho stavu. To je korektný zápis, nie je nutné prechádzať do dvoch rôznych stavov, obzvlášť v prípade, že ten nasledujúci stav má zmysel v oboch vetvách.

Pozrime sa teraz na kód. V prípade úspechu bude z pravidla `has_response` prvej úrovne extrahovaná odpoveď – `response` a tá bude pomocou funkcie `save()` uložená do kontextu pre ďalšie spracovanie. Vypíše sa debugovacia hláška a následne sa zjednodušene povedané obsah premennej request uloží do premennej message. V prípade neúspechu sa iba vypíše debugovacia hláska.

Je však dôležité, že do kontextu sa neuložila odpoveď, pretože to bude neskôr kontrolované iným pravidlom.

Tieto pravidlá sa zapisujú do súboru `tree.yaml`. Napriek názvu *tree* – strom – sa nemusí jednať o strom. Je možné sa vrátiť už do navštíveného stavu. Rovnako je možné z dvoch vetiev prejsť do rovnakého stavu. Znamená to, že sa vlastne jedná o automat, ktorý má v každom stave najviac 2 možnosti na prechod do ďalšieho stavu. Výnimkou je pravidlo typu `foreach`, ktoré bude vysvetlené neskôr.

Počiatočným stavom je pravidlo s názvom `init`. V tomto stave diagnostika vždy začína a vetví sa ďalej. Špeciálnym stavom je aj `_done`, ktorý indikuje ukončenie vetvy. Tento stav je implicitný v prípade neuvedenia ďalšieho stavu.

2.4.3.1 Python kód

Najzaujímavejšou časťou diagnostiky je práve blok Python kódu. Narozdiel od samotných pravidiel, kde sme obmedzení ich syntaxou a podobne, máme v tomto bloku plnú kontrolu nad diagnostikou a sme schopní vykonávať ľubovoľnú verifikáciu dát imperatívne. Doporučeným postupom však je tieto bloky udržiavať čo najjednoduchšie a maximum logiky umiestniť do pravidiel. V istých prípadoch je ale nutné niektoré operácie, napríklad operácie s bitmi, vykonať v Pythone.

Ďalšou dôležitou úlohou bloku kódu je volanie funkcie `event()`, ktorá generuje udalosti na základe definície. Tieto udalosti sú, ako bolo spomenuté, hlavným výstupom programu.

Interakcia s faktmi Z bloku kódu je možné pristúpiť k rámcom ktoré boli extrahované pomocou pravidiel z prvej úrovne. K faktom je možné pristupovať pomocou slovníka `_fact`, alebo zoznamu slovníkov `_results`. Rozdiel spočíva v tom, že `_results` obsahuje všetky fakty, zatiaľ čo `_fact` iba prvý z nich. To je vačšinou ten, ktorý hľadáme. K slovníku pristupujeme klasicky `_fact["FACT_NAME"]`, kde `FACT_NAME` je názov faktu z pravidiel prvej úrovne. To umožňuje pracovať s extrahovanými dátami z PCAPu. Takouto extrakciou dostaneme rámec v podobe slovníka, kde kľúčmi sú názvy polí z wiresharku a hodnotami sú hodnoty týchto polí z rámca v podobe stringu.

Kontext Kontext umožňuje ukladanie a načítanie dát na základe kľúča. Existuje globálny a lokálny kontext. Lokálny kontext je unikátny pre každú vetvu diagnostiky, globálny je spoločný pre všetky. Z kontextu sú tiež načítavané parametre pre pravidlá prvej úrovne. Do kontextu sa pristupuje pomocou funkcií.

Funkcie Behové prostredie vystavuje bloku Python kódu viacero funkcií popísaných v dokumentácií [13]. Niektoré funkcie pracujú s kontextom:

- `save(key, value)` – Uloží hodnotu *value* pod kľúč *key* do lokálneho kontextu.
- `load(key)` – Vráti hodnotu pod kľúčom *key* z lokálneho kontextu.
- `exists(key)` – Zistí existenciu kľúča *key* v lokálnom kontexte.
- `save_global(key, value)` – Ako `save`, ale pre globálny kontext.
- `load_global(key)` – Ako `load`, ale pre globálny kontext.
- `exists_global(key)` – Ako `exists`, ale pre globálny kontext.

Ďalšie generujú výstup:

- `debug(text)` – Vypíše debugovaciu hlášku.
- `event(event_name, packet)` – Vygeneruje udalosť s identifikátorom `event_name` zo súboru `events.yaml`.

Iné pomáhajú pri práci s dátami:

- `addon(addon_name, parameter)` – Zavolá addon, čiže pomocný skript, s daným parametrom. Distance podporuje viac addonov, avšak ja som použil iba addon *decode*, ktorý preloží parameter na základe slovníka v súbore `codes.yaml`. Zmysel tohoto je napríklad preklad HTTP kódov na ich názvy, napríklad 404 → `Not Found`.
- `find_field(packet, field)` – Vráti prvú hodnotu položky `field` v rámci.
- `find_all_fields(packet, field)` – Vráti všetky hodnoty položky `field` v rámci.
- `field_exists(packet, field)` – Zistí existenciu položky `field` v rámci.

Interné premenné V niektorých prípadoch vyjadrovacia schopnosť diagnostického jazyka neumožňuje jednoducho zapísať niektoré kontroly. Je napríklad nutné podmienečne prejsť do iného stavu nie na základe existencie/neexistencie faktov, ale na základe podmienky v bloku kódu. Preto je možné priamo pristupovať do internej premennej automatu zvanej `_context`, ktorá udržuje stav automatu ale tiež aj už spomínaný kontext.

Ak chceme podmienene prejsť do nejakého stavu, bez ohľadu na to, čo je uvedené vo vetve success a fail, môžme požiť nasledujúce volanie `_context.set(["_internal", "state"], STATE_NAME)`, kde `STATE_NAME` je id stavu do ktorého chceme prejsť. Existuje aj špeciálny stav `_done`, ktorý ukončí vyhodnocovanie vetvy.

Tento prístup je však nutné používať s opatrnosťou a ak je to možné sa mu vyhnúť, pretože sa modifikuje vnútorný stav diagnostického procesu. [14]

Príklad Pozrime sa teraz na príklad pravidla, ktoré má komplikovanejší kód:

```
- rule:
    id: verify_response_code
    success:
      state: retransmission
      code: |
        debug("verifying response code")
        resp = load("response")
        path = load("path")
        method_name = load("method_name")

        code = int(find_field(resp, "coap.code"))
        # CoAP code is made of 3 bit class and 5 bit detail, eg. 4.04.
        # We need to parse it out manually
        # class is a reserved keyword
        clazz = (code >> 5) & 0b111
        detail = code & 0b00011111

        coap_code_string = f"{clazz}.{detail:02d}"
        code_description = addon("decode", coap_code_string)

        if code_description == "<unknown>":
          event("unknown_response_code", method=method_name,
                  path=path, code_string=coap_code_string)

        mapping = {2: "success", 4: "client_error", 5: "server_error"}
        event_type = mapping.get(clazz, "unknown_class")

        debug(f"{event_type} {method_name} to path "
               "{path} {coap_code_string} {code_description}")

        event(event_type, resp, method=method_name, path=path,
                code_string=coap_code_string,
                code_description=code_description)
        save("message", resp)
```

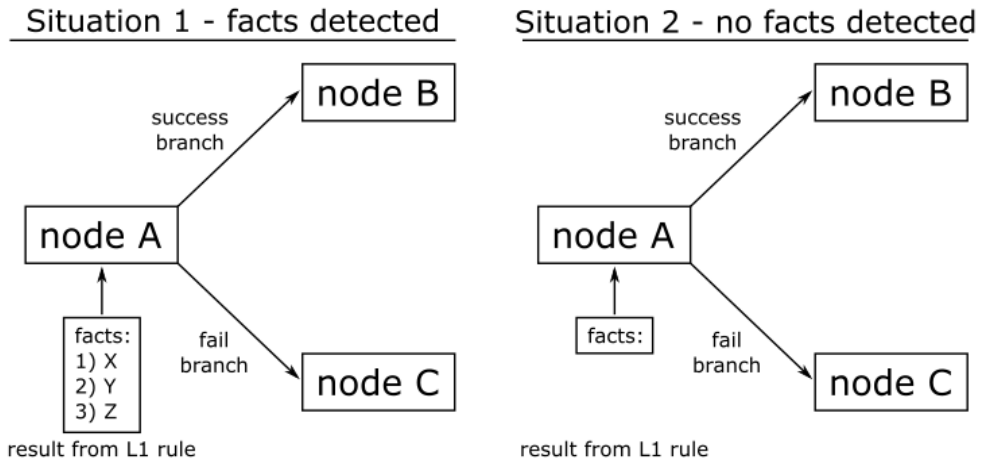
Vidíme, že pravidlo má len jedinú vetvu success, ktorá bude vykonaná vždy bezpodmienečne. Jedná sa o pomocné pravidlo, ktoré je vlastne iba logickým oddelením kódu, aby nevzniklo jedno veľké pravidlo s kvantom kódu, ale miesto toho viac malých. To sa aj ľahšie debuguje, pretože vieme, v ktorom stave nastala výnimka.

Vidíme tu načítanie premenných z kontextu, vidíme extrakciu hodnoty poľa `coap.code` a jej následné rozdelenie na class a code, ako je definované v štandarde. Následný preklad do stringovej podoby a dekódovanie kódu pomocou addonu decode na slovný popis. Ďalej vidíme vygenerovanie udalosti `unknown_response_code` v prípade neúspechu. Následne sa class preloží na typ odpovede a na základe toho sa vygeneruje príslušná udalosť. Na záver

sa pod kľúč `message` uloží rámec s odpoveďou a bude slúžiť ako parameter pre nasledujúce pravidlo.

2.4.3.2 Typ pravidiel

Popíšme si ešte spomínaný typ pravidiel. Existujú dva typy, `normal` pravidlo a `foreach` pravidlo. Rozdiel si vysvetlíme na obrázku 2.18 z dokumentácie [13]. Vidíme tu dve situácie. V prvej boli v stave A nájdené fakty, v druhej naopak žiaden fakt nebol nájdený.



Obr. 2.18: Popis správania pravidiel pri nájdených a nenájdených faktoch. Zdroj: [13]

Normálne pravidlo Toto je bežný stav, aký bo popísaný doteraz.

1. Situácia 1 – v tomto prípade bude vykonaná vetva *success* a premenná `_results` bude obsahovať fakty X, Y a Z. Premenná `_fact` bude obsahovať prvý fakt – X. Diagnostický proces bude pokračovať stavom B.

2. Situácia 2 – bude vykonaná vetva *fail*. Diagnostický proces bude pokračovať stavom B.

Foreach pravidlo Jedná sa o špeciálny typ pravidla, ktoré umožňuje „paralelne“ spustiť nezávislé vetvy diagnostického procesu pre každý nájdený fakt.

1. Situácia 1 – v tomto prípade sa diagnostika rozvetví na 3 vetvy *success*. Premenná `_fact` v prvej vetve X, v druhej Y a v tretej Z.

2. Situácia 2 – chovanie rovnaké ako pri normálnom pravidle.

Toto pravidlo je výhodné použiť napríklad v prípade, keď potrebujeme nájsť všetky dotazy a následne k nim odpovede. Pre hľadanie dotazov použijeme tento typ pravidla, čím si zaistíme, že sa diagnostika rozvetví na nezávislé vetvy pre každý jeden dotaz a následne sa už pokračuje nezávisle s hľadaním odpovede.

2.4.4 Udalosti

Udalosti, alebo tiež events sú zmyslom celej diagnostiky. Model udalostí je popísaný v dokumentácií [15] a [13]. Diagnostika má za cieľ vygenerovať zmysluplné eventy, ktoré odrážajú reále chovanie protokolu a závažnosť detekovaných udalostí.

Výstupom diagnostiky je séria JSON objektov reprezentujúcich eventy. S nimi môžu následne už pracovať iné nástroje, napríklad grafické rozhranie, ktoré eventy zobrazí administrátorovi v rozumnej podobe.

Event môže byť buť neparametrizovaný – neprijíma žiadne parametre, alebo prijíma rámec, ktorý ho vygeneroval a ktorý sa zobrazí v JSON výstupe, alebo môže obsahovať voliteľné parametre, ktoré sa tiež zobrazia vo výstupe. Zároveň sa dá týmito parametrami obohatiť hláška eventu – napríklad do nej môžme vložiť zdrojovú IP adresu.

Každý event má svoju závažnosť. Používajú sa 4 hodnoty: *error, warning, notice, information.*

- error – Indikuje, že bol nájdený nejaký problém, napríklad v konfigurácií alebo v komunikácií.
- warning – Indikuje udalosť, ktorá môže byť problematická, ale nemusí byť chybou. Príklad môže byť stratený paket. Môže sa jednať o chybne konfigurovanú sieť, výpadok zariadenia alebo jednoducho šum v médiu.
- notice – Povšimnutiahodná udalosť, napríklad pri riešení problémov.
- information – Vačšinou informuje o úspešnej operácií v protokole, napríklad otvorenie spojenia.

Eventy sa definujú v súbore `events.yaml`. Pozrime sa teraz na štruktúru zápisu eventu:

```
- event:
    name: ...
    description: ...
    severity: ...
    message: ...
    suggestion: ...
    fields:
      - name: ...
```

```
description: ...
source: ...
```

- name – Identifikátor eventu, pomocou ktorého sa naň odkazuje v kóde.
- description – Popis udalosti bez parametrov.
- message – Parametrizovaný popis udalosti.
- severity – Závažnosť udalosti.
- suggestion (nepovinné) – Informácia pre administrátora čo udalosť znamená, kde môže byť problém a ako ho odstrániť.
- fields (nepovinné) – Zoznam polí, ktoré majú byť predané eventu.
- name – Názov poľa v rámci, zvyčajne vo Wiresharkovom formáte.
- description (nepovinné) – Popis poľa pre účely dokumentácie.
- source (nepovinné) – Zdroj, z ktorého má byť čerpaná hodnota. Ak nie je špecifikovaný, použije sa hlavný rámec predaný funkcií event.

Pozrime sa teraz na príklad eventu z protokolu CoAP:

```
- event:
    name:         server_error
    description:  "CoAP server error response"
    severity:     error
    message:      "Detected CoAP server error response '{coap.code_string}
{coap.code_description}' in '{coap.method}' request to path '{coap.path}'"
    fields:
      - name: coap.method
        source: method
        description: CoAP request method
      - name: coap.path
        source: path
        description: CoAP request path
      - name: coap.code_string
        description: CoAP response code
        source: code_string
      - name: coap.code_description
        source: code_description
        description: CoAP response code description
```

Z `message` a `description` vidíme, že event reprezentuje prípad, kedy nastala chyba serveru. Severita je *error*, čo však je diskutabilné a závisí na kontexte, takže je to už na administrátorovi, aby to vyhodnotil. V `message` vidíme správu parametrizovanú hodnotami uvedenými pod `fields`.

Pozrime sa teraz na príklad eventu. Jedná sa o Client error z protokolu CoAP.

```
{
  "event-id": "client_error",
  "event-name": "CoAP client error response",
  "severity": "error",
  "description": "Detected CoAP client error response
'4.05 Method Not Allowed' in 'PUT' request to path '/'",
  "fields": {
    "frame.time_epoch": ["1375090935.086791000"],
    "frame.number": ["13"],
    "ip.version": ["6"],
    "ip.src": [],
    "ip.dst": [],
    "ip.proto": [],
    "ipv6.src": ["bbbb::3"],
    "ipv6.dst": ["bbbb::1"],
    "ipv6.nxt": ["17"],
    "udp.srcport": ["5683"],
    "udp.dstport": ["50250"],
    "udp.stream": ["6"],
    "tcp.srcport": [],
    "tcp.dstport": [],
    "tcp.stream": [],
    "coap.method": "PUT",
    "coap.path": "/",
    "coap.code_string": "4.05",
    "coap.code_description": "Method Not Allowed"
  },
  "protocol": "COAP",
  "parent-record-id": "<root>",
  "event-record-id": "ec1bafa5-b5c8-4e74-8623-c30197d8f167",
  "flow": {
    "protocol": "17",
    "source-port": "5683",
    "destination-port": "50250",
    "flow-counter": "6",
    "source-address": "bbbb::3",
    "destination-address": "bbbb::1",
    "string": "UDP@bbbb::3:5683-bbbb::1:50250"
  }
}
```


Vidíme, že JSON obsahuje položky pochádzajúce z definície eventu, placeholdery sú však nahradené hodnotami.

Ďalej vidíme `fields`, ktoré obsahujú hlavné položky z rámca ako sú IP adresy a porty. Na konci vidíme 4 položky protokolu CoAP, ktoré sa sem dostali zo sekcie `fields` v definícií eventu.

Následne vidíme položky, ktoré sú dôležité pre zaradenie eventu k iným eventom a napokon dáta o sieťovom toku.

2.4.5 Architektúra a implementácia

Diagnostika je implementovaná v jazyku Python 3. Prvým krokom je kompilácia súborov pravidiel protokolov. Tá prebieha spustením `python3 diagnostics.py build`. Tým sa všetky konfiguračné súbory skompilujú do Python kódu, ktorý je následne pri diagnostike daného protokolu spustený.

V tom spočíva sila Distance enginu – nie je nutné všetky pravidlá a stavy a prechody písať imperatívne v Pythone ale stačí ich deklarovať a Distance sa postará o ich preklad a dodržanie.

Takto vyzerá príklad vygenerovaného kódu:

```
packets1 = get_packets_from_index(index, "udp.srcport", params["request"]["udp.dstport"])
packets2 = get_packets_from_index(index, "udp.stream", params["request"]["udp.stream"])
packets3 = list(set(packets1) & set(packets2))
packets3.sort()
packets["response"] = packets3
for x in packets["response"]:
    packet["response"] = x
    if relation(make_list(find_field_values(index.packet(packet["response"]), "coap.type")), "!=",
                ["3"]) and (skip_assert_condition(params, ['request']) or ((relation(
            make_list(find_field_values(index.packet(packet["response"]), "coap.token_len")), "==",
            ["0"])) or (relation(
            find_field_values_in_list(make_list(get_param_value(params, "request")), "coap.token"), "==",
            make_list(find_field_values(index.packet(packet["response"]), "coap.token")))))) and (
            skip_assert_condition(params, ['request']) or relation(
            make_list(find_field_values(index.packet(packet["response"]), "frame.number")), ">",
            find_field_values_in_list(make_list(get_param_value(params, "request")),
                                      "frame.number"))) and relation(
            make_list(find_field_values(index.packet(packet["response"]), "coap.code")), ">=", ["32"]):
        result.append({"response": index.packet(packet["response"])})
```

Kód je pomerne nečitateľný, pretože je vygenerovaný kompilátorom. To však nie je chyba, pretože ho nikto iný než Python interpreter čítať nebude. My sa naň ale pre zaujímavosť trochu pozrieme.

Na začiatku vidíme výber rámcov spĺňajúcich prvú podmienku, následne druhú a potom vidíme ich prienik do jedného zoznamu.

Následne sú nad týmto zoznamom pomocou for cyklu vyhodnotené asercie. Vo výsledku sa objavia iba rámce, ich ktoré spĺňajú.

Z kódu vidíme, že vyhodnotenie podmienok vo faktoch je efektívnejšie (funkcia `get_packets_from_index()` je jednoduchý výber zo slovníka), než vyhodnotenie asercií, ktoré sa vyhodnocujú v cykle pre každý rámec. Preto je vhodné rámce čo najviac prefiltrovať vo faktoch až následne použiť asercie. Nevýhodou faktov je, že efektívne sú len jednoduché podmienky, ktoré sa dajú dekomponovať na vyhľadávanie v slovníku. Ak použijeme komplikovanejšiu podmienku, napríklad $<$ alebo $>$, vyhľadávanie degraduje na for cyklus. Preto je vhodné do faktov písať podmienky, ktoré je možné vyhodnotiť indexáciou. Komplikovanejšie podmienky treba dať do medzi asercie. [14]

Celkovo diagnostika pracuje tak, že pomocou programu `tshark` (patrí do rodiny Wiresharku) načítava PCAP a rámce prekladá do podoby Python slovníkov, s ktorými sa následne pracuje v diagnostických pravidlách. Takáto architektúra odľahčuje Distance od nutnosti parsovať rámce, pretože o to sa postará tshark. Ďalšia výhoda je, že tshark používa rovnaké pomenovanie polí v rámcoch ako Wireshark. To umožňuje v pravidlách písať filtre v podobnej

syntaxi, ako má Wireshark. O operátory ako `&&` alebo `==` sa však musí distance postarať sám a implementovať ich logiku.

2.4.6 Spustenie diagnostiky a príklad behu

Spusteniu predchádza kompilácia príkazom ukázaným v sekcií 2.4.5. Následne je možné spustiť diagnostiku, čo je ukázané v sekcií 3.2.3. Význam prepínačov je podľa dokumentácie [11] nasledujúci:

- `diag` – spustenie diagnostiky
- `-d` – debug výstupy
- `-p` – protokol, ktorý treba analyzovať
- `-r` – adresár s pravidlami
- `-i` – vstupný súbor
- `-t` – cesta k `tshark` spustiteľnému súboru

2.4.7 Generovanie grafickej podoby pravidiel

Distance obsahuje skript `graph_plotter.py`. Tento skript umožňuje z vytvorených pravidiel vytvoriť ich grafickú reprezentáciu. Príklad vygenerovaného grafu je napríklad obrázok 3.2, ktorý si teraz vysvetlíme.

Na začiatku vidíme pravidlo resp. stav `init`, kde diagnostika začína. Jedná sa o typ `foreach`, takže sa automat v tomto stave rozvetví pre každý rámec splňujúci podmienku (query) `is_goose`, ktorá je pravidlom prvej úrovne — je definovaná v súbore `facts.yaml`. Vetva fail, teda ak nebol nájdený žiadny rámec splňujúci podmienku, prechádza do `_done` a generuje event `no_goose`.

Vidíme, že vetva `success` má pod sebou napísané `frame`. To znamená, že sa do kontextu uložila nejaká hodnota pod kľúčom frame, v tomto prípade je to daný rámec. Nasleduje stav `unicast`, ktorý používa pravidlo prvej úrovne `is_unicast` a vidíme že toto pravidlo prijíma parameter `frame`. V prípade splnenia podmienky sa prejde do `check_state_change` a vygeneruje sa event `unicast` čo vidíme na prechodovej hrane. V prípade nesplnenia sa prejde na stav `broadcast` bez žiadneho eventu.

V stave `broadcast` sa deje analogický proces, v prípade neúspechu sa prejde do `multicast_group` kde sa kontroluje podmienka `is_standard_multicast_group`. V oboch prípadoch diagnostika prejde do stavu `check_state_change`, ibaže v prípade nesplnenia podmienky sa vygeneruje event `non_standard_multicast`.

Na grafe je stav `check_state_change` veľmi zvláštny. Má iba prechod do koncového stavu, ale žiadnu podmienku, avšak názov napovedá že sa v ňom niečo kontroluje. Intuícia nás neklame, naozaj toto pravidlo vykonáva kontroly,

a generuje 3 rôzne eventy ako vidíme pod prechodom `success`. Avšak toto všetko sa vykonáva na úrovni Python kódu, takže na úrovni grafu to nevidíme a skript to nie je schopný detailnejšie zobraziť.

2.4.8 Zhrnutie

Distance je silný nástroj pre popis diagnostických pravidiel a následnú diagnostiku protokolov. Je schopný z PCAPu extrahovať potrebné fakty spojené s činnosťou protokolu. Jeho nevýhodou však je že potrebuje mať dopredu prístupný celý PCAP. To znamená, že nie je vhodný na realtime analýzu sieťového toku, čo znamená, že o zistených chybách alebo útokoch sa užívateľ dozvie až po zachytení a analyzovaní celého toku.

Kapitola 3

Realizácia

V tejto kapitole sa pozrieme ako som naštudované špecifikácie protokolov pretransformoval do diagnostického jazyka.

3.1 Tvorba pravidiel

Ideálny spôsob tvorby pravidiel je popísaný v [16]. V skratke je postup nasledovný:

1. Zozbierať dostatok informácií o protokole a identifikovať možné chybové stavy.
2. Namodelovať jednoduchý rozhodovací graf.
3. Prepísať graf do pravidiel diagnostického enginu.
4. Otestovať zmeny.
5. Opakovať proces, kým nie je dosiahnutý požadovaný stav.

Bohužiaľ, tento spôsob je optimalizovaný pre človeka, ktorý je zoznámený s diagnostickým jazykom a už ho niekedy používal. Preto osobne doporučujem pre menej skúseného človeka, ktorý sa ešte nie je do hĺbky zoznámený s diagnostickým jazykom aby zvolil variantu, kde bod 2. a 3. zjednotí dokopy pre urýchlenie celého procesu. Následné chyby odporúčam doladiť vo viac iteráciách implementácie a testovania. Problém totiž je, že nie je triviálne preložiť ľubovoľný graf modelujúci chovanie protokolu do diagnostických pravidiel bez hĺbkových znalostí jazyka pravidiel. Človeku, ktorý však už tieto znalosti má, nerobí problém správnym spôsobom jednak namodelovať graf, aby zodpovedal vyjadrovacím schopnostiam jazyka a následne ho do tohoto jazyka prepísať.

Z toho vyplýva, že popísaný spôsob je možné aplikovať na ľubovoľný protokol. Podmienkou však je hĺbkové porozumenie fungovania diagnostiky.

3.1.1 Chybové stavy

Štúdium protokolov mi umožnilo identifikovať chybové stavy a bezpečnostné hrozby. V implementácií konfigurácie pre diagnostický engine sa tieto stavy mapujú na events. Preto som pre takéto stavy vytvoril definíciu eventu a ten som následne zaradil na správne miesto v grafe. Bohužiaľ, nie všetky bezpečnostné hrozby spomenuté v sekcií 2.3 je schopný jazyk priamočiaro popísať. Navyše sú niektoré bezpečnostné hrozby, ako napríklad odopočúvanie, nedetekovateľné.

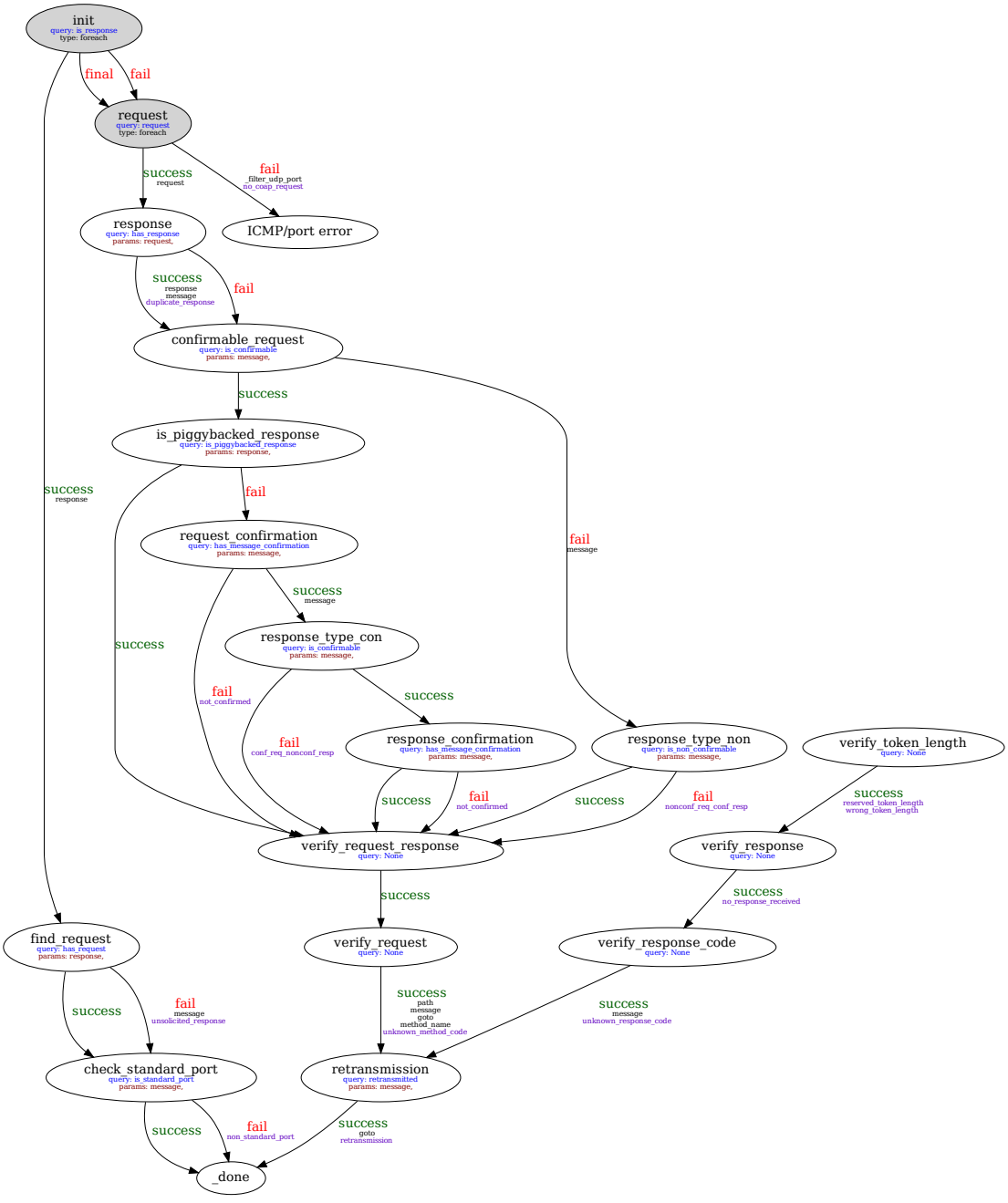
3.1.2 Optimalizácia

V procese implementácie som sa snažil pravidlá aj výkonovo optimalizovať. Následne som meral aké veľké zrýchlenie to prinieslo. Najväčším optimalizačným prínosom sa ukázal byť presun indexovateľných filtrov z `asserts` do `facts`. Napríklad pri protokole MMS to na 500 rámcovom PCAP súbore znížilo čas z cca $2,4s$ na cca $1,4s$.

3.1.3 CoAP

Diagnostický graf (obrázok 3.1) som koncipoval tak, že na začiatku sa hľadajú všetky CoAP odpovede pomocou `foreach` pravidla a následne sa ku každému hľadá dotaz, ktorý odpoveď spôsobil. Ak sa nenájde dotaz, znamená to že sa môže jednať o DoS alebo spoofing IP adresy a vygeneruje sa event `unsolicited_resp`. Následne sa v prípade úspechu aj neúspechu prejde so stavu `check_standard_port`, ktorý overí či sa jedná o komunikáciu na štandardnom CoAP porte, alebo na nejakom inom (v tom prípade sa vygeneruje event), čo môže naznačovať že ide o DoS pomocou iného UDP protokolu. Vetva následne končí.

Po skončení pravidla `init` sa prejde na pravidlo `request` (a aj v prípade že neexistuje žiadna odpoveď) ktoré je tiež typu `foreach` a tentokrát sa hľadajú všetky dotazy a následne k nim odpovede. Rozdiel je v tom, že toto pravidlo nie je schopné nájsť nevyžiadané odpovede a predchádzajúce zasa nenájde dotazy bez odpovedí. Ak sa v tomto prípade nenájde žiaden dotaz, diagnostika prechádza do protokolu ICMP, do stavu `protocol_error`. To je ďalšia schopnosť enginu, prejsť do iného protokolu a v ňom pokračovať v diagnostike. V tomto prípade sa vygeneruje event o nenájdení CoAP dotazu a skúma sa, či nebola poslaná nejaká ICMP správa, ktorá by naznačovala chybu. Toto pravidlo vykonáva ešte jednu dôležitú vec v kóde, a to je deduplikácia opakovane zaslaných (retransmitted) správ. Bohužiaľ, verzia tsharku, ktorú distance používa ešte nepodporuje pole `coap.retransmitted`, takže je nutné tieto správy deduplikovať ručne. Tu je fragment kódu z pravidla, ktorý deduplikáciu zabezpečuje:



Obr. 3.1: Vizualizácia grafu protokolu CoAP pomocou nástroja graph_plotter.py

```
key = [find_field(req, "coap.mid"), find_field(req, "udp.stream")]
if exists_global(key):
    debug("Already seen this message,"
          " not continuing (the message was retransmitted)")
    _context.set(["_internal", "state"], "_done")
else:
    save_global(key, True)
```

Najprv sa vytvorí kľuč, ktorý pozostáva z Message ID správy a UDP streamu. Následne sa kontroluje, či kľúč existuje v globálnom kontexte. Ak nie, uloží sa do kontextu. Znamená to, že daná správa je prvá nájdená. Ak ale kľúč existuje, znamená to že táto správa (jej duplikát) už bol videný a v tom prípade sa táto vetva ukončí. Na ukončenie sa použije konštrukt popísaný v odstavci Interné premenné.

To či k dotazu existuje odpoveď skúma pravidlo `response`. V prípade úspešného nájdenia odpovede sa táto odpoveď uloží do kontextu. Okrem toho však pravidlo v sebe nesie aj kód, ktorý zisťuje, či nebola obdržaná viacnásobná odpoveď, čo by mohlo znamenať pokus o spoofing, napríklad jeden z bodov popísaný v odstavci Spoofing IP adresy. V tomto prípade sa vygeneruje event.

Následne sa v oboch prípadoch pokračuje v stave `confirmable_request`, ktorý skúma či sa jedná o request typu CON alebo NON. Ak sa jedná o NON správu, pokračuje sa vetvou fail a zisťuje sa, či sa jedná o response typu NON alebo CON. Ak sa jedná o CON odpoveď, jedná sa o neštandardné chovanie, ale nie nedovolené, preto sa vygeneruje event, avšak iba so severitou `warning`. Ak sa jednalo o CON správu, pokračuje sa zisťovaním, či odpoveď prišla separátne alebo piggybacked. Ak sa jedná o piggybacked, pokračuje sa v stave `verify_request_response`, o ktorom sa dozvieme viac neskôr. Ak sa jedná o separátnu odpoveď, pokračuje sa hľadaním ACK správy, ako je definované v štandarde a vidíme to na obrázku 2.13. Ak sa ACK správa nenájde, vygeneruje sa event, ktorý indikuje stratu správy, čo môže byť spôsobené výpadkom v sieti a pokračuje sa spomínaným `verify_request_response`. V prípade úspechu sa kontroluje či je odpoveď typu CON alebo NON. Ak je NON, je to neštandardné, ale štandard to nezakazuje, takže sa vygeneruje `warning` event. Ak sa jedná o štandardnú CON odpoveď, kontroluje sa, či na odpoveď prišlo aj potvrdenie, ktoré ak neprišlo vygeneruje sa event. V oboch prípadoch sa následne pokračuje pravidlom `verify_request_response`.

Pozrime sa teraz na stav `verify_request_response`. Jedná sa o prázdne pravidlo, ktoré nič nekontroluje ani nerobí. Jediný jeho účel je iniciácia sekvencie nasledujúcich stavov, ktoré sú spoločné pre veľa vetiev. Na túto sekvenciu sa teraz pozrieme hlbšie. Je zložená z dvoch častí, kontroly requestu a kontroly response. Zaujímavosťou je aj to, že všetky stavy majú iba vetvu success a okrem jedného nemajú žiadnu `query`.

Prvým pravidlom je `verify_request`, ktoré má za úlohu skontrolovať request. Kontrola je pomerne komplikovaná a v diagnostickom jazyku nie je

priamočiary spôsob ako ju vykonať. Preto sa kontrola vykonáva v Python kóde pravidla, na ktorý sa teraz pozrieme:

```
debug("verifying request method")
mapping = {1: "GET", 2: "POST", 3: "PUT", 4: "DELETE"}

req = load("request")
path_list = find_all_fields(req, "coap.opt.uri_path") # find all path parts
path = "/" + "/".join(path_list) # merge the path parts into full path
save("path", path)

method_code = int(find_field(req, "coap.code"))

try:
  method_name = mapping[method_code]
except KeyError: # more efficient than if
  method_name = "UNKNOWN_METHOD"
  event("unknown_method_code", req)

save("method_name", method_name)

save("message", req)
save("goto", "verify_token_length")
```

V kóde vidíme použitie funkcie `find_all_fields`, ktorou sa zo správy vyextrahuje zoznam častí path z URI. Tie sa následne spoja a uložia do kontextu pre ďalšie pravidlá.

Ďalej vidíme extrakciu CoAP kód zo správy pomocou `find_field`, ktorý sa následne v try-except bloku prekladá pomocou slovníka na názov metódy. Ak sa preklad nepodarí, znamená to, že sa nejedná o jednu zo 4 štandardných metód. To nie je chyba, môže sa jednať o užívateľsky definovanú metódu, avšak nie je to úplne bežné a preto sa vygeneruje `warning` event.

Následne sa názov metódy uloží do kontextu. Ďalej sa do kontextu uloží aj request pod kľúčom „message“, čo bude parameter pre nasledujúci stav. Posledné volanie `save` ukladá pod kľúč `goto` hodnotu `verify_token_length`. Jedná sa o konštrukt, ktorý detailne pochopíme v kóde pravidla `retransmission`. Dôvod je ten, že stav `retransmission` sa v skutočnosti v sekvencií volá dvakrát. Prvý krát sa kontroluje, či bol znovuposlaný request, v druhom response. Potreboval som dosiahnuť, aby nebol stav duplikovaný aj keď vykonáva dva krát rovnaký kód. Jediný rozdiel je z ktorého stavu sa do stavu `retransmission` prišlo. Potrebujeme teda zaistiť, aby keď diagnostika príde do `retransmission` z `verify_request` , následne pokračovala stavom `verify_token_length`. Keď však do stavu `retransmission` príde z `verify_response`, chceme aby sa vetva ukončila. Na to slúži nasledujúci kus kódu v pravidle `retransmission`:

```
state = load("goto")
save("goto", "_done") # clear it
_context.set(["_internal", "state"], state)
```

Najprv sa z kontextu načíta hodnota kľúča `goto`. Následne sa „premaže“ tým, že sa do kontextu uloží `_done`. Nasleduje už známy konštrukt, ktorý nastaví nasledujúci stav na hodnotu načítanú z kontextu. Tým sa zaistí, že akú hodnotu nám predchádzajúci stav uloží pod kľúč `goto`, do takého nasledujúceho stavu sa bude pokračovať. Tento spôsob prechodu diagnostickým grafom vychádza z obmedzení súčasnej verzie jazyka.

Stav `retransmission` má v sebe okrem spomenutého kódu ešte kód na kontrolu koľko krát bola obdržaná správa s daným Message ID. Ak viac než raz, vygeneruje sa event, ktorý indikuje, že správa alebo potvrdenie sa stratilo a musela byť znovu poslaná.

Nasleduje stav `verify_token_length`, ktorý kontroluje, či dĺžka tokenu v správe zodpovedá deklarovanje dĺžke v správe a či nie je v rozsahu dĺžok, ktoré sú rezervované podľa špecifikácie. V jednom či druhom prípade sa vygeneruje event, pretože sa môže jednať o útok, napríklad pokus o zneužitie zraniteľnosti v parseri správ, čo môže viesť na pád aplikácie a následný DoS.

Nasleduje stav `verify_response`, ktorého hlavnou úlohou je overiť, či existuje odpoveď. Ak áno, pokračuje sa v kontrole odpovede, ak nie, vetva končí. To je dosiahnuté nasledujúcim kódom:

```
debug("verifying response existence")
if not exists("response"):
  event("no_response_received", load("request"),
          method=load("method_name"), path=load("path"))
  # Do not continue, because response does not exist
  _context.set(["_internal", "state"], "_done")
```

Najprv sa v kontexte hľadá kľúč `response`, ktorý bol uložený do kontextu v stave `response` v prípade existencie odpovede. Ak odpoveď neexistuje, vygeneruje sa udalosť. Neexistencia odpovede je chyba v prípade použitia 4 štandardných metód, ktoré podľa špecifikácie vyžadujú odpoveď. Nemusí to byť ale chyba pri vlastných metódach.

Môžme si klásť otázku, prečo sme udalosť už nevygenerovali už v stave `response`. Odpoveď je, že vtedy sme ešte nemali rozparsovaný request, konkrétne metódu a cestu, ktoré ako vidíme v kóde, sú predané funkcií event a podávajú viac informácií užívateľovi.

Nasleduje už známy konštrukt, ktorý v prípade neexistencie odpovede ukončí vetvu.

Predposledným pravidlom je `verify_response_code`, ktoré už bolo ukázané v odstavci Príklad v sekcií 2.4.3.1. Jeho podstatou je overiť korektnosť response kódu a preložiť ho na textovú podobu. Na začiatku vidíme načítanie odpovede a predspracovaných dát z kontextu. Následne sa z odpovede vyextrahuje položka `coap.code`, ktorá pri odpovedi zodpovedá response kódu

a rozparsuje sa na triedu (class) a detail ako je definované v špecifikácií. Následne sa z nich vytvorí stringová reprezentácia, ktorá sa pomocou addonu `decode` preloží na textový popis kódu. V prípade neznámeho kódu sa vygeneruje event o neznámom kóde so severitou `warning`, pretože sa nemusí jednať len o chybu ale aj o vlastný response kód. Nasleduje premapovanie triedy na identifikátor eventu, ktorý bude na základe neho vygenerovaný. 2 znamená úspech, event bude mať severitu `information`. 4 a 5 znamenajú chybu, event bude mať severitu `error`. Zvlášť event je určený pre prípad, kedy je trieda iná, než 2, 4 či 5. Iné hodnoty sú totiž rezervované, takže sa jedná o chybu. Posledný riadok uloží odpoveď do kontextu ako parameter pre nasledujúci stav – `retransmission`. V tomto prípade nie je nutné mu explicitne predať nasledujúci stav pomocou `goto`, pretože implicitne je nasledujúci stav `_done`. Posledný stav teda ešte skontroluje, či nebola odpoveď obdržaná viackrát, podobne ako s dotazom. Následne vetva diagnostiky končí.

Zhrnutie V protokole CoAP sa mi podarilo pokryť drvivú väčšinu stavov, ktoré môžu nastať. Do budúcna by bolo vhodné ešte pridať diagnostiku niektorých ďalších funkcií protokolu, ako je proxying a caching, podpora multicastu a veľmi vhodne by bolo implementovať aj diagnostiku ďalšieho protokolu — DTLS — ktorý je v spojení s CoAP často používaný.

Ďalší krok do budúcna by mohla byť detailnejšia diagnostika útokov na protokol, ktorá by presnejšie povedala užívateľovi o aký útok sa jedná. Znamenalo by to rozdeliť niektoré eventy na viac detailnejších eventov a tiež pridať nejaké pravidlá a Python kód.

Rovnako by bolo vhodné pridať viac informatívnych eventov o úspešných stavoch ktoré v protokole nastali.

3.1.4 GOOSE

Protokol GOOSE je v základe pomerne jednoduchý. Jedná sa iba o jednosmernú komunikáciu publisherov, ktorí posielajú správy a subscriberov, ktorý správy počúvajú.

Diagnostický graf (obrázok 3.2) som koncipoval tak, že hlavnou časťou je detekcia, či sa jedná u unicast, multicast či broadcast. Následne sa kontroluje sekvenčné číslo GOOSE správy.

Diagnostika začína stavom `init`, ktorý je typu `foreach` a ktorého úlohou je vyfiltrovať všetky GOOSE správy a pre každú z nich sa vykoná zvyšok grafu. Tento stav si vždy daný rámec uloží do kontextu a predá ho ďalším stavom, ktoré na ňom vykonávajú kontroly. Ak žiadna GOOSE správa nebola nájdená, vygeneruje sa event a diagnostika končí. Pre každú správu sa teda vykoná nasledovné. Stav `unicast` kontroluje, či je správa posielaná unicastom. Stav `multicast` kontroluje, či je správa posielaná multicastom. Podľa štandardu by totiž GOOSE správy mali byť posielané mutlicastom. Ak je teda jedna z týchto

podmienok splnená, vygeneruje sa event informujúci že sa jedná o broadcast alebo unicast.

Ak žiadna podmienka splnená nebola, prechádzame do stavu `multicast_group`, ktorý kontroluje, či sú správy posielané podľa štandardom definovaného multicastového prefixu `01:0c:cd:01`. Ak správa nepoužíva tento prefix, vygeneruje sa event.

Následne všetky stavy prejdu do stavu `check_state_change`, ktorého úlohou je generovať eventy informujúce užívateľa o priebehu „toku“ správ. Tieto „toky“ sú od seba odlíšené kľúčom, ktorý pozostáva zo zdrojovej a cieľovej adresy správ. Tento stav nemá žiadnu podmienku a má len vetvu success, ktorá prechádza do koncového stavu. Všetka logika je totiž umiestnená v Python kóde.

```
def new_state_detected():
  save_global(key, new_state)
  sqNum = int(find_field(frame, "goose.sqNum"))
  if sqNum != 0:
    event("sequence_not_zero", frame)

frame = load("frame")
key = [find_field(frame, "eth.src"), find_field(frame, "eth.dst")]
new_state = find_field(frame, "goose.stNum")
if not exists_global(key):
  debug(f"first frame for stream {key}")
  event("stream_start", frame)
  new_state_detected()
else:
  state = load_global(key)
  if state == new_state:
    debug(f"no state change for stream {key}")
  else:
    debug(f"state changed for stream {key}")
    event("stream_change", frame)
    new_state_detected()
```

Na začiatku je definovaná -funkcia `new_state_detected`, na ktorej význam sa pozrieme neskôr. Z kontextu sa načíta rámec. Z neho sa následne vyextrahuje zdrojová a cieľová Ethernetová adresa, a z týchto dvoch adries sa vytvorí kľúč. Z rámca sa tiež vyextrahuje číslo stavu GOOSE správy (`stNum`). Následne sa kontroluje existencia kľúča v globálom kontexte.

Ak kľúč neexistuje, znamená to že sa jedná o nový „tok“ a vygeneruje sa informatívny event o začatí nového toku. Zároveň je zavolaná spomínaná funkcia, ktorá do globálneho kontextu uloží číslo stavu GOOSE správy pod vytvoreným kľúčom. Okrem toho sa z rámca extrahuje aj sekvenčné číslo GOOSE správy, ktoré by pri zmene stavu či začiatku toku malo začínať od nuly.

Ak nezačína od nuly, môže to znamenať stratu rámcov v sieti, preto sa vygeneruje event so severitou `warning`.

Ak kľúč existuje, znamená to že sa jedná o už detekovaný tok a porovnáva sa jeho stavové číslo so stavovým číslo aktuálnej správy. Ak sa zhodujú, znamená to že nenastala zmena stavu. Ak sa nezhodujú, znamená to zmenu stavu, a preto sa vygeneruje informatívny event informujúci o zmene stavu. Následne sa opäť zavolá funkcia `new_state_detected`.

Potom kód stavu končí a prechádza sa do koncového stavu, čím končí vetva diagnostiky.

Zhrnutie Pre protokol GOOSE som diagnostikou pokryl hlavné aspekty protokolu. Po hlbšej štúdií štandardu IEC 61850 by bolo dobré zamerať sa aj na diagnostiku samotných posielaných hodnôt, teda aplikačných dát protokolu.

3.1.5 MMS

MMS je protokol, ktorý má pod sebou v nižších vrstvách pomerne veľkú sadu protokolov. Diagnostika sa však zameriava konkrétne na vrstvu protokolu MMS.

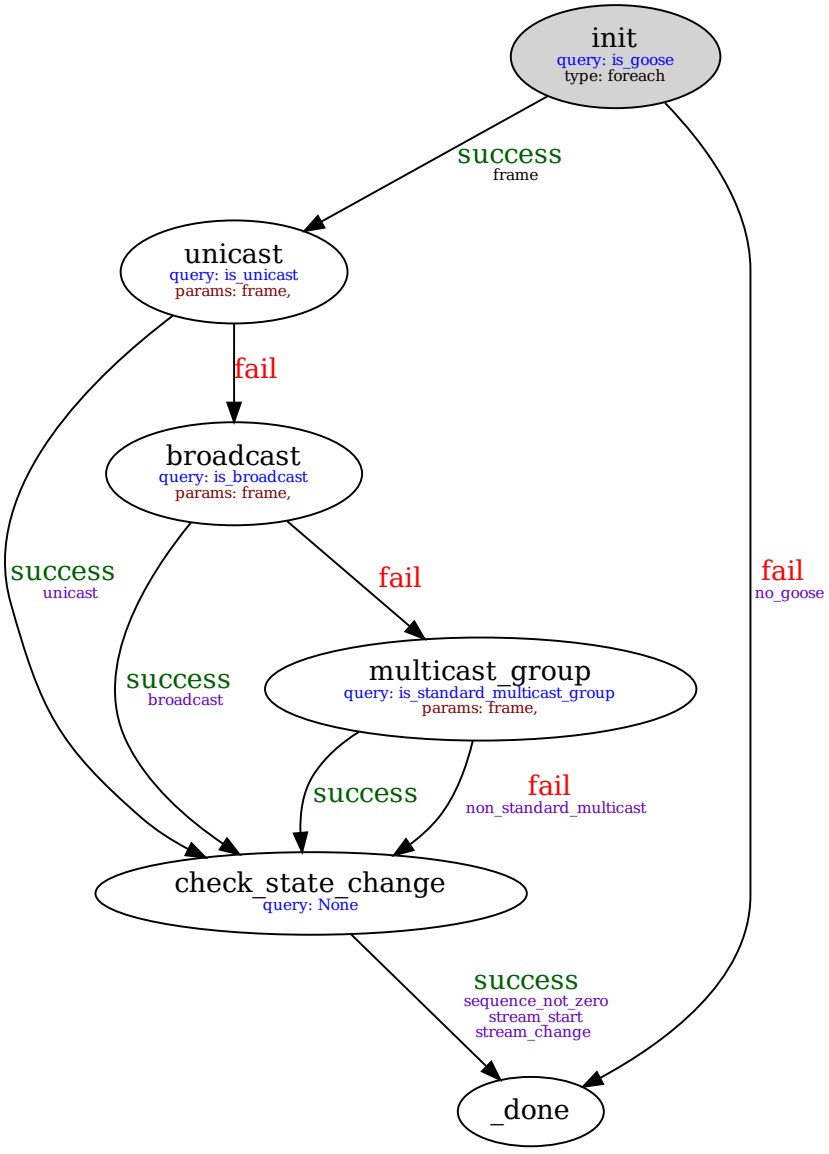
Diagnostický graf (obrázok 3.3) pozostáva z dvoch hlavných častí. Detekcia nového MMS spojenia a následné monitorovanie komunikácie až do uzavretia spojenia.

Diagnostika začína stavom `init`, ktorý je typu `foreach` a úlohou je detekcia nájsť všetky „MMS Initiate Request“ správy. Ak sa žiadna nenájde, vygeneruje sa event a pokračuje sa v protokole TCP stavom `connection refused`. Pre každú nájdenú správu sa vykoná nasledovné. Do kontextu sa uloží táto inicializačná správa a tiež identifikátor TCP streamu v ktorom bolo spojenie iniciované. Následne sa vygeneruje informatívny event o pokuse o naviazanie MMS spojenia a prechádza sa do stavu `init_response`.

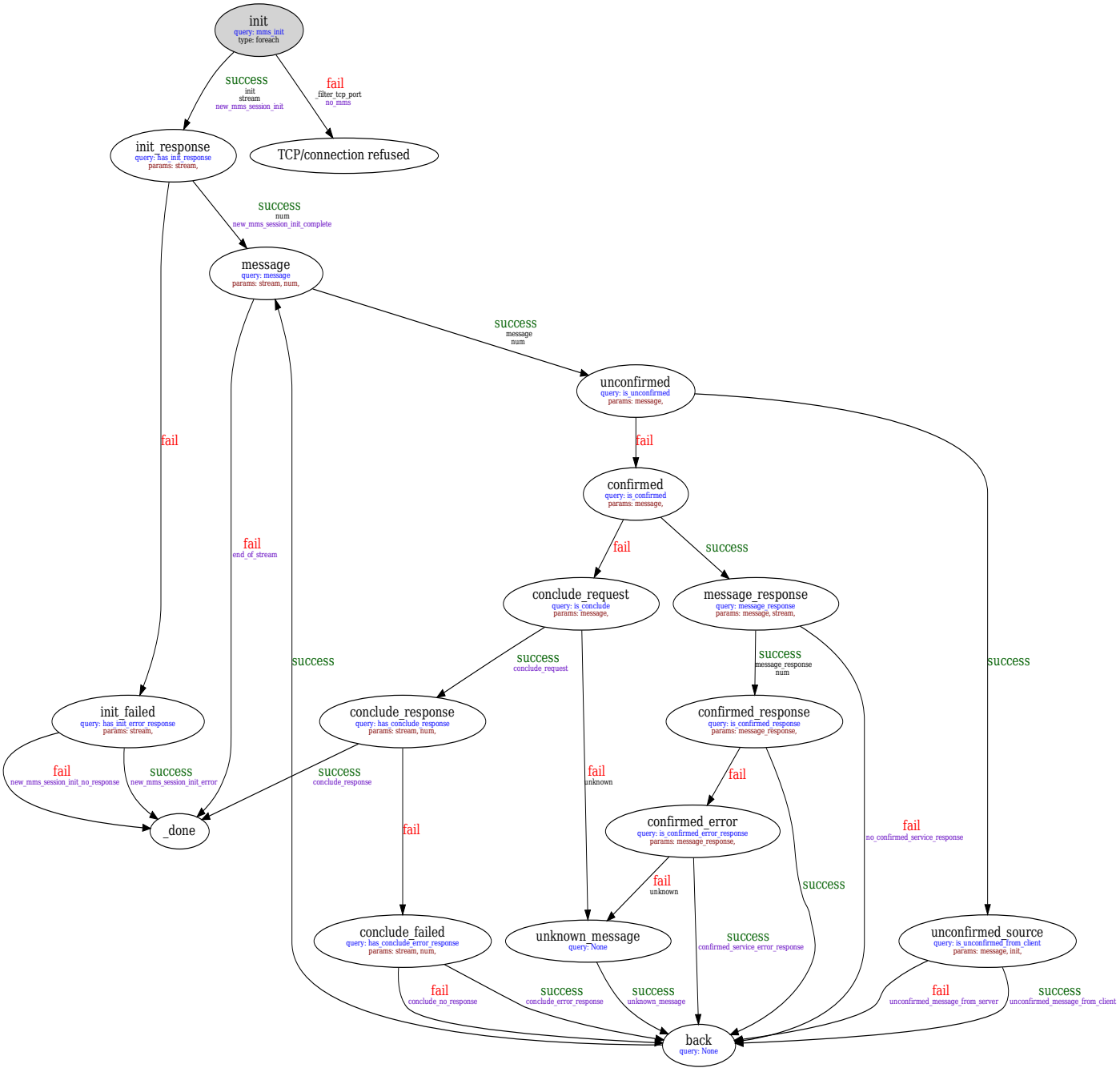
Jeho úlohou je nájsť k danej inicializácií odpoveď, to znamená potvrdenie servera, že MMS spojenie je vytvorené. Ak sa takáto odpoveď nenájde, prejdeme dostavu `init_failed`, ktorý overí dôvod, prečo neprišlo potvrdenie. Dôvody môžu byť dva, buď prišla chybová odpoveď alebo neprišlo nič. V oboch prípadoch je vygenerovaný event informujúci o danej chybe a diagnostika vetvy končí.

Ak však prišla úspešná odpoveď na iniciáciu, vygeneruje sa informatívny event o úspešnom nadviazaní spojenia. Do kontextu sa uloží poradové číslo rámca (`frame.number`).

Nasleduje stav `message`, ktorý je začiatkom cyklu opakujúcich sa stavov. Jeho úlohou je vyfiltrovať najbližšiu správu, ktorej poradové číslo je väčšie ako to uložené v kontexte. Následne túto správu uloží do kontextu pre nasledujúce pravidlá a tiež uloží nové poradové číslo rámca do kontextu – prepíše to predchádzajúce. Ak nebol nájdený žiaden rámec, znamená to chybu, že



Obr. 3.2: Vizualizácia grafu protokolu GOOSE pomocou nástroja graph_plotter.py



Obr. 3.3: Vizualizácia grafu protokolu MMS pomocou nástroja graph_plotter.py

spojenie bolo ukončené bez uzatváracej sekvencie vyžadovanej štandardom. V tom prípade je vygenerovaný event informujúci o tejto chybe a diagnostika končí.

V opačnom prípade sa prechádza na sekvenciu diagnostiky danej správy. Prvý je stav `unconfirmed`, ktorý zisťuje, či sa jedná o nepotvrdzovanú správu. Ak sa jedná o nepotvrdzovanú správu, kontroluje sa v stave `unconfirmed_source` či je poslaná od klientom alebo serverom. Nepotvrdzovaná správa má väčšinou za úlohu informovať klienta o nejakej udalosti či zmene ktorá nastala na serveri. Ak bola teda správa poslaná serverom, jedná sa o štandardné chovanie a je vygenerovaný informatívny event. Ak však správa bola poslaná klientom, nejedná sa obežné chovanie a je vygenerovaný event so severitou `warning`. V oboch prípadoch diagnostika prechádza do stavu `back`.

Stav `back`, je iba pomocný stav, ktorého jediný zmysel je prejsť naspäť do stavu `message`, čo sprehľadňuje diagnostický graf.

Vráťme sa k prípadu, keď sa nejedná o nepotvrdzovanú správu. Vtedy sa v stave `confirmed` vykonáva kontrola či sa jedná o potvrdzovanú správu. Ak sa nejedná o potvrdzovanú správu, prechádzame do stavu `conclude_request` a kontroluje sa, či sa jedná o pokus o uzavretie spojenia. Ak to nie je pokus o uzavretie spojenia, znamená to že ide o neznámu správu a prejdeme do stavu `unknown_message`, ktorý vygeneruje varovný event o tom, že bola obdržaná neznáma správa. Dôvodom môže byť aj to, že dáto správa ešte nie je v tomto diagnostickom grafe implementovaná.

Ak sa jednalo o pokus o uzavretie spojenia, vygeneruje sa informatívny event o tomto pokuse a následne prebieha v stave `conclude_response` kontrola, či protistrana poslala potvrdenie uzavretia. Ak áno, spojenie je úspešne uzavreté a diagnostika tejto vetvy reprezentujúcej jedno MMS spojenie končí. Vygeneruje sa informatívny event o úspešnom ukončení spojenia.

Ak nebolo od protistrany zaznamenané potvrdenie o ukončení spojenia, prebieha v stave `conclude_failed` zisťovanie, či protistrana poslala chybovú odpoveď, alebo neposlala nič. V oboch prípadoch sa vygeneruje príslušný chybový event. Podľa špecifikácie v prípade chyby pri uzatváraní zostáva spojenie uzavreté. Preto diagnostika prechádza do stavu `back`.

Vráťme sa k prípadu, že sa jednalo o potvrdzovanú správu. V tom prípade prehádzame do stavu `message_response`, ktorého úlohou je k správe nájsť jej odpoveď. Na to je použité pomerne komplikované nasledujúce pravidlo 1 úrovne:

```
- rule:
    id: message_response
    facts:
      - message_response: mms && stream == tcp.stream
&& mms.invokeID == message[mms.invokeID]
&& message[tcp.srcport] == tcp.dstport
```

```
params:
  - message
  - stream
```

Toto pravidlo pomocou facts filtruje všetky mms správy z daného TCP streamu, ktorých `invokeID` sa zhoduje s `invokeID` správy. Táto hodnota slúži práve na správne priradenie odpovede k správe. Okrem toho musí zdrojový port správy zodpovedať cieľovému portu odpovede, čo zaistí, že sa za odpoveď nebude považovať správa samotná.

V prípade, že nebola nájdená odpoveď na správu, je vygenerovaný event informujúci o tejto chybe a diagnostika prechádza to stavu `back`.

Ak sa odpoveď našla, uloží sa do kontextu ako parameter pre ďalšie pravidlo `confirmed_response`, ktoré overí, či sa jedná o úspešnú odpoveď alebo o chybovú odpoveď. V prípade úspechu diagnostika prechádza do stavu `back`.

Ak nejde o úspešnú odpoveď, kontroluje sa v stave `confirmed_error`, či sa jedná o chybovú odpoveď. Ak áno, vygeneruje sa event informujúci o tejto chybovej odpovedi a prejdeme do stavu `back`. Ak nie, prechádzame do už spomenutého stavu `unknown_message`.

Diagnostika sa teda rozvetví pre každé nové MMS spojenie a následne cyklicky prechádza správy daného spojenia až kým nepríde na koniec alebo k uzavretiu spojenia.

Zhrnutie Tento protokol sa ukázal byť priamočiaro mapovateľný na diagnostické pravidlá bez nutnosti písania komplikovaného python kódu. Všetka logika je ukrytá v stavoch, podmienkach a prechodoch, v kóde sa iba extrahujú položky zo správ, ukladajú sa dáta do kontextu a generujú sa eventy.

Toto je dobrý príklad použitia diagnostického jazyka. Bohužiaľ, nie každý protokol je takto dobre mapovateľný a často je vyžadované písanie Python kódu.

Rovnako ako pri protokole GOOSE, po hlbšej štúdií štandardu IEC 61850 by bolo dobré zamerať sa aj na diagnostiku samotných posielaných hodnôt, teda aplikačných dát protokolu.

3.2 Testovanie

Testovanie vytvorených pravidiel bolo nutnou súčasťou celého procesu. Vývoj prebiehal tak, že som vždy pridal nový stav alebo som stav upravil a následne som ho otestoval. Bohužiaľ, v niektorých prípadoch nie je úplne časovo jednoduché vytvoriť dáta na otestovanie daného stavu, preto kód nemá 100% pokrytie. Oblasť testovania, obzvlášť automatizovaného vidím ako smer, ktorým možno ešte túto prácu rozšíriť.

Za súčasť testovania môžme považovať aj revíziu vytvorených pravidiel spoluautorom Distance, Ing. Martinom Holkovičom, ktorá prebehla spolu s následnou videokonferenciou a doplňujúcimi otázkami v niekoľkých iteráciách.

3.2.1 Dáta

Používal som dva typy testovacích dát (PCAP súborov) — testovacie a ostré. Za testovacie som považujem tie, ktoré boli špeciálne upravené pre testovanie daného protokolu. Za ostré zasa považujem dáta z nejakej produkčnej siete, to znamená reálnu komunikáciu zariadení.

Hlavným zdrojom ostrých, ale aj testovacích dát bol môj vedúci, ktorý ich zohnal vrámci združenia CESNET. Ostré dáta som použil tak ako sú, ale tie testovacie som si upravoval, aby som pokryl maximum stavov. Na úpravu PCAP súborov som použil nástroj *Ostinato*, ktorý umožňuje zachytávanie, prehrávanie a, najdôležitejšie, editovanie rámcov z PCAP súboru.

Ďalším spôsobom, ako by bolo možné generovať testovacie dáta, by bolo napísanie skriptu, ktorý by simuloval klienta a server. Tam je však problém, že sa napríklad ťažko simuluje výpadok siete a teda vynucuje proces spoľahlivého zasielania správ (retransmission). Preto som zvolil radšej prvú variantu, s manuálnou tvorbou rámcov.

Po tom, ako som pravidlá overil testovacími dátami, prešiel som na kontrolu ostrými dátami. Tá prebiehala tak, že som sa najskôr pozrel pomocou Wiresharku na produkčný PCAP a skúmal som konkrétny protokol. Snažil som sa pochopiť, čo sa v ňom deje a „ručne“ odhaliť stavy, ktoré by mali byť diagnostikou detekované. Následne som spustil diagnostiku a skontroloval výstup, či sa zhoduje s predpokladmi.

3.2.2 Výsledky

Testovanie mi pomohlo odhaliť mnohé problémy v diagnostických pravidlách, ktoré som na základe zistení opravil. Príkladom môže by pravidlo `response_type_con`, ktoré miesto response kontrolovalo request, čo bola chyba.

Tiež sa pri testovaní na veľkých súboroch ukázalo, že neindexované vyhľadávanie môže byť pomalé, ako bolo popísané v sekcií 2.4.5. Keď som však filter prepísal do optimálnej podoby, teda dal som indexovateľnú časť filtra do časti `facts`, vyhľadávanie sa urýchlilo.

Spôsob testovania bol však pomerne zdĺhavý a pracný. Pri každej zmene bolo treba ručne spustiť diagnostiku s vybraným PCAPom a následne ručne overiť správnosť výstupu. Preto by bolo vhodné do budúcna vytvoriť možnosť tvoriť automatizované testy pre pravidlá, ktoré by mohli byť tvorcom pravidiel spúšťané pri vývoji. Tým by sa urýchlil vývoj a minimalizovala sa možnosť prehliadnutia nejakej chyby.

3.2.3 Príklady výstupu testov

Nasleduje ukážka výstupu z testu pre protokol CoAP, ktorý mal testovať prípad, keď na CON správu nepríde potvrdenie. Riadky začínajúce `COAP:` sú zobrazené, pretože bol pri behu zapnutý prepínač `-d`, teda debugovací výstup.

```
$ python3 diagnostics.py diag -d -p COAP -r protocols/ -i separate_no_ack.pcap -t /usr/bin/tshark

COAP: Found response, token=07:90
COAP: Found request response pair, token=07:90
COAP: Message with standard CoAP port.
COAP: request token=07:90 mid=59675
COAP: Response received
COAP: Is confirmable request 59675
COAP: Separate response - not piggybacked
COAP: Request confirmation received
COAP: Confirmable response
COAP: Response not confirmed
{
        "event-id": "not_confirmed",
  "event-name": "CoAP confirmable message not confirmed.",
  "severity": "error",
  "description": "CoAP message '22287' was not confirmed.",
  "fields": {
    ...
    "coap.mid": [
        "22287"
]
},
  "protocol": "COAP",
  "parent-record-id": "<root>",
  "event-record-id": "cd28ed6b-12b0-41ce-8f01-74061b5c2fa4",
  "flow": {
        "protocol": "17",
    "source-port": "5683",
    "destination-port": "33499",
    "flow-counter": "0",
    "source-address": "bbbb::3",
    "destination-address": "bbbb::1",
    "string": "UDP@bbbb::3:5683-bbbb::1:33499"
}
}
COAP: started request response validation sequence
COAP: verifying request method
COAP: Not retransmitted
COAP: next state will be verify_token_length
COAP: Token length is correct.
COAP: verifying response existence
COAP: verifying response code
COAP: success GET to path /separate 2.05 Content
{
        "event-id": "success",
  "event-name": "CoAP success response",
  "severity": "information",
  "description": "Detected CoAP success response '2.05 Content' in 'GET' request to path '/separate'",
  "fields": {
    ...
    "coap.method": "GET",
    "coap.path": "/separate",
    "coap.code": "2.05",
    "coap.code_description": "Content"
},
  "protocol": "COAP",
  "parent-record-id": "cd28ed6b-12b0-41ce-8f01-74061b5c2fa4",
  "event-record-id": "fe6798f1-3060-4fba-9095-b507fe1124e2",
  "flow": {
        "protocol": "17",
    "source-port": "5683",
    "destination-port": "33499",
    "flow-counter": "0",
    "source-address": "bbbb::3",
    "destination-address": "bbbb::1",
    "string": "UDP@bbbb::3:5683-bbbb::1:33499"
}
}
COAP: Not retransmitted
COAP: next state will be _done
```

V debug výstupe vidíme postupné prechádzanie pravidlami a vygenerovanie 2

eventov v JSON formáte. Prvý hovorí o tom, že nebolo obdržané potvrdenie správy, druhý o tom, že na GET request skončil s úspešným response kódom 2.05.

Záver

Cieľom práce bolo na základe detailnej štúdie konkrétnych ICS a IoT protokolov vytvoriť pravidlá diagnostického nástroja Distance pre tieto protokoly, aby bolo možné na základe sieťovej komunikácie odhaľovať chyby. Tieto protokoly sa majú stať súčasťou množiny protokolov podporovaných nástrojom Distance a majú slúžiť správcom sietí na automatizovanú diagnostiku problémov v konfigurácií ICS a IoT zariadení.

V práci som dôkladne preštudoval protokoly CoAP, MMS a GOOSE a ich fungovanie som detailne popísal.

Na základe štúdie protokolov som následne vykonal analýzu bezpečnostných hrozieb a chybových stavov, ktoré som previedol na udalosti (events) diagnostického nástroja Distance. Na základe štúdie a analýzy som správanie protokolu preniesol do diagnostického grafu a umiestnil v ňom udalosti na požadované miesto tak, aby odzrkadľovali chybové stavy.

V práci tiež detailne vysvetľujem fungovanie systému Distance, určeného na automatizovanú diagnostiku sieťového toku. Venujem sa vysvetleniu jazyka a jeho konštruktov, spôsoby písania pravidiel a prezentujem doporučené postupy. Ďalším prínosom mojej práce je možnosť využitia sekcie 2.4 ako manuálu pre implementátora pravidiel ďalšieho nového protokolu.

Cieľ práce sa podarilo splniť a výsledkom je teda rozšírenie množiny protokolov, ktoré je Distance schopný analyzovať o protokoly zo sféry ICS a IoT — MMS, GOOSE a CoAP. Pre protokol CoAP som navyše pridal aj detekciu útokov, čo nie je bežne súčasťou množiny diagnostických pravidiel nástroja Distance.

Systém Distance je teda po novom schopný diagnostikovať sieťovú komunikáciu aj pre spomenuté protokoly. Diagnostika bola úspešne otestovaná na ostrých dátach — dátových tokoch z produkčných sietí obsahujúcich tieto tri protokoly. Pravidlá boli tiež revidované expertom a spoluautorom Distance Ing. Martinom Holkovičom.

Osobnými prínosmi je pre mňa zoznámenie sa s oblasťou ICS a IoT, hlbšie pochopenie štandardu ICS61850 a protokolu CoAP, posun v oblasti analýzy

sieťového toku a samozrejme pochopenie diagnostického nástroja Distance.

Možné pokračovanie práce

Práca má potenciál na ďalšie rozšírenie. Preto ešte uvádzam zoznam oblastí, ktoré by bolo vhodné do budúcna preskúmať a rozšíriť:

- Hlbšie sa zamerať na štandard IEC 61850, ktorý je veľmi obsiahly.
  - Vykonať detailnú analýzu chybových stavov a hrozieb pre IEC 61850 aj na aplikačnej úrovni.
  - Túto analýzu následne premietnuť do implementácie a podchytiť možné bezpečnostné hrozby a chyby.
  - Získať testovacie dáta pre protokol SMV a implementovať jeho diagnostiku.
- Pridať do diagnostiky protokolu CoAP proxying a caching, multicastové správy a ďalšie nepovinné rozšírenia.
- Zvýšiť granularitu eventov a pridať viac informatívnych eventov pre lepšie pochopenie užívateľom.
- Vytvoriť automatizované testy, ktoré by automaticky preverili všetky vetvy a pravidlá diagnostiky.

Bibliografia

1. ACKERMAN, P. *Industrial Cybersecurity: Efficiently secure critical infrastructure systems*. Packt Publishing, 2017. ISBN 9781788395984. Dostupné tiež z: https://books.google.cz/books?id=FhlKDwAAQBAJ.
2. *Internet of Things (IoT)* [online]. Trend Micro [cit. 2020-05-20]. Dostupné z: https://www.trendmicro.com/vinfo/us/security/definition/internet-of-things.
3. MACKIEWICZ, R.E. Overview of IEC 61850 and benefits. In: 2006, s. 623–630. Dostupné z DOI: 10.1109/PSCE.2006.296392.
4. CLAVERIA, Joevis; KALAM, Akhtar. GOOSE Protocol: IED's Smart Solution for Victoria University Zone Substation (VUZS) Simulator Based on IEC61850 Standard. In: 2018, s. 730–735. Dostupné z DOI: 10.1109/APPEEC.2018.8566413.
5. MATOUŠEK, Petr. *Description of IEC 61850 Communication*. FIT-TR-2018-01, Brno, CZ: Faculty of Information Technology BUT, 2018. Dostupné tiež z: https://www.fit.vut.cz/research/publication/11832. Technická správa.
6. *Communication networks and systems for power utility automation - Part 7-2: Basic information and communication structure - Abstract communication service interface (ACSI)*. 2010. IEC 61850-7-2. International Electrotechnical Commission.
7. SHELBY, Z.; HARTKE, K.; BORMANN, C. *The Constrained Application Protocol (CoAP)* [Internet Requests for Comments]. 2014. ISSN 2070-1721. Dostupné tiež z: https://tools.ietf.org/html/rfc7252. Technická správa.
8. ARVIND, S.; NARAYANAN, V. A. An Overview of Security in CoAP: Attack and Analysis. In: *2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*. 2019, s. 655–660.

9. BORMANN, C.; SHELBY, Z. *Block-Wise Transfers in the Constrained Application Protocol (CoAP)* [Internet Requests for Comments]. 2016. Dostupné tiež z: https://tools.ietf.org/html/rfc7959. Technická správa.

10. SCHLEGEL, Roman; OBERMEIER, Sebastian; SCHNEIDER, Johannes. A security evaluation of IEC 62351. *Journal of Information Security and Applications*. 2016. Dostupné z DOI: 10.1016/j.jisa.2016.05.007.

11. RYŠAVÝ, O. *DISTANCE* [online]. CESNET [cit. 2020-05-05]. Dostupné z: https://github.com/CESNET/DISTANCE/blob/master/docs/Readme.md.

12. HOFSTEDE, R.; ČELEDA, P.; TRAMMELL, B.; DRAGO, I.; SADRE, R.; SPEROTTO, A.; PRAS, A. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys Tutorials*. 2014, roč. 16, č. 4, s. 2037–2064.

13. HOLKOVIČ, M.; RYŠAVÝ, O. *Diagnostic Language Specification* [online]. CESNET [cit. 2020-05-05]. Dostupné z: https://github.com/CESNET/DISTANCE/blob/master/docs/DiagnosticLanguageSpecification.md.

14. HOLKOVIČ, M. *Feedback k vytvoreným pravidlám* [Osobná video komunikácia 7.5.2020].

15. HOLKOVIČ, M.; RYŠAVÝ, O. *Event Model* [online]. CESNET [cit. 2020-05-05]. Dostupné z: https://github.com/CESNET/DISTANCE/blob/master/docs/EventModel.md.

16. RYŠAVÝ, O. *A systematic approach on diagnostic rule development* [online]. CESNET [cit. 2020-05-05]. Dostupné z: https://github.com/CESNET/DISTANCE/blob/master/docs/SystematicApproach.md.

Dodatok **A**

Zoznam použitých skratiek

- **BLE** Bluetooth Low Energy
- **CoAP** Constrained Application Protocol
- **DMZ** Demilitarizovaná Zóna
- **DNS** Domain Name System
- **DoS** Denial of Service
- **DTLS** Datagram Transport Layer Security
- **GOOSE** Generic Object Oriented Substation Events
- **GUI** Graphical User Interface
- **HTTP** Hypertext Transfer Protocol
- **ICMP** Internet Control Message Protocol
- **ICS** Industrial Control Systems
- **ICS** Industrial Control Systems
- **IEC** Medzinárodná elektrotechnická komisia
- **IoT** Internet of Things
- **IP** Internet Protocol
- **ISO** Medzinárodná organizácia pre normalizáciu
- **JSON** Javascript Object Notation
- **LAN** Local Area Network

- **LPWAN** Low Power Wide Area Network
- **MMS** Manufacturing Message Specification
- **MQTT** Message Queuing Telemetry Transport
- **OSI** Open Systems Interconnection
- **PCAP** Packet Capture
- **PDU** Protocol Data Unit
- **PID** Proportional Integral Derivative
- **PLC** Programmable Logic Controller
- **REST** Representational state transfer
- **SMTP** Simple Mail Transfer Protocol
- **SMV** Sampled Measured Values
- **TLS** Transport Layer Security
- **URI** Uniform Resource Identifier
- **URL** Uniform Resource Locator
- **WAN** Wide Area Network
- **YAML** Yet Another Markup Language