Universidade Estadual de Campinas
Instituto de Computação

Bruno da Silva Melo

Robustness Testing of CoAP Implementations and Applications through Fuzzing Techniques

Teste de Robustez de Implementações e Aplicações utilizando CoAP por meio de Técnicas de *Fuzzing*

CAMPINAS
2018

Bruno da Silva Melo

Robustness Testing of CoAP Implementations and Applications through Fuzzing Techniques

Teste de Robustez de Implementações e Aplicações utilizando CoAP por meio de Técnicas de *Fuzzing*

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Paulo Lício de Geus

Este exemplar corresponde à versão final da Dissertação defendida por Bruno da Silva Melo e orientada pelo Prof. Dr. Paulo Lício de Geus.

CAMPINAS
2018

\AM@currentdocname .pdf

.pdf

Universidade Estadual de Campinas
Instituto de Computação

Bruno da Silva Melo

Robustness Testing of CoAP Implementations and Applications through Fuzzing Techniques

Teste de Robustez de Implementações e Aplicações utilizando CoAP por meio de Técnicas de *Fuzzing*

Campinas, 25 de junho de 2018

"It has been said that man is a rational animal. All my life I have been searching for evidence which could support this."

(Bertrand Russel)

Agradecimentos

Primeiramente aos meus pais, Ercilia e Jorge. Obrigado pelo amor incondicional, simples e sincero. Eu amo vocês, e tenho certeza que não teria chegado até aqui sem vosso apoio às minhas decisões e confiança no meu julgamento, além de muita compreensão. Estendo esses agradecimentos à extensa família, em especial à da Tia Helenita e do Tio Benedito (Henio, Leandro...), à da Tia Tereza, Tia Mira, e à da Tia Inácia.

Agradeço também aos amigos, fundamentais em minha vida e no meu desenvolvimento como pessoa. Tenho a sorte de serem muitos, e espero não esquecer nenhum neste pequeno texto. E antes da lista...: amo muito todos vocês! Galera de Djahdema, sejam os que estão juntos comigo "desde os tempos mais primórdios...", sejam os que chegaram mais tarde mas que me são muito queridos: Jow, Shin, Ray, Brayan, Vini, Prayboy, Osama... Valeu! Espero que continuemos nos encontrando com frequência e possamos tocar nossas discussões aparentemente intermináveis. Aos amigos e gambás de São Carlos, os quais há tempos já não sei mais se constituem minha segunda ou primeira casa: Bloco 28 (Brayan (de novo!), Daniel, Condô, Leo, Luiz Otávio, Djaú, Dylon, D2, Ivanzão, Torrinha), geral do aloja com quem tive mais contato, e a gambazada (e guaxinins!) que apareceu depois: Lúcio, Hérisson, Mari, Tuani, Nahara, Polly, Giovani, Gustavo, Viniboy, Rodolfo, Perereca... Obrigado por me ensinarem tanto! Galera do BCC 09 e agregados das reps e eventos correlatos: Maurício, Molina, Sidarta, Rafinha, Marco, Mamilos, Japa, Trops, Akira, Parkinho, PT... Valeu! Cabe aqui uma atenção especial ao Maurício e ao Molina: tô com saudade, putos! Faltam palavras pra descrever o quanto eu amo vocês! Pessoal do intercâmbio: Lennon, Alan, Vinicio, Matheus, Diego, Davi, Ana, Rafael, Kenny, Winny, Tang, Elisa, Joey, Annabel, Ebube... Enfim, é muuita gente e foi um período incrível o que passamos juntos! Galera da amdocs: Portella, Gila, Tamego, Andrey, Tomita, enfim, todo o ST!—tem mais, muito mais, como em todo esse parágrafo; mas infelizmente preciso parar por aqui... Foi correria isso aí, mas teve bão também! E por último nessa seção, os amigos de Campinas, minha última parada até o momento: Tuiuiú, Luan, Leo, Mi, Gabi, Ceará, Bianca, Nara, Iás, Maria Helena, Paula... Enfim, devido à popularidade do nosso querido elo-comum ainda não-citado, vocês também são muitas e muitos, e foi (e tem sido!) ótimo conhecer todos vocês. Outro grupo com quem eu aprendo muuuito!

Agradeço ao professor Paulo, por acreditar em mim, neste trabalho, e pela paciência ao longo desse desenvolvimento. Agradeço também ao CNPq, pela bolsa de mestrado, com o adendo de que ser pago para fazer um trabalho tão complicado (e de dedicação exclusiva!) como o de uma pós-graduação é o mínimo. A situação do pós-graduando—e da Ciência como um todo—vai de mal a pior em nosso país; junto com diversas outras coisas, aliás.

Por último mas definitivamente não menos importante, agradeço à Talitha, minha companheira, namorada, amiga... Eu te amo. Espero que possamos seguir juntos por muito mais tempo nessa jornada, conhecendo muitos lugares, comendo muitas coisas gostosas, e suportando juntos a realidade dessa sociedade tão doente que nos circunscreve.

Resumo

Constrained Application Protocol (CoAP) é um protocolo da camada de aplicação padronizado pelo IETF em 2014. Segue o estilo arquitetural RESTful e tem como objetivos simplicidade e baixa sobrecarga, para ser utilizado como facilitador da Internet das Coisas (IoT). Portanto, apesar de compartilhar características em comum com o HTTP, o protocolo possui codificação binária, roda sobre UDP etc.

Com o objetivo de estudar e aperfeiçoar na prática a segurança de software para IoT, nesta dissertação nós projetamos e implementamos uma ferramenta chamada FuzzCoAP. Este sistema consiste em um ambiente completo para o teste de aspectos de robustez e segurança de aplicações e implementações lado servidor do CoAP. Cinco técnicas de *fuzzing* caixa-preta foram implementadas: Aleatória, Aleatória Informada, Mutacional, Mutacional Inteligente e Geracional.

Nós utilizamos essa ferramenta para testar um conjunto de amostras selecionadas—aplicações CoAP rodando diferentes implementações do protocolo. Para selecionar essas amostras, nós conduzimos buscas *online* procurando implementações CoAP disponíveis e detalhes a elas relacionados, como estado de maturidade e popularidade. Nós selecionamos 25 amostras (aplicações), cobrindo 25 bibliotecas (implementações) diferentes de CoAP distribuídas em 8 linguagens de programação, incluindo amostras dos sistemas operacionais específicos para IoT RIOT OS e Contiki-NG.

FuzzCoAP foi capaz de detectar um total de 100 falhas em 14 das 25 amostras testadas. Resultados experimentais mostram uma média de 31,3% na taxa de falsos positivos e 87% em reprodutibilidade das falhas (considerando uma "contagem conservadora de falhas") e uma média de 1,9% na taxa de falsos positivos com 100% em reprodutibilidade de falhas considerando uma contagem "otimista". Campanhas de *fuzzing* são executadas na ordem de minutos para a maioria das técnicas (com uma média de 45 minutos por campanha). A exceção é o fuzzer geracional, que atinge uma média de 12 horas por campanha, dado que este utiliza uma quantidade consideravelmente maior de casos de teste que as outras técnicas.

Nós fornecemos uma discussão sobre a comparação dessas cinco técnicas de *fuzzing* no nosso cenário, e sobre o quão robustas (e seguras) as implementações de CoAP testadas são. Nossos dados indicam que as técnicas são complementares entre si, corroborando com outros estudos que sugerem a combinação de várias técnicas para atingir maior cobertura e encontrar mais vulnerabilidades.

Finalmente, toda a pesquisa foi conduzida com um esforço para ser tão aberta e reproduzível quanto poderíamos torná-la. Portanto, tanto o código-fonte do FuzzCoAP como a coleção de amostras estão livremente acessíveis para o público. Também, para maximizar o impacto real na segurança de IoT, todas as falhas foram reportadas aos mantenedores das bibliotecas. Algumas já foram corrigidas, enquanto outras estão sendo manejadas no presente momento.

Abstract

The Constrained Application Protocol (CoAP) is an application-layer protocol standardized by the IETF in 2014. It follows the RESTful architectural style and aims for simplicity and low overhead, in order to be used as an enabler of the Internet of Things (IoT). Thus, even though it has features in common with HTTP, it is binary-encoded, runs over UDP etc.

With the goal of studying and improving practical IoT software security, in this dissertation we design and implement a tool called FuzzCoAP. It is a system comprising a complete environment for testing robustness and security aspects of CoAP server-side implementations and applications. Five black-box fuzzing techniques were implemented in FuzzCoAP: Random, Informed Random, Mutational, Smart Mutational and Generational fuzzers.

We use this tool to test a set of selected samples—CoAP applications running different implementations of the protocol. To select these samples, we conducted online searches looking for available CoAP implementations and related details, such as maturity status and popularity. We selected 25 samples (applications), covering 25 different CoAP libraries (implementations) distributed across 8 programming languages, including samples from IoT-specific operating systems RIOT OS and Contiki-NG.

FuzzCoAP was able to detect a total of 100 errors in 14 out of the 25 tested samples. Experimental results show an average of 31.3% false positive rate and 87% error reproducibility when considering a "conservative error counting" and 1.9% false positive rate with 100% error reproducibility for an "optimistic" one. Fuzzing campaigns are executed in the order of minutes for most techniques (with an average of 45 minutes per campaign). The exception is the generational fuzzer, which attains an average of 12 hours per campaign, given it uses a considerably larger amount of test cases than the other techniques.

We provide a discussion on how those five fuzzing techniques compare to each other in our scenario, and how robust (and secure) the CoAP implementations targeted are. Our data indicates the techniques are complementary to each other, corroborating with other studies suggesting that several techniques should be combined to achieve greater coverage and find more vulnerabilities.

Finally, the entire research was conducted with an effort to be as open and reproducible as we could make it. Thus, both FuzzCoAP's source-code as well as the collection of samples are freely available to the public. Likewise, to maximize the real-world impact on IoT security, all errors were reported to library maintainers. Some have already been fixed, while others are currently being handled, as of this text was written.

List of Figures

List of Tables

List of Listings

Contents

Chapter 1

Introduction

In this dissertation, we chose to approach a new problem—the robustness testing of an emerging application-layer protocol—in light of established techniques and practices—penetration testing through random, mutational and generational fuzzing, both "dumb" and "smart" fuzzing. To this end, we investigated how those techniques were applied to other problems and domains, such as the classical Internet and Web protocols; how to adapt them to our problem, namely an Internet of Things and Web of Things scenario; and evaluated i) the robustness state of current implementations of this protocol, CoAP; and ii) how effective these techniques are, compared to each other, when applied to this new scenario. The following pages document the findings in those directions.

1.1 Motivation

The Internet of Things (IoT) promises to increase the efficiency of our lives by providing new, value-added services through the integration of several technologies and communication solutions. Although a common definition for IoT does not exists so far, the IEEE IoT Initiative[1] has recently proposed a document [32] with exactly that goal. At the current revision, the community arrived at the following definition for a large environment scenario:

> *"Internet of Things envisions a self-configuring, adaptive, complex network that interconnects 'things' to the Internet through the use of standard communication protocols. The interconnected things have physical or virtual representation in the digital world, sensing/actuation capability, a programmability feature and are uniquely identifiable. The representation contains information including the thing's identity, status, location or any other business, social or privately relevant information. The things offer services, with or without human intervention, through the exploitation of unique identification, data capture and communication, and actuation capability. The service is exploited through the use of intelligent interfaces and is made available anywhere, anytime, and for anything taking security into consideration."*

[1] `http://iot.ieee.org/definition.html`

Considering the current technology and communications environment, the IoT vision is that society would benefit from a wide range of applications in areas such as agriculture, manufacturing, city infrastructure (e.g. mobility & transportation, energy & water distribution, environment monitoring), retail, logistics, healthcare, home & building, and many others[2].

Regarding its high-level architecture, the Internet of Things is commonly seen as a composition of different layers, each having their own responsibility. In [47], for instance, the authors review three widely-known IoT reference models: i) a 3-layer model depicting the IoT as an extension of Wireless Sensor Networks (WSNs), augmented by the use of Cloud Servers to offer services to users; ii) a 5-layer model based on service decomposition and object abstractions for edge nodes, providing a Service-Oriented Architecture (SOA) and Resource-Oriented Architecture (ROA) ecosystem; and iii) a 7-layer model with even more granularity, including concepts as Fog/Edge Computing and Data Abstraction layers between the edge nodes and the higher layers of Applications and Users. Figure 1.1 shows that 7-layer model, as proposed by Cisco at [15] and presented in [47]. In general, though, what we see is that all those reference models can be thought of as a 3-piece model (see example in Figure 1.1), herein called *edge-side layer*, *cloud/service layer* and *user/application layer*.

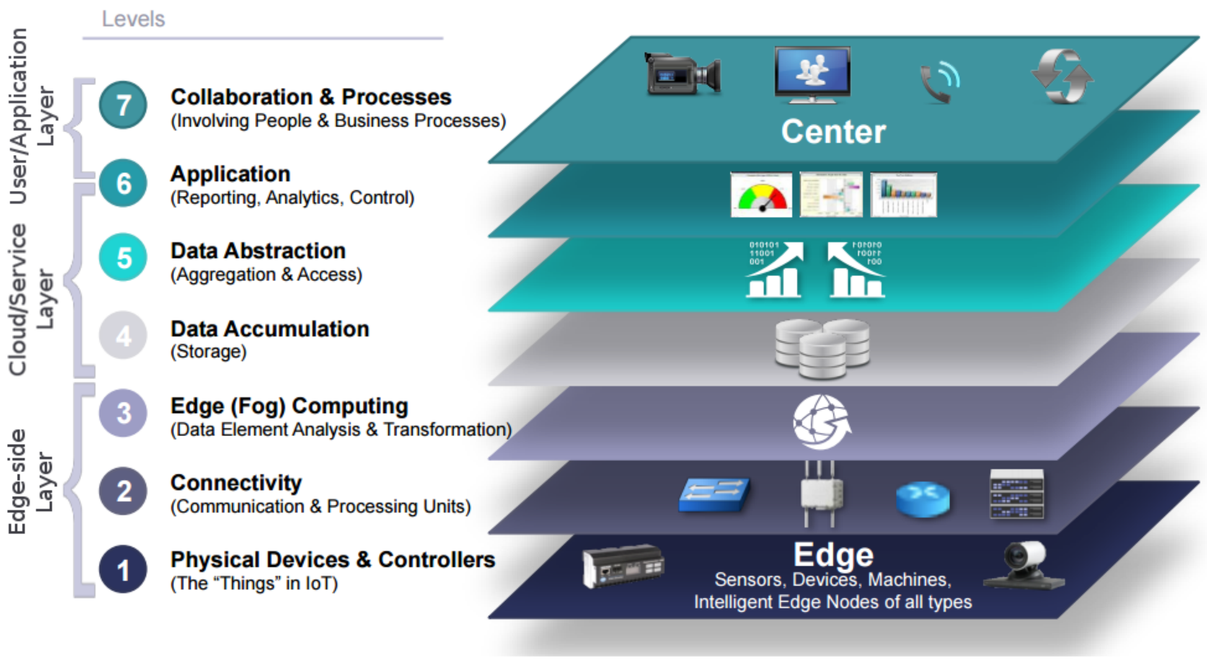

Levels
User/Application Layer
Cloud/Service Layer
Edge-side Layer
7 Collaboration & Processes
(Involving People & Business Processes)
6 Application
(Reporting, Analytics, Control)
5 Data Abstraction
(Aggregation & Access)
4 Data Accumulation
(Storage)
3 Edge (Fog) Computing
(Data Element Analysis & Transformation)
2 Connectivity
(Communication & Processing Units)
1 Physical Devices & Controllers
(The "Things" in IoT)
Center
Edge
Sensors, Devices, Machines,
Intelligent Edge Nodes of all types

Figure 1.1: Cisco's seven-layer model. Adapted from Cisco [15].

At the *edge-side layer*, sensors and actuators are responsible for bringing physical objects into the digital world. Technologies used in this layer include, but are not limited to embedded sensors, valves, Radio-Frequency Identification (RFID) tags and readers, Global Positioning System (GPS), different kinds of routers, switches and gateways, 6LoWPAN, RPL, Bluetooth and Wi-Fi.

[2]http://www.libelium.com/libelium-smart-world-infographic-smart-cities-internet-of-things/

The *cloud/service layer* includes the traditional Internet and its components. It needs to deal with heterogeneous networks such as the Internet itself, and a range of WSNs at personal, local and metropolitan areas, as well as mobile networks. It is responsible for things such as data accumulation, abstraction and service composition, and enabling SOA and ROA architecture styles [72].

Then, at the *user/application layer*, application scenarios—e.g. smart cities, smart health, smart homes—are deployed by leveraging SOA and ROA, through the use of technologies such as CoAP, Wireless Application Protocol (WAP), HTTP and Web Services. It goes without saying that resources, connections, data and services should be provided, through all these layers, in a secure manner. Therefore, one can find in the literature different research efforts, with specific techniques and methods, regarding the security particularities of each of these layers [25].

Although a number of standards for IoT communications have been proposed in recent years, and none of them have become a *de facto* standard yet, the protocol stack composed by IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs) [41, 48, 30], IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) [76], and the Constrained Application Protocol (CoAP)[9, 69] running over IEEE 802.15.4 [31] has been thoroughly explored by the research community, due to its focus on enabling direct end-to-end integration of edge devices—and initially isolated WSNs—to the Internet [57, 25]. A parallel between this protocol stack and the current Internet stack can be seen in Figure 1.2.

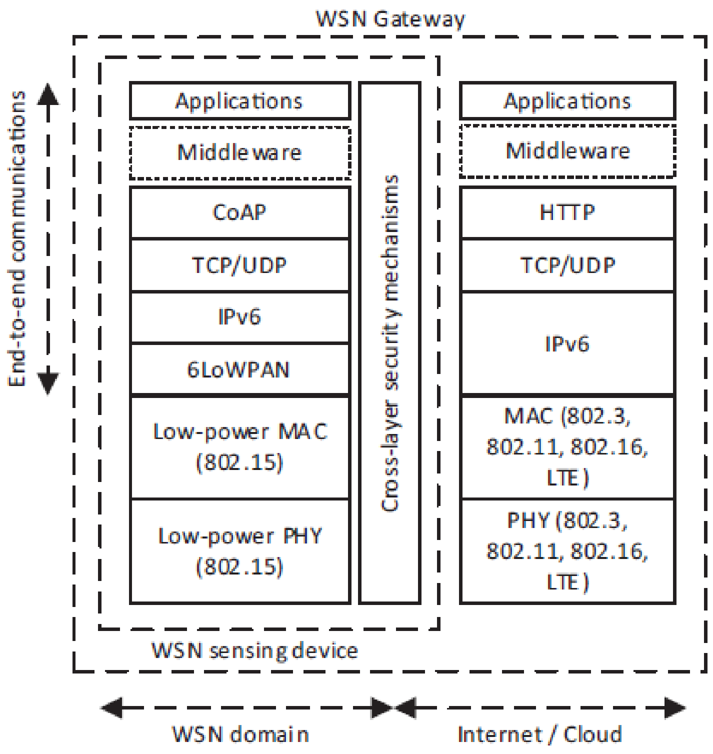


Figure 1.2: 6LoWPAN-based stack for IoT in comparison with the traditional protocol stack used in the Internet, depicting the integration of a WSN with the Internet. From Granjal et al. [25].

Running at the application layer of this protocol stack, CoAP is a RESTful [21] protocol featuring a well-defined mapping to HTTP and its methods (`GET`, `POST`, `PUT` and `DELETE`), and supporting multicast, Universal Resource Identifiers (URIs), content-type identification, resource discovery, response codes and simple subscription and caching mechanisms. Aiming for simplicity and low overhead, it requires a fixed 4-byte header. Since the messages are transported over UDP, CoAP implements a simple reliability mechanism as well. A detailed description of the protocol features, including some of its extensions proposed so far, is presented in Section 2.1.

CoAP is considered one of the most promising solutions for the IoT and machine-to-machine (M2M) communications, which can be demonstrated by the different schemes for the realization of the IoT—considering the proper integration with the traditional Internet—that have been proposed on top of it, such as a SOAP binding, using Efficient XML Interchange (EXI) format to provide SOAP Web Services in WSNs [49, 11]; a Firefox plugin for communicating with CoAP servers, called Copper [38]; and end-to-end communication schemes between web browsers and CoAP servers, based on JVM-enabled browsers [12], on Javascript-enabled browsers (called Actinium) [39], and on HTML5 Web Sockets and Javascript (called SCoAP) [23].

In general, we observe that these communication and integration solutions are converging to a Web-like environment, where complex applications can be built by mixing real-world devices and a number of services, in a practice called *mashups*, finally building up to the Web of Things (WoT) concept [26, 29, 62]. We show in Figure 1.3 the places in which CoAP is expected to be used across constrained and traditional Internet environments to implement the Web of Things architecture.

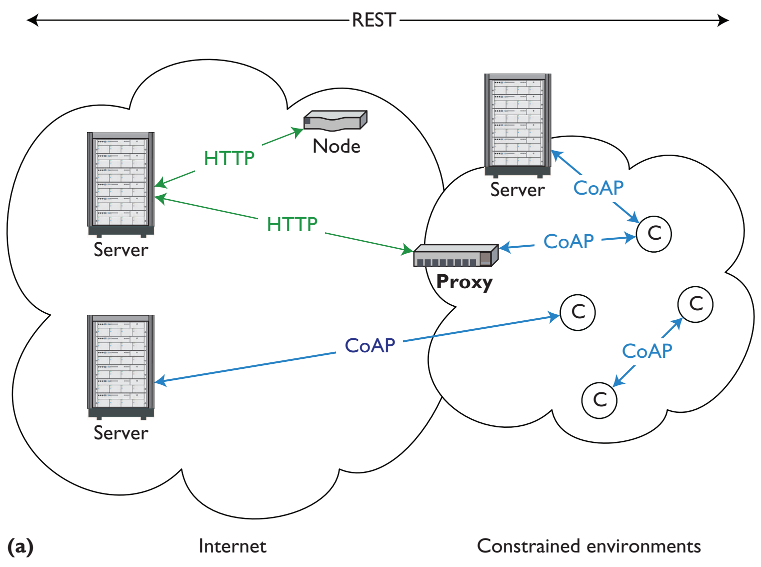


Figure 1.3: HTTP and CoAP used together to realize the Web of Things architecture. Adapted from Bormann [9].

However, considering the complex, heterogeneous scenarios IoT networks—and, furthermore, the Web of Things ecosystem—may lead us to, there are security and privacy concerns that need to be addressed before this new paradigm can be widely accepted by the public [7]. Even so, the amount of connected devices is already growing, with the most famous estimates for the number of connected devices ranging from 25 billion[3] to 50 billion[4] for the year 2020; although most of them are being criticized today, with current estimates ranging from 6.4 to 17.6 billions for 2016, and from 20.8 to 30.7 billions for 2020 [52].

Regardless of the exact figure, this increase in the number of Internet-connected devices will definitely increase the overall attack surface in the whole system, and their remote connectivity opens up new vulnerabilities to be exploited. For instance, despite the fact that the "*real* IoT" vision—with uniquely identifiable edge nodes running IPv6, reachable through a SOA or ROA architecture—is not realized yet, we can find reports of vulnerable devices in the "ad-hoc, *fake* IoT"—with devices mostly running HTTP over IPv4 behind NAT or proprietary stacks—both found in a lab [53, 58] as well as in the wild [13, 54], the biggest and most famous one probably being the Mirai botnet [40].

Thus, although this Internet-based global interconnection is desired to allow resource sharing and ubiquitous services provisioning, it is expected that IoT devices exposed to the Internet will become new targets to malicious entities. One possible attack vector is the CoAP protocol, which may be exploited by a myriad of vulnerabilities, including protocol parsing, URI processing, proxying and caching, risk of amplification, IP address spoofing etc. Threats to the *edge-side layer* of the IoT include (D)DoS, corrupted nodes, fraudulent packet injection etc. [47] System-wide, there is always the possibility of a malicious entity compromising a node to perform lateral movements on a private network.

By reviewing the literature we were able to find research performing security analysis of CoAP, but while Alghamadi et al. [2] focus on discussing the issues and limitations of using IPSec or DTLS to secure CoAP at different layers of the protocol stack—by analyzing the security features provided by each of these approaches using the X.805 security standard—, Rahman and Shah [64] present a survey-like research, mostly covering DTLS applicability to CoAP. To the best of our knowledge, there is little to no work on the practical security of this "*real* IoT" vision, and most of them focus either on cryptographic approaches [65, 27, 24], or attack scenarios performed by previously compromised CoAP nodes in a constrained network [43]. Since cryptography alone is not enough to protect software, this dissertation focus on understanding and finding out the ways in which a CoAP node could be compromised in the first place.

To that end, we leverage fuzzing techniques to discover vulnerabilities in a range of available CoAP implementations. Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected/exceptional, or random data as inputs to a computer program and investigating how such program behaves under those inputs [71]. In a traditional sense, fuzzing is used to perform robustness testing (i.e. to find bugs related to non-functional requirements such as performance or exception handling), but its use can be extended to security testing since, among other reasons, failures due to exceptions were once estimated to account for two thirds of system crashes and fifty percent of system security vulnerabilities [45].

Among the reasons to choose fuzzing techniques instead of other vulnerability identification techniques, such as vulnerability scanners, static analysis and symbolic/concolic

[3]`http://www.gartner.com/newsroom/id/2905717`
[4]`http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`

execution, we include: i) its relatively low complexity for implementation and general usability (when compared to symbolic/concolic execution) [50]; ii) the potential to identify unknown vulnerabilities (which vulnerability scanners cannot find); iii) the typically smaller numbers of false positives when compared to static analysis; and iv) given the black-box nature of this technique, the possibility of a user who might be hiring new WoT services—or the developer of an WoT solution integrating different WoT services—to use the developed tools to remotely test such services before finally selecting one of them (similarly to the situation discussed for the Web Services scenario in [42]).

Therefore, we propose a system architecture and testing methodology to run fuzzing campaigns against CoAP-based IoT applications, targeting CoAP server-side implementations. The reason for this is that considering the WoT architecture in Figure 1.3, it is expected for CoAP servers, and not clients, to be reachable in the IoT. Moreover, we develop the fuzzer tool (which we call FuzzCoAP), use it to run the campaigns on selected targets, and evaluate both each target's robustness as well as the techniques themselves when applied to this scenario. A description of the different fuzzing techniques used in this research is given in Section 2.3, with their details inside the scope of our system being further discussed in Section 3.3.

1.2 Objectives

Based on what was exposed in the previous section, our goal with this dissertation is to find out: **i)** how robust the CoAP implementations available so far are; **ii)** which potential or concrete vulnerabilities are present in those implementations; and **iii)** how the different fuzzing techniques used compare to each other in this scenario. To achieve this, we study: **a)** the CoAP protocol itself, from specification to implementation, for better understanding its role in the IoT/WoT practical realization; and **b)** software robustness and security testing through fuzzing, understanding the previous uses in the literature, in order to leverage the different existing techniques to perform this kind of testing for CoAP server targets. And finally, accomplish this by proposing a system architecture for the testing tool, implementing it, running experiments with it against selected CoAP targets, and evaluating the results both from each individual target's point of view (for i and ii) and from the techniques' point of view (for iii).

1.3 Contributions

The main contributions of this dissertation are:

- A brief review of CoAP-related specifications.
- A discussion about available CoAP implementations, including their current development status, versions of the implemented specifications, licenses, and links to download them. This basically forms a centralized collection of samples that can be easily reused by other researchers. A virtual machine with this collection and

all dependencies configured and ready to be used is available as Free Software under the GNU GPLv3 license at `https://github.com/bsmelo/fuzzcoap`, through a Vagrant[5] file.

- A collection of integrated tools including Fuzzer, Process Monitor and offline analyzers, comprising a complete test environment for CoAP server targets, which we call FuzzCoAP. This is available as Free Software under the GNU GPLv3 license at `https://github.com/bsmelo/fuzzcoap`.
- A discussion about the robustness and the security aspects of each tested target, including practical measures such as error reports delivered to the responsible developers, in order to fix potential or concrete security vulnerabilities.
- A non-exhaustive comparison between five techniques for fuzzing and how they perform when testing CoAP server targets.

Additionally, during the course of this work, the MSc candidate has published and presented one paper at a national symposium and took the role of teaching assistant for one undergraduate course, earning relevant teaching and didactic skills.

1.4 Dissertation Outline

The remainder of this document is organized as follows.

In **Chapter 2** there is a brief review of CoAP-related specifications, including the main functional aspects and discussions based on the Security Considerations section of each RFC presented. We also present existing work targeting the test of CoAP implementations and a literature review of fuzzing techniques, ranging from areas such as software dependability to security and vulnerability research.

In **Chapter 3** we present FuzzCoAP, proposing our system architecture and testing methodology. The complete testing environment is explained, both in terms of design as well as implementation. This includes the fuzzing engines developed, how each of those engines generate test cases, how those test cases are executed and how we detect a failure in the target. We also present which information is captured in an online basis, and how to analyze this information after fuzzing (offline).

We begin **Chapter 4** with data preparation, discussing the samples used in this research—the CoAP server implementations themselves. We outline how we selected our samples and present each of the implementations targeted by our tool, giving details about programming language, development status etc. Then, we present our findings regarding the robustness and security of each target tested, in terms of detected system errors. Besides discussing those specific aspects of each target, we also present a discussion on how the fuzzing techniques compare to each other when used on CoAP server targets.

In **Chapter 5** we conclude this dissertation, summarizing the obtained results and discussing the limitations of the approach taken, as well as presenting future work that can be further explored.

[5]`https://www.vagrantup.com/`

Chapter 2

Background and Related Work

In order to reliably perform fuzzing, we need to grasp not only the methods and techniques available, but the structure of what is being fuzzed. In this chapter, we summarize the standardized specifications defining the message structures and how the CoAP protocol works; briefly present existing work targeting the test of CoAP implementations; and review the research regarding fuzzing approaches and domains in which it has been previously applied, from UNIX applications and OS APIs to Web Applications, Web Services and Network Protocols. This, in turn, will be the basis toward our goal of fuzzing CoAP implementations and applications.

2.1 CoAP Specifications

The IETF Constrained RESTful Environments working group (CoRE-WG) "provides a framework for resource-oriented applications intended to run on constrained IP networks" [17] and is responsible for the documents specifying the CoAP protocol and its extensions or related functionality.

The goal of CoAP is to enable resource-oriented applications on constrained networks. These constrained networks often have very limited packet sizes (e.g. 127 bytes using IEEE 802.15.4), a high rate of packet loss and limited throughput (e.g. tens of kbit/s on 6LoWPAN). Additionally, the nodes participating on these networks consist of constrained devices, which in turn often have: limited memory (ROM and RAM), implying limited code size and stored data; limited available power (devices are normally battery-operated and expected to run for months or years without a recharge); low computing power (CPU); and, in order to save energy, they may be powered off (sleep) and periodically turned on (wake-up), a policy usually achieved by Radio Duty Cycling (RDC) protocols using the hardware's Low-Power Mode (LPM) mechanisms. The design requirements imposed to CoAP by all these different factors can be seen in Figure 2.1.

The **Base** specification for CoAP can be found in RFC7252 [69]. It provides a Request/Response interaction model, similar to the Client/Server model from HTTP. CoAP is a RESTful protocol, which means that a Client can send a CoAP Request to a Resource (identified by a URI) located at a given Server, using a Method Code (CoAP's binary-encoded counterpart to HTTP's `GET`, `POST`, `PUT`, `DELETE` methods). The Server then

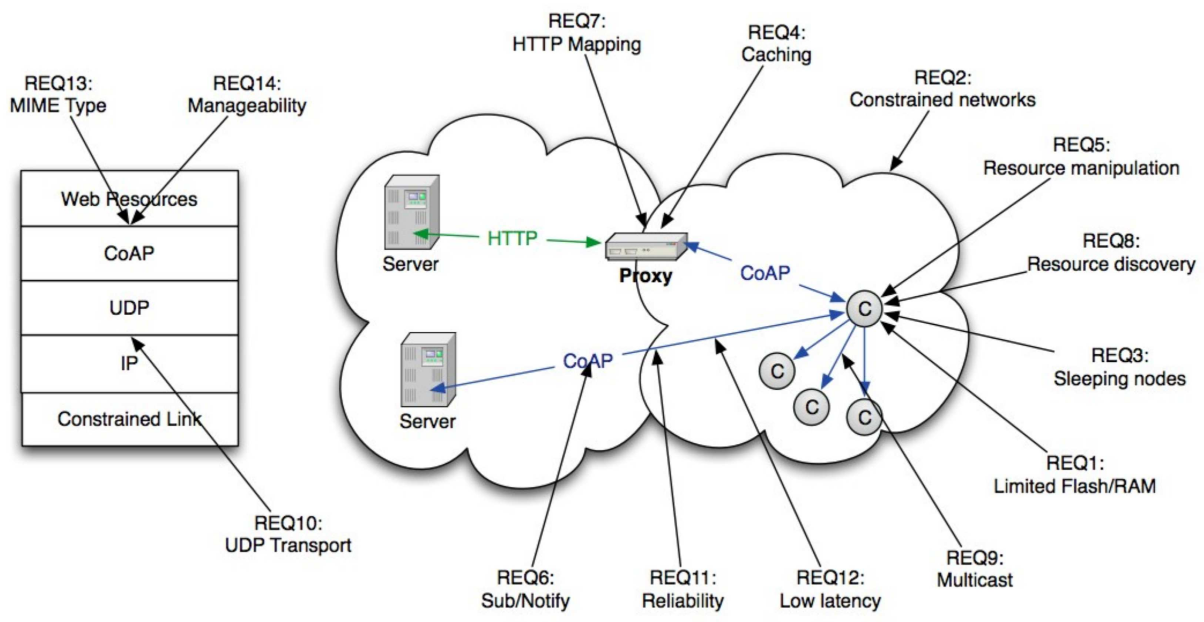


Figure 2.1: CoAP Design Requirements. From Shelby [68].

sends a CoAP Response to the Client with a Response Code (CoAP's binary-encoded counterpart to HTTP's "200 OK", "404 Not Found" etc.), and possibly a Resource representation. CoAP also offers Content-Type support (through binary-encoded Options), simple Proxying and Caching capabilities, stateless HTTP mapping and a security binding to DTLS [66].

CoAP uses a two-layer approach (although it is still a single protocol), with a messaging layer dealing with the unreliability of UDP and the asynchronous interactions, and a request/response layer above it, as shown in Figure 2.2.

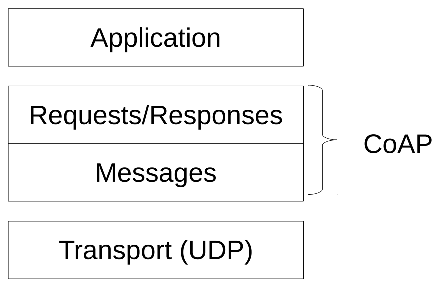


Figure 2.2: CoAP Abstraction Layers.

For optional reliability, CoAP defines four types of messages: Confirmable (`CON`), Non-confirmable (`NON`), Acknowledgement (`ACK`) and Reset (`RST`). A request can be carried over `CON` or `NON` messages, and a response can be carried over either of those as well as `RST`, or be piggybacked in `ACK` messages. A `CON` message always requires an `ACK` from the other side, so a retransmission must happen in case of packet loss. A Message ID is used to match messages and detect duplicates. A Token (not to be confused with the Message ID) is used by the request/response layer to match requests to responses—particularly

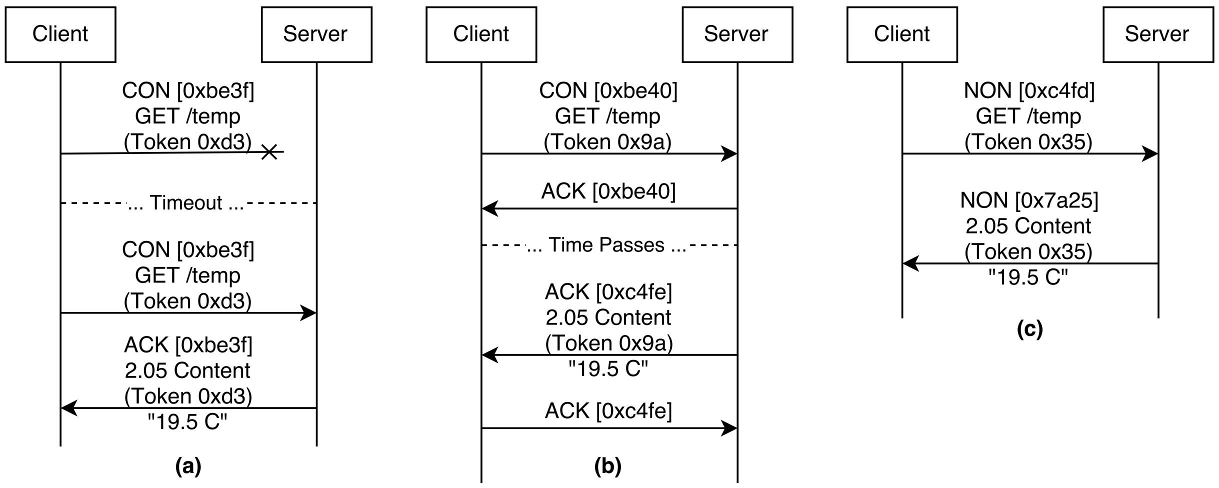


Figure 2.3: Examples of basic CoAP interactions. **(a)** `GET` request to a `/temp` resource, performed over `CON` message (and message loss followed by retransmission, reusing the message id), resulting in a Piggybacked Response. **(b)** `GET` request over `CON` message, resulting in a Separated Response scenario. When the resource is finally available at the server, the response is sent (notice the token being used, so this response can be matched to its request). **(c)** `GET` request over `NON` message, resulting in a response over `NON` message as well.

needed in the case of non-piggybacked (i.e. Separated) Responses or responses to requests carried over `NON` messages. Examples of CoAP interactions can be seen in Figure 2.3.

CoAP messages are transported over UDP (although other transport bindings are possible and even specified, such as DTLS, SMS or TCP, they are out-of-scope for this dissertation) and encoded in a simple and compact binary format. This format starts with a fixed-size 4-byte Header; followed by an optional Token value, of variable-length from 0 to 8 bytes; a sequence of zero or more Options in Type-Length-Value (TLV) format; and an optional Payload.

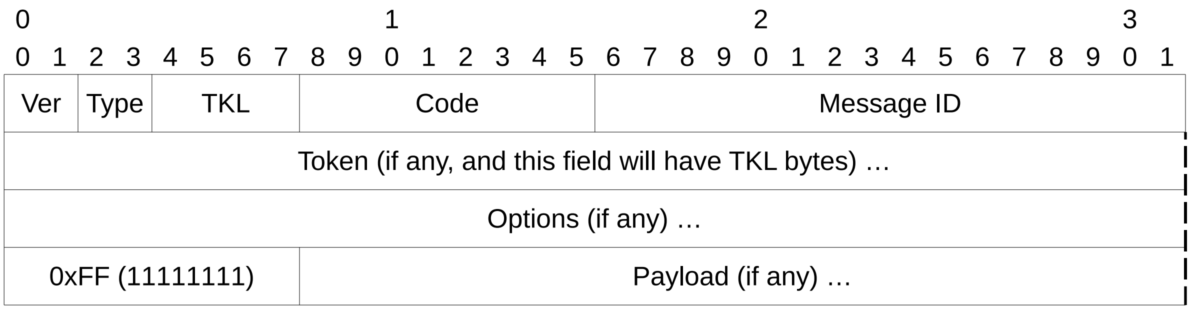


Figure 2.4: CoAP Message Format.

The CoAP Message Format is shown in Figure 2.4. A brief description for each field is given below:

Version (Ver) 2-bit unsigned integer. Indicates the CoAP version.

Type 2-bit unsigned integer. Message type, i.e. Confirmable (`0`), Non-confirmable (`1`), Acknowledgement (`2`) or Reset (`3`).

Token Length (TKL) 4-bit unsigned integer. The length of the Token field (0–8 bytes).

Code 8-bit unsigned integer, split by a mask defining a 3-bit `class` (msb) and 5-bit `detail` (lsb), written as `"c.dd"`, where `0` $\leq$ `c` $\leq$ `7` and `00` $\leq$ `dd` $\leq$ `31`. It carries a Request Method in case of a request (e.g. `0.01` means `GET`), or a Response Code in case of a response (e.g. `2.05` (Content)).

Message ID 16-bit unsigned integer. Used to detect message duplication and to match between messages. The standard defines specific rules for generating this.

Token 0–8 bytes. Used to correlate between requests and responses. The rules for generating this value are standardized as well.

Options 0–* bytes. Each Option instance in a message must specify a Number, a Length and a Value, as shown in Figure 2.5. A complete list of options defined so far by the main CoAP-related specifications is shown in Table 2.1. A description of each subfield of the option field is given below:

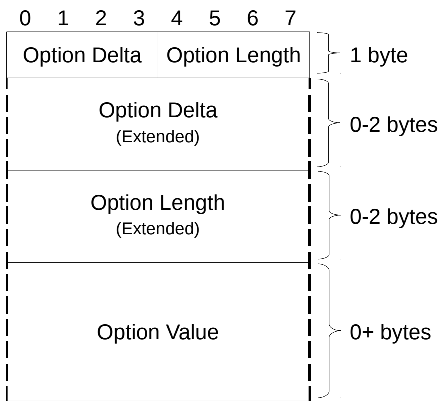


Figure 2.5: CoAP Option Format.

Option Delta 4-bit unsigned integer. The Option Number is not specified directly; instead, each option instance must appear in ascending order of their specific option numbers, and the option number of a given instance is calculated as the sum of its Option Delta with the ones of the preceding instances. Values between `0` to `12` are used directly, with `13-15` reserved for special meanings (see the extended version below).

Option Length 4-bit unsigned integer. Used to specify the length of the Option Value field. Values between `0` to `12` are used directly, with `13-15` reserved for special meanings (see the extended version below).

Option Delta (extended) 0–2 bytes. If the Option Delta is specified as `13`, an 8-bit unsigned integer indicating the option delta minus `13`. If the option delta is specified as `14`, a 16-bit unsigned integer indicating the option delta minus `269`.

- **Option Length (extended)** 0–2 bytes. If the Option Length is specified as `13`, an 8-bit unsigned integer indicating the option length minus `13`. If the option length is specified as `14`, a 16-bit unsigned integer indicating the option length minus `269`.
- **Option Value** 0–* bytes. The value itself, following a format as defined for the specific option number. Examples of option formats are `empty`, `opaque`, `uint` or `string`, as defined in [69].

Payload 0–* bytes. If present, it is prefixed by a one-byte Payload Marker (`0xFF`), indicating the end of the Options and start of the Payload. Its length is calculated from the datagram size, i.e., the payload data extends to the end of the UDP datagram.

Regarding caching, it is enabled by a simple mechanism based on freshness and validation information carried as options—Max-Age and ETag, respectively—by CoAP responses, as defined by the base specification [69]. Proxying is also specified by that document, with not only the possibility of a CoAP-CoAP proxy, but CoAP-HTTP and HTTP-CoAP "cross-proxies" as well, since it is quite simple to perform the mapping between both protocols, given that they are both based on the REST architecture.

Possible threats to the protocol, as analyzed in the Security Considerations section of RFC7252 [69] include: protocol and URI parsing and processing; communications security issues with proxying and caching mechanisms—out-of-scope in this study since this would fall more into a Systems Security approach than a Software Security one; we are, however, including all CoAP options (and this covers Max-age and Proxy-* as well) in the fuzzing process—; the risk of amplification, which could be even worse in a multicast scenario; IP address spoofing, inherent to the use of UDP; cross-protocol attacks; and specific considerations regarding constrained-nodes and constrained-networks, ranging from battery depletion attacks to physical tampering.

The **Observe** option can be used to follow state changes of CoAP resources over time, and the use of this option is specified by RFC7641 [28]. A CoAP client registers its interest in a given resource by sending a `GET` request containing the `Observe` option with value `0` to the server, which, in turn, returns the current representation of the target resource and adds the client to the list of observers of that resource. From that point onwards, whenever the state of the observed resource changes, the client receives Notifications (updates). These are additional CoAP responses to that single `GET` request it initially sent, sent until one of the following conditions are met, in which case it is removed from the server's list of observers: i) a confirmable notification message is unacknowledged (either intentionally by the client or by multiple message losses followed by a timeout); ii) the client sends an explicit Deregistration message (CoAP `GET` request containing the same token and options, including their values, as the Registration message—except for etags and the observe option, which should have the value `1`); or iii) the client rejects a notification by sending an `RST`. An example interaction using the observation feature is show in Figure 2.6.

The Security Considerations section of RFC7641 [28] lists an increased risk of amplification attacks, using the notifications mechanism; and the possible exploitation of the

Table 2.1: CoAP Option Table. Default option values cannot be assumed unless explicitly described in here. Length is expressed in bytes.

No	Name	Format	Length	Brief Description	R[d]
1	If-Match[a]	opaque	0–8	Make a request conditional on the current existence of any representation for the target resource (empty value), or on the existence of a particular representation (Etag value). Useful for protecting against accidental overwrites when updating resources.	Y
3	Uri-Host[a]	string	1–255	Internet host of the resource being requested. Default value taken from the destination IP address of the request message.	
4	ETag[a]	opaque	1–8	For differentiating between representations of the same resource varying over time, this entity-tag is used as a resource-local identifier.	Y
5	If-None-Match[a]	empty	0	Make a request conditional on the current non-existence of any representation for the target resource. Useful for protecting against accidental overwrites when creating resources.	
6	Observe[b]	uint	0–3	When used in a `GET` request, value `0` means Register and `1` means Deregister. When used in a response, the value is a sequence number for reordering detection.	
7	Uri-Port[a]	uint	0–2	Transport-layer port number of the resource. Default value taken from the destination UDP port of the request message.	
8	Location-Path[a]	string	0–255	Indicates a relative URI (together with Location-Query), used in a `2.01` (Created) response to indicate the location (relative to the request URI) of the created resource.	Y
11	Uri-Path[a]	string	0–255	One segment of the absolute path to the resource.	Y
12	Content-Format[a]	uint	0–2	Numeric identifier indicating the representation format of the message payload. E.g. `50` means `application/json`.	
14	Max-Age[a]	uint	0–4	Number of seconds indicating the maximum time a response may be cached before it is considered not fresh. Default value is `60` seconds.	
15	Uri-Query[a]	string	0–255	One argument parameterizing the resource, in `key=value` format.	Y
17	Accept[a]	uint	0–2	Numeric identifier indicating which Content-Format is acceptable to the client. Follows the same IANA registry as the Content-Format option.	
20	Location-Query[a]	string	0–255	Indicates a relative URI (together with Location-Path), used in a `2.01` (Created) response to indicate the location (relative to the request URI) of the created resource.	Y
23	Block2[c]	uint	0–3	Descriptive or control data related to the response payload.	
27	Block1[c]	uint	0–3	Descriptive or control data related to the request payload.	
28	Size2[c]	uint	0–4	Relative to the response payload. Used in a request (together with Block2) with value `0` to ask the server for a total resource size estimate, and in a response so the server can indicate the resource size estimate.	
35	Proxy-Uri[a]	string	1–1034	Absolute URI used to make a request to a forward-proxy.	
39	Proxy-Scheme[a]	string	1–255	When using a forward-proxy, if the Absolute URI is to be constructed from Uri-* options (missing Proxy-Uri), this value replaces the scheme part of the resulting URI.	
60	Size1[a]	uint	0–4	Relative to the request payload. Used in a request (together with Block1) to indicate the resource size estimate, or in a `4.13` (Request Entity Too Large) response to indicate the maximum size accepted by the server.	

[a] Specified in RFC7252 Base [69].
[b] Specified in RFC7641 Observe [28].
[c] Specified in RFC7959 Block-wise [10].
[d] Repeatable, i.e. the option can appear more than once in a message.

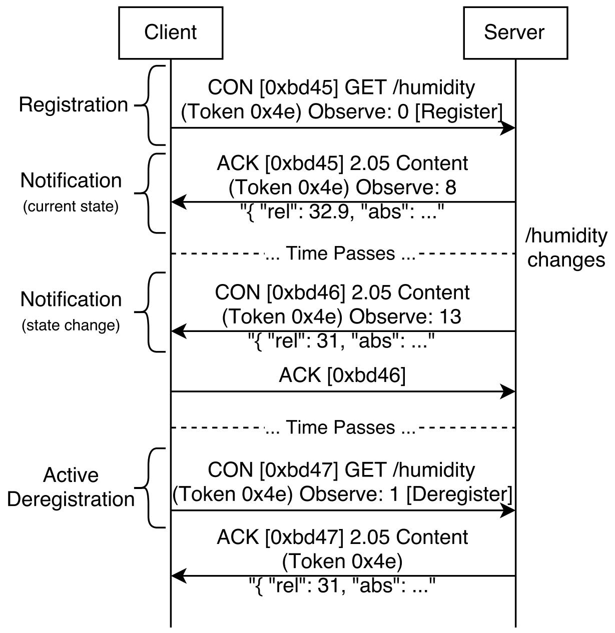

Client
Server
Registration
CON [0xbd45] GET /humidity (Token 0x4e) Observe: 0 [Register]
Notification (current state)
ACK [0xbd45] 2.05 Content (Token 0x4e) Observe: 8 "{ "rel": 32.9, "abs": ..."
... Time Passes ...
/humidity changes
Notification (state change)
CON [0xbd46] 2.05 Content (Token 0x4e) Observe: 13 "{ "rel": 31, "abs": ..."
ACK [0xbd46]
... Time Passes ...
Active Deregistration
CON [0xbd47] GET /humidity (Token 0x4e) Observe: 1 [Deregister]
ACK [0xbd47] 2.05 Content (Token 0x4e) "{ "rel": 31, "abs": ..."

Figure 2.6: Example of CoAP interaction using the Observe option. The client registers itself as an observer for the `/humidity` resource, receives notifications over time, and later decides to deregister its interest on the resource. Notice the value of the observe option and the matching tokens.

server state created to maintain the list of observers, to cause resource exhaustion, as the main threats added to the protocol by the observe feature.

In case an application needs to transfer large payloads with CoAP—for instance, for firmware updates or transferring a local log file—the **Block** option specified by RFC-7959 [10] can be used. The idea is to avoid IP-layer and/or adaptation-layer fragmentation, while enabling individual transmission of blocks, if needed, since each one is acknowledged at CoAP's messaging layer. Basically, a block-size is agreed upon in the first message exchange, and from that point onwards the client can simply continue to make normal requests, with the addition of the block option including the block number being requested. Examples of interactions using this feature are shown in Figure 2.7.

Options from RFC7959 are defined and named according to the direction of payload transfer, with `Block1` and `Size1` options pertaining to the requests' payload, and `Block2` and `Size2` pertaining to the responses' payload. A Block option can have a "descriptive usage" (e.g. `Block1` used in a `PUT` request or `Block2` used in a `2.05` response to a `GET` request) or a "control usage" (e.g. `Block2` used in a `GET` request or `Block1` in a `2.04` response to a `POST` request). The Block option is a variable-sized (0–3 bytes) `uint` encoding three subfields, as shown in Figure 2.8.

A description of each subfield of the Block option is given below:

NUM 4-, 12- or 20-bit unsigned integer. Indicates the block's relative number, within a sequence of blocks with the given size.

M 1-bit flag. In "descriptive usage", holds the value `1` if there are more blocks after this to be transferred; and `0` otherwise, which means this is the final block. In "control usage", it can serve as indication of an atomically implemented operation (see Figure 2.7 a).

SZX 3-bit unsigned integer. The "size exponent", from `0` to `6` (`7` is reserved), used to specify the block-size, according to 2^{SZX+4}; e.g. if `SZX=3`, then the block-size is $2^{(3+4)} = 128$ bytes.

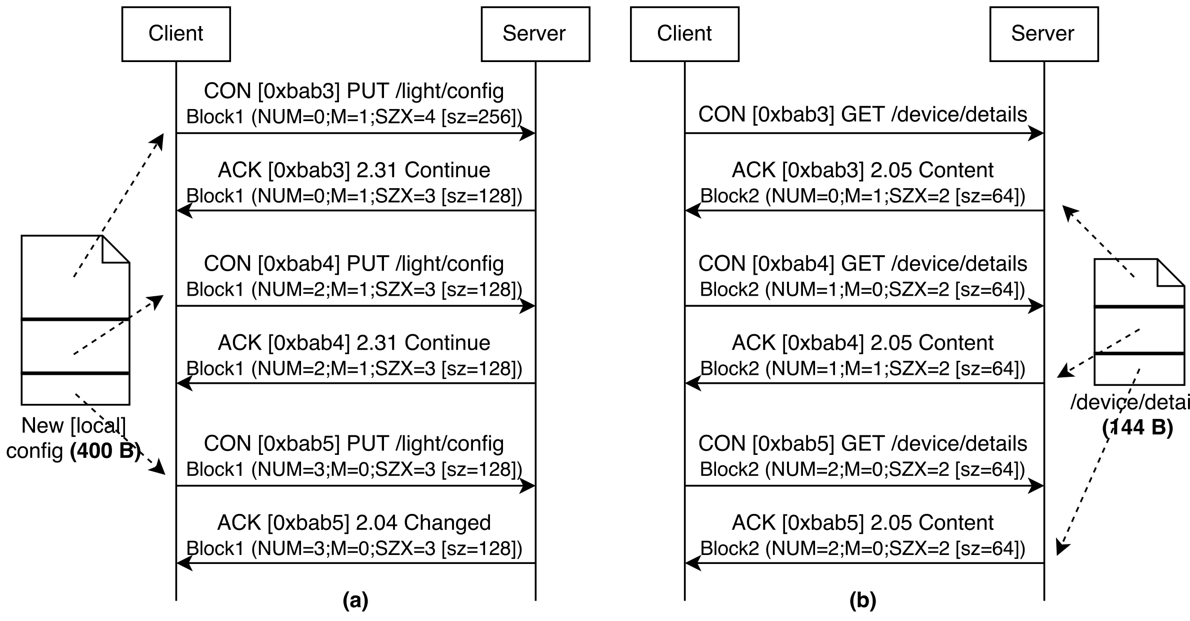

Client
Server
CON [0xbab3] PUT /light/config
Block1 (NUM=0;M=1;SZX=4 [sz=256])
ACK [0xbab3] 2.31 Continue
Block1 (NUM=0;M=1;SZX=3 [sz=128])
CON [0xbab4] PUT /light/config
Block1 (NUM=2;M=1;SZX=3 [sz=128])
ACK [0xbab4] 2.31 Continue
Block1 (NUM=2;M=1;SZX=3 [sz=128])
New [local] config (400 B)
CON [0xbab5] PUT /light/config
Block1 (NUM=3;M=0;SZX=3 [sz=128])
ACK [0xbab5] 2.04 Changed
Block1 (NUM=3;M=0;SZX=3 [sz=128])
(a)
Client
Server
CON [0xbab3] GET /device/details
ACK [0xbab3] 2.05 Content
Block2 (NUM=0;M=1;SZX=2 [sz=64])
CON [0xbab4] GET /device/details
Block2 (NUM=1;M=0;SZX=2 [sz=64])
ACK [0xbab4] 2.05 Content
Block2 (NUM=1;M=1;SZX=2 [sz=64])
/device/details (144 B)
CON [0xbab5] GET /device/details
Block2 (NUM=2;M=0;SZX=2 [sz=64])
ACK [0xbab5] 2.05 Content
Block2 (NUM=2;M=0;SZX=2 [sz=64])
(b)

Figure 2.7: Examples of CoAP interactions using the Block options. **(a)** Shows a client request containing a payload split-up in blocks, using the Block1 option. It represents an atomic update operation (`M=1` in the `ACK` messages), which means the server will only act upon the target resource after receiving all blocks. It also shows how late negotiation works, which is the block-size negotiation that happens between the first two messages: notice the increase from `0` to `2` in the `NUM` parameter from the first to the third message—the first message already sent two blocks, considering the newly agreed-upon block-size of 128 bytes instead of the 256 bytes originally used by the client. **(b)** Shows a simple block-wise `GET` to retrieve a resource, which is split in blocks by the server to be sent in separate payloads in the responses (notice the Block2 option).

Possible threats added to the protocol by the block-wise transfer feature, as stated in the Security Considerations section of RFC7959 [10], include: requests implemented non-atomically, which may lead to partially updated or corrupted resource representations; exploiting stateless servers using a high block number in a `Block1` option; and inducing buffer overflows by including misleading size indications. We can note how these possibilities are good examples for the applicability of fuzzing techniques. It is also mentioned how the use of the block-wise feature can mitigate the risk for amplification attacks, since a large resource representation would be split into smaller responses, each requiring an individual request, effectively reducing amplification factors.

For basic Resource Discovery, a well-known URI defined by RFC6690 [67] can be used to request the resources hosted by a server, which, in conformance with the same specification, must be returned using the **CoRE Link Format**. This specific link serialization

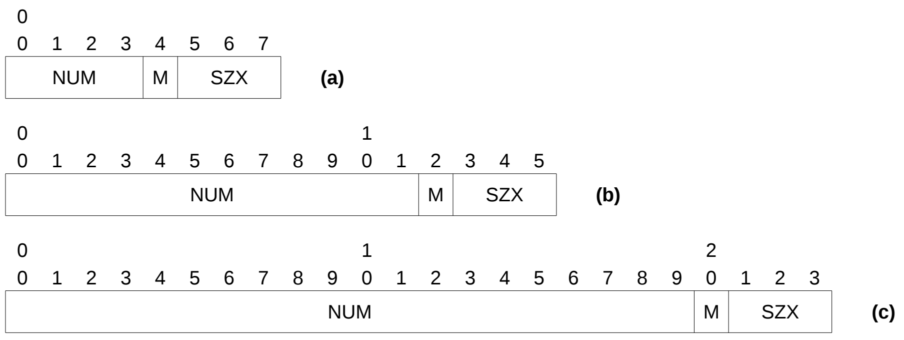


Figure 2.8: CoAP Block Option Format. This option appears either as **(a)** a 1-byte `uint`, or **(b)** a 2-byte `uint`, or **(c)** a 3-byte `uint`. Notice that only the length of the `NUM` field changes.

format is returned as a payload describing hosted resources, their attributes, and other relationships between links, as a response to a `GET` request to `/.well-known/core`—i.e. it is carried as a resource itself, as opposed to HTTP Link Headers delivered with each specific resource. The M2M use cases for this feature include: Discovery (a client could match the attributes Resource Type, Interface Description and Media Type to automatically find out the appropriate resource as well as how to interact with it); Resource Collections (nested indexes of resources containing CoRE Link Formats); and Resource Directory (like a limited search engine for constrained devices, where a centralized Resource Directory (RD) Server acts as a database of known resources, with individual CoAP servers registering their resources via `POST`. Individual CoAP clients can perform lookups using `GET`—this specific feature is being specified in an IETF draft, currently at version 13 [70]). Examples of interactions using this feature can be seen in Figure 2.9.

Besides the relatively complex grammar for the CoRE Link Format, RFC6690 also specifies filtering through the use of a query string, including the use of wildcards (`*`), which definitely increases parsers' complexity, opening room for possible vulnerabilities related to input parsing. Automatically following discovered links could also present a threat (e.g. an attacker could perform "CoRE Link Poisoning", including malicious links or removing legitimate ones from the `/.well-known/core` resource). Besides the Security Considerations section of RFC6690 [67] itself, we are also taking into account that same section from RFC3986 [44], which deals with URIs in a more general approach and includes threats such as maliciously-constructed URIs.

Finally, examples of other work being done by the CoRE-WG include the experimental **Group Communications** support defined by RFC7390 [63], although the working group itself will not further develop a reliable multicast solution [17]; IETF drafts specifying a centralized **Resource Directory** and its interfaces [70]; and an IETF draft specifying an extension with **`PATCH` and `FETCH` RESTful methods**. The complete list of the work done by the CoRE-WG can be found at [17].

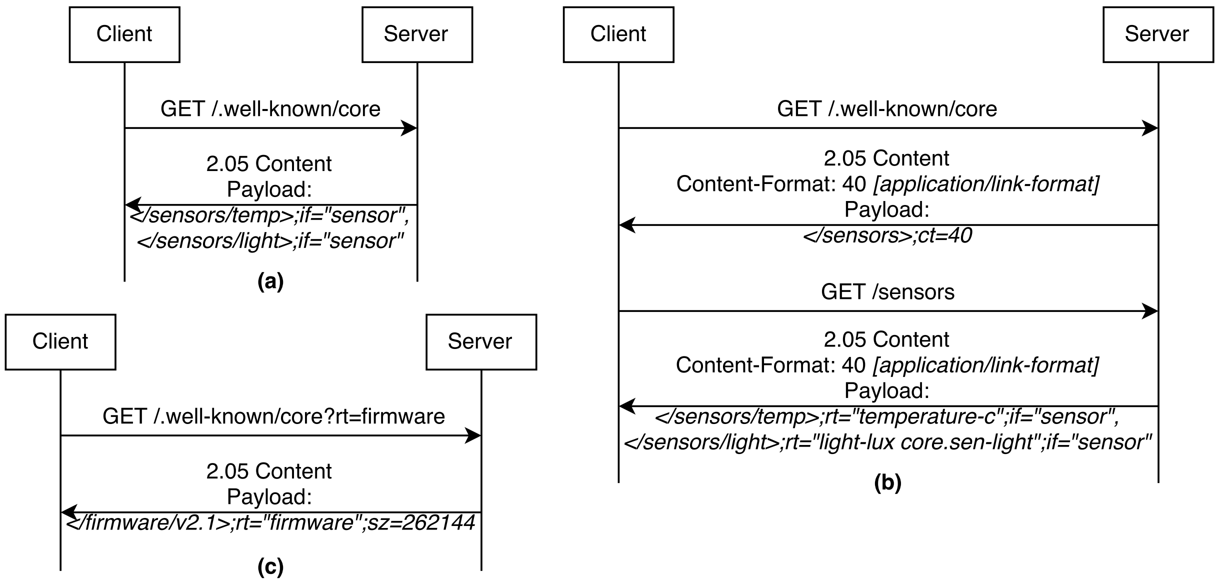

Client
Server
GET /.well-known/core
2.05 Content
Payload:
</sensors/temp>;if="sensor",
</sensors/light>;if="sensor"
(a)
Client
Server
GET /.well-known/core
2.05 Content
Content-Format: 40 [application/link-format]
Payload:
</sensors>;ct=40
GET /sensors
2.05 Content
Content-Format: 40 [application/link-format]
Payload:
</sensors/temp>;rt="temperature-c";if="sensor",
</sensors/light>;rt="light-lux core.sen-light";if="sensor"
(b)
Client
Server
GET /.well-known/core?rt=firmware
2.05 Content
Payload:
</firmware/v2.1>;rt="firmware";sz=262144
(c)

Figure 2.9: Examples of CoAP interactions using Link-Format. **(a)** Simple `GET` request to `/.well-known/core` resource, returning a payload using the CoRE Link-Format. The response contains links to two different sensors using the same Interface Descriptions (`if` attribute). **(b)** Link descriptions hierarchically organized (nested indexes). The first request returns a link to a `/sensors` resource, which, if acted upon using a `GET` request, returns links to two different sensors. Notice the Resource Type (`rt`) attribute of the `/sensors/light` resource, containing two different values separated by a space. Also note the `Content-Format` option being used to explicitly tell the payload's format. **(c)** A query filter parameter is used to match the value `firmware` inside the `rt` attribute. In the response payload, we can see the use of the Maximum Size Estimate (`sz`) attribute, which could be used by a consumer of this resource to determine if a block-wise transfer should be used to retrieve it, for instance.

2.2 CoAP Testing

Currently, to the best of our knowledge, there are only a few projects targeting the test of CoAP implementations and applications. In this section, we present and describe the key characteristics of those projects.

ETSI Plugtests

The European Telecommunications Standards Institute (ETSI) promotes events called Plugtests, in which different organizations can take part to test their own implementations of a given standard. The main goals are to improve the interoperability of products and services, support the deployment of new technologies and validate the standards themselves.

Together with the IPSO Alliance[1] and the FP7 Probe-IT project[2], ETSI has organized four IoT CoAP Plugtests so far. The third Plugtest event (CoAP#3) included

[1]`http://www.ipso-alliance.org/`

[2]`http://www.probe-it.eu/`

OMA[3] as an organizer, and included tests for LWM2M[4], and DTLS implementations as well; at CoAP#4, the last of these events so far, held in March, 2014, tests for 6LoWPAN implementations were also added to the list.

Two reports for CoAP#1 are available at [20] and [60], describing the test specification process, the two test configurations (direct Client to Server communication, and communication through a lossy gateway, randomly dropping packets), and the supporting tools used—a lossy gateway operating at transport layer, developed by the Beijing University of Posts and Telecommunications (BUPT), and a pcap analyzer used after the execution of the testing scenarios, developed by the Research Institute of Computer Science and Random Systems (IRISA), and available at [33]. The experience at CoAP#1 and the development of the supporting tools used are described by Chen et al. in [14]. The most recent test descriptions, for CoAP#4, can be found at [8]. Considering CoRE specifications only (no 6LoWPAN- or DTLS-related), it uses up-to-date RFC7252, RFC6690, and outdated Observe (draft-12) and Block-Wise Transfer (draft-14) specifications. A CoAP Client application[5] is available to automatically submit CoAP Requests to a given CoAP Server, as specified by each test description. Since the ETSI Plugtests include testing of the CoAP Clients as well, a CoAP Server application[6] is also available, exposing the CoAP Resources needed by the test descriptions, so any developer can test their Client implementations against this Server.

F-Interop

Part of the European Union H-2020 programme[7], this project was started in November 2015 and is intended to run until October 2018. The aim is to develop and provide online testing tools to perform remote Conformance, Interoperability, Scalability, Quality of Service (QoS) & Quality of Experience (QoE) and Energy Efficiency testing of emerging technologies, supporting them from research to standardization and market launch [74]. Initially, the IoT standards being supported by the program are CoAP, 6TiSCH and 6LoWPAN. Their main goals are cost-savings (time, money and other resources) and an acceleration of the standardization process, as well as the development of products based on those standards.

The proposed architecture supports different topological configurations of testing scenarios, including the use of testbeds deployed across the EU, such as Fed4FIRE[8], OneLab[9] and IoT-Lab[10]. A video demonstration of a Proof-of-Concept (PoC) CoAP interoperability testing tool is currently available at the project website[11].

[3]`http://openmobilealliance.org/`

[4]LWM2M is a Device Management protocol running on top of CoAP. More on `https://www.omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/`

[5]`http://coap.me/`

[6]`coap://coap.me/`

[7]`https://ec.europa.eu/programmes/horizon2020/`

[8]`https://www.fed4fire.eu/testbeds/`

[9]`https://onelab.eu/`

[10]`http://www.iotlab.eu/`

[11]`http://www.f-interop.eu/index.php/tools-experiments`

Robustness- and Security-related testing is not covered, and the tool currently requires the user to manually follow instructions from a GUI to perform test stimuli (e.g. using Firefox Copper to send a CoAP GET request with type CON). It also requires to manually perform input Verification & Validation (V&V) information (e.g. clicking a button to indicate success or failure of some aspect of a Server's response to a Client), providing no automation for test case execution. It does, however, compare the contents of the exchanged packets against an expected content, and present the dissected packets in the GUI, in case any further manual inspection is needed. The test specifications used are the ones from CoAP#4 Plugtests. It is also important to note that this is an ongoing research project, so further development in any of these functionalities can be expected.

Eclipse IoT-Testware

The scope of Eclipse IoT-Testware [19] is "to support conformance, interoperability, robustness and security testing of IoT devices and services via Testing and Test Control Notation version 3 (TTCN-3) test suites and cases."[12] TTCN-3[13], in turn, is a testing language used in conformance testing of communication systems, standardized by ETSI. Eclipse Titan[14] is an implementation of the TTCN-3 language consisting of a compilation and execution environment.

By using Titan—although it claims to be TTCN-3 tool independent—, Eclipse IoT-Testware aims to provide a set of TTCN-3 test suites and test cases for different IoT technologies, so application developers can have their own test environments. Started in June 2017, the initial contributions of the project focus on developing test cases for CoAP[15] and Message Queuing Telemetry Transport (MQTT) protocols. At the time of this writing, the project is still in an early phase, with a basic set of conformance test cases implemented, but still far from the ones from CoAP#4 Plugtests, for instance. Additionally, robustness and security test cases have not been implemented yet, and although an "iottestware.fuzzing"[16] repository was created inside the project, it is still empty, and no further details are available.

PEACH Fuzzer

Peach Fuzzer[17] is a commercial product, consisting of the Peach Fuzzer Platform and Peach Pits. The first is a fuzzing engine containing a GUI and capabilities for logging, monitoring the SUT and generating reports, including test coverage view. The second is a collection of prewritten test definitions, so the user can select a specification that fits the test target. It can be used to fuzz file-consuming applications, network protocols, drivers and so on. Combining automated test case generation and mutational fuzzing, it is called a "smart fuzzer".

A CoAP Peach Pit is available at [59], using up-to-date RFC7252 and outdated

[12]`https://projects.eclipse.org/proposals/eclipse-iot-testware`
[13]`http://www.ttcn-3.org/`
[14]`https://projects.eclipse.org/projects/tools.titan`
[15]`https://github.com/eclipse/iottestware.coap`
[16]`https://github.com/eclipse/iottestware.fuzzing`
[17]`http://www.peachfuzzer.com/products/peach-fuzzer/`

drafts for the Observe (draft-16) and Block-Wise Transfer (draft-17) as reference specifications. No additional information could be gathered about the use or efficiency of this fuzzer for CoAP targets; we tried to contact the company and obtain an Academic License to run it, for the sake of comparison, but those are not available for Peach Pits. There is an open source Community Version available as well, but the CoAP Peach Pit is not part of that distribution.

Defensics

Codenomicon's Defensics[18] is another commercial product used to perform fuzzing, initially developed as a span from the PROTOS research project [61]. It comes with a collection of generational model-based testing modules for more than 270 network protocols, file formats and other interfaces. Additionally, their tool provides reports and an online documentation on how to solve commonly-found problems linked by their test cases. The widely known OpenSSL's Heartbleed bug[19] was discovered while their team was improving a feature of this tool—not related to fuzzing itself, but to the discovery of failed cryptographic certificate checking, authentication bypass and privacy leaks.

A Test Suite targeting CoAP Servers is available at [16], referencing RFC7252 and RFC3986 [44] as used specifications. According to their website, it provides automated testing, ready-made test cases for CoAP Messages over UDP and UDP multicast, GUI and CLI modes and instrumentation for health-checking capability. We were not able to obtain an academic license for comparison purposes.

Table 2.2 contains a summary of the work presented in this section.

Table 2.2: Comparison of available CoAP Testing Solutions.

Project	**CoRE Reference Specification**				**Tool Support / Automated Testing**	**Robutness/Security Testing**
	Base	**Observe**	**Block**	**Link-Format**		
ETSI Plugtests	RFC7252	draft-12	draft-14	RFC6690	- Offline pcap analyzer - Automatic stimuli	-
F-Interop	RFC7252	draft-12	draft-14	RFC6690	- Real-time pcap analyzer - Manual stimuli[a]	-
IoT-Testware[b]	?	?	?	?	- Automated[c]	-
PEACH Fuzzer	RFC7252	RFC7641	draft-17	-	- Automated	Mutational Fuzzing
Defensics[d]	RFC7252	RFC7641	-	-	- Automated	Generational Fuzzing

[a] Plugtest clients such as `coap.me` and `cf-plugtest-client` could be used/integrated.
[b] Information was not found regarding the reference specifications used. It is safe to assume the authors will follow the latest specifications, though, since it is a recent project.
[c] Eclipse Titan's Executor component performs automated testing, but the actual IoT-Testware provides only the test cases.
[d] The CoAP Server test suite from Defensics also includes RFC 3986 - Uniform Resource Identifier.

[18]`http://www.codenomicon.com/products/defensics/`
[19]`http://heartbleed.com/`

2.3 Robustness/Security Testing through Fuzzing

The definition of robustness is “[the] degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions” [34]. Robustness testing, conversely, is a testing process carried out with the goal of characterizing the behavior of a system when exposed to those erroneous or exceptional conditions [4].

One commonly adopted form of robustness testing is interface error injection, which is a specific form of fault injection. This approach can be performed in two ways (Figure 2.10). The first is based on a test driver, which is a program responsible to exercise the target system or component with invalid inputs, either by using the target’s API or by sending network packets to it, for instance. The second approach is based on an interceptor, which is a program intercepting the communication between one system or component with the target, modifying/corrupting this communication (be it an API call or an HTTP request, for instance), and finally relaying it to the target [50]. In our work, even though we use the first approach, we also perform tests similar to the second approach, by modifying/corrupting previously captured packets and sending them to the target (see more in Section 3.3.2).

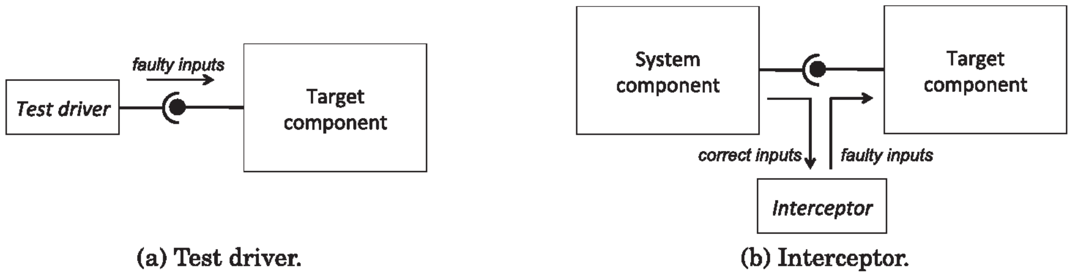


Figure 2.10: Approaches for interface error injection. From Natella et al. [50].

Additionally, the general approach for robustness testing revolves around the well-known model of fault $\Rightarrow$ error $\Rightarrow$ failure. This can be bridged to a security testing point of view by considering the specialized composite fault model introduced by [1] known as AVI (attack, vulnerability, intrusion), which “limits the fault space of interest to the composition (attack + vulnerability) $\Rightarrow$ intrusion” [51]. This leads to the notion of attack injection: by injecting interface errors consisting of potential attacks, we can monitor the target to detect signals of intrusions (through the manifestation of errors or failures), and trace this combination back to find potential vulnerabilities. Figure 2.11 illustrates the usage of this model applied to vulnerability discovery. One way to perform robustness (and, more specifically, security) testing by injecting such attacks is to use fuzzing—which is exactly what we do in this work.

In [55], Oehlert defines fuzzing as “a highly automated testing technique that covers numerous boundary cases using invalid data (...) as application input to better ensure the absence of exploitable vulnerabilities.” The author argues for the cost-effectiveness of fuzzing when compared to techniques such as functional testing; describes two ways of obtaining testing data for fuzzing—data generation and data mutation—; differentiate between intelligent fuzzers (those that leverage some knowledge of the target format)

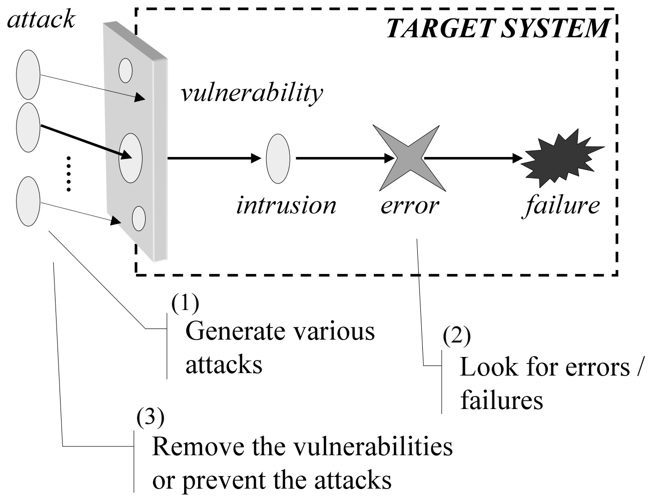


Figure 2.11: Using the composite fault model for vulnerability discovery. From Neves et al. [51].

and unintelligent fuzzers (those that, for instance, just randomly changes bits); discuss common fuzzing problems, such as the care to be taken regarding target formats using hashes or checksums, or the possible need to implement decompression and compression capabilities in the fuzzer, for formats using compression techniques; and the difficulties to check if a target application behaves correctly or not—suggesting the use of source code instrumentation and monitoring parameters, such as memory and CPU usage. In the following paragraphs we detail these concepts, presenting a brief review on the evolution of fuzzing techniques and domains in which it was applied.

One of the first studies in this area was done by Miller et al. [46]. The authors evaluated the robustness of UNIX utilities with regards to external inputs. They developed a tool called *Fuzz*, which they used to submit random data to those programs through the standard input, and were able to find that between 24% and 33% of utility programs in three UNIX systems were vulnerable to interface errors, crashing or stalling processes. This approach, which we simply call **Random Fuzzing** in this work, is the most basic and cheap one to perform fuzzing, although still capable to discover bugs [50]. The main issue with this approach is that, depending on the target being tested, it is very hard to actually find bugs without the message being instantly rejected by the System Under Test's (SUT) input parsing and sanitization mechanisms, because most of the times the message will not resemble a real protocol message or file format. This fostered research on ways to systematically generate better input streams of corrupted data, in order to enhance testing efficiency. In our work, we have implemented both a completely random generation technique as well as a slightly modified version of it, which we call **Informed Random Fuzzer**. More details are discussed in Section 3.3.1.

Another influential study in this area was done by Koopman et al., throughout more than 10 years and is better summarized in their 2008 paper [37]. The authors developed a tool called *Ballista*, using data-type-based error injection to test the robustness of POSIX-compliant OS APIs. Their approach is based on a set of predefined invalid or exceptional values for different data types. By passing these values to the system call interface (and observing the outcomes) they were able to assess the robustness of the APIs. Figure 2.12

shows an example of the input types and values used to test the `write` system call. These values were selected from the testing literature and the experience of developers. In our work we use a similar approach for the four option formats from CoAP (`empty`, `opaque`, `uint` or `string`) as well as for the data types in the CoAP header. The details of the values actually used in our system are further discussed in Chapter 3.

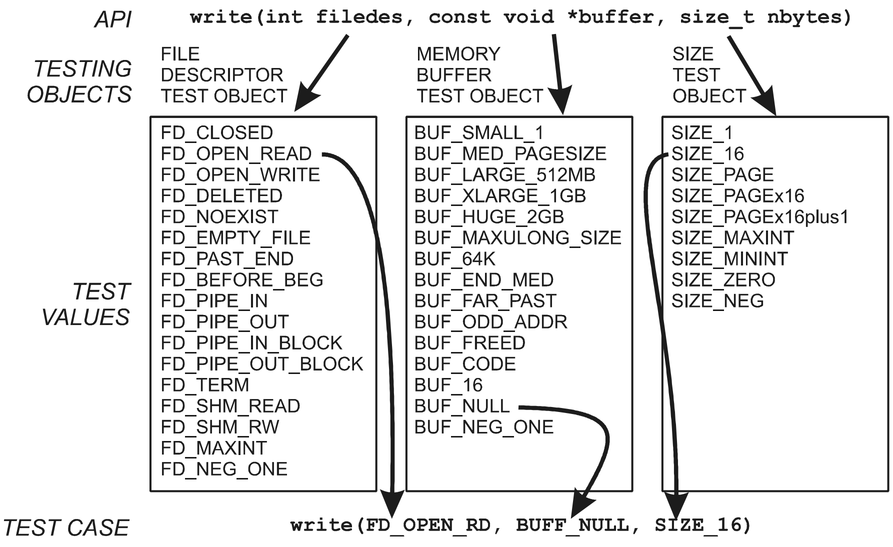


Figure 2.12: Ballista test case generation for the write() function. The arrows show a single test case being generated from three particular test values; in general, all combinations of test values are tried in the course of testing. From Koopman et al. [37].

With *Ballista*, the outcome of a test case (TC) is obtained by the error code returned by the system call and by a watchdog process responsible for monitoring the target and detecting unexpected termination or task hanging. The authors developed a *failure severity scale* called CRASH, classifying test results as:

***C*atastrophic** The OS state becomes corrupted or the machine crashes and reboots.

***R*estart** Task/process hangs and needs manual restart/force termination.

***A*bort** Task/process aborts, e.g. segmentation fault, signaling abnormal termination.

***S*ilent** No error code returned when one should be.

***H*indering** Misleading error code returned.

A more detailed description of the scale can be seen in [37]. This scale was later adapted to a Web Services environment by Vieira et al. [73], giving birth to the WSCRASH scale, and later further simplified by Laranjeiro et al. [42] to the WSAS scale, comprising only the failure modes easily observed (and distinguishable) from a Web Service's consumer point of view. In our work we use a similar scale, distinguishing the following failure modes:

***R*estart** SUT hangs and needs manual restart/force termination. Commonly caused by unhandled exceptions.

Abort SUT aborts. Commonly a segmentation fault. Our Process Monitor keeps all core dump files for further offline analysis.

Another well-known technique for data corruption is *bit-flipping*, used by Arlat et al. [6] in their *MAFALDA* tool for assessing the robustness of microkernels. Our **Mutational Fuzzer** (a.k.a. template fuzzing) uses this approach. We record good/valid CoAP conversations based on the CoAP#4 ETSI Plugtests [8], apply bit-flipping and byte-corruption functions to the captured packets, and finally replay them to the SUTs. This approach is format-agnostic, which means that the bit-flipping or byte-corruption functions do not have any knowledge of the target format being fuzzed.

Additionally, we developed what we call a **Smart Mutational Fuzzer**, which does have knowledge of the target format, and thus modifies the captured packets in a more "intelligent" manner, such as duplicating a CoAP Option field or modifying a string field to contain sensitive values such as non-printable characters. This smarter approach is based on predefined values associated to a particular data type, as discussed in [37], as well as other *data perturbation* techniques, such as the ones proposed by Offutt and Xu [56] (and later extended by De Melo and Silveira [18]), or by Vieira et al. [73]—all of them originally proposed for SOAP Web Services and adapted by us. We discuss it in details in Section 3.3.2.

For our last fuzzing engine, the **Generational Fuzzer** (a.k.a. model fuzzing, RFC- or Standards-based fuzzing), we build upon the ideas proposed by Kaksonen [36] during the development of the *PROTOS* project [61]. The authors extend the idea of domain/syntax testing, where input values are divided into input domains. These are ranges of values which are treated uniformly and thus can be used to generate test cases (making this approach a combination of equivalent class partitioning with data-type-based fuzzing). They also propose the use of attribute grammars to model the format of network protocols—similar to the *RIDDLE* library by Ghosh et al. [22], in which the authors adopt a grammar similar to the Backus-Naur Form (BNF) to generate syntactically correct, yet erroneous inputs.

These models are then used to generate test cases by creating exceptional data of particular attribute classes [3]. In our work, we use the Scapy[20] framework to model the CoAP protocol format and generate test cases based on attribute type and CoAP packet/message formats, such as filling out a Uri-Path option with a large, predefined string comprised only of `"\%"`, filling out the message ID field with values 0 and 65535 (maximum 16-bit unsigned integer) etc. More details are available in Section 3.3.3.

We note the increasing complexity regarding the aforementioned techniques, from the most basic purely random fuzzing, to a generational fuzzing scheme which is often equivalent in effort to writing a real implementation of a protocol or a format parser, passing through intermediates such as basic bit-flipping for mutational fuzzing to format-aware mutations, the latter having similar complexity to a generational fuzzing scheme. We also note the need to recalculate checksums (UDP checksums in our case) after mutating a previously captured packet/message template, in our case only needed for the mutational and smart mutational fuzzers.

[20]`https://github.com/secdev/scapy`

Additionally, we are not using source code instrumentation, as we consider that to be too much intrusive for a black-box approach intended to be used remotely. In turn, we monitor the SUTs by sending periodic health-checks (or heartbeats), listening to signals emitted by them and watching return codes. Finally, we are able to pin out which test cases failed (and information regarding the failure itself, with the corresponding error) by crossing this monitoring information with the SUT's output logs. We are also able to reproduce (offline) failed TCs by using packet information stored during the fuzzing campaign. A more detailed description of our system is given in Chapter 3.

Chapter 3

System Architecture and Testing Methodology

In this Chapter we present the FuzzCoAP tool, a complete test environment for CoAP server targets. The system architecture is proposed and, as the role of each entity of the system is detailed, we also describe our testing methodology. This is done by explaining the details of every step of the execution of a fuzzing campaign.

There are three basic entities in the architecture of our system: the *Process Monitor*, the *Fuzzer Controller* (a.k.a. *the Fuzzer*), and the *System Under Test (SUT)* itself. The Fuzzer Controller is composed by three fuzzing engines, each responsible for a different test case generation technique, namely *Random*, *Mutational* and *Generational* engines, plus a *Workload Executor*. Figure 3.1 shows this system architecture in a high level view. The role of each entity will be detailed in this chapter, as we also explain details of

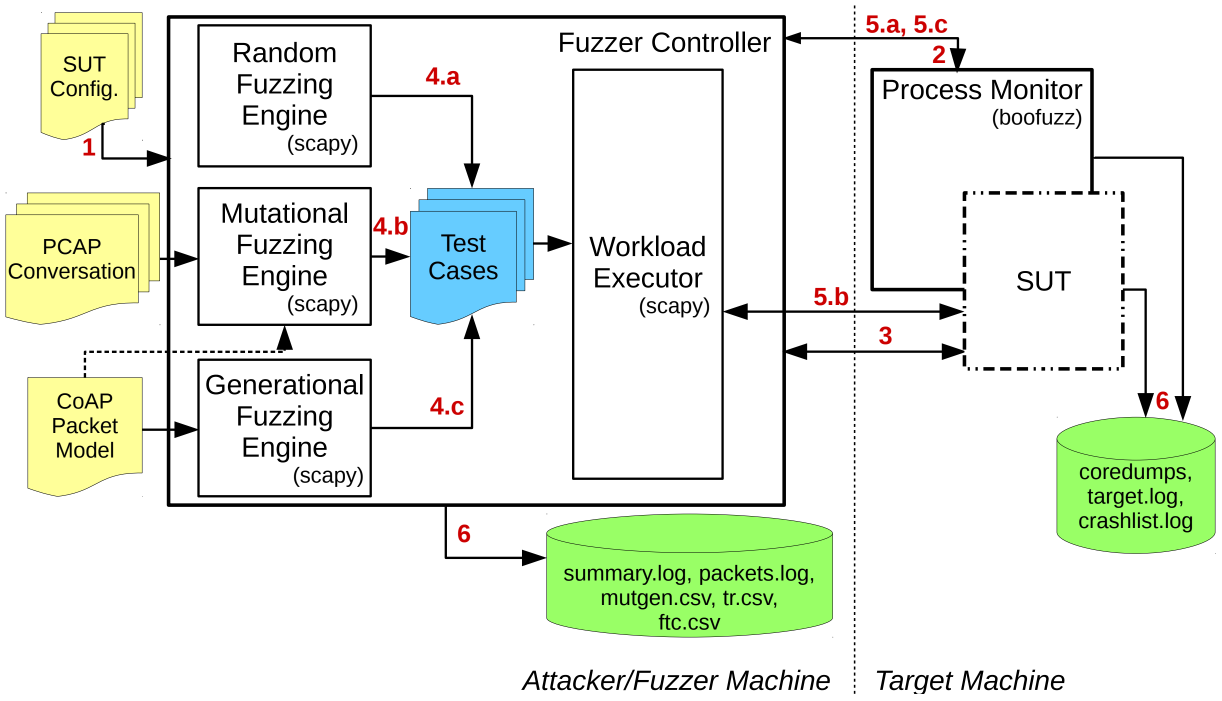

SUT Config.
1
PCAP Conversation
CoAP Packet Model
Fuzzer Controller
Random Fuzzing Engine (scapy)
Mutational Fuzzing Engine (scapy)
Generational Fuzzing Engine (scapy)
4.a
4.b
4.c
Test Cases
Workload Executor (scapy)
6
summary.log, packets.log, mutgen.csv, tr.csv, ftc.csv
5.a, 5.c
2
Process Monitor (boofuzz)
5.b
3
SUT
6
coredumps, target.log, crashlist.log
Attacker/Fuzzer Machine
Target Machine

Figure 3.1: Architecture of the testing system.

what happens during the execution of a fuzzing campaign—denoted, in sequence, by the red numbers in the figure. Although this architecture is influenced by Allen et al. [3], Neves et al. [51], and Antunes and Vieira [5], we should note that none of the approaches presented by these authors: i) detail the inner workings of the entity equivalent to the *Process Monitor*; and ii) allows the usage of multiple test case generation engines for comparison of different fuzzing techniques. Also, none of the tools developed in those works are available as Free or Open Source Software.

The first step (Figure 3.1-**step1**) is to fill out a configuration file with information from the target application—see Listing 1. This is a simple Python file (the whole system was developed in Python) where, for each target we want to support in the system, we add a python dictionary containing the following **m**andatory fields:

`dictionary name` the SUT name (an identifier, line 3);

`smart_cmd` the full command line to (re)start the SUT (lines 4–5);

`time_to_settle` the time, in seconds, to wait before considering the SUT up-and-running (line 9);

`heartbeat_path` the path of the resource to be used for heartbeats (line 10), which must be a resource for which the SUT always replies to a simple `GET` request.

```
1  BASE_DIR="/home/fuzz/apps"
2  #(...)
3  "libcoap-server": {
4      "start_cmd": "%s/libcoap/coap-server -A %s -p %d -v 9" %
5          (BASE_DIR, aut_host, aut_port), # M
6      "env": {
7          "ENV_VAR_1": "env_var_value",
8      }, # Opt.
9      "time_to_settle": 1, # M
10     "heartbeat_path": [("Uri-Path", ".well-known"), ("Uri-Path", "core")], # M
11     "default_uris": ["reg"], # Opt.
12     "bin_file": "%s/libcoap/coap-server" % (BASE_DIR), # Opt.
13     'strings': ['page', 'count', 'resource-param'], # Opt.
14 }, #(...)
```

Listing 1: *SUT Configuration File* - Example for `libcoap-server` application

Moreover, **opt**ional fields can be used to specify:

`env` environment variables to be exported before the SUT's execution (lines 6–8);

`default_uris` a list of default URIs the SUT exposes but does not list under `.well-known/core` (line 11), which the Fuzzer is unable to obtain automatically (see further in step3);

`bin_file` the binary file (ideally with debug symbols) used for offline, posterior analysis of generated core files through the GNU Debugger[1] `gdb` (line 12);

`strings` a list of user-supplied strings to be used during the test case generation for this specific SUT (line 13).

At this point, the *Fuzzer* can be started and **step2** in the process is triggered. In this step, the *Fuzzer* communicates with the *Process Monitor*, through Remote Procedure Calls (RPC), sending the relevant parameters from the SUT configuration file to it, so the *Process Monitor*, in turn, is able to properly start the SUT. This entity is described next.

3.1 Process Monitor

The *Process Monitor* is a standalone entity responsible for handling and monitoring the execution of the SUT itself. The entity was adapted from the Boofuzz[2] tool. It listens on a given TCP port and supports, among others, the following methods through RPC:

`alive` Checks if the connection between Fuzzer and Process Monitor is still alive;

`set_*` Sets SUT-specific parameters, such as the start command and environment variables;

`start_target` Starts the SUT by spawning it as a subprocess and monitoring its exit status in a new thread;

`stop_target` Stops the SUT, either by issuing a custom command or by using OS signals to force termination;

`restart_target` Restarts the SUT (`stop_target` followed by a `start_target`);

`pre_send` Ensures the SUT thread is operational before sending a test case. This will also add a mark to the output logs of the SUT, so we can later relate each test case, by its number, to a particular slice of that log file (see more in Section 3.4 and Listing 12);

`post_send` Returns the status of the SUT after a test case is received, so the *Fuzzer* can check if the target crashed—this is purely based on exit status and signal handling, and thus only able to detect an **A**bort failure in our scale. For detecting **R**estart failures we have implemented the healthcheck/heartbeat mechanism further described in Section 3.4 and Listing 9.

During actual test execution from a fuzzing capaign (step5.a through 5.c) the Process Monitor will be responsible for collecting core files generated by the SUT and saving them with a name that can be linked back to the test case that generated it (e.g. `TC_3125.dump`),

[1] `https://www.gnu.org/software/gdb/`
[2] `https://github.com/jtpereyda/boofuzz`

saving information regarding the exit status for each test case causing **A**bort failures at `crashlist.log`, and writing to `target.log`, together with the SUT itself. Details regarding the actual information collected in each of these files are presented in Section 3.4. Currently, it works for Unix processes only, although it could be extended to interface with hardware debuggers, in order to monitor the status of applications running on real IoT hardware (e.g. connected through USB or JTAG cables at the target machine).

3.2 Fuzzer Controller

The *Fuzzer Controller* (or *Fuzzer*) is the other standalone entity, and is composed by the test case generation engines plus a workload executor. This entity is responsible for controlling the actual testing process, by executing a fuzzing campaign. We will detail the role of each of its inner components in the next sections.

Before further diving into those details, regarding the test execution flow, at this point the SUT should be up and running. Before the test cases are generated, **step3** is triggered. At this step, the Fuzzer submits a `GET .well-known/core` request directly to the SUT, in order to obtain, among other things: a list of available links (a.k.a. paths, URIs or resources) the SUT exposes, as well as other relevant information about those links, e.g. if a given link supports the observe feature; which resource type it returns; which standard interface it follows, if any. For this single request we rely on the CoAPthon[3] library to easily perform a GET request using blockwise transfer instead of implementing block handling ourselves, since this list of resources, due to its possible large size, is commonly split up in blocks. The response to this message is parsed and produces two lists, one containing the extracted paths and the other containing the extracted strings. Both lists are later used by the test generation engines—the general idea is to increase the chance of reaching deeper levels in the SUT's code by targeting resources it actually supports and fuzzing/filtering through strings formed by key-pair values it actually has, among others. We revisit this idea with concrete examples and further explanation in Section 3.3. Listing 2 shows an example for the `libcoap-server` application.

```
# Payload of the CoAP message received
# in response to the GET ./well-known/core request
</>;title="General Info";ct=0,</time>;if="clock";rt="Ticks";title="Internal
↪  Clock";ct=0;obs,</async>;ct=0

# Output lists
Extracted Paths:
['async', '.well-known/core', 'time']
Extracted Strings:
['rt="Ticks"', 'title="Internal Clock"', 'ct=0', 'title="General Info"', 'obs',
↪  'if="clock"']
```

Listing 2: Information extracted in **step3**

[3]`https://github.com/Tanganelli/CoAPthon`

3.3 Test Case Generation Engines

The Fuzzer Controller has three test case generation engines, each one focused on a different way of obtaining test data for fuzzing: random data (**step4.a**), data mutation (**step4.b**) and data generation (**step4.c**). In this section we describe the inner workings of each of those engines.

3.3.1 Random Fuzzing

Random data is the easiest, cheapest one to be obtained for fuzz testing. The Random Fuzzing engine uses Python's `random` module—which in turn relies on an OS-specific source of randomness; in the case of our experiments, Linux's `/dev/random`—to generate test data. Inside our engine, we actually generate test data in two ways: a purely random one, which we call *Random Fuzzer* and a slightly smarter one, which we call *Informed Random Fuzzer*. We describe each of those next.

Random Fuzzer

The *Random Fuzzer* generates test cases consisting of raw UDP packets with random bytes. Half the packets have their lengths randomly chosen to be no bigger than 2176 bytes, as it is the suggested upper bound for CoAP messages (see Section 4.6 from RFC7252 [69]), and the other half uses boundary (or "singular") values around important powers-of-two from 0 (empty UDP datagram) until 65507, which is the limit for the data length of a UDP datagram. The idea behind this is that, although the data itself is random, these data lengths are more inclined to reveal potential off-by-one, integer overflow and buffer overflow errors. Hence, given these two generators, we can configure a parameter K_{random} to obtain the number of test cases generated for a random fuzzing campaign:

$$N_{\text{TCs}}^{\text{random}} = 2 \cdot K_{\text{random}} \tag{3.1}$$

Listing 3 contains the possible lengths used for the second half of the test cases. We present an example packet generated, as dissected by Wireshark[4], in Listing 4.

```
[0, 1, 2, 3, 4, 5, 7, 8, 9, 15, 16, 17, 31, 32, 33, 63, 64, 65, 127, 128, 129, 255,
↪  256, 257, 511, 512, 513, 1023, 1024, 1025, 2047, 2048, 2049, 4095, 4096, 4097,
↪  8191, 8192, 8193, 16383, 16384, 16385, 32752, 32753, 32754, 32767, 32768, 32769,
↪  65506, 65507]
```

Listing 3: 50 "singular" values are used as possible lengths of the UDP datagrams.

Informed Random Fuzzer

The *Informed Random Fuzzer* is a modification of the *Random Fuzzer* which uses the path information gathered in step3 (see Listing 2). The idea is to improve the odds of a

[4]`https://www.wireshark.org/`

```
Constrained Application Protocol
    10.. .... = Version: 2
    ..01 .... = Type: Non-Confirmable (1)
    .... 0001 = Token Length: 1
    Code: Unknown (147)
[Malformed Packet: CoAP]
# Packet as Hex + ASCII Dump:
0000   91 93 f3                                         ...
```

Listing 4: Example of a *Random Fuzzer* test case consisting of a 3-byte UDP packet. Note that since a CoAP header has 4 bytes, this is marked by Wireshark as a malformed CoAP packet.

test case actually reaching parts of the SUT responsible for handling the requests for a given resource/URI. Instead of raw UDP packets, the test cases generated by this engine are actual CoAP packets, although not necessarily valid (i.e. the version field can still be invalid, the code field can still carry a response code instead of a request method etc.). Three types of packets are generated, for which the target CoAP resources are chosen randomly, in an uniformly distributed manner:

Header Only Random bytes at the header fields, accompanied by CoAP Uri-Path options carrying the path to one of the resources obtained in step3. An example is shown in Listing 5.

```
Constrained Application Protocol, Acknowledgement, Unknown 253, MID:11505
    00.. .... = Version: 0
    ..10 .... = Type: Acknowledgement (2)
    .... 0001 = Token Length: 1
    Code: Unknown (253)
    Message ID: 11505
    Token: 8b
    Opt Name: #1: Uri-Path: separate
        Opt Desc: Type 11, Critical, Unsafe
        1011 .... = Opt Delta: 11
        .... 1000 = Opt Length: 8
        Uri-Path: separate
# Packet as Hex + ASCII Dump:
0000   21 fd 2c f1 8b b8 73 65 70 61 72 61 74 65        !.,...separate
```

Listing 5: Example of a *Header Only* test case from the *Informed Random Fuzzer*. Consisting of the 4-byte header, a 1-byte token and a 9-byte option, it was generated and sent to the `'separate'` URI, obtained in step3 for the yacoap-piggyback application.

Empty Payload Similar to the previous one, but in addition to the Uri-Path options carrying the target resource URI, it includes random bytes where other CoAP options should be. This slice of "garbage options" can range from 0 to 65503 bytes, using similar values as those from Listing 3—the last two values in the list becomes actually smaller, since this is not a UDP raw packet, but instead contains the additional CoAP header (4 bytes) and the Uri-Path options carrying the target path. We present an example in Listing 6.

```
Constrained Application Protocol, Reset, Unknown 37, MID:25820
    01.. .... = Version: 1
    ..11 .... = Type: Reset (3)
    .... 1011 = Token Length: 11
    Code: Unknown (37)
    Message ID: 25820
    Token: a891603e9d10afc8eb3fe4
    Opt Name: #1: Uri-Path: separate
    [Expert Info (Warning/Malformed): Invalid Option Number 2201]
    Opt Name: #2: Unknown Option: 52 31 21 96 ae 13
    [Expert Info (Warning/Malformed): option longer than the package]
# Packet as Hex + ASCII Dump:
0000   7b 25 64 dc a8 91 60 3e 9d 10 af c8 eb 3f e4 b8  {%d...`>.....?..
0010   73 65 70 61 72 61 74 65 e6 07 81 52 31 21 96 ae  separate...R1!..
0020   13 d8 da e2 5f 75 f7 46                          ...._u.F
```

Listing 6: Example of an *Empty Payload* test case from the *Informed Random Fuzzer*. We can see that in addition to the `'separate'` Uri-Path option, it has a slice of 16 random bytes where more CoAP options should be. In this specific case, Wireshark is not even able to correctly dissect the complete packet.

Random Payload Also similar to the first one, but instead of adding a slice of "garbage options", the only options present are still the CoAP Uri-Path options carrying the target resource URI, followed by a *payload marker* (`0xFF`), then finally followed by a payload consisting of 0–65503 random bytes. The length of this payload is determined analogously to the length of the "garbage options" slice from the previous type of test case. Listing 7 shows an example.

```
Constrained Application Protocol, Confirmable, Unknown 169, MID:23948
    00.. .... = Version: 0
    ..00 .... = Type: Confirmable (0)
    .... 0101 = Token Length: 5
    Code: Unknown (169)
    Message ID: 23948
    Token: 331008db58
    Opt Name: #1: Uri-Path: piggyback
    End of options marker: 255
    Payload: Payload Content-Format: text/plain; charset=utf-8 (no Content-Format)
        Payload Desc: text/plain; charset=utf-8
        [Payload Length: 31]
Line-based text data: text/plain
    \301\204\246\236j6YB 9\316\034\206;\332NM\323<\214$\235\234}\243e\343\234 %/
# Packet as Hex + ASCII Dump:
0000   05 a9 5d 8c 33 10 08 db 58 b9 70 69 67 67 79 62  ..].3...X.piggyb
0010   61 63 6b ff c1 84 a6 9e 6a 36 59 42 20 39 ce 1c  ack.....j6YB 9..
0020   86 3b da 4e 4d d3 3c 8c 24 9d 9c 7d a3 65 e3 9c  .;.NM.<.$..}.e..
0030   20 25 2f                                          %/
```

Listing 7: Example of a *Random Payload* test case from the *Informed Random Fuzzer*. In this case, it was sent to another URI from the same SUT (`'piggyback'`), and we can see it has a 31-byte random payload.

Therefore, given these three generators, we can configure a parameter $K_{\text{inf-random}}$ to obtain the number of test cases generated for an informed random fuzzing campaign:

$$N_{\text{TCs}}^{\text{inf-random}} = 3 \cdot K_{\text{inf-random}} \tag{3.2}$$

3.3.2 Mutational Fuzzing

Inside our *Mutational Fuzzing* engine we also generate test data in two ways: one based on bit-flipping and byte-corruption, the most straightforward techniques for mutating previously obtained data, which we simply call *Mutational Fuzzer*; and a slightly smarter one, based on more careful data perturbation techniques and boundary values, as discussed in Section 2.3, which we call *Smart Mutational Fuzzer*.

As the primary input for both fuzzers, we need initial data before we can actually mutate it. In our system, this initial data should be a set of valid CoAP requests. Common good candidates are messages from real-world interactions between a working CoAP client and a given CoAP server, or messages used as part of functional test cases—which we can assume were already designed to exercise at least some boundary or negative cases, besides the most common or expected positive cases.

We adopt the second approach, by using wireshark to capture the interactions consisting of actual CoAP packets implementing the test descriptions defined by Bormann [8] for the CoAP#4 ETSI Plugtest, as discussed in Section 2.2. These functional/interoperability-focused test cases used were the ones implemented in the Californium[5] CoAP library. By running one of the tools distributed with the library, called `cf-plugtest-client`, we can exercise all test cases from CoAP#4 against a given CoAP server, and thus capture all packets using wireshark, so they can later be used as inputs by our mutational engine—this is the *PCAP Conversation* represented in Figure 3.1. Based upon this data we can finally generate our test cases for fuzzing. In the next sections we describe each mutation approach used.

Mutational Fuzzer

As mentioned before, the bit-flipping and byte-corruption strategy is rather straightforward. The *Mutational Fuzzer* loads up the previously captured *PCAP Conversation* and, for each packet in there, it generates a user-defined fixed number (K_{mut}) of test cases using Scapy's `CorruptedBits` and `CorruptedBytes` classes. The first one *"flips a given percentage or number of bits in the packet"* (using a basic bitwise-xor operation), while the second one *"corrupts a given percentage or number of bytes from the packet"* (adding a random number between 1 and 255 to the original byte, modulo 256 to avoid overflow). We are using the default of 1% for both cases, meaning 1% of the packet's bits will be flipped or 1% of its bytes will be corrupted. Therefore, given these two mutation strategies and the number of packets in the *PCAP Conversation* ($N_{\text{pkts}}^{\text{PCAP}}$), we can obtain the number of generators (or templates) produced:

$$N_{\text{templates}}^{\text{mut}} = 2 \cdot N_{\text{pkts}}^{\text{PCAP}} \tag{3.3}$$

And, from this, the number of test cases generated for a mutational fuzzing campaign:

$$N_{\text{TCs}}^{\text{mut}} = K_{\text{mut}} \cdot N_{\text{templates}}^{\text{mut}} \tag{3.4}$$

[5]`https://www.eclipse.org/californium/`

```
# Original packet:
Constrained Application Protocol, Confirmable, GET, MID:750
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 1000 = Token Length: 8
    Code: GET (1)
    Message ID: 750
    Token: 0c443f4fa4c9c990
    Opt Name: #1: Etag: 17 29
    Opt Name: #2: Uri-Path: validate
# Original packet as Hex + ASCII Dump:
0000   48 01 02 ee 0c 44 3f 4f a4 c9 c9 90 42 17 29 78  H....D?O....B.)x
0010   76 61 6c 69 64 61 74 65                          validate

# Packet generated by bit-flipping:
Constrained Application Protocol, Confirmable, GET, MID:750
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 1000 = Token Length: 8
    Code: GET (1)
    Message ID: 750
    Token: 0c443f4fa4c9c990
    Opt Name: #1: Content-Format: Unknown Type 5929
    [Expert Info (Warning/Malformed): Invalid Option Number 19]
    Opt Name: #2: Unknown Option: 76 61 6c 69 64 61 74 65
# Bit-flipping packet as Hex + ASCII Dump:
0000   48 01 02 ee 0c 44 3f 4f a4 c9 c9 90 c2 17 29 78  H....D?O......)x
0010   76 61 6c 69 64 61 74 65                          validate

# Packet generated by byte-corruption:
Constrained Application Protocol, Confirmable, GET, MID:750
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 1000 = Token Length: 8
    Code: GET (1)
    Message ID: 750
    Token: 0c443f4fa4c9c990
    Opt Name: #1: Etag: 17 29
    Opt Name: #2: Uri-Path: va\037idate
        Opt Desc: Type 11, Critical, Unsafe
        0111 .... = Opt Delta: 7
        .... 1000 = Opt Length: 8
        Uri-Path: va\037idate
# Byte-corruption packet as Hex + ASCII Dump:
0000   48 01 02 ee 0c 44 3f 4f a4 c9 c9 90 42 17 29 78  H....D?O....B.)x
0010   76 61 1f 69 64 61 74 65                          va.idate
```

Listing 8: Example of one CoAP packet obtained from the plugtest set and two test cases derived from it, generated by the *Mutational Fuzzer*. The mutation highlighted in yellow was made by the bit-flipping technique, turning `0x42 (0100 0010)` into `0xc2 (1100 0010)`. The other mutation, highlighted in cyan, was made by the byte-corruption technique and turned a `0x6c (0110 1100)` into a `0x1f (0001 1111)`.

Listing 8 shows an example of an original packet from the PCAP Conversation and two possible mutations obtained from it, one using bit-flipping and the other using byte-corruption. We note how wireshark's interpretation of the packet changes drastically even when a single bit is flipped (the most significant bit (MSB) of the byte in position 12, changing it from `0x42` to `0xC2`, highlighted in yellow), in case that bit is located in an important protocol field—here, the Option Delta. This shows the relevance of bit-flipping, despite its simplicity, especially when dealing with binary-encoded formats or protocols, as is the case with CoAP. The second mutation, through byte-corruption, did not change wireshark's interpretation so drastically, but a non-printable character (`0x1F`, octal `037`, known as Unit Separator) was inserted in the middle of the string representing the target URI (highlighted in cyan), which definitely represents an exceptional case to be handled by the SUT.

Smart Mutational Fuzzer

The *Smart Mutational Fuzzer* yields test cases based on the same collection of packets (*PCAP Conversation*), but instead of using basic bit-flipping or byte-corruption, it leverages the knowledge of the CoAP protocol format to mutate specific option values or entire fields. For every option in each base packet inside that collection, this engine will produce a number of test cases based on the option format, following the rules from Table 3.1 and Table 3.2. The rules presented in Table 3.1 are applied to a parameter value, such as a CoAP option value or the value of the payload field. Table 3.2, on the other hand, presents rules applied to a whole field, such as a CoAP option (including delta, length, extended and value fields) or the pair paymark + payload. Table 3.3 shows examples of possible mutations for a string parameter, and Table 3.4 shows examples of possible mutations for the uint format.

To illustrate, if this engine finds a packet containing an Uri-Host (string) plus an Uri-Port (uint) option, with no payload, it will generate $8 + 9 + 2 \cdot 3 = 23$ test cases. In addition to those, it will also generate targeted test cases, which are modified versions of the TCs originally generated, containing Uri-Path options targeting a known-path (randomly chosen) of the SUT, as obtained in step3. Hence, the single packet with two options and no payload from the previous example would yield $2 \cdot 23 = 46$ test cases. The idea with the targeted test cases is similar to that mentioned for the *Informed Random Fuzzer*: improving the odds of a TC actually reaching request handlers.

Therefore, given these mutation rules, we can obtain the number of test cases generated for a smart mutational fuzzing campaign:

$$N_{\text{TCs}}^{\text{smart-mut}} = 2 \cdot \Bigg[\overbrace{8 \cdot N_{\text{opts}}^{\text{string}} + 6 \cdot N_{\text{opts}}^{\text{opaque}} + 9 \cdot N_{\text{opts}}^{\text{uint}} + 7 \cdot N_{\text{opts}}^{\text{empty}} + 7 \cdot N_{\text{pkts}}^{\text{payload}}}^{\text{parameter mutations}} + \underbrace{+\,3 \cdot \left(N_{\text{opts}}^{\text{string}} + N_{\text{opts}}^{\text{opaque}} + N_{\text{opts}}^{\text{uint}} + N_{\text{opts}}^{\text{empty}} + N_{\text{pkts}}^{\text{payload}} \right)}_{\text{field mutations}} \Bigg] \tag{3.5}$$

where:

$$\begin{aligned} N_{\text{opts}}^{\text{string}} &= \text{Number of string options across all packets from the } \textit{PCAP Conversation} \\ N_{\text{opts}}^{\text{opaque}} &= \text{Number of opaque options across all packets from the } \textit{PCAP Conversation} \\ N_{\text{opts}}^{\text{uint}} &= \text{Number of uint options across all packets from the } \textit{PCAP Conversation} \\ N_{\text{opts}}^{\text{empty}} &= \text{Number of empty options across all packets from the } \textit{PCAP Conversation} \\ N_{\text{pkts}}^{\text{payload}} &= \text{Number of packets from the } \textit{PCAP Conversation} \text{ containing a payload} \end{aligned}$$

The constants from Equation 3.5, in order of appearence, are related to: generation of both untargeted and targeted TCs (2), number of string mutation rules (8), number of opaque mutation rules (6), number of uint mutation rules (9), number of empty mutation rules (7), number of payload mutation rules (7) and number of field mutation rules (3).

Table 3.1: Parameter mutation rules used by the *Smart Mutational Fuzzer*. Adapted from Vieira et al. [73].

Format	**Test Name**	**Parameter Mutation**
string	StrEmpty	Replace by empty string
	StrPredefined (5)	Replace by predefined string[a]
	StrAddNonPrintable	Add nonprintable characters to the string
	StrOverflow	Add characters to overflow max size
opaque	OpaqueEmpty	Replace by empty binary string
	OpaquePredefined (4)	Replace by predefined binary string[b]
	OpaqueOverflow	Add characters to overflow max size
uint	UintNull	Replace by null value
	UintAbsoluteMinusOne	Replace by -1
	UintAbsoluteOne	Replace by 1
	UintAbsoluteZero	Replace by 0
	UintAddOne	Add one
	UintSubtractOne	Subtract 1
	UintMaxRange	Replace by maximum value valid for the parameter
	UintMinRange	Replace by minimum value valid for the parameter
	UintMaxRangePlusOne	Replace by maximum value valid for the parameter plus one
empty	EmptyPredefined (4)	Replace by predefined [binary] string[c]
	EmptyAbsoluteMinusOne	Replace by -1
	EmptyAbsoluteOne	Replace by 1
	EmptyAbsoluteZero	Replace by 0
payload	PayloadEmpty	Replace by empty payload string
	PayloadPredefined (5)	Replace by predefined payload string[d]
	PayloadAddNonPrintable	Add nonprintable characters to the payload string

[a] Random amount between the specific option's minimum and maximum length, of either one of the following characters or character sequences: `‘\x00’`, `‘8’`, `‘#’`, `‘U+1F60D’`, `‘%’`

[b] Same, but for the following: `‘\x00’`, `‘\xff’`, `‘U+1F60D’`, `‘%’`

[c] One of the following: `‘\xff’`, `‘#’`, `‘U+1F60D’`, `‘%’`

[d] Random amount between 1 and 65502 of either one of the following: `‘\x00’`, `‘\xff’`, `‘#’`, `‘U+1F60D’`, `‘%’`

Table 3.2: Field mutation rules used by the *Smart Mutational Fuzzer*.

Test Name	**Field Mutation**
FieldNull	Replace by null value
FieldRemove	Remove field from the packet
FieldDuplicate	Duplicate field from the packet

Table 3.3: *Smart Mutational Fuzzer* - String parameter example

Original String Option - Uri-Host: ‘`sensors.example.com`’	
Test Name	**Mutated Parameter**
StrEmpty	`''`
StrPredefined_\x00	`'\x00\x00\x00'`
StrPredefined_8	`'8888888888'`
StrPredefined_#	`'#####...' (len=255)`
StrPredefined_U+1F60D	`'U+1F60D'`
StrPredefined_%	`'%%%%'`
StrAddNonPrintable	`'sensors.example.com\x7f'`
StrOverflow	`'sensors.example.com%%%%...' (len=256)`

Table 3.4: *Smart Mutational Fuzzer* - Uint parameter example

Original Uint Option - Accept: 50	
Test Name	**Mutated Parameter**
UintNull	`null`
UintAbsoluteMinusOne	-1
UintAbsoluteOne	1
UintAbsoluteZero	0
UintAddOne	51
UintSubtractOne	49
UintMaxRange	65535
UintMinRange	0
UintMaxRangePlusOne	65536

3.3.3 Generational Fuzzing

The third and last engine is responsible for the *Generational Fuzzer*. It works by producing value generators (or templates), based on the *CoAP Packet Model*—a description of the protocol format using Python and Scapy—, which in turn generates test cases following predefined rules. These value generators can be classified as format-based (Table 3.5), option-based (Table 3.6) or message-based (Table 3.7). Format-based generators yield data based on a given CoAP Option format (string, opaque, uint, empty), option-based generators yield data based on a specific CoAP Option type (e.g. Uri-Query and Location-Query options expect values in a `key=value` format, where the `key`, in general, assumes a known attribute name), and message-based ones generate data based either on a CoAP header field or the entire packet.

Besides the aforementioned protocol-based classification, value generators can be classified regarding the data generation mode as well: *Random*, which generates random data (of a particular format, option or header field, for instance), and *Singular*, which generates special data instead (e.g. a 4096-byte string composed only by '%' characters, an Uri-Query with value `title=\x00`, or a message id with value 65535). The *Singular* mode basically encompasses boundary, exceptional and carefully-crafted, focused values, which may in turn be based on the parameter format (Table 3.5), the CoAP option (Table 3.6) or the CoAP message itself (Table 3.7).

In addition to the value generators, we have the packet templates. A packet template can generate actual CoAP packets, with mostly valid header fields (i.e. the version field is always `1`, type is always `CON` or `NON`, token always has 0–8 bytes etc.). There are 5 types of packet templates:

A - PacketSingular Mostly valid header with a slice of “garbage bytes” composed by a singular string, which can span through the CoAP Options all the way to the payload field.

B - PacketPayloadEmpty Similar to the previous one, but limits the “garbage bytes” to the options fields, ensuring an empty payload.

C - PacketPayloadSingular Similar to the previous one, but instead of ensuring an empty payload, it ensures a payload is present and is composed by a singular string.

D - PacketPayloadEmptyPathKnown Similar to the PacketPayloadEmpty, but includes Uri-Path options carrying the path to one of the resources obtained in step3. It can also be seen as a specialization of the “Empty Payload” packet type from the *Informed Random Fuzzer*, which, instead of carrying a random slice of “garbage options”, carries a singular one.

E - PacketPayloadSingularPathKnown Similar to the PacketPayloadSingular, with the addition of the Uri-Path options as well. It can also be seen as similar to the “Random Payload” of the *Informed Random Fuzzer*, but instead of a random one, it carries a payload comprised of a singular string.

The idea with the combination of these generators/templates is to gradually achieve better odds of reaching critical parts of the SUT, with critical data. For instance, a StrRandom value when used with packet templates A, B and C are just random values within an untargeted packet. In conjunction with packet templates D and E, although the generated option value will still be random, it will in turn then be targeted towards a known Uri-Path from the SUT and, finally, by combining a StrSingular value with packet templates D and E, we get specially-crafted (commonly boundary or known to cause issue) values sent towards known Uri-Paths. The same reasoning applies to option-based value

Table 3.5: *Generational Fuzzer* - Format-based generation rules.

Format	Test Name	Parameter Generation	Templates Used
string	StrRandom	Random string with random length (min, max)	A, B, C, D, E
	StrSingular	Singular string with singular lengths (min, max)	D, E
opaque	OpaqueRandom	Random binary string with random length (min, max)	A, B, C, D, E
	OpaqueSingular	Singular binary string with singular lengths (min, max)	D, E
uint	UintRandom	Random number (min, max)	A, B, C, D, E
	UintSingular	Singular number (min, max)	D, E
empty	EmptyRandom	Random number (min, max)	A, B, C, D, E
	EmptySingular	Singular number (min, max)	D, E

Table 3.6: *Generational Fuzzer* - Option-based generation rules. All tests are constructed using packet templates D and E.

Option Type	Test Name	Parameter Generation
Uri-Query, Location-Query	QueryStrRandom	String formed by a valid query attribute name with random attribute value. The value has singular lengths (min, max).
	QueryStrSingular	String formed by a valid query attribute name with singular attribute value. The value has singular lengths (min, max).
Uri-Port	PortSingular	Well-known port numbers, such as 53, 8080 and 61615–61633.
Content-Format, Accept	ContentFormatSingular	Well-known content format identifiers, such as 41, 50 and 60.
Block1, Block2	BlockSingular	Blocks (uints) formed by a singular number concatenated with all numbers between 0 and 15.

generators in comparison with format-based ones: they are specially-crafted not only with regards to that options' format, but to its semantics as well—and, additionaly, are always combined with packet templates D and E for a targeted approach.

Therefore, given these generation rules, we can obtain the number of generators produced for a given option:

$$N^o_{\text{gen}} = \overbrace{3 + 2 \cdot N^{\text{SUT}}_{\text{paths}}}^{\text{format random}} + \overbrace{2 \cdot N^{\text{SUT}}_{\text{paths}}}^{\text{format singular}} + \overbrace{B^o_{\text{rand\_t}} \cdot 2 \cdot N^{\text{SUT}}_{\text{paths}}}^{\text{option random}} + \overbrace{B^o_{\text{sing\_t}} \cdot 2 \cdot N^{\text{SUT}}_{\text{paths}}}^{\text{option singular}} \tag{3.6}$$

where:

$$O = \{\, o \mid o \text{ is a CoAP Option type} \,\}$$
$$S = \{\, \text{Uri-Query}, \text{Location-Query}, \text{Uri-Port}, \text{Content-Format}, \text{Accept}, \text{Block1}, \text{Block2} \,\}$$
$$S' = \{\, \text{Uri-Query}, \text{Location-Query} \,\}$$
$$B^o_{\text{rand\_t}} = \begin{cases} 1, & \text{if } o \in S' \\ 0, & \text{otherwise} \end{cases}, \qquad B^o_{\text{sing\_t}} = \begin{cases} 1, & \text{if } o \in S \\ 0, & \text{otherwise} \end{cases}$$
$$N^{\text{SUT}}_{\text{paths}} = \text{Number of paths obtained for the SUT in step3}$$

And, from there, the number of generators produced in a given generational fuzzing

Table 3.7: *Generational Fuzzer* - Message-based generation rules.

Protocol Field	**Test Name**	**Parameter Generation**	**Templates Used**
All	AllFieldsSingular	Singular binary string with singular lengths (min, max). The slice occupied by this string depends on the Packet Template.	A, D, E
Message ID	MIDSingular	Singular number (min, max)	E
Token	TokenSingular	Singular binary string with singular lengths (min, max)	E

campaign:

$$N_{\text{gen}}^{\text{gen}} = \overbrace{5}^{\text{message generators}} + \sum_{\forall o \in O} N_{\text{gen}}^{o} \tag{3.7}$$

Furthermore, in order to obtain the number of actual test cases generated for a given campaign, we need to consider that, for the format-based generators, the number of singular strings, binary strings or numbers produced (K_{format}^{o}) depends on the given CoAP option, since it is based on two things: that specific option's minimum and maximum lengths, which can be seen in Table 2.1, and the number of user-defined crafted elements of a given option format. Besides, the number of random ($K_{\text{opt\_r}}^{o}$) or singular ($K_{\text{opt\_s}}^{o}$) strings, binary strings or numbers produced by the option-based generators also depends on the specific option, since it is based on special values relevant for that given option only (e.g. well-known port numbers are only relevant for Uri-Port, while `key=value` strings are relevant only for Uri-Query and Location-Query options). Thus, we define the following constants:

$$\begin{aligned} K_{\text{gen}} &= \text{User-defined fixed number of TCs for the <Format>Random generators} \\ K_{\text{format}}^{o} &= \text{Number of <Format>Singular elements produced for option } o \in O \\ K_{\text{opt\_r}}^{o} &= \text{Number of <Option>Random elements produced for option } o \in S' \\ K_{\text{opt\_s}}^{o} &= \text{Number of <Option>Singular elements produced for option } o \in S \end{aligned}$$

And, by plugging these constants in Equation 3.6, we get the number of test cases generated for a given option:

$$N_{\text{TCs}}^{o} = K_{\text{gen}} \cdot \overbrace{(3 + 2 \cdot N_{\text{paths}}^{\text{SUT}})}^{\text{format random}} + K_{\text{format}}^{o} \cdot \overbrace{(2 \cdot N_{\text{paths}}^{\text{SUT}})}^{\text{format singular}} + K_{\text{opt\_r}}^{o} \cdot \overbrace{(2 \cdot N_{\text{paths}}^{\text{SUT}})}^{\text{option random}} + K_{\text{opt\_s}}^{o} \cdot \overbrace{(2 \cdot N_{\text{paths}}^{\text{SUT}})}^{\text{option singular}}$$

From which, finally, we can obtain the total number of test cases generated for a given generational fuzzing campaign:

$$N_{\text{TCs}}^{\text{gen}} = \overbrace{K_{\text{gen}}^{\text{all}} \cdot 3 + 49 + 603}^{\text{message TCs}} + \sum_{\forall o \in O} N_{\text{TCs}}^{o} \tag{3.8}$$

where

$$K_{\text{gen}}^{\text{all}} = \text{User-defined fixed number of TCs for the AllFields generators}$$

3.4 Workload Executor, Gathered Information & Offline Analyzers

After one of the available engines generates the test cases in step4, **step5** can be started. This step is mainly performed by the *Workload Executor*. At this step, the general workflow is to: **a)** signal a new incoming TC to the *Process Monitor*; **b)** send the actual packet consisting of that TC to the SUT; and **c)** through the Process Monitor, check the status of the SUT after receiving the TC and take the necessary measures to be able to proceed and send the next TC. Throughout this step numerous information are gathered and stored in files for a posterior, offline analysis. In this section we detail this workflow, how failures are detected, which information is gathered and how they are used.

3.4.1 Test Case Execution and Failure Detection Mechanisms

The first two substeps are straightforward. To signal a new incoming test case to the Process Monitor (**step5.a**), we call its `pre_send` RPC method (mentioned in Section 3.1), which just ensures the SUT thread is alive and marks the SUT's output log (`target.log` in Figure 3.1). These marks are later used by the offline analyzer to automatically relate each TC number to a particular slice of logs produced by that TC, in order to obtain information regarding a particular possible failure and related error (see more, with examples, in Section 3.4.2). Then, to send the actual packet (**step5.b**), we simply use Scapy to assemble the generated packet, send it over the network, and wait for a response during a given timeout interval. This response (if present) is used during the third and last substep.

```
 1  ProcMon.post_send():
 2    # Checks the SUT subprocess status
 3    if not is_alive():
 4      # SUT is dead, fill out report...
 5      crashlist.write(tcNo, exitStatus)
 6      # ... And save core file if available
 7      save_coredump(tcNo, corefile)
 8    return is_alive()
 9
10  # exitStatus examples:
11  #     'Segmentation fault'
12  #     'Stopped with signal <SIGNAL>'
13  #     'Terminated with signal <SIGNAL>'
14  #     'Exit with code - <CODE>'
15  #     'Process died for unknown reason'
```

Listing 9: Pseudocode for **step5.c**. The piece on the left is executed by the *Fuzzer* at the Attacker/Fuzzer Machine while the *Process Monitor* is responsible for the one on the right at the Target Machine.

The last substep, **step5.c**, is more complex and we detail it with the help of Listing 9. First, based on the details of the template being currently executed (i.e. `templateDetails`, e.g. Generational Fuzzer's PacketPayloadEmptyPathKnown for a Uri-Host option with StrSingular), it determines if this TC is a targeted test case (line 2), in which case it saves this TC (the pair `tcNo` and `pkt`, the actual packet sent) to an internal list of *last*

TCs sent to a specific URI (line 3). This list can keep a predefined, configured value of MAX_{uri} elements. In our experiments, this is set to 5. This information is useful for error reproduction if such an error is related to consecutive packets sent to the same URI, for instance if it only happens in case a CoAP `GET` request is issued after a CoAP `DELETE` one to the same URI.

Then, if this TC was produced by a "deterministic" mutator or generator (e.g. Smart Mutational Fuzzer's StrOverflow instead of any template from Random Fuzzer or Mutational Fuzzer), we save the mutated/generated value as well (lines 4–5). This is done for potential statistical purposes, but this specific information (represented by the `mutgen.csv` in Figure 3.1) is not currently used by us. Next, we save the TC inside another internal list, which keeps a predefined number of *last TCs sent to the SUT* (line 6). This information is also useful for error reproduction. In this case, if such an error is related to a sequence of actions which can be captured by the last MAX_{all} packets received by the SUT, we are able to reproduce it later. In our experiments we set this number to 5.

After that, we check if the TC packet was answered by the SUT: if so, we merely update the counter of currently unanswered packets (`unansCount`) to 0 and finish (lines 8–9), so the next TC can be sent (continuing the loop from steps 5.a to 5.c again); otherwise, we increment that counter. Then we need to determine if the SUT crashed. As mentioned before, we can make a distinction between an **a**bort failure and a **r**estart failure. First, the *Fuzzer* calls the `post_send` method from the *Process Monitor* through RPC (line 15). At the target machine, the *Process Monitor* will check if the SUT subprocess is still alive and, if not, it will fill out the `crashlist.log` file with information regarding the exit status of the SUT, as well as saving a `core` file in case one was generated; then, it will return, through RPC, the liveness status of the SUT to the *Fuzzer*—this whole part is presented in lines 1–8 of the right-hand side portion of Listing 9. This information is useful for understanding the root cause of a failure, as well as distinguishing errors from each other (see examples in Section 3.4.2). Back to the *Fuzzer*, if a failure was indeed detected using this method, we can classify it as an **a**bort failure (line 16).

In case the previous method did not detect a failure (i.e. the SUT subprocess is still alive), it might still be possible the SUT task just hanged—due to an infinite loop, an unhandled exception etc. In order not to have an additional heartbeat/healthcheck packet sent between each unanswered TC, we only do it if the number of currently unanswered packets is above a configurable threshold MAX_{unans} (in our experiments, 5). If this is the case, we send the heartbeat (line 18), which is just a basic CoAP packet known to be always answered by that specific SUT (commonly a `GET .well-known/core`, but this is actually configurable in a per-SUT basis, as illustrated in Listing 1, line 10). If this heartbeat packet is not answered, we can infer a **r**estart failure occurred (line 19).

Finally, if either an **a**bort or a **r**estart failure was detected (line 20), we conclude the SUT has crashed. We then save the available details of the TC which caused this crash (line 22), such as number, target URI and base template for mutation/generation—this is later used to find out which CoAP options are related to which particular errors, for example, and is represented by the `ftc.csv` in Figure 3.1. We also save both internal lists to be used when trying to reproduce this error (`packets.log` from that same Figure). Examples of this information can also be seen in Section 3.4.2. Then, we restore the necessary counters and lists to zero/empty (lines 24–26) and restart the SUT through an RPC call to the *Process Monitor* (line 27). At this point, the SUT should be up-and-running again and ready to receive the next test case.

The last thing we do before actually executing the next test case is to apply a basic heuristic to avoid finding too much of the same error. Our assumption is that, given a combination of template and option type (or target URI, for example) being fuzzed, it is more likely this combination will reach similar parts of the SUT, thus rendering a—possibly large—number of the same error. We do this by defining configurable thresholds per option classes, which are verified after each TC is executed (e.g. proxy- or response-related options can be defined to stop trying their templates before other, more common request options). To illustrate: if a template has generated 604 TCs but 50 failures were already detected at TC #253 in this sequence, we drop the remaining $604-253=351$ test cases and proceed to the TCs from the next template. Additionally, when the execution of a given template terminates (by this heuristic or by reaching the last TC of the template), we save the details of this template's execution, represented by `tr.csv` in Figure 3.1 (see examples in Section 3.4.2), and finally proceed to the next TC.

3.4.2 Gathered Information & Offline Analyzers

We gather information for mainly four purposes: distinguishing between different errors; identifying information regarding the possible root cause of a failure; reproducing an error; and extracting metrics regarding fuzzer execution, such as elapsed times, number of TCs generated and executed, number of crashes per template, etc. In this section we show examples and formats of the information gathered and how we automated the analysis of part of this information. The collection of the scripts mentioned in this section is what forms the *Offline Analyzers*.

Distinguishing Errors and Identifying Root Causes

Three files are used to distinguish between different errors and to obtain information on their root causes:

`crashlist.log` Produced by the *Process Monitor*, it lists all abort failures, each with a summary message. An example can be seen in Listing 10;

```
[08:30.54] Crash : Test - 530 Reason - Segmentation fault
[08:31.27] Crash : Test - 778 Reason - Exit with code - 1
[08:32.06] Crash : Test - 1124 Reason - Exit with code - 0
[08:35.31] Crash : Test - 3406 Reason - Segmentation fault
```

Listing 10: Snippet example of the `crashlist.log` file from the riot-native-gcoap-server application.

`coredump` Also produced by the *Process Monitor*. One `core` file is generated for every crash with reason "Segmentation Fault". In Listing 11 we show an example of one `core` file opened in `gdb` and the stacktrace obtained from it;

```
Core was generated by `/home/bruno/Dropbox/coap-apps/libnyoci/src/plugtest/.libs/nyoci-plugtest-server'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  nyoci_node_list_request_handler (node=0x7ffcc02b41d0) at nyoci-list.c:88
88                      if(prefix[0]) prefix = NULL;
(gdb) bt
#0  nyoci_node_list_request_handler (node=0x7ffcc02b41d0) at nyoci-list.c:88
#1  0x00007f64dd1becd2 in nyoci_node_router_handler (context=0x7ffcc02b41d0) at nyoci-node-router.c:81
#2  0x00007f64dd3c9d43 in nyoci_handle_request () at nyoci-inbound.c:542
#3  0x00007f64dd3ca15b in nyoci_inbound_packet_process (self=self@entry=0x2028010,
↪   buffer=buffer@entry=0x7ffcc02b3d70 "B\001\310\373\367\a3", '\200' <repeats 127 times>,
↪   "\210separate%2A%7C", packet_length=packet_length@entry=149, flags=flags@entry=0) at
↪   nyoci-inbound.c:429
#4  0x00007f64dd3ce987 in nyoci_plat_process (self=self@entry=0x2028010) at nyoci-plat-net.c:922
#5  0x0000000000401990 in main (argc=<optimized out>, argv=0x7ffcc02b4688) at main-server.c:150
```

Listing 11: Snippet example of a `core` file from the libnyoci-plugtest application. We obtain the TC number associated with this crash from the filename (in this case, `TC_26531.dump`).

`target.log` For restart failures and for abort failures with reasons different from "Segmentation Fault", this file, produced by the SUT itself (and "annotated" by the *Process Monitor*) is used. An example can be seen in Listing 12.

```
⋮
[11:48.08] pre_send(57)
11:48:08.018 [Thread-7] INFO  org.ws4d.coap.core.rest.CoapResourceServer - read ressource: null
Exception in thread "Thread-7" java.lang.NullPointerException
    at org.ws4d.coap.core.rest.BasicCoapResource.init(BasicCoapResource.java:74)
    at org.ws4d.coap.core.rest.BasicCoapResource.<init>(BasicCoapResource.java:70)
    at org.ws4d.coap.core.rest.CoapResourceServer.createResourceFromRequest(CoapResourceServer.java:306)
    at org.ws4d.coap.core.rest.CoapResourceServer.onRequest(CoapResourceServer.java:271)
    at org.ws4d.coap.core.connection.BasicCoapServerChannel.handleMessage(BasicCoapServerChannel⌋
    ↪  .java:129)
    at org.ws4d.coap.core.connection.BasicCoapSocketHandler$ReceiveThread⌋
    ↪  .handleIncommingMessage(BasicCoapSocketHandler.java:335)
    at org.ws4d.coap.core.connection.BasicCoapSocketHandler$ReceiveThread.run(BasicCoapSocketHandler⌋
    ↪  .java:221)
[11:48.08] pre_send(58)
⋮
```

Listing 12: Snippet example of the `target.log` file from the jcoap-plugtest application. It shows the consecutive marks made during **step5.a**, as well as the output produced by the SUT when processing a test case during **step5.b**. In this case, from the annotation made by the *Process Monitor*, we can see that TC #57 caused an exception in the SUT.

The format of `crashlist.log` and the `core` files are SUT-independent. This way it was rather straightforward to implemented a script, `an_crashlist.py`, which parses the `crashlist.log` file and, for each test case number in there, reads the corresponding `core` file (if available). Then, by interfacing with the `gdb` debugger, we load the SUT binary configured in Listing 1 together with the `core` file, obtaining the stacktrace from a given error. To be able to distinguish between errors, we assign an error identificator at this point, composed by `filename|line_no|function_name`, based on the deepest frame from the stacktrace from which we can extract a filename. To illustrate: the error depicted in Listing 11 would be identified as `src/libnyociextra/nyoci-list.c|88|nyoci_node_list_request_handler`. Using this identifier we can find out which errors are duplicated and which ones are unique errors.

For the `target.log` file, however, the stacktrace format is dependent on the SUT—mainly on the SUT language, so there is still some degree of code reuse and generalization. For this we implemented another script, called `an_target.py`, which, based on the SUT, parses the `target.log` file and, similarly to the previous one, is able to assign an error identificator to each failure found (as well as associating that error to a given test case number from the fuzzing campaign, by using the annotation marks produced by the *Process Monitor*). Although dependent on the programming language of the SUT, a given parser is fairly simple and is usually implemented with about 20 lines of Python code. By following the patterns in our script, users of FuzzCoAP can easily extend it to support SUTs using programming languages not initially explored in our experiments. To illustrate the outputs of this script using the example from Listing 12, in that case the script can detect an error caused by TC #57, which would be identified as `BasicCoapResource.java:74:java.lang.NullPointerException`. Note that we replace the `function_name` portion of the identifier by an `exception_name`, when the last one is present.

Finally, by identifying only the unique errors detected in each SUT, we can later come back to those stacktraces to perform a manual Root Cause Analysis (RCA). This manual analysis is assisted not only by the stacktraces, but by an inspection of the contents from the test cases actually causing each of those errors as well, obtained by an association of that information with the information from `packets.log`. After our experiments, this was the information provided to the developers of the SUTs so they could not only easily reproduce the issues, but also have a starting point for investigation and correction of the errors detected by us.

Reproducing Failures and Errors

For failure and error reproduction we basically use the information gathered on the `packets.log` file, consisting of the last packets sent to the SUT (up to MAX_{all}) and last packets sent to that specific URI target (up to MAX_{uri}). The script developed to do this, called `an_packets.py`, tries to reproduce each failure from a specified input list of test case numbers by replaying the packets related to each of those test cases to the SUT. If a failure does happen, a check is performed to verify if the (re)produced error is the same as the one triggered during the actual fuzzing campaign. In our experiments, each packet was replayed at least three times before rulling out an error as non-reproducible (see more in Section 4.2).

Using the example from Listing 13, to reproduce the failure detected during execution of TC #4141, FuzzCoAP can use the three last packets received by the SUT at that point when it crashed (packets from TCs #4139, #4140 and #4141), as well as one packet from the *last sent to specific URI* list (from TC #4141). We note how these two lists usually contain distinct packets (or number of packets, for that matter), but the last packet of each list will always be the same, which is the packet from the TC in which the

```
⋮
Crash detected on TC: 4141
Currently Unanswered Packets: 3
TC: 4139
0000  5102D8570CB472696F7405626F617264 Q..W..riot.board
0010  433A5C43454E5C434F4E             C:\CEN\CON

TC: 4140
0000  5302387E312F8CB472696F740576616C S.8~1/..riot.val
0010  7565D081D082D083D084D085D086D087 ue..............
0020  D088D089D08AD08BD08CD08DD08ED08F ................
...[truncated for better readability]...
00a0  D188D189D18AD18BD18CD18DD180D18F ................

TC: 4141
0000  4101BFE9D8B472696F7405626F617264 A.....riot.board
0010  0D310E0E0E0E0E0E0E0E0E0E0E0E0E0E .1..............
0020  0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E ................
0030  0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E ................
0040  0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E ................

Last Packets sent to this Uri-Path (riot/board/^N...[<repeats 62 times>]...): 1
TC: 4141
0000  4101BFE9D8B472696F7405626F617264 A.....riot.board
0010  0D310E0E0E0E0E0E0E0E0E0E0E0E0E0E .1..............
0020  0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E ................
0030  0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E ................
0040  0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E0E ................

Crash detected on TC: 4317
⋮
```

Listing 13: Snippet example of the `packets.log` file from the riot-native-nanocoap-server application, displaying the last packets sent before a failure was detected on TC #4141.

failure was actually detected—not necessarily the packet which caused the failure, since detecting a failure only after a few (up to MAX_{unans}) other TCs are executed is a common scenario for **r**estart failures, those in which the SUT just hangs. So, to actually reproduce it, our tool would send the packet from TC #4139, then check the SUT's health through a heartbeat. If the SUT has not crashed, it would send the next packet, and continue this until all elements from the first list were tried, when it would repeat the same process for the second list. We argue this is a simple yet effective way of reproducing a given failure (and related error) without saving every packet sent during the fuzzing campaign, which would be unfeasible. This is further discussed and supported by FuzzCoAP's high reproduction rates and low false positive rates in Chapter 4.

Extracting Fuzzing Campaign Execution Metrics

The last two files we use are `tr.csv` and `ftc.csv`. The first one, which is an acronym for ***T**emplate **R**esults*, is a list of all templates used in the campaign (note that the components of the Template Details vary between different engines) together with the number of crashes, number of TCs generated, number of TCs actually executed (which might be different from the previous one due to our heuristic mentioned at the end of Section 3.4.1) and the time taken to run that template. An example is shown in Table 3.8. In this example we show all templates used for the CoAP header and for the Block2 option during

Table 3.8: Fragment example of the `tr.csv` file from the Generation Fuzzer campaign against the openwsn-server application.

Option Name	Template Details				Crashes	Executed TCs	Generated TCs	Run Time
	Packet Template	Generator Type	Generator Rule	URI ID				
header	PacketSingular	Message-based	AllFieldsSingular	-	50	280	1000	87.01
header	PacketPayloadEmpty	Message-based	AllFieldsSingular	-	50	389	1000	168.56
header	PacketPayloadSingular	Message-based	AllFieldsSingular	-	1	1000	1000	43.4
header	PacketPayloadEmptyPathKnown	Message-based	MIDSingular	-	1	49	49	3.43
header	PacketPayloadSingularPathKnown	Message-based	TokenSingular	-	1	603	603	32.28
Block2	PacketSingular	Format-based	UintRandom	-	7	50	50	16.96
Block2	PacketPayloadEmpty	Format-based	UintRandom	-	5	50	50	20.01
Block2	PacketPayloadSingular	Format-based	UintRandom	-	0	50	50	2.1
Block2	PacketPayloadEmptyPathKnown	Format-based	UintRandom	0	4	50	50	24.74
Block2	PacketPayloadSingularPathKnown	Format-based	UintRandom	0	1	50	50	3.32
Block2	PacketPayloadEmptyPathKnown	Format-based	UintSingular	0	14	97	97	72.14
Block2	PacketPayloadSingularPathKnown	Format-based	UintSingular	0	6	97	97	11.64
Block2	PacketPayloadEmptyPathKnown	Option-based	BlockSingular	0	50	454	976	242.78
Block2	PacketPayloadSingularPathKnown	Option-based	BlockSingular	0	6	976	976	55.71

the execution of a Generational Fuzzer campaign against the openwsn-server application. From these numbers we can obtain aggregated values for a given option, a given target URI or even for the entire campaign. These values and their aggregations form the metrics discussed in Chapter 4. We note how the maximum number of crashes is 50 due to the threshold we use for the aforementioned heuristic in our experiments. Similarly, the numbers from the Generated TCs field can be explained by the parameters we use, in our experiments, for Equation 3.8—we further discuss these experimental details in Chapter 4 as well.

The second file, which is an acronym for ***F****ailed* ***T****est* ***C****ases*, is a list of all TCs in which a failure was detected, containing the following fields: Option Name, Template Details and TC Number. Since the reported TC is the one in which the crash was detected, not the one which actually caused the SUT to crash, what we do is to merge this information with the information obtained through either `an_crashlist` or `an_target` to obtain an accurate piece. We show an example of this, after merging, in Table 3.9. The template details field shown in this example is just a condensed/internal version of the one shown for `mr.csv`, with no information lost. We note, by the difference in the numbers between the Detected On and the Crashed On fields, how a **r**estart failure can be detected at most up to $MAX_{\text{unans}} = 5$ TCs later. Finally, for the third line, the emptiness of the last two fields indicates a false positive—usually caused by timeout issues between the Fuzzer and the SUT.

Table 3.9: Fragment example of the `ftc.csv` file from the Generation Fuzzer campaign against the openwsn-server application. From this file we can identify which failure detections are false positives (the ones for each there is no respective Error ID).

Option Name	Template Details	Detected On	Crashed On	Error ID [truncated]
Block2	D_O_BlockSingular_0	21023	21019	../coap/coapOption.py:381:NotImplementedError
Block2	E_O_BlockSingular_0	21028	21024	../coap/coapOption.py:175:AssertionError
Block2	E_O_BlockSingular_0	21083		
Block2	E_O_BlockSingular_0	21984	21984	../coap/coapOption.py:203:ValueError
Block2	E_O_BlockSingular_0	21989	21985	../coap/coapOption.py:203:ValueError

Chapter 4

Experimental Evaluation

This chapter presents our findings. First we describe which samples were selected to be used as SUTs, and outline our criteria to look for and choose these samples. Then, we present the experimental results of running fuzzing campaigns against the selected samples and discuss these results in terms of detected errors, error reproducibility rates, false positive rates and execution time, among others. This discussion considers both possible points of views: per fuzzing technique and per sample.

4.1 Data Preparation - Applications, Implementations and Products using CoAP

To find target systems to be tested, we had to look for candidates consisting of any application listening for CoAP packets at a given UDP port. To that end, we ran searches with keywords "coap", "lwm2m" and "onem2m"[1] in the following bases: GitHub[2], BitBucket[3], GitLab[4], Open Hub[5], Krugle[6], Codeplex[7], Grepcode[8], Google Code[9], Sourceforge[10] and Google Search[11] (first 10 pages). Additionally, we considered the lists of protocol implementations available at the official website for the CoAP protocol[12] and at the Wikipedia article for CoAP[13]. A complete list with every CoAP implementation (or library) we found—as well as cloud services offering a CoAP API and 6 commercial products using the protocol—is shown in Appendix A.

With the list of available implementations built, we have manually inspected each

[1]Onem2m is a platform architecture standard with CoAP bindings. More on `http://www.onem2m.org/`

[2]`https://github.com/`

[3]`https://bitbucket.org/`

[4]`https://gitlab.com`

[5]`https://www.openhub.net/`

[6]`http://www.krugle.org/`

[7]`https://archive.codeplex.com/`

[8]`http://www.grepcode.com/`

[9]`https://code.google.com/archive/`

[10]`https://sourceforge.net/`

[11]`https://www.google.com`

[12]`http://coap.technology/`

[13]`https://en.wikipedia.org/wiki/Constrained_Application_Protocol`

Table 4.1: CoAP Implementations/Libraries targeted in the experimental study. Key: ✔–Full Support, ●–Partial Support and ✘–No Support. Reference links to the respective repositories are available in Appendix A.

CoAP Implementation	**RFC 7252 Base**	**RFC 7641 Observe**	**RFC 7959 Block-Wise Transfer**	**RFC 6690 Link-Format**	**Language**	**License**
aiocoap	✔	✔	✔	✔	Python	MIT
Californium	✔	✔	✔	✔	Java	EPL/EDL
canopus	✔	✔	✔	✔	Go	Apache-2.0
cantcoap	✔	✘	✘	✘	C++	BSD-2-Clause
CoaPP	✔	✔	✘	✔	C++	MPL 2.0
CoAPthon	✔	✔	✔	✔	Python	MIT
Erbium	✔	✔	✔	✔	C	BSD-3-Clause
FreeCoAP	✔	●	✔	●	C	BSD-like
gcoap	✔	✔	✘	●	C	LGPLv2.1
gen_coap	✔	✔	✔	✔	Erlang	MPL 1.1
go-coap	●	✘	✘	✘	Go	MIT
java-coap	✔	✔	✔	✔	Java	Apache-2.0
jcoap	✔	✔	✔	✔	Java	Apache-2.0
libcoap	✔	✔	✔	✔	C	GPLv2/BSD-2-Clause
LibNyoci	✔	✔	✔	✔	C	MIT
microcoap	●	✘	✘	✘	C	MIT
mongoose-coap	✔	✘	✘	✘	C	GPLv2/Commercial
nanocoap	●	●	✘	●	C	LGPLv2.1
nCoAP	✔	✔	✔	✔	Java	BSD-3-Clause
node-coap	✔	✔	✔	✔	Javascript	MIT
openwsn	●	✘	●	✘	Python	BSD-3-Clause
ruby-coap	✔	✔	●	✔	Ruby	MIT
Soletta-CoAP	✔	✔	✘	✘	C	Apache-2.0
txThings	✔	✔	✔	✔	Python	MIT
YaCoAP	●	✘	✘	✔	C	MIT

implementations' code repository and/or website looking for information from which we could infer a maturity and popularity status. This information included, but was not limited to: versions of the reference standards implemented, frequency of updates to the repository (as well as the date from the last update), number of forks and stars the project had (since most of the results are located in bases with these metrics, such as GitHub) and applications using that library. In possession of this information and considering platform constraints—e.g. discarding libraries requiring different Operating Systems or specific embedded hardware—we filtered the initial list with 78 implementations down to 26 relevant candidates, and finally to 25 libraries, since we were not able to successfully install one of those. The list with all the 25 libraries targeted in this study is presented in Table 4.1.

Finally, for each targeted library we have selected one application to be used as a sample in our study. For most cases, the sample is an example server application distributed with the library itself. We briefly describe each of these samples below, including which version of the underlying library was tested (composed of the prefix of the commit hash and the commit date) and a "Short ID" (between parenthesis) to be used in Section 4.2 for better readability of the presented results. The fragments describing the underlying libraries were mostly obtained from the repositories presented in Table 4.1 and is based on developers' claims:

`aiocoap-server (aiocoap)` Example server exposing `.well-known/core` plus 3 resources: one triggering block-wise transfer, another using separate responses and an observable one. The underlying library, aiocoap, uses Python 3's asynchronous I/O to facilitate concurrent operations while maintaining a simple to use interface and not depending on anything outside the standard library. Version `a2c2abe @ 2017-05-11` was used;

`californium-plugtest (calif)` Example server exposing `.well-known/core` plus 28 resources, including resources triggering block-wise transfers, using separate responses and observable ones. The library, Eclipse Californium, is a Java 7 implementation of CoAP for IoT cloud services. Thus, the focus is on scalability and usability instead of resource-efficiency like for embedded devices. Yet Californium is also suitable for embedded JVMs. The version used was `0533403 @ 2017-07-27`;

`canopus-server (canopus)` Server application[14] created by us by putting together snippets from the `simple`, `block1` and `observe` examples distributed with the library, just so we could obtain a single sample with all this functionality. It exposes `.well-known/core` plus 6 resources, including resources triggering block-wise transfers and observable ones. The library, canopus, is a Go implementation of the protocol. Version `e374f5b @ 2018-02-07` of the library was used. Since this application is not distributed with the library, although composed simply and exclusively from snippets that are, it has its own version: `4a79a8d @ 2018-04-11`;

`cantcoap-server (cant)` Example server exposing a single resource with piggybacked response. The underlying library, cantcoap, is a C++ implementation with a focus on simplicity. The library only provides Protocol Data Unit (PDU) construction and de-construction, and users are expected to deal with retransmissions, timeouts and message ID matching themselves. Version `14e7afc @ 2016-09-26` was used;

`coapp-server (coapp)` Example server exposing 4 resources, including an observable one. CoaPP, the used library, is a C++11 implementation for which additional details on its goals and design philosophy were not found. The sample uses version `dc271bf @ 2017-04-12` of the library;

`coapthon-server (thon)` Example server exposing `.well-known/core` plus 12 resources, including resources triggering block-wise transfers, using separate responses and observable ones. The underlying library, CoAPthon, is a Python 2.7 implementation focused on developers' usability. Version `b6983fb @ 2017-07-06` of the library was used;

`contiking-native-erbium-plugtest (erbium)` Contiki OS[15] is one of the most popular open source operating systems for the Internet of Things. Contiki-NG[16], in turn, started as a fork of the Contiki OS focusing on dependable (secure and reliable)

[14]`https://github.com/bsmelo/canopus`
[15]`http://www.contiki-os.org`
[16]`http://contiki-ng.org/`

low-power communication and standard protocols. It is currently more actively maintained than Contiki OS itself, and easier to use and test inside a Linux process. Erbium[17] is a CoAP implementation written in C and available in both Contiki OS and Contiki-NG. In our study, version `9777ac4 @ 2018-02-03` of Contiki-NG was used. The tested application, using Erbium, is an example server exposing `.well-known/core` plus 20 resources, including resources triggering block-wise transfers, using separate responses and observable ones;

`freecoap-server (free)` Example server exposing 3 resources, including one triggering block-wise transfer and an observable one. FreeCoAP is an implementation of the CoAP protocol in C for GNU/Linux. The library version used was `948b01a @ 2018-01-18`;

`riot-native-gcoap-server (gcoap)` RIOT OS[18] is another popular open source operating system for the IoT. Gcoap is a CoAP implementation written in C and supported by RIOT. It provides a high-level interface for writing CoAP messages via RIOT's `sock` networking API. By internalizing network event processing, an application only needs to focus on request/response handling. Internally, gcoap depends on the `nanocoap` package for base level structs and functionality. Running inside a Linux native process, the sample used is an example server exposing 2 resources with piggybacked responses. The version used was `23f4f9b @ 2017-09-08`;

`gen_coap-server (gen)` Example server exposing a single `.well-known/core` resource. The underlying library, gen_coap, is a pure Erlang implementation of the protocol. The sample uses version `c820035 @ 2017-06-23` of the library;

`ibm-crosscoap-proxy (ibm-go)` This sample is a CoAP-to-HTTP translator proxy, built in Go, called crosscoap[19], intially developed by developerWorks[20], an open source division from IBM. It is a UDP server which translates incoming CoAP requests to corresponding HTTP requests which are sent to a backend HTTP server. The HTTP responses from the backend are translated back to CoAP and sent over to the CoAP client. It exposes no resources by default, depending on what is exposed by the HTTP backend. Version `bcdf74f @ 2016-03-22` of the application was used. Internally, this sample relies on the go-coap library to handle CoAP packets. Version `ddcc806 @ 2017-02-13` of the library was used;

`java-coap-server (java-mb)` Server application[21] created by us, since this library did not distribute a server-side example. It exposes `.well-known/core` plus 4 resources. Java-coap, the underlying library, is a Java SE 8 implementation of the protocol developed by ARM mbed[22] to be used within their cloud services. Version

[17]`http://people.inf.ethz.ch/mkovatsc/erbium.php`
[18]`http://www.riot-os.org/`
[19]`https://github.com/ibm-security-innovation/crosscoap`
[20]`https://developer.ibm.com/code/2016/02/20/crosscoap-puts-coap-to-work-for-you/`
[21]`https://github.com/bsmelo/java-coap`
[22]`https://www.mbed.com/en/`

`23e62f3 @ 2018-03-21` of the library was used. Since this application is not distributed with the library, it has its own version: `52a9b80 @ 2018-03-22`;

`jcoap-plugtest (jcoap)` Example server exposing `.well-known/core` plus 3 resources, including an observable one. The library used, jCoAP, is a Java 6 implementation of the CoAP protocol and is part of the bigger Web Services for Devices (WS4D)[23] project. Version `673fa68 @ 2017-06-21` of the library was used.

`libcoap-server (libcoap)` Example server exposing `.well-known/core` plus 2 resources, including one using separate responses and an observable one. Libcoap is a C implementation for devices that are constrained by their resources such as computing power, RF range, memory, bandwidth or network packet sizes. It is designed to run on embedded devices as well as high-end computer systems with a POSIX API. It was recently ported to the Win32 API as well. Library version `44f392d @ 2017-07-18` was used;

`libnyoci-plugtest (nyoci)` Example server exposing `.well-known/core` plus 6 resources, including resources triggering block-wise transfers, using separate responses and observable ones. LibNyoci, the library used in this sample, is a highly-configurable CoAP stack developed in C, which is suitable for a wide range of devices, from bare-metal sensor nodes with kilobytes of RAM to Linux-based devices with megabytes of RAM. Version `4a984c2 @ 2017-07-08` of the library was used;

`riot-native-microcoap-server (micro)` Another sample using RIOT OS. Microcoap is actually an OS-independent CoAP implementation written in C and supported as external package by RIOT. It is branded as a small CoAP implementation for microcontrollers. Running inside a Linux native process, the sample used is an example server exposing `.well-known/core` plus 2 resources with piggybacked responses. The library version used was `ef27289 @ 2016-02-05`, linked by RIOT OS `23f4f9b @ 2017-09-08`;

`mongoose-server (mongoose)` Example server exposing a single resource with piggybacked response. This sample uses the CoAP implementation from Mongoose[24], an embedded web server & networking library. On the market since 2004, Mongoose is used by a vast number of open source and commercial products—running even on the International Space Station. Version `2516da1 @ 2018-03-12` of the library was used in this sample;

`riot-native-nanocoap-server (nano)` Another sample using RIOT OS. Nanocoap is a CoAP implementation written in C and supported by RIOT. It provides CoAP functionality optimized for minimal resource usage, in a philosophy similar to that of `cantcoap`. Running inside a Linux native process, the sample used is an example server exposing `.well-known/core` plus 3 resources with piggybacked responses. The version used was `23f4f9b @ 2017-09-08`;

[23]`http://ws4d.org/`
[24]`https://cesanta.com/`

`ncoap-server (ncoap)` Example server exposing `.well-known/core` plus 2 resources, including resources triggering block-wise transfers and an observable one. The library used in this sample is nCoAP, a Java 7 implementation based on Netty, an asynchronous and event-driven network application framework. Version `15d5a76 @ 2018-03-20` of the library was used;

`node-coap-server (node)` Simple example server exposing a single resource with piggybacked response. The node-coap library used by this sample is a Node.js (Javascript) client and server library for CoAP modeled after Node.js's `http` module. Library version `e4bbe97 @ 2018-03-18` was used;

`openwsn-server (openwsn)` Simple example server exposing a single resource with piggybacked response. The library used is a Python implementation of the protocol and is part of the bigger OpenWSN[25] project from UC Berkeley. This sample uses library version `bdf2cac @ 2017-07-14`;

`ruby-david-server (ruby-dv)` Example server called `david`[26] exposing `.well-known/core` plus 8 resources, including resources triggering block-wise transfers, using separate responses and observable ones. David is a CoAP server developed in Ruby and uses the ruby-coap library internally. It has a `Rack` interface and its goal is to enable the usage of Rack-compatible web frameworks for the Internet of Things. Ruby-coap version `86c8419 @ 2016-01-26` was used, and since `david` is distributed separately, it has its own version: `b9413ce @ 2018-03-04`;

`soletta-coap-server (soletta)` Simple example server exposing a single resource with piggybacked response. The CoAP library used is part of the bigger Soletta[27] project, developed by `01`, Intel's open source division. It is a framework written in C and aims to ease the software development for IoT devices. Version `0e492a8 @ 2017-06-08` was used;

`txthings-server (txthings)` Example server exposing `.well-known/core` plus 4 resources, including resources triggering block-wise transfers, using separate responses and observable ones. It uses txThings, a Python library based on Twisted, an asynchronous I/O framework and networking engine. Version `3ad1e3a @ 2016-05-01` was used;

`yacoap-piggyback (yacoap)` Example server exposing `.well-known/core` plus 2 resources, including one using separate responses. This sample uses YaCoAP, a C implementation that started as a fork from microcoap. Library version `d32b128 @ 2016-11-25` was used;

Due to the number of programming languages (8) used across all samples, and thus the number of different application development environments and building tools involved, it is not straightforward to get all these samples properly running for a given experiment.

[25]`http://www.openwsn.org/`
[26]`https://github.com/nning/david`
[27]`https://solettaproject.org/`

Thus, we provide a virtual machine (through a Vagrant file) with this collection and all dependencies configured and ready to be used, available as Free Software under the GNU GPLv3 license at `https://github.com/bsmelo/fuzzcoap`. We highlight this as a contribution facilitating future research investigating any practical aspects of CoAP implementations.

4.2 Results and Discussion

In this section we present the experimental results of the fuzzing campaigns executed. All experiments were performed on a Lenovo Y570 laptop with Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz processor and 8Gb of RAM. This machine was running Linux Mint 17.3 distribution with GNU/Linux 3.19.0-32-generic x86_64 kernel.

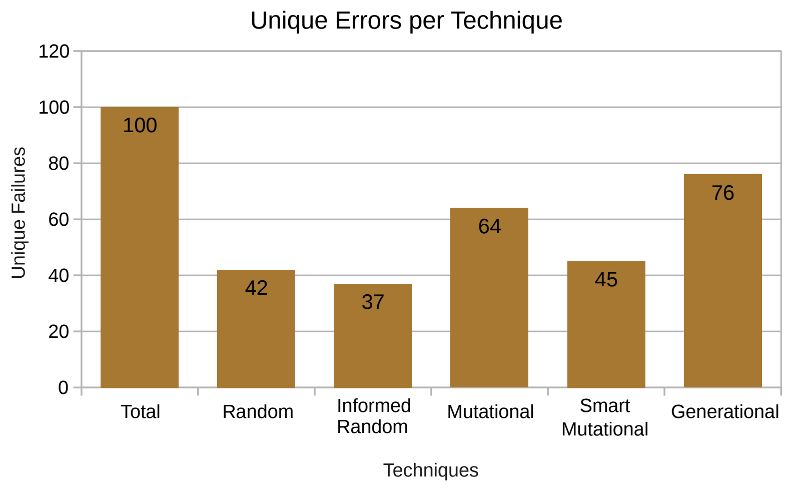


Figure 4.1: Chart of unique errors uncovered per fuzzing technique. Although it shows the generational and mutational fuzzers with the highest scores, we can also see how far from the total (100) these scores are. This suggests that the fuzzing techniques used are complementary to each other—in other words, there are no full intersections between the sets of uncovered errors per technique, as further detailed in Figure 4.2.

Of the 25 samples being tested, FuzzCoAP found at least one error in 14 (or 56%) of them. As shown in Figure 4.1, a total of 100 unique errors were detected across all samples. Figure 4.2 contains a detailed visualization of the same data presented in Figure 4.1. Based on this visualization, we are able to draw one basic conclusion: the techniques are mostly complementary with regard to each other. From the 100 errors, only 13 were detected by all 5 techniques, while other 24 errors were exclusively detected by only one of the techniques—with the generational fuzzer having the highest "exclusiveness score" of 12 errors.

Figure 4.3 further drills down this data, not only on a per-technique but on a per-sample basis too. Only the samples for which at least one error was sucessfully detected are displayed. We highlight that although for half (7 out of 14) of the samples displayed, a single technique was able to uncover the total errors uncovered by the set of all techniques

[28]Generated using InteractiVenn `http://www.interactivenn.net/`

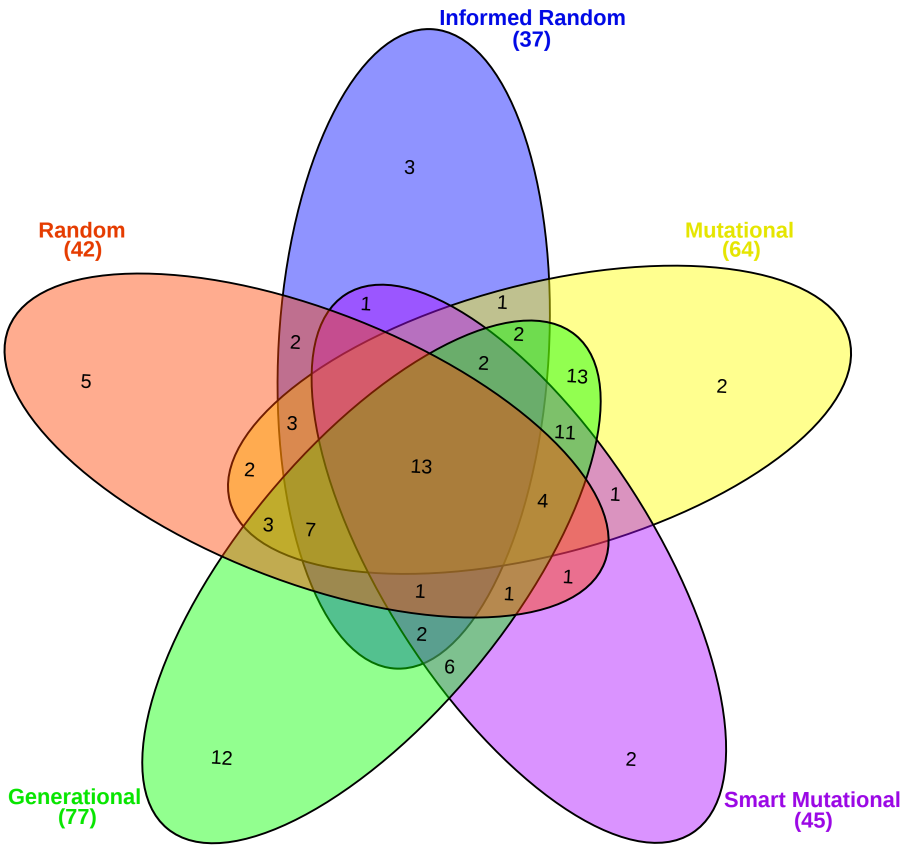


Figure 4.2: Intersection of errors discovered across all samples and techniques[28]. Empty regions contain 0 elements; the number was removed for better readability.

for that sample (a "perfect score"), this technique was not necessarily one of the complex or more sophisticated ones; corroborating the conclusion that the techniques are complementary. From these 7 samples for which a "perfect technique score" was attained, the generational fuzzer was the most successful technique, covering 4/7 samples, followed by the mutational fuzzer with 3/7 samples, finally followed by the three other techniques—random, informed random and smart mutational fuzzers—with 2/7 samples. Considering only the samples for which the total uncovered errors (across all techniques) is greater than one, only the generational fuzzer (for the `nyoci` sample) and the mutational fuzzer (for `openwsn` and `thon`) were able to attain a "perfect technique score".

When FuzzCoAP detects a failure, we say an alarm is raised. Information regarding that alarm, including latest TCs executed and logs, is saved as detailed in Section 3.4.1. Then, during offline analysis, every alarm is checked—in the semi-automatic manner described in Section 3.4.2—to become either a confirmed failure (those for which there is evidence of the related error) or non-confirmed (mostly due to the time-based detection heuristic by heartbeats). Additionally, any confirmed failure can be either reproducible or not reproducible. Reproducible failures are those failures we are able to reproduce, a posteriori, using information from its respective alarm (i.e. by replaying the TCs from `packets.log` we obtain the same stacktrace, or error, found in either `target.log` or `core.dump`). Non-reproducible failures are those from which, although in possession of evidence generated during the fuzzing campaign (after all, they are confirmed failures), we

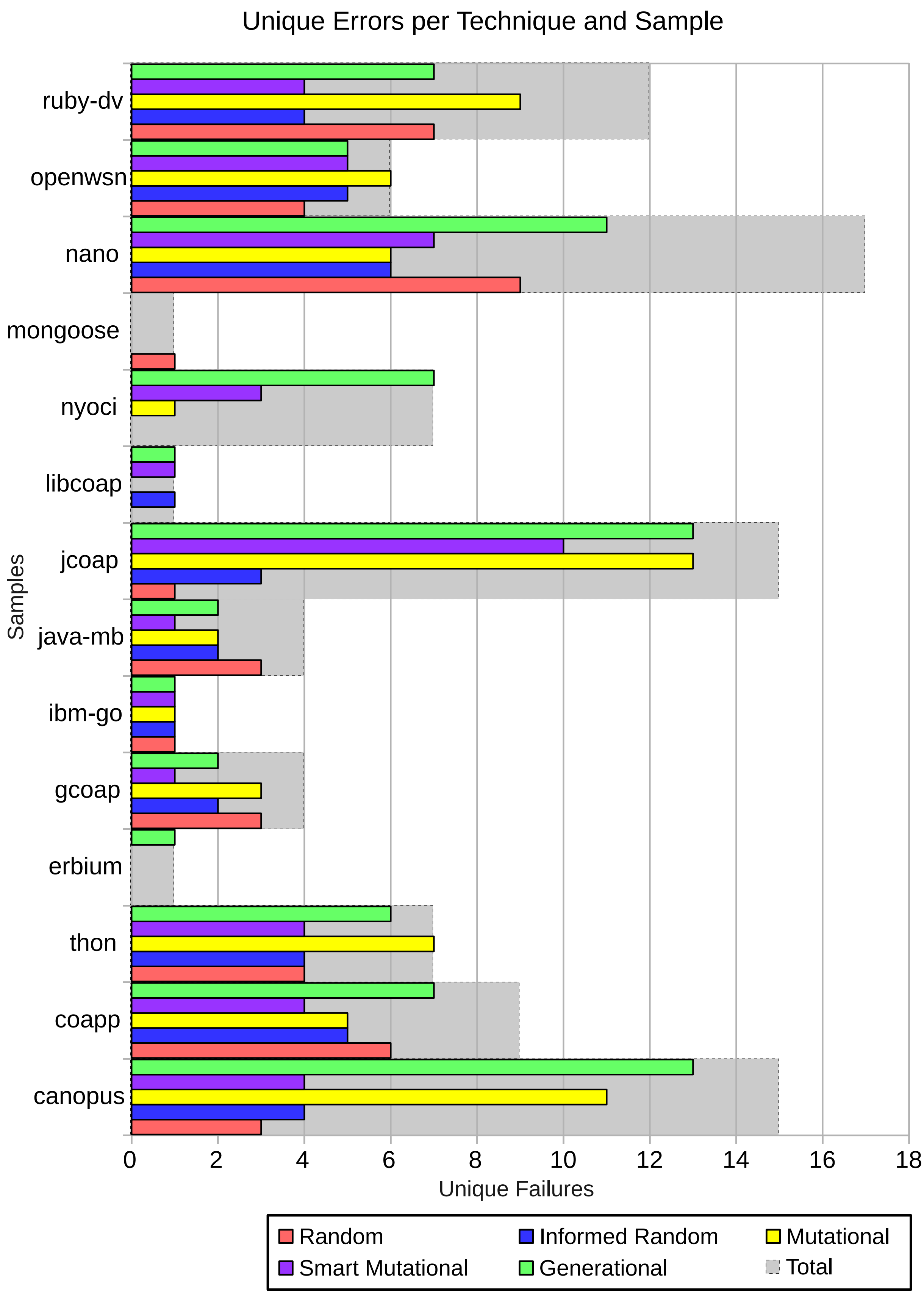


Figure 4.3: Chart of unique errors uncovered per sample and technique. Considering only the samples for which at least one error was uncovered, in only half of them there was a single technique that was able to uncover all errors—but not necessarily the same technique—, corroborating once more the complementary nature of the techniques studied.

are not able to consistently reproduce them—this might happen due to race conditions, for instance.

Based on those classes, we can classify an alarm as either a True Positive (TP) or a False Positive (FP). True Positives are reproduced, confirmed failures and errors. False Positives are non-confirmed failures. This raises a question regarding how to classify the remaining group: non-reproducible, confirmed failures. In the charts to follow we use both possible counting approaches: "optimistic", in which non-reproducible failures are considered true positives (data with this approach is displayed with a star (*) in the charts and tables); and "conservative", in which non-reproducible failures are considered false positives (data with this approach is displayed with no remarks in the charts and tables). Figure 4.4 summarizes this decision process.

Experiment Phase	**Fuzzing Campaign**	**Offline Analysis**	
Artifacts Used (classifiers)	Process Monitor and heartbeats heuristic	`target.log, core.dump`	`packets.log, target.log, core.dump`

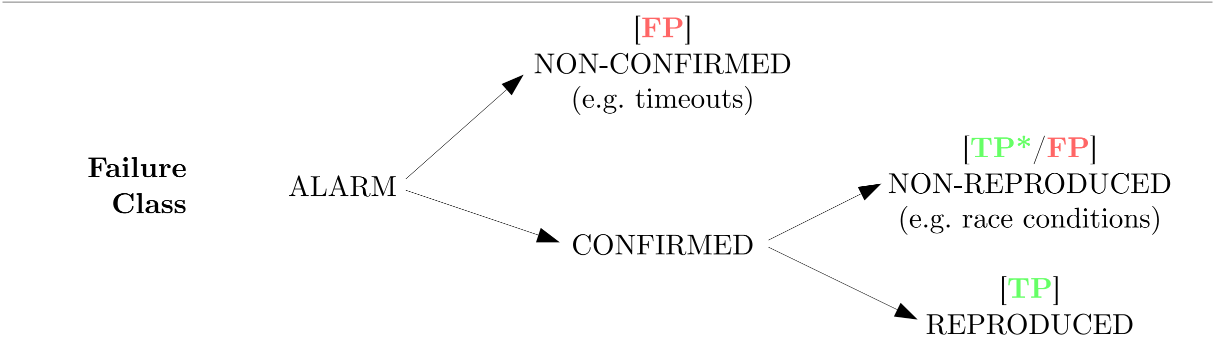
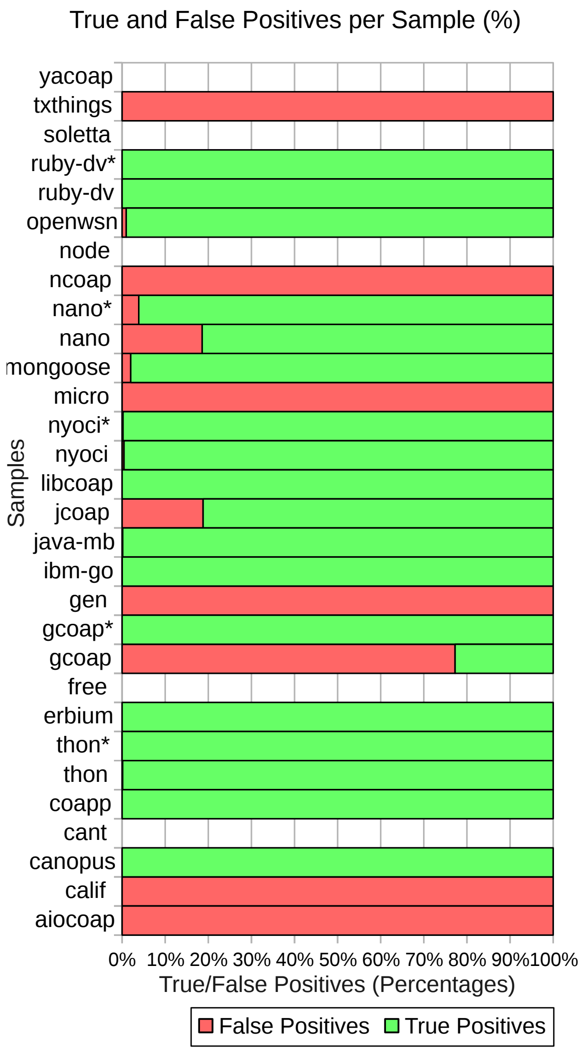
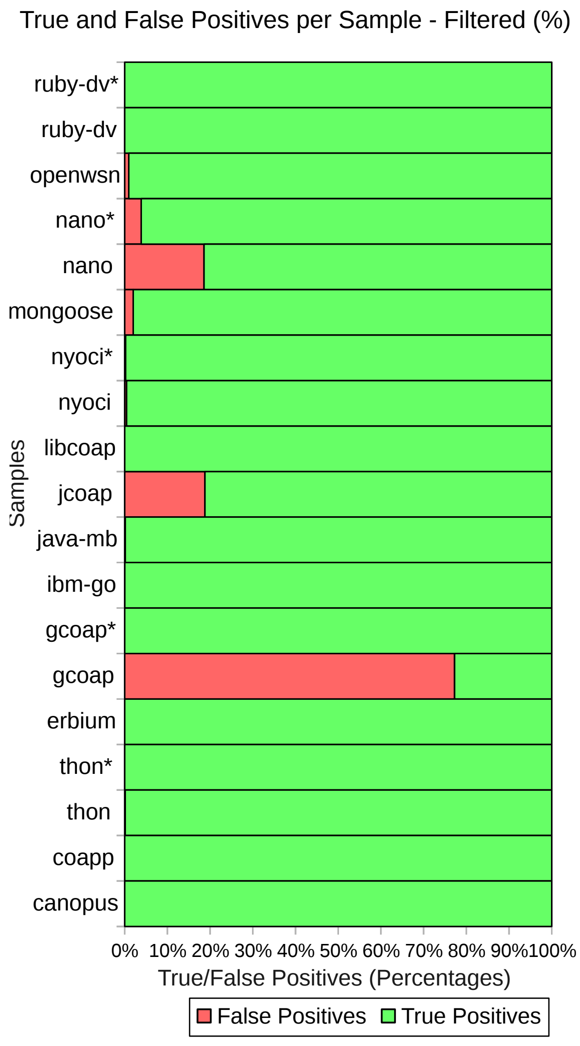

Failure Class
ALARM
[FP] NON-CONFIRMED (e.g. timeouts)
CONFIRMED
[TP*/FP] NON-REPRODUCED (e.g. race conditions)
[TP] REPRODUCED

Figure 4.4: Classification process for True and False Positives.

We argue that FuzzCoAP has a low false positive rate. The only way for a false positive to occur is due to timeouts on the heartbeat heuristic—and as explained in the previous Chapter, it only happens for possible **r**estart failures; alarms for **a**bort failures are always true positives, due to the nature of the Process Monitor. The proportional charts in Figure 4.5 supports this claim. Empty lines are samples which did not generate any alarm. If we look at the chart in (a), displaying all 25 samples, we see 7 samples (out of 20 if we exclude the empty ones) dominated by false positives. However, if we filter out this chart, displaying only the samples for which at least one of the alarms was a true positive, we obtain the chart in (b). In this chart, only 1 bar (out of 14) is dominated by false positives; and it is a "conservative counting" bar—meaning that these failures occurred and were confirmed, they are just intermitent for some unknown reason. From the chart in (b) and considering the "optimistic counting" bars, we can see that the only sample presenting a high false positive rate is `jcoap`. We performed a further inspection and found out that, for a number of test cases, this sample tends to block (either due to a time-consuming processing task or due to too much logging output), causing the heartbeat heuristic to timeout and (incorrectly) assume a **r**estart failure. This could be easily improved by a configurable heartbeat timeout per SUT—trading-off with execution



(a)



(b)

Figure 4.5: Charts of True and False Positives per Sample (%). (a) Displays all samples. (b) Filtered to display only the samples with at least one true positive.

time for that specific SUT as well, of course. Additionally, the average number of alarms from the filtered out samples (`txthings`, `ncoap`, `micro`, `gen`, `calif` and `aiocoap`) is 49.7, with half of them (`txthings`, `ncoap` and `calif`) triggering less than 15 alarms each. The same improvement suggested for `jcoap` applies.

Figure 4.6 displays the same data (true and false positives per sample), but in absolute numbers. Large true positive numbers suggest the presence of "shallow failures": failures that are so easily triggered it makes it difficult to reach deeper parts of the SUT. This is a common problem with fuzzing, even more so when testing new or immature software. After further inspection, delivery of the reports and follow-up discussions with the maintainers, we have confirmed this is the case for `thon`, `openwsn` and `ruby-dv`. Others are currently being handled, but the trend exists.

Regarding failure and error reproducibility, 87% of the confirmed failures uncovered by FuzzCoAP were consistently reproduced. This number is, of course, based on the "conservative counting"; "optimistic counting", which includes all confirmed failures—those for which the tool was able to gather evidence of an error, but occurs intermitently—,

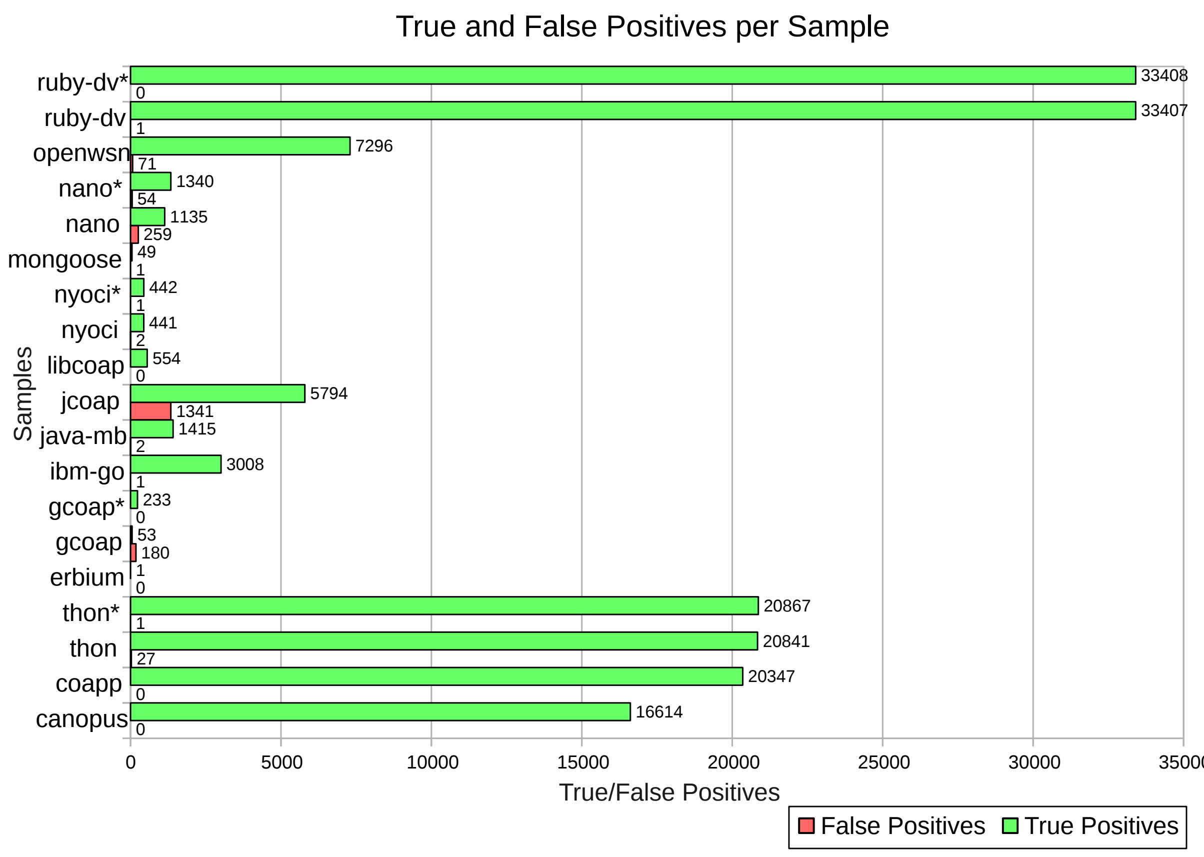


Figure 4.6: Chart of True and False Positives per Sample, in absolute values—filtered to display only the samples with at least one true positive.

would render a score of 100%. Every packet (or test case) was replayed at least three times before a given failure or error was ruled out as non-reproduced. Figure 4.7 displays error reproducibility data, based on the "conservative counting", per individual sample. It also shows that 11/13 (or 85%) of those non-reproduced errors belong to samples developed in C (`nano`, `nyoci` and `gcoap`), a low-level language more prone to race conditions and other issues that might impact consistent reproducibility. Further supporting this claim, 10/11 (or 91%) of these non-reproduced errors belong to samples based on a specific IoT OS (RIOT OS), relying on custom timer, scheduler and IPC implementations. These implementations are not as mature as those of a widespread OS as GNU/Linux, and could have a greater impact on reproducibility (again due to higher chances of race conditions and similar issues) as well.

Furthermore, in Figures 4.8 and 4.9 we present data regarding the execution times per sample, for each fuzzing technique implemented. The data series labeled as "real" is formed by the actual execution times of the fuzzing campaigns. The "projection" one shows how long it would take to run the same campaign, if the heuristics to avoid finding too much of the same errors—presented at the end of Section 3.4.1—were not in place. Throughout all fuzzing techniques, we see these heuristics having most of their impacts on the same set of samples (`ruby-dv`, `openwsn`, `java-mb`, `ibm-go`, `thon`, `coapp`, `canopus`). These samples are basically the same ones with large true positive rates presented in Figure 4.6. We can make two inferences from this: i) those heuristcs greatly reduce execution time which

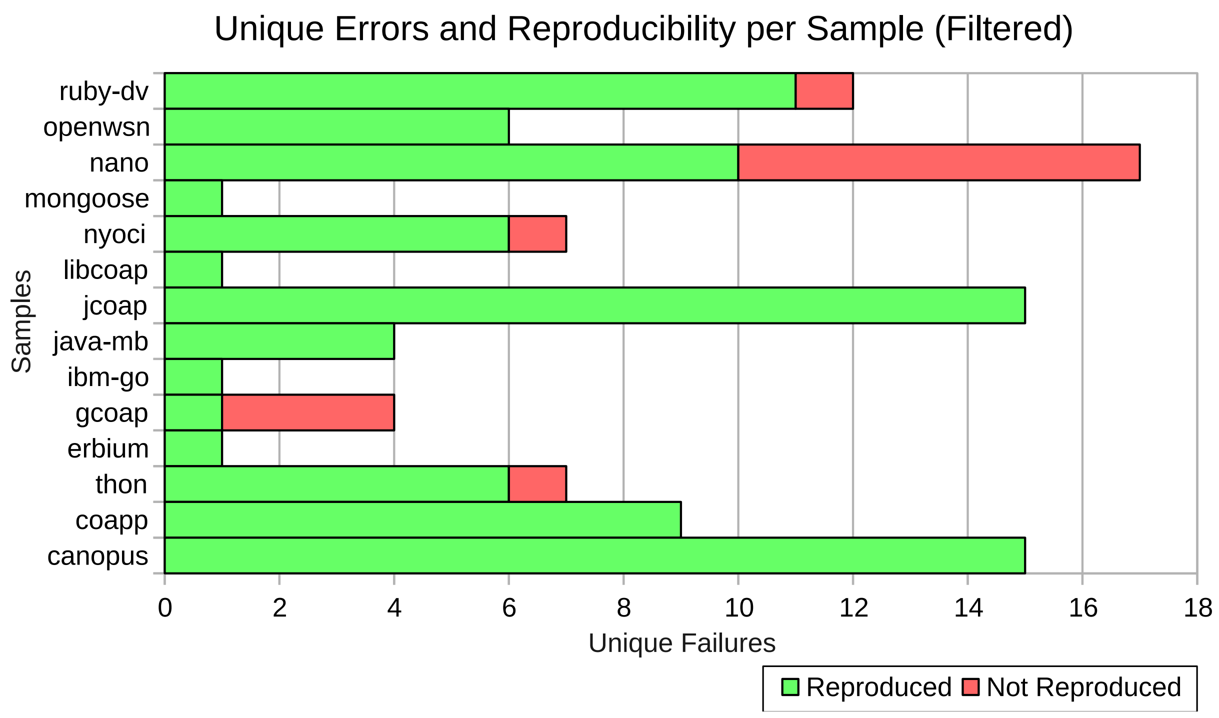


Figure 4.7: Chart of Error Reproducibility per Sample, considering the "conservative counting".

would be wasted in finding the same "shallow failures" (by an average of 823 minutes per sample, or 54% of the projected time, when considering all techniques); ii) given the large numbers from Figure 4.6, there is still room for improvement on those heuristics. Additionally, we associate the increase in the order of magnitude (easily seen from the mean lines), from all other techniques to the generational fuzzer, to the fact that this fuzzer is the only one with a variable amount of generated test cases. This amount is proportional to the number of paths of a given sample (see Equations 3.6 and 3.8), and we highlight `ruby-dv` (9), `erbium` (21), `thon` (13) and `calif` (29) as the biggest targeted samples in that sense.

Finally, we reiterate the real-world impact on IoT security made by this research. All errors were reported to respective library maintainers, accompanied by simple Python scripts to easily reproduce each error themselves. Currently, there is an ongoing follow-up process between us and the maintainers, and some of the errors were already fixed. Our goal is to maximize these fixes with the community.

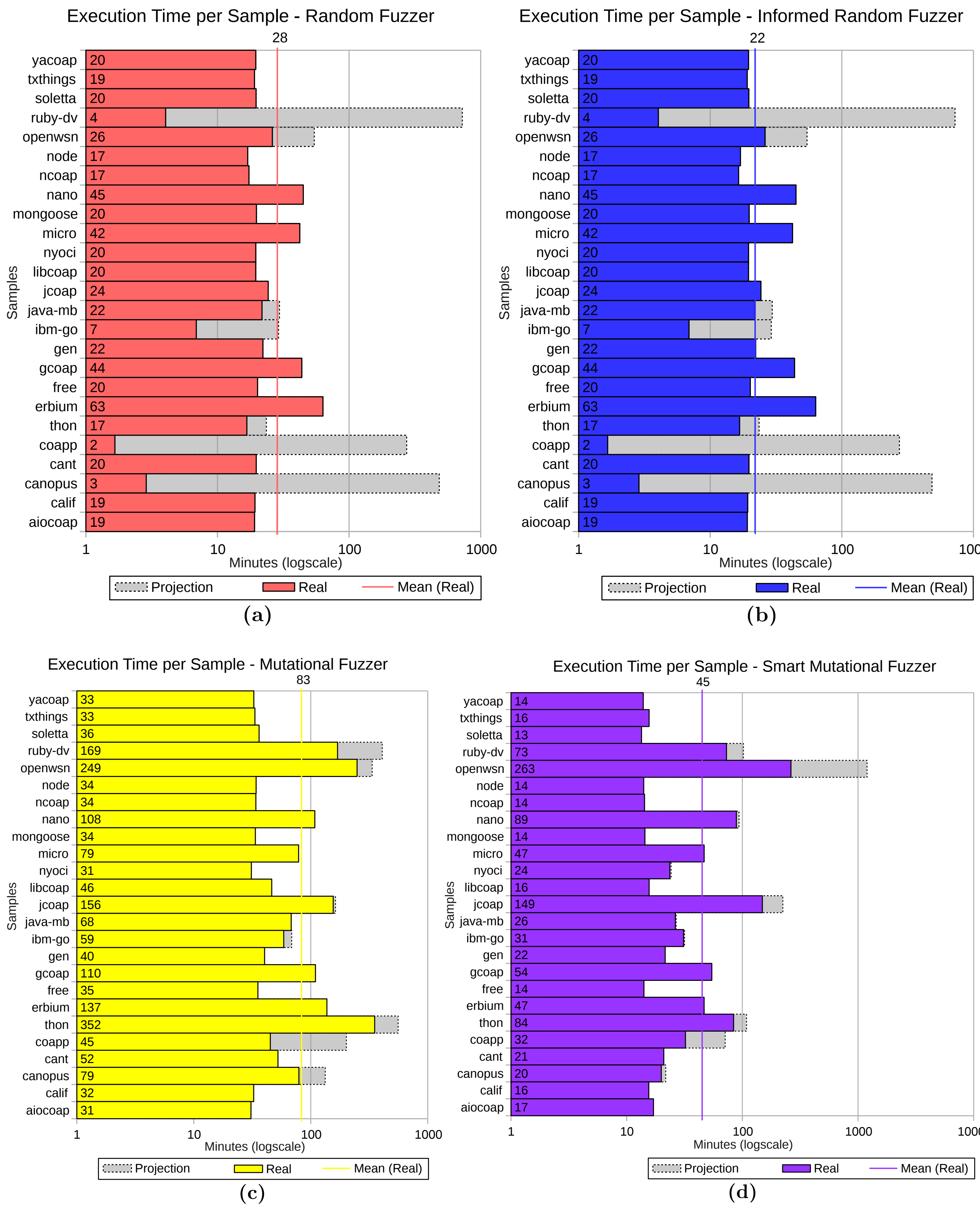


Figure 4.8: Charts of Execution Times per Sample (1). (a) Random Fuzzer. (b) Informed Random Fuzzer. (c) Mutational Fuzzer. (d) Smart Mutational Fuzzer. Colored vertical lines displays the average between all samples.

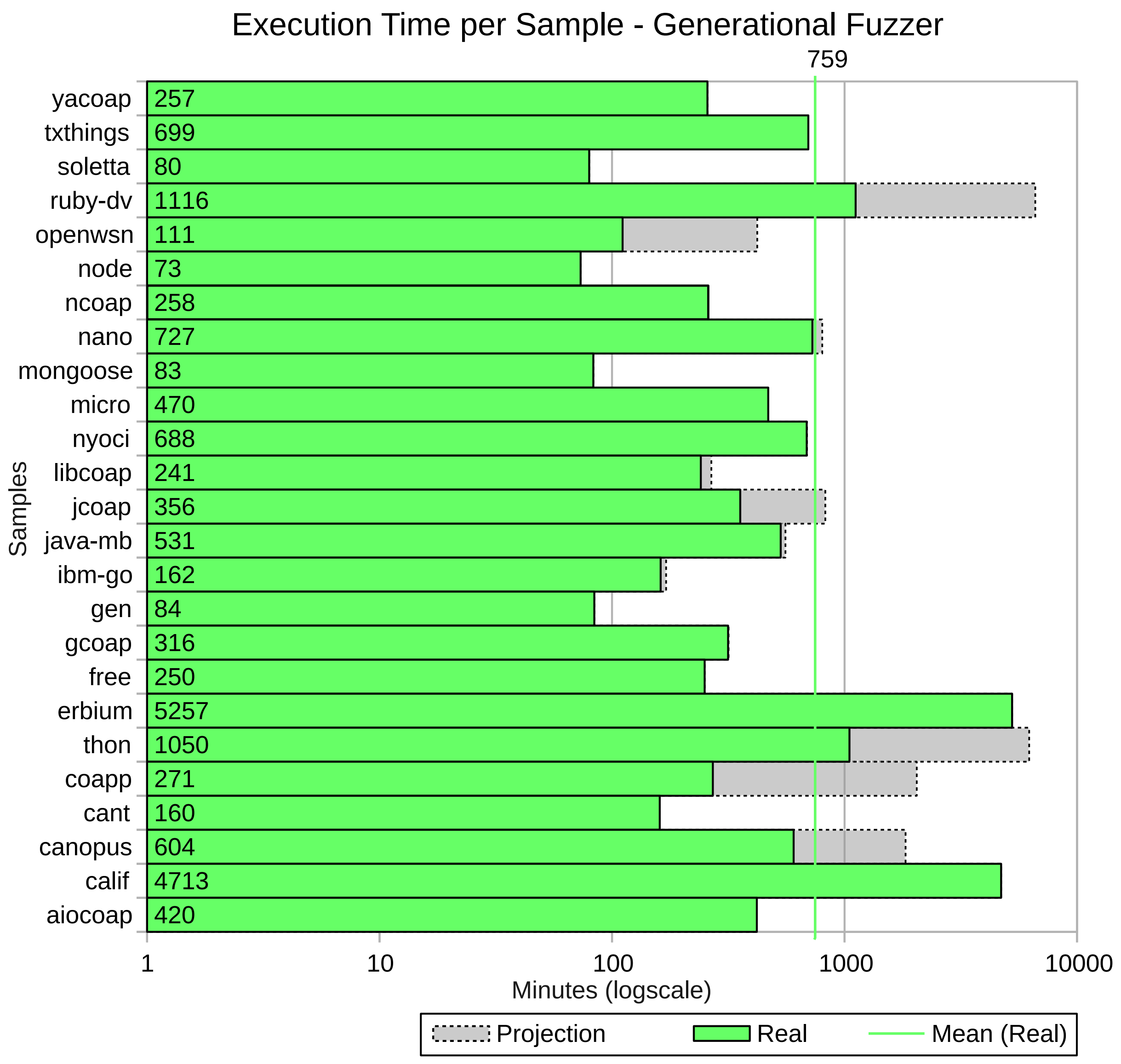


Figure 4.9: Chart of Execution Times per Sample (2). Generational Fuzzer. The colored vertical line displays the average between all samples.

Chapter 5

Conclusion

Motivated by the growth of research and applications for an Internet of Things scenario and considering possible security issues in emerging technologies, in this dissertation we approached the problem of robustness/security testing of the CoAP protocol. In this chapter we wrap up the work by summarizing what was done, pointing out its limitations and suggesting work that could be further developed in this area.

In Chapter 1, an introduction to the Internet of Things and Web of Things concepts was presented, as well as to the Constrained Application Protocol (CoAP) as a technology enabler for those concepts. Likewise, an overview was given on why security is an important concern in the IoT domain, including examples of security incidents in this domain, and what kind of work has been done regarding CoAP security. Based on this, a decision was taken to explore practical IoT software security, by using fuzzing techniques to test the robustness/security status of available CoAP implementations. The chapter ended with an outline of the objectives, contributions and dissertation structure.

Chapter 2 presented an overview of the CoAP protocol through its main specifications, allowing one to better understand the protocol itself. Then, existing research and tools for testing CoAP implementations was presented, highlighting not only the need for a study to better understand the current status of CoAP implementations, but also the need for an open, free, readily available tool to test robustness and security aspects of CoAP applications and implementations. The open tools found can only perform functional (conformance and interoperability) tests, while tools that could be used to perform robustness/security testing of the protocol are not open. Finally, relevant work regarding fuzzing techniques was presented, together with their usage and evolution. This served as the basis for the development of a CoAP-focused testing environment and fuzzer, a system we called FuzzCoAP.

The design and implementation of that system was presented in Chapter 3. Every aspect of every entity involved in the fuzzing process was detailed, ranging from the Process Monitor, responsible for monitoring the status of a given System Under Test (SUT), through the online failure detection mechanisms from the Workload Executor and to the five fuzzing techniques used to generate test cases—how they work, what they do and why it was done. Additionally detailed were the information collected during a fuzzing campaign and how that information is used afterwards, during offline analysis, to distinguish between different errors, reproduce these failures and errors and extract execution metrics.

Finally, in Chapter 4, the experimental results were presented, starting by explaining the data preparation phase, showing how and why the samples were found and selected. Then follows a discussion of the obtained results, focusing on interpreting what the data points to in terms of specific fuzzing techniques and individual samples. It is highlighted that this discussion is highly tied to the discussion presented in Chapter 3, which explains design decisions, implemented heuristics and other aspects directly related to the resulting experimental data.

The tool was able to detect a total of 100 errors in 14 out of 25 tested samples, with an average false positive rate of 31.3% and 87% of error reproducibility. Considering only the samples presenting at least one true positive (i.e. at least one confirmed failure), the average false positive rate attained is 1.86%. Fuzzing campaigns executed with FuzzCoAP take an average of 12 hours using the generational fuzzer, and 45 minutes using the other techniques implemented—considering specific, configurable parameters and thresholds. Besides the practical, real-world impact of these findings—errors were reported and are currently being fixed by library maintainers—, our data corroborates Johansson et al.'s [35] and Winter et al.'s [75], which suggests that several techniques should be combined to achieve greater coverage and find more vulnerabilities. The techniques implemented are complementary to each other, with a presumably more complex technique being not necessarily better than a simple technique—most of the time, they are able to uncover different errors.

Another contribution was the centralized collection of samples. Since the samples use different programming languages, and thus different build systems, the task of setting up an environment to deal with all of them is not trivial. To aid future research involving available CoAP implementations, we provided a Virtual Machine with this collection and all dependencies configured and ready to be used.

5.1 Limitations

One must be aware that the results and comparisons between the fuzzing techniques implemented might not be generalizable to other domains. In fact, one may see this as another motivation for this work: understand how each technique behaves in the specific scenario of testing the CoAP protocol.

Additionally, concerns regarding the quality of the tested samples were highlighted. Since a black-box technique is used, actual applications to be targeted are needed, although through the applications one is actually testing the underlying libraries. Since CoAP is a new protocol with very few commercial products using it as of now, most samples are demo/example server applications that ship with the respective libraries. While this is considered a possible limitation, it is also an interesting finding regarding the current adoption status of the protocol itself.

Regarding root cause analysis of the errors found by FuzzCoAP, it was not performed since it would require a time-consuming, non-repeatable effort: to properly investigate particular errors and find out their causes, we would need to know the details of each implementation, their architectures, programming languages used etc. Instead, as previously stated, we have chosen to report the errors to the respective developers, who with proper knowledge of the implementations, are able to understand and fix the errors with greater ease. Thus, we are currently not able to say if a given sample leak some data, for instance; but we do know some buffer overflow errors found are indeed exploitable, as is the case for LibNyoci[1] and Contiki-NG's Erbium[2].

Finally, in the interest of full disclosure, this author has also contributed to the development of one of the tested libraries: Soletta-CoAP. Contributions were minor and originated from a Google Summer of Code[3] project, in which the original author developed an LWM2M security layer for the Soletta Framework. Since LWM2M uses CoAP as a transport, some modifications were made in the Soletta-CoAP API. Due to the nature of these changes and their minor overall impact in the library itself, there is no conflict of interests.

5.2 Future Work

FuzzCoAP itself can be further improved and extended with regards to a number of things:

- Making the timeout configurable on a per-SUT basis could improve the heartbeat heuristic and lower the false positive rate.
- The heuristic to avoid finding too much of the same error could be improved as well: since a given SUT suffering from the problem of "shallow failures" will exhibit this problem for all techniques, the observation of the results from a first run with a fast technique (e.g. random fuzzer) could be used to tune the configurable thresholds before running a technique that takes more time. This would also lower the false positive rate. In case the user of the tool is the developer of the SUT, she could even fix the shallow failure before running longer campaigns.
- Adding packet models (descriptions of the protocol format written in Python using Scapy) to better support CoAP Resource Directory, oneM2M and LWM2M targets.
- Multiple testing threads, simulating multiple clients. It would mean higher complexity on reproducibility, but should be possible and could uncover new errors. Even though, this would probably have a bigger impact on LWM2M targets than on plain CoAP ones, since LWM2M is stateful, unlike CoAP.

Finally, after studying, implementing and better grasping the challenges of fuzz testing, it would be interesting to explore other approaches and techniques, in either the IoT scenario or in a different one. This could include research on evolutionary fuzzing, guided fuzzing, improved SUT monitoring, online error identification and classification etc.

[1] `https://github.com/darconeous/libnyoci/pull/12`
[2] `https://github.com/contiki-ng/contiki-ng/issues/425`
[3] `https://summerofcode.withgoogle.com/archive/2016/projects/5629807966552064/`

Bibliography

[1] A. Adelsbach, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J.-C. Laprie, B. Pfitzmann, D. Powell, B. Randell, J. Riodan, and Others. Maftia conceptual model and architecture. Technical report, 2001.

[2] T. A. Alghamdi, A. Lasebae, and M. Aiash. Security analysis of the constrained application protocol in the internet of things. In *Second International Conference on Future Generation Communication Technologies (FGCT 2013)*, pages 163–168. IEEE, nov 2013.

[3] W. H. Allen, C. Dou, and G. A. Marin. A model-based approach to the security testing of network protocol implementations. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 1008–1015, 2006.

[4] N. Antunes and M. Vieira. Security testing in SOAs: Techniques and tools. In *Innovative Technologies for Dependable OTS-Based Critical Systems*, pages 159–174. Springer Milan, Milano, 2013.

[5] N. Antunes and M. Vieira. Designing vulnerability testing tools for web services: approach, components, and tools. *International Journal of Information Security*, pages 1–23, 2016.

[6] J. Arlat, J.-C. Fabre, and M. Rodriguez. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, 2002.

[7] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[8] C. Bormann. Test descriptions for CoAP#4. `https://github.com/cabo/td-coap4`. Access in March 03, 2017.

[9] C. Bormann, A. Castellani, and Z. Shelby. CoAP: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, mar 2012.

[10] C. Bormann and Z. Shelby. RFC7959 - block-wise transfers in the constrained application protocol (CoAP), 2016.

[11] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi. Web services for the internet of things through CoAP and EXI. In *2011 IEEE International Conference on Communications Workshops (ICC)*, pages 1–6. IEEE, jun 2011.

[12] M. Castro, A. J. Jara, and A. F. Skarmeta. Enabling end-to-end CoAP-based communications for the web of things. *Journal of Network and Computer Applications*, 59(C):230–236, jan 2016.

[13] Check Point Software. Media alert: Check point researchers discover ISP vulnerabilities that hackers could use to take over millions of consumer internet and wi-fi devices. `http://www.checkpoint.com/press/2014/media-alert-check-point-researchers-discover-isp-vulnerabilities-hackers-use-take-millions-consumer-internet-wi-fi-devices/`. Access in April 09, 2015.

[14] N. Chen, C. Viho, A. Baire, X. Huang, and J. Zha. Ensuring interoperability for the internet of things: Experience with CoAP protocol testing. *Automatika - Journal for Control, Measurement, Electronics, Computing and Communications*, 54(4):448–458, 2013.

[15] Cisco. The internet of things reference model. `http://cdn.iotwf.com/resources/71/IoT_Reference_Model_White_Paper_June_4_2014.pdf`, 2014.

[16] Codenomicon. CoAP server suite | codenomicon. `http://www.codenomicon.com/products/defensics/datasheets/coap-server.html`. Access in February 25, 2017.

[17] CORE-WG. Constrained restful environments (core). `https://datatracker.ietf.org/wg/core/charter/`. Access in March 07, 2017.

[18] A. C. V. De Melo and P. Silveira. Improving data perturbation testing techniques for web services. *Information Sciences*, 181(3):600–619, 2011.

[19] Eclipse. Eclipse IoT-testware | projects.eclipse.org. `https://projects.eclipse.org/projects/technology.iottestware`. Access in April 11, 2018.

[20] ETSI. ETSI CTI CoAP plugtests guide first draft v0. `http://www.etsi.org/plugtests/CoAP/Document/CoAP_TestDescriptions_v015.pdf`. Access in March 03, 2017.

[21] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. Ph.d. thesis, University of California, Irvine, 2000.

[22] A. K. Ghosh, M. Schmid, and V. Shah. Testing the robustness of windows NT software. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257)*, pages 231–235. IEEE Comput. Soc, 1998.

[23] N. K. Giang, Minkeun Ha, and Daeyoung Kim. SCoAP: An integration of CoAP protocol with web-based application. In *2013 IEEE Global Communications Conference (GLOBECOM)*, number December, pages 2648–2653. IEEE, dec 2013.

[24] J. Granjal and E. Monteiro. End-to-end transparent transport-layer security for internet-integrated mobile sensing devices. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 306–314. IEEE, may 2016.

[25] J. Granjal, E. Monteiro, and J. Sa Silva. Security for the internet of things: A survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials*, 17(3):1294–1312, 2015.

[26] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. *Architecting the Internet of Things.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[27] J. Han, M. Ha, and D. Kim. Practical security analysis for the constrained node networks: Focusing on the dtls protocol. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 22–29. IEEE, oct 2015.

[28] K. Hartke. RFC7641 - observing resources in the constrained application protocol (CoAP), 2015.

[29] J. Heuer, J. Hund, and O. Pfaff. Toward the web of things: Applying web technologies to the physical world. *Computer*, 48(5):34–42, may 2015.

[30] J. Hui and P. Thubert. RFC6282 - compression format for IPv6 datagrams over IEEE 802.15.4-based networks, 2011.

[31] IEEE. *IEEE Standard for Local and metropolitan area networks, Part 15.4: Low-Rate Wireless Personal Area Networks.* 2011.

[32] IEEE. Towards a definition of the internet of things (IoT), 2015.

[33] IRISA. Passive validation tool for CoAP [TIPI - testing internet protocols interoperability]. `http://www.irisa.fr/tipi/wiki/doku.php/Passive_validation_tool_for_CoAP`. Access in March 03, 2017.

[34] ISO/IEC/IEEE. *ISO/IEC/IEEE 24765 : 2017(E): ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary.* IEEE, 2017.

[35] A. Johansson, N. Suri, and B. Murphy. On the selection of error model(s) for OS robustness evaluation. *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*, pages 502–511, 2007.

[36] R. Kaksonen. A functional method for assessing protocol implementation security. *VTT Publications 448*, 2001.

[37] P. Koopman, K. Devale, and J. Devale. Interface robustness testing: Experience and lessons learned from the ballista project. *Dependability Benchmarking for Computer Systems*, pages 201–226, 2008.

[38] M. Kovatsch. Demo abstract: Human-CoAP interaction with copper. In *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, pages 1–2. IEEE, jun 2011.

[39] M. Kovatsch, M. Lanter, and S. Duquennoy. Actinium: A restful runtime container for scriptable internet of things applications. In *2012 3rd IEEE International Conference on the Internet of Things*, pages 135–142. IEEE, oct 2012.

[40] B. Krebs. Hacked cameras, DVRs powered today's massive internet outage. `https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/`. Access in April 28, 2017.

[41] N. Kushalnagar, G. Montenegro, and C. Schumacher. RFC4919 - IPv6 over low-power wireless personal area networks (6LoWPANs): Overview, assumptions, problem statement, and goals, 2007.

[42] N. Laranjeiro, S. Canelas, and M. Vieira. Wsrbench: An on-line tool for robustness benchmarking. In *Proceedings - 2008 IEEE International Conference on Services Computing, SCC 2008*, volume 2, pages 187–194, 2008.

[43] R. d. J. Martins, V. G. Schaurich, L. A. D. Knob, J. A. Wickboldt, A. S. Filho, L. Z. Granville, and M. Pias. Performance analysis of 6LoWPAN and CoAP for secure communications in smart homes. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 1027–1034, 2016.

[44] L. Masinter, T. Berners-Lee, and R. T. Fielding. RFC3986 - uniform resource identifier (URI): Generic syntax, 2005.

[45] R. A. Maxion and R. T. Olszewski. Improving software robustness with dependability cases. In *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 496. IEEE Computer Society Press, 1998.

[46] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, dec 1990.

[47] A. Mohsen Nia and N. K. Jha. A comprehensive study of security of internet-of-things. *IEEE Transactions on Emerging Topics in Computing*, 5(4):586 – 602, 2016.

[48] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. RFC4944 - transmission of IPv6 packets over IEEE 802.15.4 networks, 2007.

[49] G. Moritz, F. Golatowski, and D. Timmermann. A lightweight SOAP over CoAP transport binding for resource constraint networks. In *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, pages 861–866. IEEE, oct 2011.

[50] R. Natella, D. Cotroneo, and H. S. Madeira. Assessing dependability with software fault injection. *ACM Computing Surveys*, 48(3):1–55, feb 2016.

[51] N. F. Neves, J. Antunes, M. Correia, P. Veríssimo, and R. F. Neves. Using attack injection to discover new vulnerabilities. In *Proceedings of the International Conference on Dependable Systems and Networks*, volume 27513, pages 457–466, 2006.

[52] A. Nordrum. Popular internet of things forecast of 50 billion devices by 2020 is outdated - IEEE spectrum. `http://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated`. Access in April 28, 2017.

[53] S. Notra, M. Siddiqi, H. Habibi Gharakheili, V. Sivaraman, and R. Boreli. An experimental study of security and privacy risks with emerging household appliances. In *2014 IEEE Conference on Communications and Network Security*, pages 79–84, 2014.

[54] L. O'Donnell. 8 DDoS attacks that made enterprises rethink IoT security. `http://www.crn.com/slide-shows/internet-of-things/300084663/8-ddos-attacks-that-made-enterprises-rethink-iot-security.htm`, apr. Access in April 27, 2017.

[55] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy Magazine*, 3(2):58–62, mar 2005.

[56] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, sep 2004.

[57] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler. Standardized protocol stack for the internet of (important) things. *IEEE Communications Surveys & Tutorials*, 15(3):1389–1406, 2013.

[58] M. Patton, E. Gross, R. Chinn, S. Forbis, L. Walker, and H. Chen. Uninvited connections: A study of vulnerable devices on the internet of things (IoT). *2014 IEEE Joint Intelligence and Security Informatics Conference*, pages 232–235, 2014.

[59] PEACH. CoAP peach pit user guide. `http://www.peachfuzzer.com/wp-content/uploads/CoAP.pdf`. Access in February 25, 2017.

[60] Probe-IT. CoAP white paper. `http://www.probe-it.eu/wp-content/uploads/2012/04/CoAP-whitePaper.pdf`. Access in March 03, 2017.

[61] PROTOS. Protos - OUSPG. `https://www.ee.oulu.fi/research/ouspg/Protos`. Access in February 25, 2017.

[62] D. Raggett. The web of things: Challenges and opportunities. *Computer*, 48(5):26–32, may 2015.

[63] R. A. Rahman and E. Dijk. RFC7390 - group communication for the constrained application protocol (CoAP), 2014.

[64] R. A. Rahman and B. Shah. Security analysis of IoT protocols: A focus in CoAP. In *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, pages 1–7. IEEE, mar 2016.

[65] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt. Lithe: Lightweight secure CoAP for the internet of things. *IEEE Sensors Journal*, 13(10):3711–3720, oct 2013.

[66] E. Rescorla and N. Modadugu. RFC6347 - datagram transport layer security version 1.2, 2012.

[67] Z. Shelby. RFC6690 - constrained restful environments (CoRE) link format, 2012.

[68] Z. Shelby. Constrained application protocol (CoAP) tutorial - youtube. `https://www.youtube.com/watch?v=4bSr5x5gKvA`. Access in March 10, 2017.

[69] Z. Shelby, K. Hartke, and C. Bormann. RFC7252 - the constrained application protocol (CoAP), 2014.

[70] Z. Shelby, M. Koster, C. Bormann, and P. van der Stok. I-D - CoRE resource directory, 2017.

[71] M. Sutton, A. Greene, and P. Amini. *Fuzzing - Brute Force Vulnerability Discovery*. 2007.

[72] R. T. Tiburski, L. A. Amaral, E. D. Matos, and F. Hessel. The importance of a standard security architecture for SOA-based IoT middleware. *IEEE Communications Magazine*, 53(12):20–26, dec 2015.

[73] M. Vieira, N. Laranjeiro, and H. Madeira. Benchmarking the robustness of web services. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pages 322–329. IEEE, dec 2007.

[74] C. Viho and F. Sismondi. F-interop - remote conformance & interop testing. `http://www.f-interop.eu/images/Articles/20160922_F-Interop_TPAC-WoT.pdf`. Access in February 24, 2017.

[75] S. Winter, C. Sârbu, N. Suri, and B. Murphy. The impact of fault models on software robustness evaluations. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 51, New York, New York, USA, 2011. ACM Press.

[76] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RFC6550 - RPL: IPv6 routing protocol for low-power and lossy networks, 2012.

Appendix A

Applications, Implementations and Products using CoAP

This appendix contains additional results from the internet searches conducted in Section 4.1. First, Table A.1 displays every CoAP implementation we found. Then, in Table A.2, we display information on Cloud Services or Platforms supporting CoAP. Finally, in Table A.3, we list every commercial product we found that uses CoAP. These results can be used as a starting point for other research on practical aspects of CoAP applications, implementations, products and its uses.

Table A.1: List of all CoAP Implementations/Libraries found. The "Last Updated" field was checked in March 2018. Implementations with no marks for the RFCs are the most immature ones, for which little to no information was found, except for the source code.

CoAP Implementation	**RFC 7252 Base**	**RFC 7641 Obs.**	**RFC 7959 Block**	**RFC 6690 Link**	**Language**	**License**	**Repository or Reference(s)**	**Last Updated**
aadya	✓	✓	●	●	Elixir	LGPLv3	`https://gitlab.com/ahamtech/coap/aadya`	12/03/18
aiocoap	✓	✓	✓	✓	Python	MIT	`https://github.com/chrysn/aiocoap`	07/03/18
anjay	✓	✓	✓	✓	C	Apache-2.0	`https://github.com/AVSystem/Anjay/tree/master/src/coap`	08/01/18
ArduinoCoAP					C++	GPLv2	`https://github.com/dgiannakop/Arduino-CoAP`	14/06/13
bosch-xdk-coap	✓	✗	✓	✓	C	Commercial	`https://xdk.bosch-connectivity.com/documents/37728/286250/XDK110_CoAP_Guide.pdf`, `http://xdk.bosch-connectivity.com/xdk_docs/html/group__coapgroup.html`	07/09/17
Californium	✓	✓	✓	✓	Java	EPL, EDL	`https://github.com/eclipse/californium/`	17/02/18
canopus	✓	✓	✓	✓	Go	Apache-2.0	`https://github.com/zubairhamed/canopus`	07/02/18
cantcoap	✓	✗	✗	✗	C++	BSD-2-Clause	`https://github.com/staropram/cantcoap`	26/09/16
Catarinum	●	●	●	✗	C#	?	`https://github.com/marcelcastilho/Catarinum`	08/05/12
ccoap	●	✗	✗	✗	C	Apache-2.0	`https://github.com/ipflavors/ccoap`	22/04/13
coap-on-lon					C#	MIT	`https://sourceforge.net/projects/coap-on-lon/`	29/12/16
coap-rs	✓	✗	✗	✗	Rust	MIT	`https://github.com/Covertness/coap-rs`	06/12/17
CoAP-simple-library	●	✗	✗	✗	C++	MIT	`https://github.com/hirotakaster/CoAP-simple-library`	10/03/18
CoAP.NET	✓	✓	✓	✓	C#	BSD-3-Clause	`https://github.com/smeshlink/CoAP.NET`	21/07/16
coapBlip	●	●	✗	✗	C/nesC	BSD-2-Clause	`http://tinyos.stanford.edu/tinyos-wiki/index.php/CoAP`	29/07/14
CoAPEmbedded					C++	MIT	`https://github.com/Tanganelli/CoAPEmbedded`	21/01/17
coapex	●	✓	✗	✗	Elixir	?	`https://github.com/lucastorri/coapex`	24/03/16
coapi					C++	MIT	`https://github.com/linkineo/coapi`	18/03/17
CoAPLib	✓	✗	●	●	C++	?	`https://github.com/jfajkowski/CoAPLib`	14/06/17

Table A.1 continued from previous page

CoAP Implementation	RFC 7252 Base	RFC 7641 Obs.	RFC 7959 Block	RFC 6690 Link	Language	License	Repository or Reference(s)	Last Updated
CoAPP	✓	✓	✗	✓	C++	MPL 2.0	`https://github.com/zerom0/CoaPP`	12/04/17
CoAPSharp	✓	✓	✓	✓	C#	LGPL	`http://www.coapsharp.com/`	12/06/14
CoAPthon	✓	✓	✓	✓	Python	MIT	`https://github.com/Tanganelli/CoAPthon`	01/02/18
Copper	●	✓	✓	✓	Javascript	BSD-3-Clause	`https://github.com/mkovatsc/Copper`	06/12/17
dart-coap	✓	✓	✓	✓	Dart	MIT	`https://github.com/shamblett/coap`	03/11/17
eCoAP	●	✗	✗	✗	C	MIT	`https://gitlab.com/jobol/ecoap`	30/01/16
Erbium	✓	✓	✓	✓	C	BSD-3-Clause	`https://github.com/contiki-ng/contiki-ng/tree/develop/os/net/app-layer/coap`	05/03/18
erika-coap	●	✗	✗	✗	C	GPLv2+LE	`http://rtn.sssup.it/index.php/research-activities/middleware-of-things/middleware-of-things/11-research-activities/35-coaperika`	2011
ESP-CoAP	●	●	●	●	C++	GPLv3	`https://github.com/automote/ESP-CoAP`	13/11/17
excoap					Elixir	MIT	`https://github.com/mbialon/excoap`	20/09/15
FreeCoAP	✓	●	✓	●	C	BSD-like	`https://github.com/keith-cullen/FreeCoAP`	18/01/18
gcoap	✓	✓	✗	●	C	LGPLv2.1	`https://github.com/RIOT-OS/RIOT/tree/master/sys/net/application_layer/gcoap`	16/02/18
gen_coap	✓	✓	✓	✓	Erlang	MPL 1.1	`https://github.com/gotthardp/gen_coap`	23/06/17
geog-coap					?	?	`https://sourceforge.net/projects/geog-server-embedded/`	07/10/16
go-coap	●	✗	✗	✗	Go	MIT	`https://github.com/dustin/go-coap`	14/02/17
h5.coap	●	●	●	✗	Javascript	MIT	`https://github.com/morkai/h5.coap`	27/01/14
hcoap	●	✗	✗	✗	Haskell	BSD-3-Clause	`https://github.com/lulf/hcoap`	25/02/16
iCoAP	●	●	●	✗	Objective-C	MIT	`https://github.com/stuffrabbit/iCoAP`	31/01/18
IoTivity-Constrained	✓	✓	✓	●	C	Apache-2.0	`https://github.com/iotivity/iotivity-constrained/tree/master/messaging/coap`	14/02/18
java-coap	✓	✓	✓	✓	Java	Apache-2.0	`https://github.com/ARMmbed/java-coap`	10/01/18

Table A.1 continued from previous page

CoAP Implementation	RFC 7252 Base	RFC 7641 Obs.	RFC 7959 Block	RFC 6690 Link	Language	License	Repository or Reference(s)	Last Updated
jcoap	✔	✔	✔	✔	Java	Apache-2.0	`https://gitlab.amd.e-technik.uni-rostock.de/ws4d/jcoap`, `http://ws4d.org/ws4d-jcoap/`	08/11/16
libcoap	✔	✔	✔	✔	C	GPLv2, BSD-2-Clause	`https://github.com/obgm/libcoap/`, `https://libcoap.net/`	06/03/17
libnyoci	✔	✔	✔	✔	C	MIT	`https://github.com/darconeous/libnyoci`	20/12/17
linted-coap	●	✘	✘	✘	C	Apache-2.0	`https://gitlab.com/linted/coap/`	22/07/16
lobaro-coap	✔	✔	✔	✔	C	MIT	`https://github.com/Lobaro/lobaro-coap`	05/03/18
lobaro-coapgo	✔	✔	✔	✔	Go	MIT	`https://github.com/Lobaro/coap-go/`	05/03/18
LosCoAP	●	✘	✘	✘	C	BSD-3-Clause	`https://github.com/zhoubo85/LosCoAP`	26/08/17
mbed-coap	✔	✔	✔	✔	C	Apache-2.0	`https://github.com/ARMmbed/mbed-coap`	14/03/18
microchip-coap	●	✘	●	✘	C	Commercial	`http://ww1.microchip.com/downloads/en/AppNotes/00002512A.pdf`, `http://ww1.microchip.com/downloads/en/DeviceDoc/release_notes_coap_library_v1_0_0.pdf`	08/08/17
microcoap	●	✘	✘	✘	C	MIT	`https://github.com/1248/microcoap`	05/02/16
MinT	●	✘	✘	✘	Java	GPLv2	`https://github.com/soobinjeon/MinT`	13/09/17
mongoose-coap	✔	✘	✘	✘	C	GPLv2, Commercial	`https://github.com/cesanta/mongoose/tree/master/examples/coap_server`	12/03/18
mr-coap	●	●	●	●	Java	BSD-3-Clause	`https://github.com/MR-CoAP/CoAP`	21/07/14
nanocoap	●	●	✘	●	C	LGPLv2.1	`https://github.com/kaspar030/sock/tree/master/nanocoap`	11/12/17
nCoAP	✔	✔	✔	✔	Java	BSD-3-Clause	`https://github.com/okleine/nCoAP`	29/03/17
node-coap	✔	✔	✔	✔	Javascript	MIT	`https://github.com/mcollina/node-coap`	21/02/18
node-coap-ed					Javascript	?	`https://github.com/errordeveloper/node-coap`	05/06/13
node-coap-old					Javascript	?	`https://github.com/errordeveloper/node-coap-old`	17/05/12

Table A.1 continued from previous page

CoAP Implementation	RFC 7252 Base	RFC 7641 Obs.	RFC 7959 Block	RFC 6690 Link	Language	License	Repository or Reference(s)	Last Updated
nodemcu-coap	●	✗	✗	✗	C/Lua	MIT	`https://github.com/nodemcu/nodemcu-firmware/blob/master/app/modules/coap.c`	02/08/16
NZSmartie.CoAPNet	✓	✗	●	●	C#	Apache-2.0	`https://github.com/NZSmartie/CoAP.Net`	19/03/18
openthread	✓	✓	✗	✗	C++	BSD-3-Clause	`https://github.com/openthread/openthread/tree/master/src/core/coap`	26/02/18
openwsn	●	✗	●	✗	Python	BSD-3-Clause	`https://github.com/openwsn-berkeley/coap`	31/08/17
particle-coap	●	✗	✗	✗	C++	?	`https://github.com/hirotakaster/CoAP`	02/08/17
PhpCoAP					PHP	?	`https://github.com/cfullelove/PhpCoap`	04/02/15
PicoCoAP	●	✗	✗	✗	C	BSD-3-Clause	`https://github.com/exosite-garage/PicoCoAP`	11/08/15
qcoap					C++	MIT	`https://github.com/romixlab/qcoap`	12/12/15
QtCoap	●	●	●	✓	C++	GPLv3	`https://github.com/t-mon/qtcoap`	22/09/16
ruby-coap	✓	✓	●	✓	Ruby	MIT	`https://github.com/nning/coap`	26/01/16
scala-CoAP					Scala	?	`https://github.com/fbertra/scala-CoAP`	17/12/16
Soletta-CoAP	✓	✓	✗	✗	C	Apache-2.0	`https://github.com/solettaproject/soletta/tree/master/src/lib/comms`	23/08/16
SSN_Milli_CoAP	✓	●	✗	●	C++	MIT	`https://github.com/kurtgo/SSN_Milli_CoAP`	01/06/17
SwiftCoAP	✓	✓	✓	●	Swift	MIT	`https://github.com/stuffrabbit/SwiftCoAP`	17/04/17
tcoap	●	●	●	✗	C	BSD-2-Clause	`https://github.com/Mozilla9/tiny-coap`	16/02/18
TinyCoAP	●	●	✗	●	C/nesC	BSD-2-Clause	`https://github.com/AleLudovici/TinyCoAP`	11/11/13
tokio-coap					Rust	MIT, Apache-2.0	`https://github.com/azdle/tokio-coap`	19/11/17
txThings	✓	✓	✓	✓	Python	MIT	`https://github.com/mwasilak/txThings`	17/01/18
Waher.Networking.CoAP	✓	✓	✓	✓	C#	Waher	`https://github.com/PeterWaher/IoTGateway/tree/master/Networking/Waher.Networking.CoAP`	09/02/18

Table A.1 continued from previous page

CoAP Implementation	**RFC 7252 Base**	**RFC 7641 Obs.**	**RFC 7959 Block**	**RFC 6690 Link**	**Language**	**License**	**Repository or Reference(s)**	**Last Updated**
webiopi-coap	●	✘	✘	✘	Python	Apache-2.0	`http://webiopi.trouch.com/DOWNLOADS.html`	02/10/15
YaCoAP	●	✘	✘	✔	C	MIT	`https://github.com/RIOT-Makers/YaCoAP`	25/11/16

Table A.2: Cloud Services and Platforms Supporting CoAP.

Platform/Application Name	**Reference Repository**	**Additional Notes and References**	**Language**
Thingsboard	`https://github.com/thingsboard/thingsboard`	Uses Californium. `https://thingsboard.io/docs/reference/coap-api/`	Java
AVSystem Coiote	Closed Source	Device Management Server, including support for CoAP and LWM2M. `https://www.avsystem.com/products/coiote/`	Java
Creator Device Server	`https://github.com/CreatorDev/DeviceServer`	LWM2M Management Server. Uses CoAP.net. `http://console.creatordev.io/`	C#
FIWARE Orion NGSI Lab	`https://github.com/telefonicaid/fiware-orion`	Supports CoAP and LWM2M. Probably uses node-coap. `https://catalogue.fiware.org/enablers/backend-device-management-idas`; `https://github.com/telefonicaid/lightweightm2m-iotagent`; `https://account.lab.fiware.org/`	C++, Javascript
ARM mbed Cloud	Closed Source	Supports CoAP and LWM2M. `https://connector.mbed.com/`; `https://cloud.mbed.com/`	?
thethings.io	Closed Source	Supports CoAP. `https://thethings.io/`	?
ARTIK Cloud Services	Closed Source	Supports CoAP and LWM2M. Uses Californium. `https://my.artik.cloud/`; `https://developer.artik.cloud/documentation/data-management/coap.html`; `https://developer.artik.cloud/`	Java
Chordant	Closed Source	Supports CoAP, LWM2M and oneM2M. `https://www.chordant.io/`	?
Exosite Murano	Closed Source	Uses libexositecoap, which in turn uses PicoCoAP. `http://docs.exosite.com/portals/coap/`; `https://exosite.com/iot-platform/`	?
Meshblu's Octoblu	`https://github.com/octoblu/meshblu`	Uses node-coap. `https://meshblu-coap.readme.io/docs/devices`; `https://github.com/octoblu/meshblu-core-protocol-adapter-coap`	Javascript
Autodesk Fusion Connect	Closed Source	Supports CoAP. `https://autodeskfusionconnect.com/`	?
SiteWhere	`https://github.com/sitewhere/sitewhere`	Uses Californium. `http://www.sitewhere.org/`	Java

Table A.3: Commercial Products using CoAP.

Product	References
Molex Lights CoAP	http://www.transcendled.com/
Cisco Digital Building Solution	https://www.cisco.com/c/dam/en/us/solutions/collateral/workforce-experience/digital-building/digital-building-partner-guide.pdf; https://www.cisco.com/c/dam/en/us/solutions/collateral/workforce-experience/digital-building/molex-digital-building-design-guide.pdf; https://www.cisco.com/c/dam/en/us/solutions/collateral/workforce-experience/digital-building/molex-digital-building-implementation-guide.pdf; http://www.microchip.com/design-centers/ethernet/ethernet-of-everything/firmware/page/2; https://www.cisco.com/c/en/us/solutions/digital-ceiling/partner-ecosystem.html
Microchip Smart Lighting	http://www.microchip.com/promo/smart-connected-lighting; http://ww1.microchip.com/downloads/en/AppNotes/00002512A.pdf; http://ww1.microchip.com/downloads/en/DeviceDoc/release_notes_coap_library_v1_0_0.pdf
IKEA Tradfri	https://www.ikea.com/us/en/catalog/categories/departments/lighting/36812/
U-Blox SARA R4/N4 LTE Modules	https://www.u-blox.com/en/product/sara-r4n4-series; https://www.u-blox.com/sites/default/files/SARA-R4-N4_ProductSummary_%28UBX-16019228%29.pdf
Centero WEE Thread Module	http://centerotech.com/product/wee-thread-module/#how_to_buy