Università della Calabria

Department of Computer Science, Modeling,
Electronics and Systems Engineering (DIMES)

PhD in Information and Communication Technologies

AI-driven Attack Detection and Binary Analysis for Enhancing IoT Systems Security

Supervisor:

Prof. Giancarlo Fortino

Candidate:

Claudia Greco

XXXVI Cycle

Contents

1

Introduction

The spread of Internet of Things (IoT) devices and their full integration into everyday life is one of the major factors defining the current technology landscape. With the embedding of computational power and persistent connectivity to the Internet, an ever-increasing number of everyday objects are now IoT devices and created new levels of automation, efficiency, and convenience. IoT devices are being used in a wide range of applications and ecosystems, including smart homes, healthcare, transportation, industrial settings, and daily living. By gathering and transmitting data, smart objects are enabling new possibilities for innovation and improvement. Considering their constant use and the access they have to our data, ensuring that these devices are safe is an urgent concern, exacerbated by the fact that they still lack of adequate security and safety measures, resulting in frequent disastrous security breaches and jeopardized privacy. The reason behind this concern is that while our world has become deeply interconnected with IoT devices, our ability to protect them from malicious soft-

ware, known as malware, and security attacks of various types has not kept pace. The tools and techniques we use to perform security analysis and assessment have lagged behind the rapid expansion of the IoT ecosystem.

IoT devices are indeed extremely appealing to attackers for several factors, rooted in the their resource limitations, their complex nature, and the diversity of technologies involved within the IoT paradigm. IoT devices are often designed with constrained computational capabilities, short battery life and modest memory resources. These characteristics are primarily dictated by the need to minimize device costs and energy consumption. As such, traditional security techniques, conceived and implemented for systems with wide computational power and memory, are unsuitable for IoT devices. For instance, cryptographic operations integrated into security protocols, can be computationally demanding and, consequently, resource-intensive. Adapting these mechanisms to IoT devices without compromising their functionality poses major challenges. This implies that resource efficiency is a factor to be taken into account when devising security solutions for IoT. The lack of standardized security measures in the IoT domain, combined with the inherent limitations of IoT devices, creates a fertile ground for for malicious actors, who see them as low-hanging fruits. To address this pressing issue, it is imperative to find innovative ways to adapt and enhance existing security analysis and assessment methods specifically for IoT environments.

When it comes to ensuring the security of an IoT system, we end up colliding with a vast landscape of technical challenges and responsibilities. System security cannot be treated as an individual and isolated facet, but rather as an intricate array of interconnected elements, each of which demands meticulous attention. IoT systems are complex ecosystems comprising many different hardware and software parts, each of which presents potential vulnerabilities. At the core of the IoT architecture lies the hardware layer, including sensors, actuators, and communication devices, which are susceptible to various forms of attacks, including physical tampering and eavesdropping. The software layer, which includes embedded firmware, operating systems, and application software, plays a central role in the security of IoT systems. Vulnerabilities in software can expose the system to a wide array of threats, such as remote code execution and malware injection. Ensuring software security requires following rigorous coding practices, program analysis, and the application of security patches to mitigate vulnerabilities. Every program running within the system necessitates rigorous analysis and testing to assess its safety and identify potential vulnerabilities. Nevertheless, enhancing software security is often compounded by the unavailability of source code, frequently not provided by developers. Hence, binary code analysis assumes a pivotal role in this context. Securing communication networks and protocols, which enable data transmission and command propagation, is also a task of paramount importance.

Security breaches at this layer may result in eavesdropping, man-in-the-middle attacks, and data leaks. Securing communications requires the deployment of robust encryption mechanisms, stringent authentication protocols, and the implementation of secure communication standards. The combination of all these components in IoT systems leads to a significantly large attack surface. Hence, in order to ensure the overall security of an IoT system, it is imperative to secure the integrity of each individual component comprising the system, but also considering the interactions among them. The complexity of the IoT ecosystem is further exaggerated by the diversity in the types of device components provided by different developers, with different protocols, frameworks, operating/file systems, and hardware architectures.

The IoT ecosystem involves indeed a wide range of diverse technologies. A plethora of specialized, resource-constrained devices exist, ranging from tiny microcontrollers to powerful single-board computers. These devices often employ diverse and custom-tailored operating systems and hardware platforms to cater to the specific demands of their respective applications. For instance, a wearable fitness tracker may utilize a real-time operating system (RTOS) tailored for low-power operation, while a connected home security camera may employ a unique combination of hardware and firmware designed for real-time image processing.

In the realm of traditional systems, standardized security measures prevail due to the large use of a few operating systems (e.g.

Windows, macOS, Linux) running on homogeneous hardware architectures (x86 or x86-64) and standardized TCP/IP protocols, along with commonly recognized applications. On the contrary, the Internet of Things presents an entirely distinct scenario, where a profusion of innovative hardware architectures, operating systems, and communication protocols complicates the security landscape. In the IoT domain, we encounter a spectrum of hardware architectures, including ARM, MIPS, SuperH, and PowerPC, and a new breed of operating systems such as FreeRTOS and VxWorks. Moreover, communication technologies in the IoT domain transcend traditional boundaries, encompassing ZigBee, BLE (Bluetooth Low Energy), SDR (Software Defined Radio), and NFC (Near Field Communication). This variety of possible technologies aggravates the challenge of defining standardized security techniques, especially if compared to the more limited scope of traditional systems.

1.1 Contributions

The scope of this thesis is to present the comprehensive contribution of my research conducted in the domain of *IoT security* throughout the doctoral program. In this regard, this thesis summarizes a journey of exploring and analyzing the state-of-art of security techniques, pinpointing their limitations and possible gaps when dealing with the IoT ecosystem, and subsequently conceiv-

ing and experimenting with innovative approaches, which are also herein presented [52, 46, 53, 51, 47, 17]. My research interest has been moving across different aspects of security, but with a major focus on the analysis of programs at the binary level. Within this thesis, the issue of analyzing binaries running on IoT devices is thoroughly examined from various perspectives. Specifically, code obfuscation and its uses for both software protection and malicious purposes is discussed, along with the importance of finding new approaches to detect obfuscated code and malicious code within programs, suitable in both traditional and IoT devices. Also, the challenges of adapting traditional binary analysis techniques, such as fuzzing, to IoT environments are discussed along with the proposal of an innovative approach.

Other aspects of IoT security are explored, such as that of attack detection. Specifically, the problem of jamming detection in wireless communication is addressed, besides that of Cross-Site Request Forgery attacks.

1.2 Organization

The remainder of this thesis is structured in two parts. The first part, which includes Chapters 2, 3, and 4 is centered on the analysis of programs at a binary level, in the realm of the detection of obfuscated code, malicious code, and possible vulnerabilities. Chapter 2 introduces the fundamental principles of code obfusca-

tion and signature scanning, a technique traditionally employed in malware detection. This chapter explains the inherent connection between obfuscated code and malicious code, highlighting the correlations we can discern between the detection of obfuscated code and that of malicious code, despite the presence of obfuscation. Chapter 3 provides an exhaustive examination of contemporary techniques prevalent in the scientific literature for code obfuscation detection. Additionally, it introduces a framework for the analysis and detection of obfuscated code. The chapter expounds upon the intricacies of this framework, emphasizing its potential to advance the field of obfuscation detection. Chapter 4 delves into a comprehensive overview of the traditional techniques applied in the domain of malware detection, along with their constraints when dealing with obfuscated code or IoT systems. Furthermore, an innovative methodology is introduced to address and surmount these limitations, paving the way for groundbreaking approaches in the realm of malware detection. Chapter 5 introduces the fundamental concepts of static and dynamic analysis. This section underscores the challenges inherent in applying dynamic analysis techniques, such as fuzzing, within the context of the Internet of Things (IoT), while also highlighting the limitations of current solutions in addressing these challenges. Within this chapter, a novel methodology designed to surmount these limitations is discussed. The second part of this thesis, which includes Chapter 6 and Chapter 7, concerns the detection of security attacks in specific

scenarios. Chapter 6 presents an analysis of common approaches employed for the detection of jamming attacks in wireless networks and discusses a novel approach based on machine learning algorithms to improve the detection of such attacks, proving its efficacy in challenging scenario such as that of drone networks. Chapter 7 introduces Cross-Site Requests Forgery attacks and describes the proposal of a deep learning-based approach for the detection of vulnerabilities related to such attacks. Finally, Chapter 8 brings together the key ideas portrayed in the previous chapters, offering a synthesis of insights and reflections.

Part I

Code Obfuscation and Malware Analysis

2

Code obfuscation

Malware, short for *malicious software*, refers to computer programs specifically crafted to perform malicious actions on computers, networks, and electronic devices. These actions may include unauthorized access, data theft, and system disruption. The consequences of malware infections can range from data breaches and financial losses to the compromise of critical infrastructure systems. It follows that the detection of malware is a priority in any digital system, whether it is traditional or IoT. Nevertheless, traditional security solutions, which are primarily designed for standard computing platforms, often fall short in effectively identifying malware specifically tailored for IoT devices. In order to address this emerging threat landscape, it is imperative to invest in the development of cross-device malware detection systems. However, the diverse range of IoT devices, each with its unique operating system and hardware architecture, makes it difficult to develop universal malware detection methodologies. It follows that IoT security solutions should be designed to transcend the inherent differences in

device types and architectures, enabling comprehensive protection against malware attacks across the IoT spectrum.

The most popular technique employed to detect the presence of malware within binaries is *signature scanning*. Signature scanning, mainly employed by antivirus systems, involves the meticulous search for specific invariant byte sequences within files or data streams, with the intention of identifying patterns associated with known threats or indicators of compromise. Specifically, anti-virus systems collect and store signatures in a database, that are used as a ground truth for new comparisons. When a user initiates a scan or accesses a file, the anti-virus starts examining the code of the program and its binary code, comparing it against the signatures in the database. Although this methodology proves successful to detect malware that has been seen before, its efficacy can be limited when dealing with binaries compiled for architectures other than the one related to the signature, or when the code has been obfuscated.

Code obfuscation is a software protection technique that is meant to make a program harder to understand for an attacker. It consists in applying a set of transformations that alter the form but not the behavior of the program. The use of obfuscation makes the process of reverse engineering significantly more difficult, thereby offering a degree of protection for valuable code. However, it is worth mentioning that obfuscation is not only used for code pro-

tection but also for malicious purposes: generally it can be used to hide malicious functionality within seemingly legitimate programs, but it is in the field of malware that obfuscation finds the most fertile ground. By using obfuscating transformations, malware writers are able to create variations of their malware that maintain the same functionality but have different forms and do not contain the known malicious patterns, thus evading detection by anti-virus programs based on signature scanning.

In this context, it is clear that detecting the presence of obfuscation within software executables is extremely important. Determining whether an executable has undergone an obfuscating transformation can help in detecting the presence of malicious code. However, the task of obfuscation detection is not as straightforward as it may seem, especially when attempting to verify the presence of obfuscation in a large number of binaries without actually executing them to avoid running malicious code. Moreover, obfuscation detection is not only important for malware detection but can also be used to evaluate the strength and stability of software protection solutions.

The analysis presented involves automatically extracting static features from binaries that are compiled using various compilers, optimization levels, and obfuscated using distinct transformations. Thanks to ML and XAI it is possible to dig deeper the effects of obfuscation on binaries and to provide a way to easily and effectively detect the presence of obfuscating transformations.

The rest of this chapter is organized as follows: Section 2.1 provides basic background on code obfuscation and its transformations, while Section 2.2 provides an insight on how obfuscation techniques have been employed by malicious actors for malware writing, describing how obfuscating transformations have been integrated in well known malware code to perform evil tasks.

2.1 Notions of Code Obfuscation

Code obfuscation refers to the process of transforming the syntax of a software without modifying its semantics, with the aim of making it harder for a reverse engineer to analyze it, with the goal of extracting information from it or altering its behaviour. This process is typically carried out by an obfuscation tool, or simply *obfuscator*. Given a program P, the obfuscator generates an obfuscated program P' by applying a set of transformations $T = \{t_1, t_2, ..., t_n\}$ which make P' much more harder for an adversary to analyze than the original code. These transformations are semantic-preserving: P' will be syntactically different but semantically equivalent to P. Most of the them consist in introducing bogus dead code or adding complexity to the control-flow. An obfuscator can manipulate the program before or after its compilation from the source code to the executable, or even while it is running. In the first two cases, the obfuscation is said to be *static*, while in the other it is said to be *dynamic*. In the case of static

obfuscation, transformations are applied before the actual execution of the program, while dynamic obfuscation implies that the transformations are repeatedly implemented while the program is running.

The transformations carried out by the obfuscator are not without a cost, since they often lead may lead the resulting program to be larger, slower, and with higher memory footprint. To limit the cost raised by the obfuscation, one or more transformations may be applied only to a limited piece of code or data, corresponding to the information that needs protection. Following the taxonomy proposed by Collberg et al. in [34], we can classify the code transformations in *layout transformations*, *control transformations*, and *data transformations*.

2.1.1 Layout Transformations

Layout obfuscation involves modifying the structural arrangement of a program, such as by renaming identifiers or removing debugging data, with the aim of reducing the informative content of the software code for reverse engineers. Since these operations are unidirectional, involving actions like substituting identifiers with arbitrary symbols and removing comments, unnecessary functions, and debugging information, the majority of layout obfuscations are irreversible. Although layout obfuscations do not prevent reverse engineers from deciphering the program through an examination

of the obfuscated code, they do, nonetheless, raise the cost associated with the process.

2.1.2 Control Transformations

These transformations are designed to limit an attacker's ability to track a program's control flow and construct analytical structures like control flow graphs. Control transformations encompass actions such as inserting deceptive control flow instructions, concealing the destinations of branch instructions, or flattening the program to obscure any evidence of structured programming.

2.1.3 Data Transformations

Data structures within the source application are obscured through a conversion process that transforms their original form into a more complex representation, rendering them resistant to analysis. We can conceptualize a data transformation as a function $E()$ that converts a variable V into an obfuscated representation. Consequently, all potential values that V can take on, along with any valid operations the program may perform on V, must also be translated into this novel representation.

Up to this point, code obfuscation has been presented as a potent weapon against the reverse engineering or tampering of malicious software. However, it is crucial to acknowledge that code obfuscation comes with significant drawbacks. First and foremost,

it does not constitute a definitive solution and, even if the program has been sufficiently obfuscated, the attacker is expected to be able to reverse engineer it eventually. Herein, it is fundamental to underscore that the primary goal of code obfuscation is not to permanently hinder reverse engineering, but rather to defer it as much as possible.

2.2 Use of Code Obfuscation in Malware

Although code obfuscation was originally developed for intellectual property protection of software, over time it became a precious resource for virus writers. This is because, even if many different detection strategies have been proposed, the most widespread methodology used by nowadays antivirus software is signature scanning [62]. A signature is an invariant sequence of bytes, most of the times extracted from the application code or raw file content, that is useful in order to uniquely identify a specific malware. In order to overcome this detection strategy, malware writers came up with the idea of *metamorphic* viruses, i.e. applications that reshape themselves while infecting new targets, so that their new variants will be able to perform the same tasks of the preceding generations while having a different syntactic structure [169]. This often complex task is performed with the aim of changing the invariant patterns used as malware signatures and it is achieved by means of a plethora of different code transformations, includ-

ing, but not limited to: permutation, garbage insertion, expansion, shrinking, register swaps, encryption [34]

Runtime packers (self-extracting archives that unpack themselves in memory when executed) and, especially, code obfuscation techniques are used more and more in order to make malware hard to analyze by a reverse engineer analyst and to detect by a signature scanner. In accordance to [13], more than 80% of malicious software makes use of runtime packers, and 50% of new malware samples are obtained by simply repacking existing malware. A similar trend can be observed when dealing with obfuscation, where many proposed approaches [44, 104, 125, 70] are able to successfully avoid antivirus detection based on signatures. From an historical point of view [169, 61], excluding packing, the first obfuscation technique seen in the wild is encryption [137, 138, 163]. Malware encryption works by encrypting the executable body that will be decrypted at runtime by a decryptor. The cryptographic keys used to encrypt the program are different at every infection, resulting in an always different encrypted body. Cascade, Win95/Mad and Win95/Zombie [151] are some of the first malwares relying on this methodology, sometimes even stacking multiple encryption layers, as in the case of Win32/Coke. It is important to notice that, even if the body is always different at every infection, the decryptor in most of the cases stays the same, so it is good place to search for invariant patterns to be used as signature. This limitation led to the birth of a new malware variety: *polymorphic viruses* [125].

By leveraging a plethora of different strategies [163, 31, 80] polymorphism allows to apply transformations even to the decryption routines, thus avoiding them to be an easy source of signatures. Win95/Marburg and Win95/HPS are the first malware using a real 32-bit polymorphic engine. In order to easily revamp non obfuscated malware into polymorphic, several polymorphic engines have been implemented, e.g. *"The Mutation Engine"* (*MtE*) [137]. Even if polymorphic viruses are quite effective in evading signature based antivirus scanners, they can easily be detected with dynamic approaches, since the body of the virus will be unencrypted in memory at runtime and using sandboxing techniques for controlled malware execution [137, 138] can lead to a convenient analysis and signature identification. To avoid the execution in controlled environments, different armoring techniques arose [137], but many of them turned to be ineffective with the improvements of sandboxing mechanism. An important milestone in malware history has been reached with the advent of metamorphic viruses [44, 70, 138, 163, 31, 80]. At each infection iteration, metamorphic viruses transform their own body, by applying mostly syntactic transformations, so that the resulting code is different from the generating one while preserving its functionalities.

There are various techniques employed by metamorphic and polymorphic malware, such as:

- *Register swapping*: while generating a new transformation of a virus the registers used in various instruction are altered.

It is a technique historically attributed to the malware writer Vecna for his virus Win95/Regswap [163], that can be easily dismantled by leveraging wildcard based signatures and regular expression scans.

- *Instruction substitution* refers to replacing certain instructions or groups of instructions with others that have the same functionality but are syntactically different [80].
- *Garbage instructions insertion* involves adding instructions that are not necessary for the execution of the program, with the goal of varying the malware body[11, 163, 80]. These instructions can be single operations or sequences that do not affect the state of the program, and may even be located in areas that will never be executed ("dead-code").
- *Transposition* refers to various techniques for reordering instructions while maintaining the original flow of the program [31]. One such technique involves randomly reordering some instructions and then using unconditional jumps to reconstruct the original flow. Another approach is to isolate independent groups of instructions and modify their order, eliminating the need for unconditional jumps. However, finding independent groups of instructions can be difficult, so developers often use simpler techniques, such as reordering subroutines within the malware. This approach was used by the Win32/Ghost malware and can result in up to $n!$ different variants, where n is the number of subroutines.

- *Code integration*, one of the most sophisticated techniques used in malware writing, was first used by the virus writer Zombie in Win95/Zmist (Zombie Mistfall). It involves decompiling the program to be infected, inserting malware code in between, and then reassembling the original program and malware into a single executable.

3

Code obfuscation detection

Binary obfuscation techniques are commonly employed to protect code against reverse engineering and piracy. Unfortunately, besides being used for legitimate purposes, virus writers also resort to obfuscation to evade antivirus detection mechanisms based on signature scanning. The capability of detecting obfuscation within software plays a crucial role in several scenarios, firstly and most importantly that of the ongoing fight against malware. The detection of obfuscated programs can be the first step to recognize potentially malicious software: obfuscation is indeed frequently employed by malware writers to make it more difficult for both security analysts and antivirus systems to understand and analyze malware. Herein, obfuscated code can be symptomatic of the presence of malicious code and, its prompt detection can therefore let researches to take the necessary actions, such as quarantining the software or deleting it. Furthermore, the detection of obfuscated programs can be a valid resource in the evaluation of software protection. Obfuscation detection can indeed be the first step in

the process of de-obfuscating a software. If a reverse engineer is able to detect the application of obfuscating transformations over the software, it can indicative of an inappropriate level of protection. Specifically, the detection of obfuscated code is crucial for the evaluation of software protection, malware detection, and virus scanner evaluation.

Detecting obfuscation is a task full of difficulties owing to the wide range of possible obfuscation transformations and the indistinguishability of obfuscated code. Researchers have dedicated significant effort to the development of methods for the detection of obfuscated code in software. These methods aim to identify instances of obfuscated code and classify the types of transformations applied to it. However, there is no standardized approach for the detection of obfuscated code. For this reason, I gave my contribution to this research field by proposing a new methodology, based on Machine Learning (ML) and eXplainable Artificial Intelligence (XAI), to explain the fundamental aspects that allow for the identification of binary obfuscation, that lead to the publication of the article [52]. The rest of this Chapter is organized as follows: Section 3.1 provides a comprehensive state-of-the-art of scientific literature on obfuscation detection. Section 3.2 introduces background notions of eXplainable Artificial Intelligence (XAI). Section 3.3 explains the methodology proposed by my co-authors and I in [52] and Section 3.4 shows the results that we obtained.

3.1 State of the art

Obfuscating transformations can be implemented on different programming languages and applications. It follows that, the particular methods employed for obfuscation may vary based on the target program and the objectives of the obfuscation process. Since almost all the academic research done on obfuscation detection targets technologies such as Android, Javascript, and binary code, the remainder of this section discusses scientific papers in these three categories.

3.1.1 Android

All the scientific papers related to the application of obfuscation detection to Android scenarios rely on the use of machine learning and deep learning techniques. Jiang et al. [66] presented a hybrid neural network model denoted as GCN-LSTM, generated from the integration of the Grapgh Convolutional Network (GCN) with the Long Short-Term Memory (LSTM) architecture. A dataset of Control Flow Graphs (CFGs), adjacency matrices, and basic blocks of functions was employed to train the model. The model was able to perform obfuscation detection in Android applications at both function level and APK-level. To discern and classify the presence of obfuscation in Android applications, Conti et al. [35] propose a range of deep learning models, employing statically extracted bytecode features. Besides using traditional machine learning methods,

they leverage algorithms tailored for image recognition and natural language processing. Ultimately, through the amalgamation of a hybrid model, they optimize the synergistic strengths of each approach. Their primary focus revolves around four obfuscation categories: class encryption, reflection, text encryption, and identifier renaming, culminating in promising outcomes. Bacci et al. [10] use binary classification models such as Support Vector Machines (SVM), Random Forest (RF), and Multi-Layer Perceptron (MLP) to perform obfuscation detection over a set of features statically obtained. The classifiers were used to detect specific obfuscating transformations such as changing package name, identifier renaming, data encoding, call indirections, code reordering, and junk code insertion. BLADE was presented by Sihag et al. [147] as an obfuscation-resilient framework for malware detection from opcode segments, based on machine learning. Algorithms such as J48 and k-Nearest Neighborhood (kNN) were used to detect obfuscation and distinguish among several obfuscating transformations. AndrODet, proposed by Mirzaei at al. [102], is a model based on online learning algorithms, including Decision Tree, SVM, kNN and Random Forest, trained over datasets of statically extracted features. The proposed framework is able to identify obfuscating transformations such as identifier renaming, string encryption, and control flow obfuscation. Machine learning classifiers such as Random Forest, AdaBoost, Extra Tree, and Linear SVM were used by Park et al. [115] over a dataset of vectorized Java codes to

detect obfuscation in Android applications at class-level. Dong et al. [43] also used machine learning to detect obfuscating transformations such as identifier renaming, string encryption and Java reflection. Another machine learning framework using SVM was used by Wang et al. [159] to detect the obfuscator used to implement obfuscation choosing among ProGuard, Allatori, DashO and others on Android APKs. The model was trained over a dataset of strings extracted from the data section of the program bytecode, including names of packages, classes, methods and fields. OBFUSCAN [161] was developed by Wermke to detect obfuscating transformations applied by the ProGuard obfuscator, such as name overloading, debug data obfuscation, annotation obfuscation, string encryption, and DEX data encryption. Mohammadinodooshan et al. [103] introduced a Naive Bayes-based generative classification approach, specifically tailored for the identification of obfuscated strings.

3.1.2 Javascript

Choi et al. [30] present a tool for the detection of obfuscated strings in Javascript codes. Statically extracted features such as n-gram, entropy and word size were employed to compare regular and malicious Javascript codes. JSObfuscDetector [67] is a framework for the detection of obfuscated Javascript code, based on an ensemble learning approach. The detection is performed by analyzing notable changes either in lexical aspects and in the structure of

code. Several papers [5, 105, 73, 19] discuss the use of features obtained from the Abstract Syntax Tree representation of Javascript codes as input for machine learning models. JSOD [5] is a static JavaScript obfuscation detector that extracts semantic information from the AST, Moog et al. [105] rely on features extracted both automatically and manually from the AST augmented with information related to control and data flow and tokens, while NOFUS [73] employs features representing context and text of the AST nodes. Finally, Blanc et al. [19] apply pattern matching to compare the AST representation of code against subtrees representing obfuscating transformations. Sheneamer et al. [**?**] proposed a machine learning-based framework for detecting Java code clones and obfuscated code in target pairs of codes. Their framework relies on the use of an ensemble of various state-of-the-art classification models, such as Naïve Bayes, LibLinear SVM, Random Forest, and others, fed with features derived from Java bytecode dependency graphs (BDG), program dependency graphs (PDG) and abstract syntax trees (AST). The framework was tested for the detection of obfuscating transformations such as contraction, expansion, loop transformation, and renaming.

3.1.3 Binary Code

While the scientific literature abounds with numerous papers addressing obfuscation detection in Android and JavaScript environments, the research on obfuscation detection in binary programs

remains relatively underexplored. REDIR [148] is a framework for static detection of anti-debugging obfuscating transformations that relies on a rule engine sensemaking process with the aid of intermediate representation. However, the choose of BAP as intermediate language limits the efficiency of the proposal, due to the inability of BAP to produce anti-debugging methods that generate cycles such as loops. Salem et al. [135] discuss the use of Term Frequency Inverse Document Frequency (TF-IDF) features to feed machine learning models, including Naive Bayes and Decision Tree. Tofighi-Shirazi et al. [154] present a framework based on the combination of semantic reasoning and advanced machine learning. Jiang et al. [66] use a hybrid neural network model called GCN-LSTM, resulting from the combination of Graph Convolutional Network (GCN) and Long Short-Term Memory (LSTM) to detect obfuscation in x86 assembly code at function-level. DynODet [78] is a dynamic obfuscation detection tool based on binary instrumentation, able to distinguish six obfuscating transformations commonly found in benign code, such as self-modification, section mislabel obfuscation, dynamically generated code, unconditional to conditional branch obfuscation, exception-base obfuscation, and overlapping code sequences, from obfuscation in malicious code.

Various studies have tackled the issue of Binary Code Similarity Analysis (BCSA), aiming to discern similarities between code segments optimized at different levels and under obfuscation

transformations. BCSA finds utility in several security-related domains, such as detecting plagiarism and uncovering vulnerabilities. Nonetheless, conducting novel research in this realm is challenging due to the absence of easily interpretable machine learning techniques and a scarcity of publicly available source code and datasets in the existing literature. BCSA also can be employed the task of obfuscation detection, as it allows for the comparison of obfuscated binary code from a program with a known, non-obfuscated version of the same program. This enables the identification of discrepancies that may arise as a consequence of obfuscation. TREX [117] is a machine learning-based tool able to match functions similar from a semantic perspective, by looking at the semantics of their execution. TREX was trained on the execution semantics of functions inferred from their dynamic traces. OFCI [122] enables the identification of function clones in obfuscated binaries. This framework examines execution traces to inspect the effect of known function calls on function similarity. However, the framework results show mixed precision and recall values. Kim et al. [79] proposed TIKNIB, a BCSA utility that employs simple interpretable models. The results achieved by TIKNIB resemble those of state-of-the-art deep leatning-based tools. IFattn [65] is a tool for binary code semantic similarity detection which merges basic presemantic features yielding to more expressive semantic features.

3.2 Explainable Artificial Intelligence

Making AI models understandable to human users is a major difficulty and has given rise to the important topic of Explainable AI (XAI). It is especially pertinent in the field of cybersecurity, where AI-based cybersecurity solutions may have an advantage over conventional solutions in terms of performance, but may fall short in terms of being able to explain and make sense of the outcomes (ranging from detection and prediction to reasoning and decision-making) to humans. A widely accepted taxonomy of XAI techniques is as follows:

- *Model Agnostic/Specific*: If the interpretation technique is limited to a certain model, it is considered model-specific. Otherwise, it is referred to as model-agnostic. Agnostic approaches can make any machine learning model more interpretable, but they may or may not have access to internal model data such as weights and structural information.
- *Intrinsic/Extrinsic*: Models can be either interpretable in and of themselves (intrinsic) or require interpretation techniques to be applied after training in order to be made interpretable (extrinsic). Intrinsic models include decision trees and other straightforward models, while the application of interpretation techniques post-training is extrinsic.
- *Local/Global*: Local interpretation refers to techniques that represent a single instance of a model's behavior, while global

interpretation refers to techniques that describe the complete dataset.

3.2.1 Model Interpretability using SHAP

The SHapley Additive exPlanations (SHAP) method is an approach rooted in cooperative game theory and specifically in the calculation of the Shapley value [145]. This value is a measure of the influence of a player in forming coalitions. A coalitional game in game theory is composed of a set of N players and a characteristic function, v, which maps player subsets, $S \subseteq 1, 2, \cdots, N$, to a real value, $v(S)$, that represents the collective payoff obtained by the players in the subset when they cooperate. The Shapley value is calculated as a weighted average of a player's marginal contributions to all possible player subsets, where the marginal contribution, $\Delta v(i, S)$, of player i with respect to coalition S is defined as the additional value generated by the inclusion of player i in the coalition. In the context of XAI, given a model prediction, $m(f_1, f_2, \cdots, f_d)$, features 1 through d can be viewed as players in a game, where the payoff, $v(S)$, with $S \subseteq f_1, f_2, \cdots, f_N$, is a scalar prediction calculated from a subset of feature values. The calculation of the Shapley value, ϕ_i, of each feature can then be seen as the contribution of the feature to the model prediction:

$$\phi i = \sum S \subseteq F \setminus i \frac{|S|!(|F| - |S| - 1)}{|F|!} \Delta v(i, S) \qquad (3.1)$$

where F is the set of all features, S is a subset of F, and $\Delta v(i, S)$ is the marginal contribution of feature i, i.e., the difference between the model prediction with feature i and the model prediction without feature i. Computing the exact Shapley value requires evaluating the characteristic function a factorial number of times and the algorithm has a computational complexity of $O(2^n)$, where n is the number of features. This makes it computationally infeasible to train a ML classifier on all possible feature subsets and to compute the exact Shapley value. To address this issue, several approximation techniques for efficiently calculating the Shapley value have been proposed in recent years. These methods differ in their assumptions and computational methods for approximating $m(i, S)$ in Equation 3.1 with $E[f(x)|x_S]$, which is the expected value of the function conditioned on a subset S of the input features.

3.3 Proposal

The methodology presented consists of a machine learning-based approach for obfuscation detection with the application of XAI to detect how features related to program properties were affected from specific transformations. In order to conduct our study, two different datasets were built of 8610 tuples and 12 features each, combining non obfuscated and obfuscated programs. The methodology followed during the process of creation of the two datasets is summarized in Figure 3.1. First of all, the GNU `coreutils`

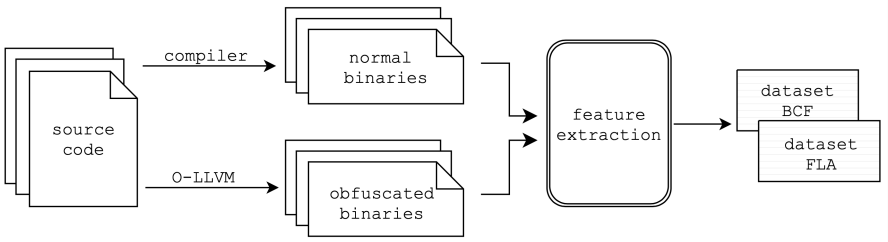


Fig. 3.1. *Dataset generation*

utilities[1] using both the GNU Compiler Collection[2] (`GCC`) versions 4.9.4, 5.5.0, 6.4.0, 7.3.0, and 8.2.0, as well as `Clang`[3] versions 4.0, 5.0, 6.0, and 7.0 were compiled. Compiler optimization too was taken into account, since it plays a role resembling that of obfuscation, by transforming the original code into a new form in order to improve performance and/or code size at the expense of compilation time and possibly the ability to debug the program Five different optimization levels were used, ranging from `O0` to `O3`, plus the code size optimization option `Os`. The reason behind the choice of taking into consideration multiple optimization levels is to empower the classification process by enhancing the classification model to be able to distinguish between obfuscating transformations and optimization. Conversely, the obfuscated binaries were generated by compiling the source code by means of `Obfuscator-LLVM` [71]. For each dataset a distinct obfuscating transformation was utilized: *Control-Flow Flattening* and *Bogus Control Flow*. As the name implies, the *Control Flow Flatten-*

[1] https://www.gnu.org/software/coreutils/
[2] https://gcc.gnu.org/
[3] https://clang.llvm.org/

ing [158, 88] transformation completely flattens the control flow graph of a program. M ore in detail, the connections between basic blocks, including both conditional and unconditional edges, are redirected to a dispatcher node. The dispatcher node employs an artificial variable to determine the subsequent block to which to jump. This variable is updated at the conclusion of each individual basic block. The *Bogus Control Flow* transformation works by modifying a function call graph by adding new basic blocks, situated prior to the original basic blocks. This new basic blocks incorporates an opaque predicate, followed by a conditional jump to the original basic block. Additionally, the original basic block is replicated, and randomly selected junk instructions are inserted to fill the duplicated block. This method offers a modified function call graph, as the execution path is altered through the use of the opaque predicate and the conditional jump. The utilization of junk instructions in the replicated basic block serves to further obfuscate the code. We implemented a feature extraction script to obtain a set of 12 features, reported in table 3.1, including the class label, representing relevant properties of the target program. It is important to note that all the features are function related, since the chosen obfuscating transformation can be applied to specific functions. The mechanism of feature extraction utilized in this study is purely static in nature. This approach was deemed necessary to simulate a realistic scenario where the execution of obfuscated code may result in the execution of malicious

code. Hence, a static extraction methodology was employed with the intention of eventually incorporating our approach into a detection system so that alerts can be generated when obfuscating transformations are identified.

Table 3.1. Extracted features

Feature	Meaning
f_cyc_complex	Cyclomatic complexity
f_cycle_cost	Function cycles cost
f_loop_count	Loop count
f_n_bb	Number of basic block
f_n_locals	Number of local variables
f_arit	Percentage of arithmetic instructions
f_logic	Percentage of logic instructions
f_control_flow	Percentage of control flow instructions (e.g. `jmp`)
f_stack	Percentage of stack operations (e.g. `push` and `pop`)
f_memory	Percentage of memory access operations
f_program_flow	Percentage of program flow instructions (e.g. `call` and `ret`)

Our datasets were preprocessed and fed into a *XGBoost* classifier with a *k-fold cross validation* step. The choice of XGBoost came from its strengths, such as the ability to produce interpretable models, as the algorithm is based on decision trees that can be visualized and analyzed. This makes it easier to understand the relationships between features and predictions, and to provide explanations for individual predictions. Additionally, XGBoost has been integrated with various XAI techniques, such as feature importance scores and partial dependence plots, which can

help to provide further insights into the decision-making process of the model. In our analysis, we use the SHapley Additive exPlanations (SHAP) [97] as a game theoretic approach to explain the output of our machine learning model due to its several advantages over other popular methods. One of the main advantages of using SHAP is that it is an extrinsic technique, which makes it suitable for ex-post analysis. SHAP is also model-agnostic, which means that it can be used with a wide range of models, including XGBoost. This is in contrast to other popular methods such as Lime [127] and Anchors [128], which are local-based and have limitations when it comes to being used with different models. Additionally, while Gradient-weighted Class Activation Mapping (GRAD-CAM) [140] and other gradient-based approaches are extrinsic, they are model-specific and may not be suitable for all models. The use of SHAP not only provides a local interpretation of a prediction, but it also provides a global interpretation. This means that it can provide a comprehensive understanding of how a prediction was made, taking into account all of the features used in the model. This is especially important for interpretability and understanding the reasoning behind a prediction. In this paper, we make use of a TreeExplainer [95], which is an efficient algorithm for computing SHAP values for additive tree-based models such as random forests and gradient boosting machines. The core idea behind the algorithm is to estimate $E[f(x)|x_S]$ by observing the proportion of training set samples that match the condition x_S

Table 3.2. Detection results

	Control-Flow Flattening	Bogus Control-Flow
Accuracy	0.996	0.984
Precision	0.997	0.985
Recall	0.974	0.888
F1-score	0.985	0.934

and fall into each leaf node. The major advantage of this method is that it reduces the computational complexity of exact SHAP value computation from exponential to low order polynomial for trees and sums of trees (since the SHAP values of a sum of two functions is the sum of the original functions' SHAP values)[96]. Finally, it is important to notice that SHAP does not rely on a feature independence assumption.

3.4 Results

The results of the obfuscation detection task are presented in Table 3.2. It can be observed that exceptional rates are achieved in both datasets, reinforcing the hypothesis that obfuscation detection can be effectively performed through a machine learning (ML) based approach utilizing static features. Despite the robust evidence of the quality and efficacy of the obfuscation detection approach provided by the results, the focus of this study extends beyond mere detection performance. Of greater interest is gaining insights into the obfuscation process by analyzing how each obfuscating transformation influences the features and using this

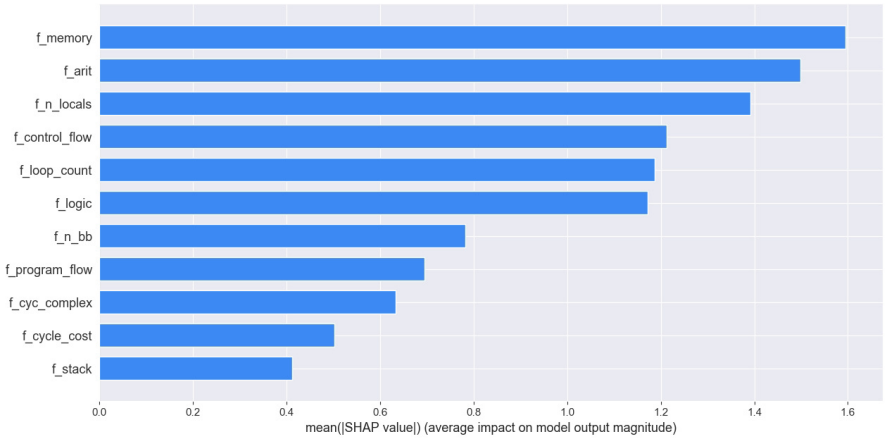


Fig. 3.2. *Global Feature Importance: Bogus Control Flow Transformation*

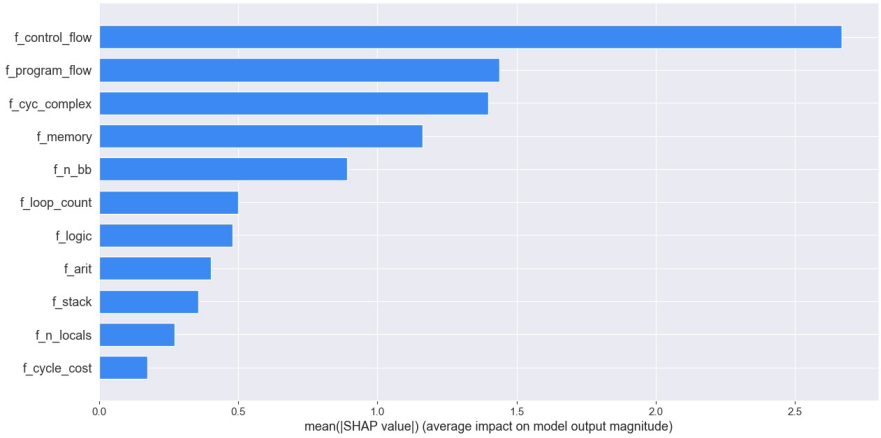


Fig. 3.3. *Global Feature Importance: Flattening Transformation*

information to detect obfuscation or invariant features. Invariant features, in particular, are valuable as they can be utilized to generate malware signatures that are resistant to obfuscation. As depicted in Figures 3.2, 3.4, 3.3 and 3.5, the features involved in each obfuscating transformation are notably distinct and they offer many points of reflection.

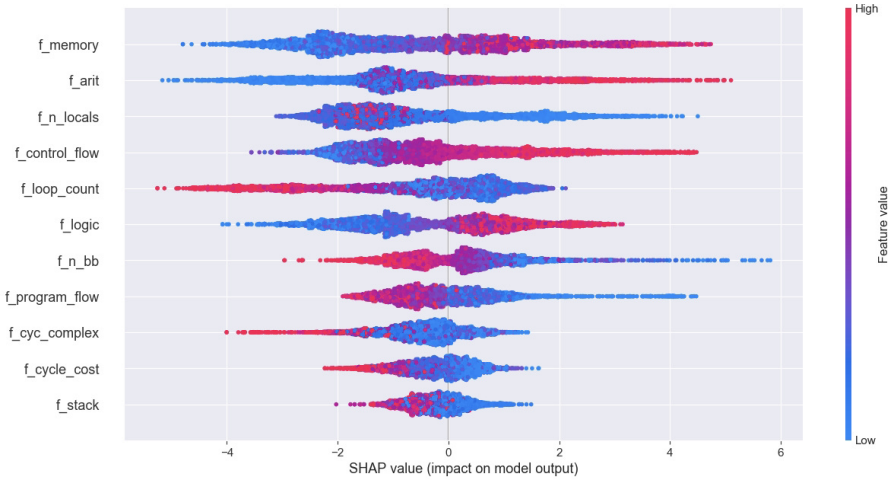


Fig. 3.4. *Beeswarm Plot: Bogus Control Flow Transformation*

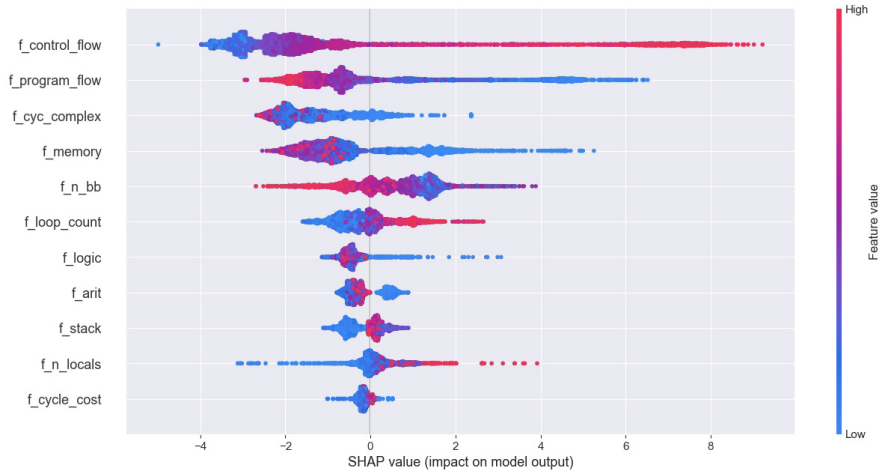


Fig. 3.5. *Beeswarm Plot: Flattening Transformation*

It is extremely interesting, for example, to note how some features are significative only when they assume a low or high value. As an example, let us take into consideration the value of cyclomatic complexity (a measure of the number of linearly independent paths through the function source code). In Figure 3.5 we can ob-

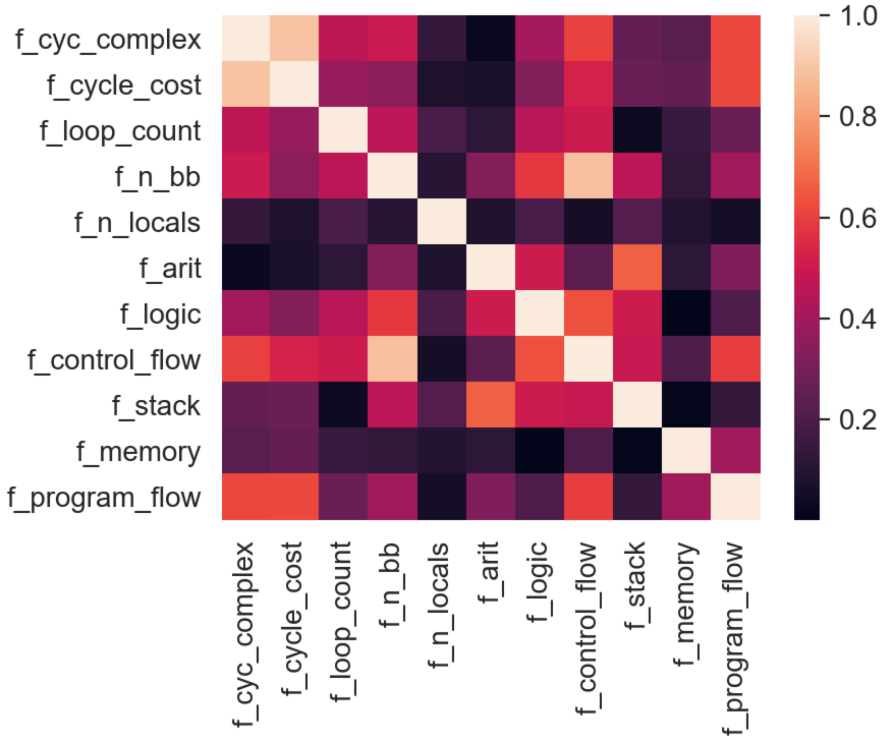


Fig. 3.6. *Correlation Plot: Bogus Control Flow Transformation*

serve that when its value is high, then there are high chances that the program has not been flattened, but we cannot state the inverse if the value is high. This outcome of the XAI procedure is of great significance as it enables us, through examination of the extracted features of a given sample, to determine which obfuscating transformation was applied. This information is extremely valuable in the task of deobfuscation and reverse engineering. The results of the correlation matrices in figures 3.6 and 3.7 generated using SHAP suggest that there is a low level of correlation among the features in the datasets. We can observe that the simi-

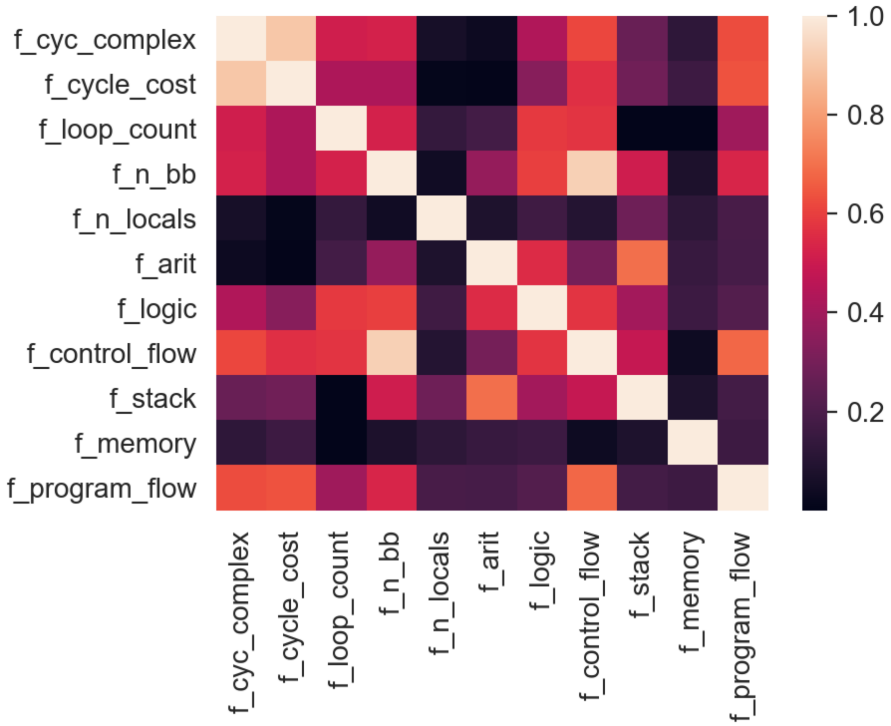


Fig. 3.7. *Correlation Plot: Flattening Transformation*

larity of the experiments, both related to control transformations, is reflected on the similarity of the correlation matrices. The low level of correlation indicates that the features in the datasets are relatively independent and therefore likely to provide unique information about the transformations under investigation.

4

Malware Detection

The proliferation of the Internet of Things (IoT) has given rise to a significant issue in malware detection, whereby a substantial portion of malware comprises repurposed versions of existing malware, recompiled to function on new architectures such as ARM and MIPS. This poses a formidable challenge to signature-based malware detection systems, as the compiled variants differ significantly due to the distinct instruction sets employed by IoT devices. Consequently, virus authors easily achieve one of their primary objectives, namely, evading antivirus scanners through the straightforward adaptation of their preexisting source code. To address this problem, it becomes evident that a paradigm shift is required. Instead of solely focusing on the generated code, a higher level of comprehension must be embraced, wherein considerations pertaining to the instruction set and architecture cease to be decisive factors in categorizing a program as malicious or benign. In this Chapter, the proposal of a novel approach for enhancing signature scanning called SigIL [47] is discussed, along with an

analysis of current state-of-the-art solutions and their main limitations. The novelty of the proposal made by my co-authors and I relies in a shift of the focus of analysis, from a binary level to its intermediate representation. The rest of this Chapter is organized as follows: in Section 4.1 an overview of traditional malware detection techniques is provided. Section 4.2 discusses current research carried out in the realm of signature scanning and pattern matching. Section 4.3 provides basic notions of Intermediate Representation and Section 4.4 introduces SigIL and explains its inner workings.

4.1 Traditional Malware Detection Techniques

The traditional approach to malware detection is primarily rooted in the utilization of signatures and cryptographic hashes. A signature is an invariant sequence of bytes, most of the times extracted from the application code or raw file content, that is useful in order to uniquely identify a specific malware. More formally, following the definition in [20], Let P represent the set of software programs, and let $M \subset P$ be the set of malicious programs. We associate M with a set of signatures, denoted as $\mathbb{M}$. A detector, denoted as $D : P \times \mathbb{M} \rightarrow 0, 1$, operates on these signatures. Therefore, we assert that a program p is considered detected if there exists a signature $m \in \mathbb{M}$ such that $D(p, m) = 1$. When we apply this definition to the conventional signature scanning method, the set $\mathbb{M}$

encompasses the collection of all known malware string signatures. The corresponding detector is defined as follows: $D(p, m) = 1$ if m is a substring of p, and $D(p, m) = 0$ otherwise.

In the field of malware detection, a program signature can be compared to those of known malicious programs, in order to detect if the target program is benign or not. Another conventional approach utilized in the realm of malware detection is based on the utilization of cryptographic hashes. A cryptographic hash, also known as a message digest, is a fixed-length value derived from a cryptographic hash algorithm. This algorithm takes a program of arbitrary size as input and produces a hash that represents the program. By comparing the cryptographic hash of a target program to that of a known program, malicious behaviors can be detected. Traditional techniques such as signature scanning and cryptographic hashing exhibit significant weaknesses, as they are only capable of detecting a file if it contains the exact sequence of bytes representing the signature, or if two files are entirely identical. In signature-based detection, even a slight alteration of a single bit within the byte sequence representing a signature can lead to the failure of the signature matching mechanisms, resulting in false negatives. Similarly, in the case of cryptographic hashes, even the change of a single bit of the input program will produce a completely different cryptographic hash as the output. This is a limit in the malware detection application, where virus writers rely on obfuscation to hide the functionality of their code.

4.2 State of the art

Several approaches have been explored with the aim of identifying malicious patterns through signature scanning and pattern matching techniques. One of the most popular tools in this regard is YARA [8], which is commonly utilized within many antivirus software. YARA is a powerful pattern-matching tool that allows for the creation and detection of signatures based on sequences of bytes. By employing a flexible rule-based syntax, YARA enables analysts to define complex patterns and conditions for identifying specific malware or other program attributes. While YARA is a valuable resource, it is important to acknowledge its limitations: one of the primary constraints lies in the reliance on byte-level signatures, which can be susceptible to obfuscation techniques employed by adversaries. Malicious actors may employ various methods to evade signature-based detection, such as encryption, polymorphism, or packing, which can alter the byte-level representation of the program and render the signatures ineffective. Alternative approaches are employed for malware detection to assess program similarity, by means of different types of signatures, such as program hashes. Hash-based signatures involve generating unique hash values for programs and comparing them with known hashes to determine similarity. This approach is effective in identifying identical or near-identical programs, regardless of

their byte-level representation. Hashdeep[1] employs rolling hash algorithms to calculate and store file or data block hashes, enabling efficient comparison for similarity detection. However, it may not capture fine-grained differences within files, potentially leading to false negatives. Fuzzy hashing, utilized by tools like ssdeep [81] and tlsh [111], involves dividing files into blocks, computing block hash values, and concatenating them to create a fuzzy hash. Ssdeep can handle variations within files, offering a more robust measure of similarity compared to traditional cryptographic hash functions, along with a scoring mechanism to quantify similarity. Tlsh employs locality-sensitive hashing, focusing on statistical data properties to calculate a hash value representing local file structure, suitable for identifying similar files within large datasets. SDhash [131] utilizes locality-sensitive hashing (LSH) to identify file similarities based on the statistical distribution of short contiguous blocks within files, making it valuable for near-duplicate or closely related file detection.

Some recent endeavors have delved into the domain of graph matching for enhanced detection capabilities, where Control Flow Graphs (CFGs) are employed as program signatures. Specifically, these approaches involve the generation of CFGs from the programs under analysis, which are subsequently compared against a database of CFGs associated with known malware instances [20, 6, 26].

[1] https://github.com/jessek/hashdeep

Several approaches leveraging machine learning and deep learning techniques have also been proposed for malware detection [27, 52, 136] or other analysis techniques based on ad-hoc encoding of security related data [60, 59, 56]. Some of these methods specifically exploit features extracted from intermediate program representations [121, 171]. This entails using the intermediary forms of programs to extract meaningful characteristics, ultimately enhancing the performance of machine learning models in identifying malware and uncovering irregular patterns in software.

Other useful approaches to mitigate the variability introduced by metamorphic transformations are code optimization [120], imposing an order on the instructions [85] and code normalization [21]. These techniques all share the common objective of transforming a program into a canonical form that is simpler in terms of structure or syntax, while preserving the original semantics. However, the effectiveness of these techniques is still limited to the original architecture and instruction set, rendering them inadequate for the task of detecting the same malware repurposed for different architectures.

4.3 Intermediate Representation

Intermediate Representation (IR) refers to a form of representation of source code internally generated by compilers during the process of compilation into executable code. The conversion op-

erated by compilers from source code to executable code can be seen as a translation of a program written in one language into another. In order to perform this translation, compilers rely on inner structures such as a front end - able to understand the source program and map it into the intermediate representation - and a back end - that reads the intermediate representation of code and translates it into the target machine language. While on one hand, the utilization of intermediate representation introduces an additional step in the translation process from source code to target code, on the other hand, it presents numerous advantages, including heightened abstraction, establishing a clearer demarcation between the front end and the back end, and introducing the potential for re-targeting and cross-compilation. Additionally, the generation of an intermediate representation of code enables new optimizations which are applicable on this form of code. An IR can take on various forms: semantic graphs such as an Abstract Syntax Tree (AST) and Control Flow Graph (CFG), tuples such as Static Single Assignment (SSA), stack codes and more. The choice of a particular IR form in a compiler depends on several factors, including the source programming language, the target architecture, and the transformations and optimizations applied by the compiler. The source programming language plays a crucial role in determining the appropriate IR form: different languages have specific syntactic and semantic features, and the IR needs to capture and represent those accurately. For example, an AST is often

used as an intermediate representation for languages with complex grammatical structures, as it maintains the hierarchical relationships between different elements of the code. The target architecture is a critical aspect, since compilers often generate machine code specific to a particular architecture. To optimize the generated code for the target architecture, the IR must be structured in a way that enables efficient instruction scheduling, register allocation, and other architecture-specific optimizations. Different types of Intermediate Representations (IR) exist, varying in their expressiveness, depending on the desired outcome. They can be categorized as:

- High-level IRs: These are highly readable and closely resemble high-level programming languages.
- Medium-Level IRs: Mid-level IRs offer a certain level of abstraction while maintaining independence from the underlying architecture.
- Low-level IRs: In this case, the intermediate representation aligns closely and is even dependent on the specific architecture for which the program is being compiled. However, it explicitly addresses memory management concerns, such as stack pointer management and parameter passing mechanisms.

IRs find utility not only within the scope of compilers for the purpose of optimizing the resultant target code, but also in the domain of binary analysis tools. The latter employ IRs as a means

to abstract and encapsulate any architectural aspect of a binary program that pertains to the specific architecture for which it has been compiled.

4.4 SigIL

SigIL (Signature scanning on Intermediate Language) is a novel framework for conducting signature scanning, with the objective of detecting malicious behaviors within compiled programs. While traditional signature scanning techniques target specific patterns within sequences of bytes, we shift our focus to the IR of the code. To achieve this, the framework that my co-authors and I proposed relies on the generation of a set of rules, exhibiting a syntax akin to that of YARA, yet referencing a specific intermediate language rather than byte sequences. By shifting our focus to the IR, our framework offers several distinct advantages. Firstly, it enables a higher level of abstraction, allowing for the detection of malicious behaviors independent of the underlying architecture's nuances. This enhances the versatility and adaptability of the scanning process, facilitating the detection of threats across multiple architectures without the need for architecture-specific rule sets. Secondly, the compactness of IR, due to features such as the simplified representation of code, allows for concise rule composition, eliminating unnecessary complexity and facilitating more streamlined and compact rule definitions. It is important to underline

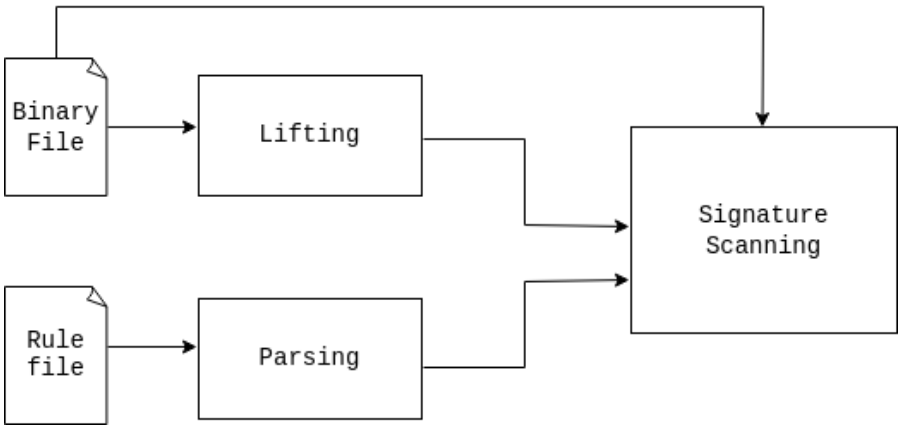


Fig. 4.1. SIGIL *structure*

that there is a wide range of software available for performing the translation from machine code to IR, commonly referred to as "lifting". Therefore, in contrast to the development of complex software for code normalization, code optimization, and other ad hoc approaches, it is unnecessary to implement brand new complicated software. Instead, we can leverage existing projects that are routinely utilized by compilers and binary analysis tools for the lifting phase. Moreover, our framework's approach mitigates the challenges posed by code obfuscation techniques: by analyzing the code's IR, our rules can capture patterns and behaviors that persist even when traditional byte sequences are obscured or modified. This resilience to obfuscation enhances the accuracy and reliability of the scanning process, ensuring a higher detection rate for malicious activities.

Our tool operates by taking two essential inputs: a target binary file and a rule file. The target binary file serves as the pri-

mary source of analysis, while the rule file provides the necessary guidelines for identifying specific patterns or signatures within the binary file. The architecture of our tool, as depicted in Figure 4.1, consists of three interconnected modules, each serving a crucial role in the analysis process. These modules are as follows: the binary file lifting module, the rule file parsing module, and the signature scanning module.

The lifting module performs the transformation of binary files into an IR. We chose to use LLVM IR [89] as intermediate language within our framework. We employ `McSema` [2] and `llvm-dis` [3] to obtain the IR from the binaries under analysis. LLVM IR is a low-level language that is employed by the LLVM framework during the compilation process. It provides a set of instructions that closely resemble machine code and can be optimized and compiled into native machine code for various target architectures. The motivations behind our preference are due to its inherent characteristics. LLVM IR operates at a higher level of abstraction as it represents a program's behavior and semantics independently of the source or target language. It provides a structured and language-agnostic representation. LLVM IR is structured in a three-address form. This form allows for more flexibility in expressing operations and dependencies between instructions. LLVM IR also has the advantage of representing types, such as 'I32' for

[2] https://github.com/lifting-bits/mcsema
[3] https://llvm.org/docs/CommandGuide/llvm-dis.html

a 32-bit integer, without being tied to specific registers. This abstraction from registers makes LLVM IR independent of the target platform, meaning it can be optimized and generated into machine code for different architectures. This feature allows us to abstract code from specific hardware platforms, making it an ideal fit for the vast and diverse architectures encountered in the realm of the IoT.

The parsing module consists of a software component that, based on a specific grammar, validates the correctness of the syntax of the language by performing lexical and syntactic analysis, adhering to the rules defined in the grammar. We imlemented this module by using the `Lark` parsing toolkit[4]. It conducts lexical analysis of the rule files text by using the Look-Ahead Left-to-Right (LALR) algorithm, ultimately resulting in the creation of an Abstract Syntax Tree (AST) to capture the hierarchical structure and semantics of the rules. Finally, the signature scanning module is responsible for comparing the IR of the code against the provided rules. By representing the rules as an AST, the tool gains several advantages in terms of matching code against the rule set. The AST serves as a structured blueprint, providing a clear understanding of the rule hierarchy and the corresponding conditions for matching. It enables the tool to navigate through the binary file's representation and identify potential matches based on the rules' defined patterns. The AST representation of the rules simplifies

[4] https://github.com/lark-parser/lark

the verification process, as it allows for precise rule evaluation. Each node within the AST corresponds to a specific rule or condition, and by traversing the tree, the tool can efficiently check if the code in the binary file adheres to the rules' requirements. The AST facilitates the systematic examination of code sections, making it easier to identify and validate matches.

Dealing with data and raw code.

Another crucial aspect of our tool is the ability to define signatures based on the data of a software rather than its code. As depicted in Figure 4.1, the signature scanning module takes both the lifted program and the program itself as inputs. This enables the module to search for the presence of a given signature not only in the lifted code but also in raw code sections (in case of unaligned malicious code) and the data sections of the binary. Consequently, we can define signatures that are associated with specific strings, IP addresses, or other conventional patterns commonly utilized in malware signature creation, even if they are unrelated to sequences of instructions.

Wildcard scanning support.

SigIL fully supports and integrates wildcard scanning, a valuable technique utilized in signature-based malware detection systems to improve the flexibility and effectiveness of signature matching algorithms. By incorporating wildcard characters or placeholders

into signature patterns, SigIL is able to identify malware variants that share similarities but are not exact matches. Wildcards allow for matching any value or a range of values at specific positions within the scanned data, thereby facilitating the detection of subtle modifications or obfuscation techniques employed by malware.

4.5 Example

In this section, we present a signature in the IR domain that enables the detection of the `socket` function invocation and verifies the connection to a specific IP address. This signature is designed to be architecture-agnostic and allows for the identification of a given behaviour across different architectures. This demonstration serves as a preliminary showcase, providing insight into SigIL's operation and capabilities. The signature, named `socket_and_ip_detection`, is defined in the SigIL rule syntax as shown in *Code* 4.1.

```
1 rule socket_and_ip_detection {
2     meta:
3         description = "Detects socket func
    and presence of a given IP addr"
4
5     condition:
6         exists(
7             call(
```

```
8                   return_type = "i32",
9                   name = "socket",
10                  arguments = [
11                      { type = "i32" },
12                      { type = "i32" },
13                      { type = "i32" }
14                  ]
15              )
16          ) and
17          exists(
18              data(
19                  name = "
    command_and_control_ip",
20                  type = "i8*",
21                  content = "111.222.333.444"
22              )
23          )
24  }
```

Listing 4.1. *Example of SigIL usage*

The signature consists of two conditions. The first condition uses the `exists` keyword with a `call` instruction to detect the invocation of the `socket` function. The `call` instruction uses `"i32"` as both the `return_type` and the `arguments` to match the `socket` function prototype. The second condition utilizes the `exists` key-

word with a `data` instruction to detect the presence of a specific IP address. By combining these conditions, the signature allows for the cross-architecture detection of connections to command and control servers. By abstracting away architectural specifics, when applied using the SigIL tool, this signature facilitates the identification of malware across different architectures, providing robust and consistent detection capabilities.

5

Firmware dynamic analysis

Conventional analysis methods are often not suitable for the IoT environment: the dynamic analysis of the firmware of these devices typically requires that code is not executed in the device's native execution environment, but in a controlled one. The reason behind this is manifold. First of all, the use of dynamic analysis on the native device may require expensive hardware which may not be readily available to the analyst and that may be prone to damage during testing. Also, it can be hard to supply inputs to guide analysis (as happens with fuzzing), and debugging. As explained in [134] using fuzzing, it is hard to detect memory corruption vulnerabilities on low-cost bare-metal devices lacking security mechanisms, because of the lack of visible effects. While, in theory, the possibility of using the device's ports for debugging exists, these ports are often obscured or inaccessible. Additionally, with physical hardware it is not feasible to perform concurrent executions, which is essential for dynamic analysis.

For this purpose, we rely on the emulation of the firmware, better known as *firmware re-hosting*. This process separates the firmware from the hardware and emulates it to run on a different architecture without the need for actual hardware. Firmware re-hosting offers several benefits, including the ability to execute in a controlled environment, use debuggers for greater insight, concentrate solely on software components, and benefit from scalability. However, that of firmware re-hosting is a challenging task due to the fact that the firmware frequently retrieves input directly from the device peripherals, which can have their own unique access definitions and different configurations and interfaces.

Several solutions have been proposed in literature for firmware re-hosting, based on approaches such as hardware-in-the-loop, low-level abstractions, learning, or symbolic execution. Despite the significant progress that has been made in the area of automated re-hosting and analysis of firmware, the current solutions come with drawbacks. While the hardware-in-the-loop approach has its advantages, it is often not feasible due to the difficulty or impossibility of obtaining real hardware, or the risk of damaging expensive components during large-scale automated analysis. Approaches based on abstractions usually incorporate binary instrumentation techniques to intercept calls to functions that interact with the hardware. Binary instrumentation adds a substantial overhead to an already slow emulation environment and severely impacts the performance of dynamic vulnerability discovery tech-

niques like fuzzing, which requires an high number of executions of binary code to take place.

This Chapter has different goals: we analyze the current solutions adopted in literature to enable dynamic analysis on the re-hosted firmware and their limitations. We offer our point of view on how progress can be made in this context in order to enable faster vulnerability discovery processes by using traditionally employed techniques such as fuzzing. In particular, we discuss our idea of replacing the device peripherals and the interactions between the firmware and hardware with high level operations, bringing all the firmware functioning at the software level. This Chapter directly addresses a crucial aspect within the broader domain of the device edge cloud continuum. By exploring the security challenges and vulnerabilities of IoT devices, specifically focusing on firmware re-hosting and dynamic analysis, our research aligns with the overarching theme of advancing paradigms, architectures, and applications in this interconnected landscape.

The rest of this Chapter is organized as follows: Section 5.1 provides some basic knowledge on firmware re-hosting and outlines a background on program analysis techniques, such as fuzzing, and different analysis levels. Section 5.2 gives a deeper explanation of the firmware re-hosting problem, along with an analysis of the challenges researches have to face when dealing with such problem. Section 5.3 explores current approaches employed to overcome the difficulties met when re-hosting firmware. Finally, Section 5.4 de-

scribes a novel approach to enable security assessment techniques such as fuzzing on re-hosted firmware that my co-authors and I presented in the article [46].

5.1 Background

Program analysis plays a crucial role when dealing with the security assessment of software systems and, over the years, there has been significant progress in the development of new techniques and methodologies to accomplish this task. Great effort has been put to make the process of program analysis scalable, leading to the development of dynamic analysis tools such as fuzzers and symbolic execution engines, that allow to analyze binary programs that, as often occurs, do not come with their source code. Popular examples in the security community of tools for fuzzing and symbolic execution are AFL [1] and angr [146]. However, program analysis tools, especially when performing dynamic analysis, need high levels of parallelism and scalability to function properly, which necessitates moving the execution into an emulated environment.

With the advent of IoT, the use of such tools, firstly meant to be applied on desktop and mobile systems, has been extended to various devices and their firmware. The necessity to shift the execution to a virtual environment in the case of firmware is supported by its dependency on the hardware, which limits the ability to observe its behavior. For example, in order to perform security testing of a

firmware by means of fuzzing, it would be often necessary to substitute the values derived from peripheral interactions with inputs generated by a fuzzer. It follows the execution must be taken out of its native environment and performed into an emulated and controlled environment, without involving physical components. The process of emulating a firmware in a way that accurately replicates its behavior on real hardware is referred to as *re-hosting*. Firmware re-hosting allows to thoroughly examine and manipulate firmware in manners that are not feasible on physical hardware and offers many benefits to analysts, including the ability to limit the scope of analysis to software components alone, while providing scalability. In this way, no physical embedded component of the device is necessary for the security analysis process to be held and the program execution can be attached to debuggers, making it possible to get a more in-depth understanding of program execution. Unfortunately, the task of firmware re-hosting is not without challenges, since while running the firmware interacts with peripherals. It follows that in order to gain a properly functioning emulation of the embedded system, firmware re-hosting implies modeling, along the CPU, the behavior of the device peripherals. Peripherals fall in two categories: *on-chip* peripherals include components such as timers, bus controllers, networking elements, and serial ports, while *off-chip* peripherals include sensors, actuators, external storage devices, and other circuit board circuitry that is accessed via on-chip peripherals. Specifically, on-chip pe-

ripherals, such as General Purpose Input/Output (GPIO) or bus interfaces like Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI), intermediate the firmware and off-chip peripherals communications and are typically controlled by the CPU through Memory Mapped Input/Output (MMIO), allowing programs to access them via memory. The absence of these components can result in the firmware crashing or producing outcomes that deviate from those generated when real hardware is employed and since a good many of system functions involve interactions with both on-chip and off-chip peripherals, realizing a proper emulation of them is vital.

5.1.1 Vulnerability discovery techniques

As we stated, there are some popular dynamic analysis techniques employed during the vulnerability discovery process of a system that benefit from its emulation. In this Section we briefly outline some of the most popular dynamic analysis methodologies.

Fuzzing

The goal of fuzzing is to identify inputs that cause the program to behave in unexpected ways or even crash, revealing the presence of vulnerabilities, bugs or other security issues in the programs that could be exploited by attackers. During fuzzing, a large number of values randomly o semi-randomly generated by a *fuzzer* are given as input to the target program, which then is monitored for

possible crashes, errors, or unexpected outputs. Popular fuzzers are AFL [1], libfuzzer [1] and Honggfuzz [2].

Concolic execution

combines symbolic execution and concrete execution to analyze the behavior of computer programs. It runs the program with concrete inputs while maintaining a symbolic state. It is also known as *dynamic symbolic execution* and it is different from static symbolic execution as it explores only one path at a time, determined by the concrete inputs. To explore different paths, the technique "flips" path constraints and uses a constraint solver to calculate concrete inputs that result in alternative branches.

Binary instrumentation

is used to gather information about the behavior and performances of an executable by adding code to the compiled binary. Instrumentation is meant to track and record data about program execution, such as function call information, memory accesses, and performance metrics, which can be used used for a variety of purposes, such as debugging, testing, profiling, and other security analysis. The instrumentation process is performed at the machine code level, making it platform-independent, and can be used to analyze a wide range of software, including low-level system code,

[1] https://llvm.org/docs/LibFuzzer.html
[2] https://github.com/google/honggfuzz

firmware, and high-level applications. However, binary instrumentation can also introduce overhead at the program execution, as the added code can slow down the program and consume more memory.

5.1.2 Analysis levels

The software security analysis of an embedded system can be performed at different levels: full-system, process level, or application level.

- With **full-system emulation** the firmware is meant to run inside a virtual environment recreated by an emulator, which is supposed to mimic any component of the original system, from the processor to the hardware peripherals. This approach is able to test the system in every aspect, allowing to generate the same data and behavior of the original target system, but it is the hardest to achieve, as well as slower when compared to other levels of emulation. Full-system emulation is possible by means of base emulators such as QEMU and Simics, although they only provide a small set of peripherals, which does not cover the wide and diverse range of possible hardware in embedded systems.
- The analysis at **process level** allows the emulation of specific processes behavior inside the target system. The execution of the processes can be performed inside the native system or a different hardware platform with an operating system provid-

ing an execution environment that resembles the native one. Emulators such as QEMU and Simics allow the process level analysis though the user mode emulation. Process level emulation is faster than the full-system emulation, however the results of the analysis may differ from the reality if the emulated execution environment is not faithful to the original, thus compromising the vulnerability discovery process.

- Analysis at the **application level** consists in analyzing the single application that can run in the native target system. This can be done both statically by extracting application-specific data or dynamically, by running the application itself. The limitation of the static approach is that reducing the analysis to the evaluation of statically extracted data can detect existing vulnerabilities in the specific application, but not within the system that interacts with that application. In the case of dynamic analysis, the execution is usually carried in the native execution environment. This analysis level is faster than the others, since it does not require the emulation of the system or process, however the emulation may not be accurate if the target application depends on native hardware features not supported by the execution environment.

A problem encountered when full system emulation is not involved is that the host platform used to run the firmware is not necessarily capable of supporting the use of dynamic analysis tools. This results from the fact that IoT systems are generally

lightweight and have reduced computing and storage resources, compared to traditional systems.

5.2 Motivation

Firmware re-hosting in IoT devices constitutes a significant challenge due to the wide heterogeneity involved in both hardware and software components, especially in comparison to desktop and mobile systems where the standardized execution environments and limited number of operating systems and architectures make the issue much easier to handle. The creation of a single generic emulator that is capable of hosting transparently a certain firmware turns out to be a highly impractical goal due to the remarkable diversity of existing embedded systems and architecture designs, as well as the proprietary nature of some chip designs. This diversity results from the combination of various hardware architectures (x86, ARM, MIPS, and so on), different types of embedded peripherals, multiple operating systems, and customized configurations and interfaces. The conjunction of these factors leads to a long list of realizable embedded systems, making it challenging to design a general emulator for firmware re-hosting.

These challenges have encouraged the development of a variety of emulation solutions. A widely adopted approach is the *hardware-in-the-loop* (HITL) [170, 72, 82, 152], in which the firmware is only partially emulated and whenever an unsupported I/O operation

is attempted, the request is redirected to the hardware itself. In HITL, the firmware interacts with a hardware platform that mimics the real hardware. The platform might be an actual piece of hardware or a hardware simulator, and provides the necessary peripheral interfaces and inputs/outputs that the firmware requires to function correctly. This lessens the need for access to the real target hardware and enables testing and evaluation of the firmware in a controlled environment. Various studies employ operating system and/or hardware abstractions [29, 37, 33] to take advantage of the abstraction layer provided by the firmware. An operating system abstraction is a layer of software that provides a standard interface for the firmware to interact with the hardware, while an hardware abstraction provides a similar layer of abstraction, but it is specific to the hardware being used. Such abstractions provide high-level representations of the underlying hardware, thus enabling the firmware to interact with the hardware in a manner that is independent of the actual hardware's implementation. Conversely, other studies aim for full-system emulation [55, 45, 25] without the presence of actual hardware. With full-system emulation the behavior of an entire embedded system, including the firmware and the underlying hardware, is recreated in a virtual environment. These works focus on the automated creation of models that describe the interactions between firmware and hardware, allowing for the replaying of these interactions without a direct connection to the specific device or even the learning of these in-

teractions through models generated from recorded real interactions. Once the software and hardware models have been created, they can be integrated into a virtual environment that simulates the behavior of the real system. The virtual environment can be used to test and evaluate the firmware in a manner that is independent of the underlying hardware. This makes it possible to test and evaluate firmware on different hardware platforms, or even on platforms that do not yet exist, without the need for access to the actual hardware.

Although much improvement has been made to handle the firmware re-hosting challenge, the currently approaches come with limitations. HITL-based solutions are effective in enabling interactivity and allowing testers to utilize dynamic analysis tools to input data into the firmware. However, this method introduces latency in the forwarding process, thus impeding the execution speed, reducing parallelism and scalability, and limiting its performance as a testing approach. Furthermore, HITL still entails a substantial tie between the firmware and hardware. Methods that rely on operating system or hardware abstractions exceed the HITL drawbacks, though these approaches have limitations in terms of the types of firmware they can handle. Indeed, in order to accommodate a broad spectrum of firmware, it is crucial for emulators to be devoid of high-level abstractions. Finally, learning-based solutions still require interactions with real hardware to collect data on the peripherals behavior. To overcome the limitations

of the currently adopted methodologies for re-hosting, we propose a novel approach that takes advantage of binary rewriting techniques. Binary rewriting is meant to modify the behaviour of a compiled program without having its source code or recompiling it, while keeping the binary executable. Binary rewriting can be classified as either static or dynamic. Static binary rewriting involves making modifications to the binary file and saving them permanently, while dynamic binary rewriting modifies the binary while it's being executed, without making any permanent change.

Several binary rewriting methodologies have been proposed in literature, including both static [14, 87, 126, 157, 155] and dynamic [139, 32, 58, 22, 94] solutions. Binary rewriting can be applied in a variety of ways, such as monitoring a program during execution, optimizing it, and emulating it [160].

Finally, many proposals build the emulation on top of base emulators such as QEMU [16] and Simics [98]. The exclusive use of these tools for hardware emulation is indeed discouraged since they are only able to emulate a restricted list of CPUs and peripherals and support a limited number of possible configurations [16], and may also require significant analyst intervention [98].

5.3 State-of-the-art approaches and their limitations

Firmware re-hosting has drawn a lot of interest in the literature, and various works came up introducing solutions to address the hardware emulation challenge with the ultimate goal of enabling faster security assessment of firmware.

HITL-based approaches

Several works such as Avatar [170], Avatar2 [109], PROSPECT [72], SURROGATES [82], Inception [36] and Charm [152] pursue the HITL approach, proposing partially emulation of firmware with unsupported I/O requests redirected to real peripherals, which still implies a strong dependence from the hardware. These works propose, in order to conduct dynamic analysis, to partially offload the execution of firmware to actual hardware, thus compromising scalability.

Abstractions-based approaches

Other works design emulation on top of OS abstractions [29, 37] or Hardware Abstraction Layers (HALs) [33]. Firmadyne [29] is a full system emulation tool for automated large-scale dynamic analysis of firmware. When a firmware image is provided to Firmadyne, the tool extracts the file system and performs an analysis to determine

the hardware specifics. Then, a pre-built Linux kernel that corresponds to these specifics is employed and an initial emulation is conducted using QEMU to infer the system and network configuration. Costin et al. [37] present a framework for scalable security testing of embedded web interfaces using dynamic analysis tools. The framework relies on the emulation of firmware images via QEMU, by replacing the system native kernel with a default kernel - for a specific CPU architecture - supported by QEMU. The limit of relying on OS-abstraction is that only a reduced number of firmware are supported by the framework, indeed only Linux-based firmware is handled by both [29, 37]. HALucinator [33] relies on a technique called High-Level Emulation (HLE) to perform dynamic analysis on firmware in embedded systems. The authors leverage the use of Hardware Abstraction Layers (HALs) commonly used by firmware developers to simplify their jobs, as a basis for re-hosting and analyzing firmware. The technique works by first identifying the library functions responsible for hardware interactions in a firmware image, and then providing high-level replacements for these functions in a full-system emulator such as QEMU. The authors demonstrated the practicality of HLE for security analysis by supplementing their prototype system, HALucinator, with a fuzzer to locate multiple previously-unknown vulnerabilities in firmware middleware libraries. In [91] a technique called para-rehosting to make re-hosting of microcontroller (MCU) software to commodity hardware smoother is proposed. The authors implemented a

portable MCU (PMCU) using the POSIX interface, which models common functions of the MCU cores and accurately replicates the common behaviors of real MCUs. They abstracted and modeled common functions of MCU cores and proposed HAL-based peripheral function replacement, in which high-level hardware functions are replaced with an equivalent back-end driver on the host, allowing for incremental plug-and-play library porting. Both [33] and [91] present interactive and hardware-independent environments. However, these environments are built over the assumption that the firmware relies on HALs, which may not always be the case. In order to be able to deal with a wider range of firmware, emulators should be abstraction-free, meaning that they should not depend on high-level constructs.

Learning-based approaches

Other approaches have been proposed to automatically create emulators for firmware, which require knowledge of how the firmware interacts with the peripherals, by capturing and reproducing data generated during I/O interactions to model the hardware behavior. This allows large-scale and interactive executions, but inevitably necessitates trace recording from within the device itself, thereby restricting accurate execution in the emulator to only the recorded program paths. In [55, 149, 45, 25, 68] the peripherals behavior is learned and the interactions between the firmware and hardware are modeled in order to enable virtualized execution of firmware

without implement peripheral emulators at all. Pretender [55] and Conware [149] gather a set of observations of the low-level interactions between the firmware and the original peripherals by means of HITL or code instrumentation. Pretender then utilizes machine learning to generate models of the memory-mapped input/output (MMIO) operations and interrupt-driven peripherals, which can replace the physical hardware. In contrast, Conware generates composable automata representations based on the collected recordings to model the peripherals, which can be merged to build generalized models. Both Pretender and Conware intend to emulate arbitrary firmware without having to instrument the actual firmware, however they imply an access to the physical hardware during the training phase in order to gather the observations needed for the emulation and require instrumentation to detect interactions with the hardware. Also, Pretender only provide models for interrupt-based firmware, that therefore are not generic and suitable for non interrupt-based firmware. P^{2}IM [45] is a framework for MCU firmware *approximate* emulation, that does not provide model for the physical peripherals themselves, but treats them as black boxes. Whenever the running firmware requests interactions with the peripherals, it is provided with acceptable inputs that simply satisfy internal checks and do not cause the execution to halt or crash. P^{2}IM does not require a deep knowledge about the peripherals behavior, allowing only a small set of values to be considered as possible inputs to the firmware. Although this

functioning allows hardware-independent emulation, it reduces the ability to represent effectively complex firmware logic.

Symbex-based approaches

Other works achieve firmware re-hosting by means of symbolic execution [25, 68, 172]. Laelaps [25] and Jetset [68] propose a symbolic execution-based approach to infer the behavior of the peripheral expected by the firmware. Laelaps is a concolic execution based firmware re-hosting framework that combines concrete and symbolic execution. It uses a full system emulator such as QEMU to run the firmware and gain the inner state of the execution and switches to symbolic execution whenever an access to an unimplemented peripheral is attempted, in order to find valid input that lead the execution to a path that resembles a realistic behavior. In order to prevent the path explosion, Laelaps relies on the Context Preserving Scanning Algorithm (CPSA) heuristics, which are able to infer inputs valid for for the near future execution, but which may cause execution crash in the long term. Jetset is a tool that relies on symbolic execution for inferring how the peripheral devices are expected to behave in the interaction with the firmware. This inferred are used while the firmware is running into an emulator such as QEMU in order to reproduce a device target functionality. Path explosion is mitigated using guided symbolic execution with a variation of Tabu Search to minimize the distance to the goal. However, following this approach, at each

branch the direction to take is chosen by looking at the distance to the goal, which can make it difficult to model more complex behaviors. µEmu [172] uses symbolic execution to extract valuable information and to build a knowledge base that is used to emulate the peripherals behavior during firmware re-hosting. As it carries the knowledge extraction process, µEmu tries to avoid the path explosion by switching to another path only when the current one is found invalid. However, it fails to emulate complex peripherals behaviors.

Approaches	**Articles**	**Strenghts**	**Limitations**
HITL	[170], [109], [72], [82], [36], [152]	– Interactivity; – Dynamic analysis enabled	– Hardware dependency; – Limited scalability; – Latency
Abstraction-based	[29], [37], [33]	– Hardware independence;	– Abstractions not always available
Learning-based	[55], [149], [45], [25], [68]	– Hardware independence after training;	– Hardware dependency during dataset recording;
Symbex-based	[25], [68], [172]	– Extensive path exploration; – Possibility to automate test case generation	– Potential path explosion; – Problematic complex behavior modeling

Table 5.1. Strenghts and limitations of existing approaches

5.4 Proposal

We propose an approach to enable dynamic security assessment techniques such as fuzzing on firmware. Our proposal rely on binary rewriting to obtain a full-system emulation of firmware, ensuring hardware independence as well as interactivity and overcoming the limitations experienced in the current approaches. By providing innovative insights and practical solutions to the security concerns surrounding IoT devices, this Chapter contributes directly to the advancement of knowledge and practices within the realm of the device edge cloud continuum. Our research underscores the significance of addressing vulnerabilities in IoT devices and highlights the potential for improved paradigms, architectures, and applications in ensuring the integrity and resilience of this interconnected ecosystem. Besides the drawbacks already discussed in Section 5.2, most of the current approaches involve the use of binary instrumentation to intercept the invocations to functions related to I/O interactions with peripherals and forward them to their replacement models. Since the instrumented code is meant to be executed in a virtual environment outside the firmware execution environment, its use is significantly slowing down the process.

In our proposal, we completely bypass that step by fully replacing the interactions between the firmware and the hardware with code. We integrate the embedded peripherals behavior at a high level by firmware rewriting, avoiding the involvement of lower

level abstractions such relying on OS assumptions or using HALs. In this way we enable the application of vulnerability assessment techniques based on a large number of executions and possibly crashes of the binaries under analysis.

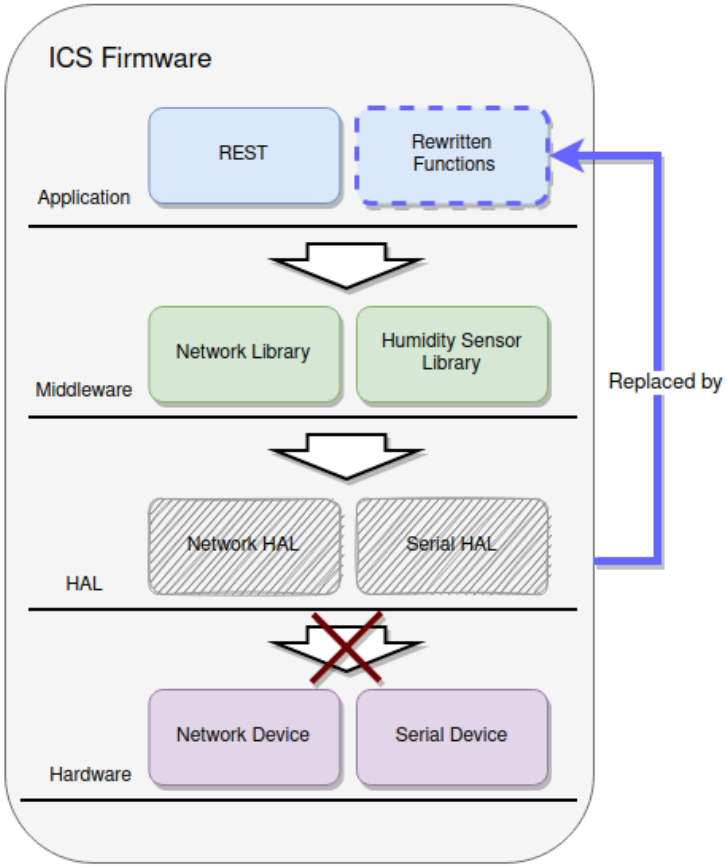


Fig. 5.1. Example: ICS rewritten firmware

The process involves the following steps: i) Portions of the binary code that constitute interactions with the peripherals must be identified. This step can be accomplished through various methods such as manual reverse engineering, debugging firmware in a hardware-in-the-loop environment, or locating HAL functions via library matching. ii)Once these functions are identified, their behavior must be rewritten to ensure successful emulation of the firmware, which is a challenging task that relies on manual development due to its difficulty to automate. However, literature suggests that several approaches can be used to automatically develop models to replace the original functions by recording firmware interactions with hardware peripherals, as discussed in Section 5.3. iii)To perform the actual replacement of functions that interact with hardware with ad-hoc implementations, we rely on binary rewriting techniques. This is the most significant aspect of our proposal as it avoids intercepting calls using binary instrumentation approaches, thereby significantly speeding up the emulation process. The binary rewriting step can be performed in several ways, some of which have already been introduced in Section 5.3.

The most significant benefit of rewriting, compared to current approaches, is that the firmware can be treated as normal software from an emulation perspective. As a result, the entire re-hosting process is much faster, and vulnerability assessment techniques such as fuzzing can be easily adopted without incurring excessive slowdown due to binary instrumentation.

In fact, what happens during the normal operation of a device is that numerous interactions occur with the hardware. By choosing to use binary instrumentation techniques to identify these interactions, the peripheral emulation process undergoes a significant slowdown.

For example, consider an Industrial Control System (ICS) that leverages REST APIs to expose the status of a humidity sensor. In particular, when a request is for humidity data is held through such APIs, the ICS calls a library function supplied by the manufacturer, that in turn invokes another function meant to interact with the peripherals, which is an HAL function provided by the microcontroller vendors, able to read the humidity value from the chip, by means of a serial communication. It is well known that the REST service relies on the HTTP protocol, which is implemented on top of a TCP communication, that in our example is put on in a library using an HAL to communicate with the Ethernet port. Even considering such a simple scenario, we can identify several interactions between firmware and hardware, e.g. function calls to the micro-controller HAL and communication with the Ethernet port. Our idea consists in replacing these functions with custom implementations, in order to be able to dissolve the firmware-hardware bonds and achieve complete independence from the underlying hardware, as illustrated in Figure 5.1. By rewriting the functions related to I/O interaction, we are able to

achieve firmware emulation without the need of instrumenting the emulation.

Part II

Attack Detection

6

Jamming detection

The rapid evolution of the computational and sensory capabilities of increasingly small devices has made the use of networks of mobile small nodes, like Unmanned Aerial Vehicles (UAVs) or *drones*, widespread and reasonably inexpensive. As flying devices without humans on board drone networks are ideal for supporting strategic missions in which they are used for surveillance and intelligence operations and, if armed, also for offensive missions. In recent years drone networks have been also used to other areas of interest, spanning from civilian to commercial domains, such as remote sensing, reconnaissance, search and rescue, commercial aerial surveillance, film-making, disaster relief, relay communication, etc. [57, 144, 12].

Given the increasing deployment of these networks, it is critical to make them as secure as possible. There are numerous threats that undermine the security of drone networks and, more generally, of wireless networks. Some of these threats can be addressed through appropriately designed network architectures and rout-

ing protocols [100, 76, 114, 12, 24]. Existing security mechanisms applied to drone networks are based on cryptography to ensure message privacy and node authentication. However, there are some attacks that, by their nature, cannot be addressed by conventional security mechanisms and therefore need further defense. Radio interference attacks, or jamming attacks, are a typical example of these threats.

Drone networks, often called FANETs (for Flying Ad hoc Networks), are particularly subject to jamming, as drone in the network communicate using wireless radio links, which are subject to interference. Jamming is considered the wireless version of a Denial-of-Service (DoS) attack [63]: the attacker may damage the operations of nodes within its radio range, preventing other devices from communicating correctly. Although it is a well-known problem in wireless networks, jamming is still an open issue and to date there is no effective solution to this threat.

6.0.1 Research context and motivations

Many solutions have been proposed for reacting to jamming attacks, such as frequency hopping [110, 167] and channel hopping [42]. Clearly, in order to react to a jamming attack, we must detect it first. Unfortunately, this is not an easy task since there is no single network parameter which, by itself, allows us to recognize a jammed condition. Also, jamming makes it impossible for the network nodes to communicate with each other. However, this

can also happen due to other network conditions, such as congestion or non-malicious interference [41]. Therefore, it is necessary to combine more than one metric to ascertain whether impaired communication is due to jamming or not.

Most studies carried out on jamming concern general wireless scenarios. Works focusing on jamming attacks on drone networks are lacking, arguably because detecting jamming in these networks is made even more challenging because of the nature of drones themselves, namely, small devices, with limited computational resources and power consumption constraints.

To tackle this challenge, on-device Artificial Intelligence (AI) can be applied, due to its communication efficiency. However, current solutions relying on AI techniques follow a centralized approach [74, 165, 18]. This kind of solutions is not appropriate for drone networks because of their highly mobile and decentralized nature, their highly mobile nodes, low response times and power constraints.

6.0.2 Objectives and contributions

In this Chapter a new framework that my co-authors and I designed for detecting jamming attacks in drone networks is presented [53]. Our proposal consists of a distributed supervised learning-based on-device jamming attack detection security framework that relies on an analysis of network parameters whose combination can be a symptom of jamming. Particularly, we iden-

tify throughput, Packet Delivery Rate (PDR) and Received Signal Strength Indicators (RSSI) as metrics that may noticeably vary when a jamming attack occurs, and whose combination can be therefore used to detect jamming.

We choose these features as attributes of the datasets used to induce our classification models, according to supervised machine learning approaches.

The framework operates following two main steps:

1. given a set of packet traces, it extract a series of default features;
2. once a value is assigned to each feature, it detects whether jamming is occurring or not.

We propose a distributed solution, in which each node is responsible to detect its own jamming condition without relying on any other network entity to discover if an attacker is jamming it. The detection is performed through two machine learning (ML) techniques which are *Multi-layer Perceptron* and *Decision Tree* whose operation is detailed in section 6.3.3.

The proposed framework is evaluated with datasets from publicly available standardized jamming attack scenarios by CRAWDAD [124], and analyze its applicability to drone networks also evaluating our framework with datasets generated through NS3 simulations. Then, an analysis of the obtained results, expressed in terms of accuracy values, is conducted proving the capability to

detect known attacks and more. In particular, a comparison of the two ML techniques in various network communication settings is provided, verifying that the Multi-layer Perceptron has a higher generalization capability overcoming the Decision Tree (96% Vs 50%) in larger and more complex communication scenarios for which had not been trained. Since the goal of the proposed framework is not only to provide a framework that is capable of jamming detection, but also to be used in the context of drone networks, an analysis of the required hardware resources is also conducted, bearing in mind the limited resources with which drones are generally equipped.

The rest of this Chapter is structured as follows. Section 6.1 provides background concepts about drone networks and jamming attacks. Section 6.2 introduces the main techniques known in literature to counteract jamming attacks. In Section 6.3 a proposal for jamming detection is presented, highlighting the characteristics of the metrics on which it is based. In order to make the make the discussion clear and precise, Section 6.3.3 provides some introductory concepts on the machine learning techniques employed in the proposal. Results from the performance evaluation of our framework is reported in Section 6.4, comparing the performance obtained by using different communication technologies and networks.

6.1 Background

This section presents the issues related to the emerging drone networks technologies also presenting few schemes for jamming classification, possible strategies for jamming detection and the main jamming indicators useful to detect jamming attacks.

6.1.1 Drone networks: Single-UAV and multi-UAV systems comparison

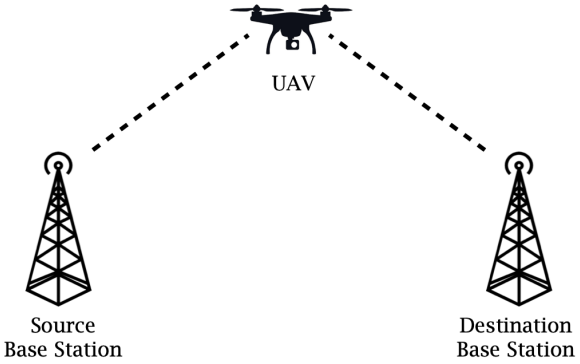
A drone network is a wireless system whose nodes can be UAVs (Unmanned Aerial Vehicles), commonly known as *flying drones*, consisting of unmanned flying devices, usually controlled by a computer embedded on board, called Flight Computer System (FCS), via a pre-programmed software. Its early usage consisted of single-UAV systems, which are networks with just one drone, executing one or many tasks [7].

In single-UAV applications it is very simple to set up a communication environment, since all the ground nodes are linked to the aerial node in a star topology network strategy and therefore they can easily communicate with each other indirectly [133]. However, these systems pose challenges in the peer-to-peer communication such as sending more data or expanding the communication range, since if a drone flies outside the coverage of the ground base, it loses connection. Furthermore, in order to carry powerful processors, sensors and communication hardware, the drone of a single-UAV

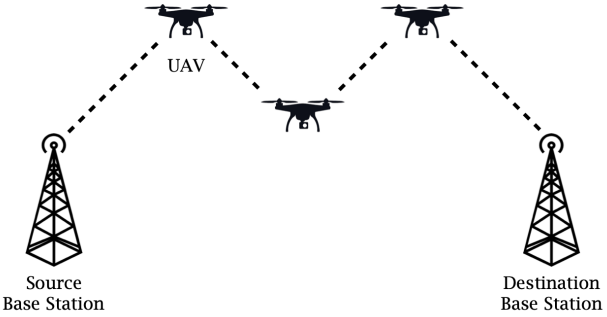
system is generally huge and this can be dangerous and expensive in case of failure.

Single-UAV systems have been in use for a long time, until the spread of a trend towards miniaturization, from the 1990s, has led to the development of small tools and equipment. Since small drones have lower acquisition and maintenance costs, this phenomenon has made them affordable to academics and even hobbyists, allowing their use in other sectors besides the military one [28]. With a smaller and lighter architecture, drones are hard to detect and do not represent a real threat to human life or the environment in which they move. Moreover, they can be transported by hand without any support and can be reused in different applications. However, the capability of a mini-UAV is restricted by many aspects, such as power, sensing and computation, and therefore these devices are designed to be integrated into multi-UAV systems. Within a multi-UAV system, drones must coordinate and collaborate, thus leading to the creation of a system whose capabilities exceed those of a single drone. Indeed, drone systems consisting of multiple small devices can offer several advantages over the use of just one large, in terms of cost, scalability, survivability, fault tolerance and other aspects [15].

Multi-UAV systems allow to expand the area of interest of a mission as well as to achieve goals faster, parallelizing individual tasks. Moreover, the failure of one drone does not imply the failure of the entire task, which can be carried out by the other UAVs.



(a) Single-UAV system



(b) Multi-UAV system

Fig. 6.1. UAV systems topologies

Ultimately, the many advantages offered by multi-UAV systems compared to single-UAV systems have the cost of greater communication and coordination complexity.

The topologies of single-UAV and multi-UAV networks are shown in Figure 6.1, and the main differences between these systems are summarized in Table 6.1.

Table 6.1. Single-UAV, Multi-UAV systems comparison

	Single-UAV	Multi-UAV
Size	Large and heavy	Small and light
Cost	High	Low
Portability	Needs support	Hand-carried
Dangerous	High	Low
Survivability	Single drone life	Fault tolerance
Scalability	Low	High
Multitasks capability	Low	High
Goal achievement time	High	Low
Complexity	Low	High

6.1.2 Jamming attacks

Jamming attacks are the equivalent of Internet Denial-of-Service (DoS) attacks for radio signals. In particular, jamming is the act of disturbing radio (wireless) communications by causing their Signal-to-Interference-plus-Noise-Ratio (SINR) to decrease, typically transmitting on the same frequency and with the same modulation as the signal to be disturbed.

To understand how jamming attacks work, it is important to distinguish them from interference. Interferences are unintentional forms of disruption, usually due to device malfunctions, accidentally caused; on the contrary, jamming attacks are conducted by an attacker with the malicious intention to interrupt or prevent

legitimate communications in the network. A pictorial representation of jamming is shown in Figure 6.2.

Radio users also distinguish between *intentional* and *unintentional* jamming. We talk about unintentional jamming when a node transmits on a busy frequency without first checking whether it is in use, or when it is not able to hear stations using that frequency.

In general, attackers jam the ongoing transmissions via injecting malicious signal to the wireless channel in use.

The entity - either a malicious or unintentional wireless device - that launches such radio interference is referred to as *jammer* or *interferer*.

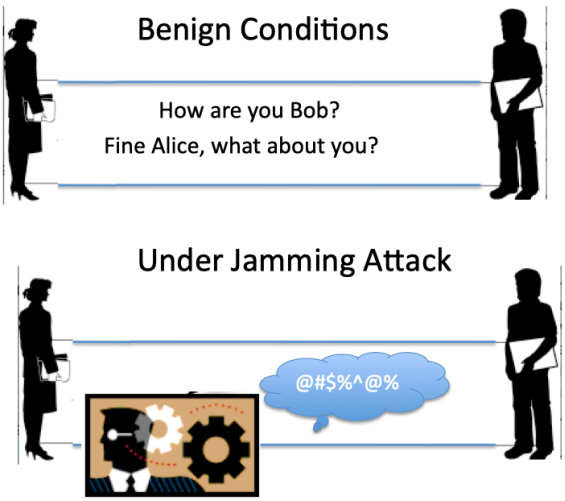


Fig. 6.2. Pictorial representation of a jamming entity [118].

In order to avoid the occurrence of unintentional jamming phenomena, CSMA (Carrier Sense Multiple Access) is often used. It is a media access control (MAC) protocol according to which each node in the network, before transmitting on a shared medium, verifies the absence of other traffic. This protocol by itself actually is not enough to avoid collisions, and other mechanisms are usually adopted. Some types of jammers, as explained below, transmit without respecting the CSMA. We can define a jammer as a device that does not comply to legitimate PHY or MAC protocols and intentionally tries to hinder communications by interfering with transmitter or receiver.

The effect of a jammer depends on several aspects, such as its radio transmitter power, location and influence on the network or the targeted node.

There are many ways in which a jammer may jam a wireless network, among which it chooses the most effective one, and there are also many different ways to classify them. Some divide jamming techniques in two main categories, which are *noise techniques* and *repeater techniques*.

A detailed classification of jamming attacks, summarized in Figure 6.3, among with a description of a jammer possible operating modes is proposed in [166].

The analysis was conducted by focusing on reactive jamming, which is more sophisticated compared to constant jamming. Reactive jammers are considered the most critical adversarial threat to

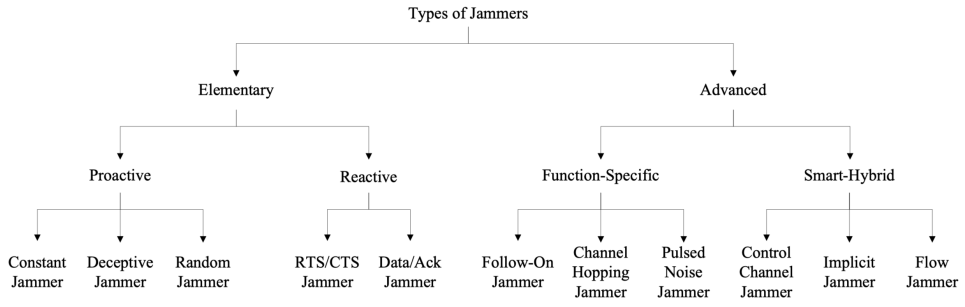


Fig. 6.3. Types of Jammers in Wireless Networks [54]

compromise networks since they operate by injecting interfering signals only when target devices are transmitting. In particular, during reactive jamming, interference is only generated when a transmission is taking place over the network, staying quiet when the channel is idle, in contrast to constant jamming, which consists in a nonstop interference. This makes reactive jamming not only cost-effective for the jammer, but also more challenging to detect, track and compensate.

In any case, the goal of a jammer is to interfere with legitimate wireless communications, and it can achieve this by acting both in transmission and in reception, that is by preventing the source node from sending packets or the destination node from receive legitimate packets.

6.2 Related works

Several techniques are known in the literature for detecting and reacting to jamming attacks, as well as some metrics whose value

can be symptomatic of the presence of a jammer in action. Clearly, these techniques vary according to the type of jammer considered and its characteristics. Also, as seen before, not all jamming attacks are equally difficult to detect. For example, since reactive jammers are triggered only when packet transmission occurs over the network, they may be not so easily detected.

Once a jammer and the area in which it acts have been recognized, the other nodes in the network can counter the attack by redirecting traffic, changing channels or moving to a safe location. Some well known techniques are *channel hopping* and *frequency hopping*.

The channel hopping is the most popular countermeasure to jamming and works particularly well in the case of proactive jamming. It consists in continuously changing the communication channel after a certain period of time.

The frequency hopping, instead, selects the frequencies considered "good" (not jammed) and, if anomalous interferences are detected on those used, it changes the transmission frequencies.

In addition, many techniques can be combined, such as the direct-sequence spread spectrum (DSSS) and the frequency-hopping spread spectrum (FHSS) [108]. The DSSS uses a modulation that makes the transmitted signal wider in bandwidth than the information bandwidth [162]. In this way, the attacker detects data signals as white noise and it is not able to identify the communication radio band. The FHSS is used to avoid interference, instead.

Game theory has also often been used to model jamming attack scenarios. More specifically, the interaction between a network node and a jammer can be formulated as a Stackelberg game, in which the latter, acting as a *follower*, determines its transmission power based on the action observed by the former, which instead acts as a *leader* [64, 164].

Almost all the studies carried out on jamming concern general wireless scenarios, and the literature lacks work that focuses on this attack in the specific context of drone networks, which is even more challenging because of its characteristics.

However, it is worth considering that drone networks are usually made up of small devices, with limited computational resources and power consumption constraints.

Recently, on-device Artificial Intelligence (AI) has gained significant notice thanks to its communication efficiency [101]. This is an unexplored territory: very few works apply Machine Learning for the detection of jamming attacks. In [74] an unsupervised machine learning approach is proposed through the use of clustering to detect jamming in the radio frequency communication between two vehicles. A framework based on supervised machine learning techniques for detecting jamming in optical networks is presented in [18]. In [165] the use of Reinforcement Learning against jamming attacks is evaluated in a VANET, including relying UAVs.

The success gained by *Reinforcement Learning* (RL) in the decision-making problems has led to the application of Q-learning

(QL) to build anti-jamming wireless communications. According to this approach, the jammer, base on the environment, chooses its jamming action in order to maximize the reward [48]. This is long-term cumulative reward, which is determined through a series of time events. The Q-learning is a RL method, which can learn the optimal strategy based on the long-term cumulative reward with an end-to-end approach.

Although all these solutions take advantage of machine learning methods against jamming attacks, they rely on a centralized approach, which is not suitable for UAV networks.

A centralized solution is impractical for several reasons: drones have high mobility and may not always be in the communication range of the ground station, and also they usually require an immediate response. In addition, continuous communications to a centralized system is not effective, since it can cause severe power consumption [133].

6.3 Proposal

Detecting jamming attacks is a critical step towards building a secure and reliable wireless network. This is not a simple task, since a jammer can operate in several different ways; moreover, we have to distinguish jamming attacks scenarios from other legitimate network situations, such as congestion.

Jamming attacks detection can be implemented on dedicated devices or in customized programs running on the communication devices themselves. Generally, the latter case is preferred, since it involves lower costs.

In this section, I present my proposal for jamming attacks detection to be executed on each node of the network, whose objective is to indicate, starting from raw packets, the presence or absence of jamming.

In particular, I examine the metrics that can be symptoms of a jamming attack. I also present two supervised machine learning techniques, and explain how they can be used within a specific framework, combined to the chosen measures, for the detection of jamming attacks in wireless networks. In particular I used *i)* a Multi-Layer Perceptron [113] *ii)* and a Decision Tree [132].

The considered scenarios are controlled, which means that the network nodes have information on both sides of the communication in which they participate, transmitter and receiver.

The framework was developed using Python 3.7.2 and its code relies on the SciKit-Learn library to implement Machine Learning techniques [116].

Since there is no publicly available dataset relating to jamming attacks in drone networks, which was the problem of interest, my co-authors and I used the traceset collected in a VANET communications scenario [124], made available to the scientific community

through CRAWDAD [83], an archive that stores wireless trace data, as already done in [106].

This decision is justified by the following reasons: *i)* FANETs (Flying Ad hoc NETworks) [133], [77] are a subset of VANETs and the two networks therefore share common characteristics; *ii)* communication takes place using the 802.11p protocol, which, although meant for VANET, is perfectly suited to FANET, as explained in [156]; *iii)* according to the Federal Aviation Administration, drones usually maintain the same altitude during their flight, so the third dimension of the UAVs in a FANET can be considered fixed.

6.3.1 Jamming indicators analysis

During a jamming attack, the value of some network parameters is generally subject to alterations and therefore they can be used as jamming indicators. However, there are many papers showing that there is no metric which is itself enough to conclude that a jamming attack is in progress [168]. According to these observations, we decided to analyze the combined effect of three metrics, which are Throughput, PDR and RSSI.

Throughput It is a measure used to quantify the actually used transmission capacity of a telecommunication channel. Its value is lower than that of the theoretical capacity, which expresses the maximum transmission frequency at which data can travel. Throughput is the amount of data transmitted in a unit of time;

therefore, it is usually expressed in bits per second (bit/s or bps), and sometimes in data packets per second (p/s or pps).

Packet Delivery Ratio (PDR) It is defined as the ratio of the number of packets successfully received by a destination node, compared to the number of packets sent by a source node. If a node does not receive any packet at all, the PDR is 0, however, in order to estimate the PDR, an exchange of messages must take place between the nodes of the network and, since a jamming attack can be interrupted at any time, this exchange of messages must be very frequent.

Received Signal Strength Indicator (RSSI) It is a measure of the relative power present in a received radio signal in a wireless network; thus, the greater the RSSI value, the stronger the signal. Generally, the RSSI is not provided from the receiving device to the user; however, IEEE 802.11-based devices make their RSSI values available to the user.

The goal of a jamming attack is to downgrade the quality of the channel surrounding a node, and therefore essentially reduce the ability of the node to send or receive packets. These three measures can undergo variations in both cases in which a jammer attacks the sender, which may not be able to transmit, and when it attacks the destination, which may not be able to receive packets.

PDR is a central measure for detecting jamming attacks: it is effective enough in distinguishing jamming scenarios, in which it

may assume very close to 0 values, from congestion scenarios, in which it usually assumes lower values, but not so close to 0, as explained in [168]. However, it is not nearly as effective at distinguishing other network dynamics that can downgrade the PDR value just like a jamming attack would. This occurs for example when the sender moves out of the communication range of the receiver, or when its battery is discharged.

To address this question we rely on the results obtained by *Xu et al.* [168], which show that an enhanced jammer detection, able to differentiate between these two scenarios, can be achieved by combining the values of PDR and Signal Strenght (SS). In particular, they show that, in a normal interference-free scenario, a high signal strength implies a high PDR and a low signal strength implies a low PDR. On the other hand, if the PDR is low, the SS is not necessarily low.

Two cases in which a node of the network X may have low PDR values are the following: its neighboring nodes are down or the node X is jammed. In the first case the SS values are also low, in the second they are high. This signal strength consistency-based consideration allows us to distinguish the two cases, and then reduce the number of false positive.

This is what happens when the Reactive jammer is active analyzed in [123]: a period of no reception, which means PDR=0, starts and ends without any corresponding decrease or increase of the RSSI.

In conclusion, there is no single network parameter that, by itself, is enough to determine that a jamming attack is in progress; therefore, it is preferred to combine multiple metrics rather than develop detection techniques that focus exclusively on the values of one of them [168, 106].

6.3.2 Feature Extraction

Since the dataset contains raw packets, a feature extraction is necessary in order to proceed with the analysis. By feature extraction, we mean the process of creating a new set of features out of the original one, which came with the raw data. This is done through some algorithm or transformation dependent on the nature of the data.

First of all, the CRAWDAD dataset was analyzed and processed in order to generate the dataset to be used during the classifier training and setting phases.

The data relating to each communication in the CRAWDAD traceset is stored in two files, respectively containing packets sent by the transmitter node and packets received by the receiver node. Since these two files are not synchronized, the packets stored in one of them do not necessarily coincide with all and only the packets stored in the other. Therefore, a first step of data processing consists in identifying the packets, on the transmitter and receiver side, belonging to the same time interval.

Each sender side tuple represents a packet which, among other things, contains the sending timestamp, the protocol adopted (C2X for data packets and MGM for signaling packets) and the packet size (194 bytes for data packets, shorter lengths for signaling packets). Each tuple on the receiver side includes the same information, as well as the RSSI. A second step therefore consists in filtering, on both sides, the data packets.

At this point, the remaining packets are divided among smaller and fixed-size time windows, in order to aggregate the data of the packets linked to each window in a single instance.

The basic idea is to compute, starting from the data associated with a window, the minimum, maximum, average and standard deviation values of Throughput, PDR and RSSI.

The instances extraction was done through a Python script whose steps are summarized by Algorithm 6.3.2.

The resulting dataset is made up of 12 features, 4 for each metric, plus the class label. Regarding the size of the windows, we set it to values equal to 2, 3, 4, 5 seconds. For each of these values, two datasets were produced: in one, the windows follow one another without overlapping and each packet falls into only one of them; in the other, there is a 50% overlap between each window and the next. In conclusion, eight datasets were generated, overall.

Table 6.2 shows for each of the generated datasets the number of instances and how many of them are labeled as jamming attacks.

Algorithm 1 Instance Extraction

Input: Sent packets *s_pck*, received packets *r_pck*, window size *win_size*

Output: A labeled instance

Identify packets from *s_pck* and *r_pck* falling within the same time interval

Filter data packets

Split packets into *win_size* sized windows

for *each window* **do**

compute *min, max, mean, dev_std* Throughput

compute *min, max, mean, dev_std* PDR

compute *min, max, mean, dev_std* RSSI

associate label

end

Table 6.2. Class distribution

Datasets			
Window size	**Instances**	**Normal**	**Jamming**
2 sec	1085	310	775
2 sec with overlap	2167	620	1547
3 sec	715	204	511
3 sec with overlap	1417	403	1014
4 sec	530	151	379
4 sec with overlap	1052	299	753
5 sec	419	118	301
5 sec with overlap	826	232	594

As we can see, the datasets are not balanced with respect to the class label: the percentage of instances belonging to the Jamming

class is approximately 72%, while the Normal instances constitute the remaining 28%, in each dataset.

6.3.3 Training and testing

The datasets obtained as explained in the previous section were used to train and evaluate the classifiers.

As classifiers we analyzed the behavior of Multi-Layer Perceptrons and Decision Trees, adopting the k-fold cross validation technique, with $k = 5$ to split the dataset into training set and test set. The training set and test set generated at each step of this process are balanced with respect to the distribution of labels in the starting dataset.

An Artificial Neural Network (ANN) [84] is a computational model, inspired by the human brain's neural network. It is made up of layers of interconnected nodes, called *neurons*, which are processing units, and adapts its own structure based on external or internal information, that flows through the network during the training phase. An ANN can have a variable number of layers, but it always has at least two: the *input layer* and the *output layer*. The neurons that follow the input layer and precede the output layer are organized in *hidden layers*. The network receives signals through its input layer, the nodes of which are connected with the internal nodes, arranged on the various layers. Each node processes the received signals and transmits the result to subsequent nodes. The last layer of neurons constitutes the output layer. In

the construction of a neural network, neurons are assigned an *activation function*, which defines the output of that node given an input. A feed-forward neural network, commonly called a *feed-forward network*, is an ANN where connections between neurons do not form cycles, unlike recurrent neural networks. The simplest feed-forward network is the Single-Layer Perceptron (SLP). A SLP consists only of an input layer and an output layer, without any hidden layer. Each input neuron is connected to each output neuron. In other words, this type of neural network has a single layer that performs data processing, hence the name. A feed-forward network having at least one hidden layer is called a Multi-Layer Perceptron (MLP) [49]. Each layer has connections coming from the previous layer and going to the next, therefore the propagation of the signal occurs forward without cycles and without transverse connections, as shown in Figure 6.4.

In this case, a MLP is built with one single hidden layer, consisting of three neurons. The network uses the rectified linear unit function as activation function, which returns

$$f(x) = max(0, x).$$

A Decision Tree (DT) consists of a classification model, in which each path from the root node to a leaf node represents a conjunction of conditions that lead to the classification of an instance. Each internal node represents a set of instances and defines a

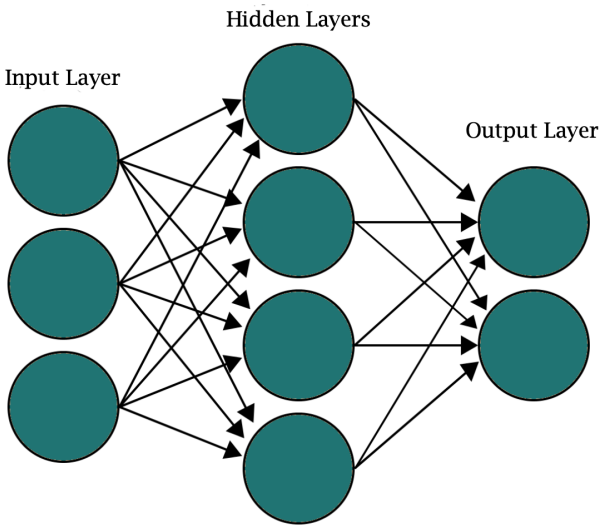


Fig. 6.4. Multi-Layer Perceptron

split, that is a condition on a feature of the dataset, on the basis of which the data is divided. Its branches represent possible outcomes. The root node represents the set of all instances, while the leaf nodes make up the class labels. The structure of a decision three is shown in Figure 6.5. A decision tree is built using learning techniques starting from the initial data set, which is divided into the two subsets we know: the training set, on the basis of which the tree structure is created, and the test set, which is used to test the accuracy of the predictive model thus created.

In order to define the split of each node, it is necessary to identify, from time to time, the relevance and importance of each of the attributes, thus place the most important as the basis of the split test. Here the Gini index was chosen as test split criteria. The

Gini index or Gini impurity is a statistical measure of distribution, which measures the degree or probability of a particular variable being wrongly classified when it is randomly chosen.

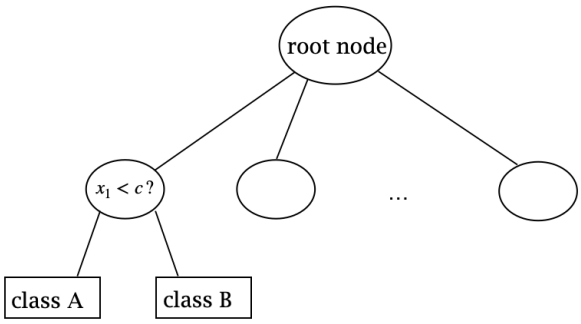


Fig. 6.5. Decision Tree

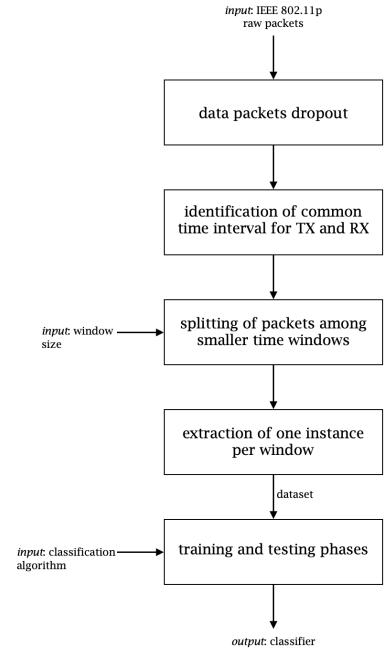
Formally, it is defined as:

$$Gini = 1 - \sum_{i=1}^{n} p_i^2$$

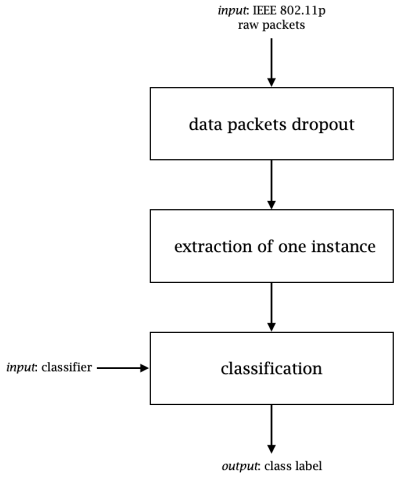
where p_i is the probability of an object being classified to a particular class.

While building the decision tree, it is preferable to choose the feature with the least Gini index as the root node.

The process of extracting the dataset and building the classifier is schematized in Figure 6.6.a.



(a) Online operation



(b) Offline operation

Fig. 6.6. Operation modes of the framework.

6.3.4 Jamming detection

Once the classifiers have been trained and tested offline, as explained in the previous sections, they can be used in real-time. The basic assumption is that the necessary information, which is the trace data on both the transmitter and receiver sides, is available on the node. The task is designed to collect traces for a period of time equal to the chosen window size. The processing of this data trace, which consists of the steps discussed above, leads to one unlabeled instance, which is given as input to the classifier selected among those available on the node.

The online operation of the framework is schematized in Figure 6.6.b.

6.4 Performance Evaluation

In this section an analysis of the proposed framework and its applicability in a drone network scenario is carried out, testing the various configurations and evaluating different indicators.

In particular, the quality of the proposal was tested on different tracesets coming out from both real and simulated communication scenarios, with the aim of investigating a wider range of cases.

6.4.1 Basic concepts for model evaluation

In order to evaluate the model, some basic concepts are introduced. Given a binary classification problem having as possible outcomes Positive and Negative labels, we denote by:

- True Positive (TP) $\rightarrow$ an outcome *correctly* predicted by the classifier as belonging to the Positive class;
- True Negative (TN) $\rightarrow$ an outcome *correctly* predicted by the classifier as belonging to the Negative class;
- False Positive (FP) $\rightarrow$ an outcome *incorrectly* predicted by the classifier as belonging to the Positive class;
- False Negative (FN) $\rightarrow$ an outcome *incorrectly* predicted by the classifier as belonging to the Negative class;

These definitions above can be also generalized to non-binary classification problems. At this point, we denote by:

- **Accuracy** - the fraction of correctly classified instances formally defined as $Accuracy = (TP+TN)/(TP+TN+FP+FN)$
- **Precision** - the fraction of correctly Positive classified instances over all the Positive classified instances formally defined as $Precision = TP/(TP + FP)$
- **Recall** - the fraction of correctly Positive classified instances over all the correctly classified instances formally defined as $Recall = TP/(TP + FN)$

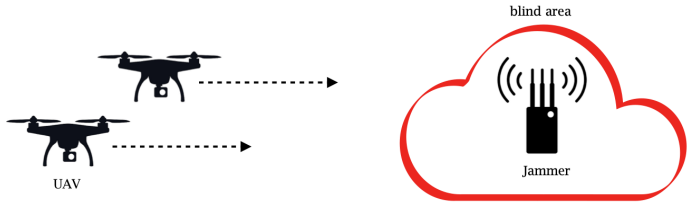
- **F-measure** - the harmonic mean of precision and recall. It is an accuracy metrics and, formally, it is defined as $F = 2 \cdot (Precision \cdot Recall)/(Precision + Recall)$

To evaluate the trained classifier, we compare the labels predicted by it on the test set with the real ones, assigned by the domain expert. In this way we obtain the TP, TN, FP and FN values to be used to calculate the metrics. In the case of *k-fold* cross validation these measures are calculated as the average of their values obtained at each execution.

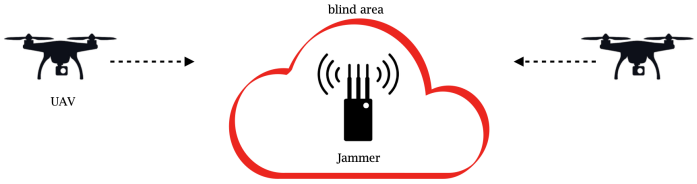
6.4.2 Attacker Model

We assume that, according to the configuration of the experiments set in the collection of data traces, the attacker is a reactive jammer. A reactive jammer aims to compromise the reception of a message. Hence, it starts jamming only when it observes a network activity to occur on a certain channel [166]. Therefore, in order to be active, it must first detect a transmitter and to be able to impair packet delivery, it must also generate sufficient interference power at the receiver. This attack is less energy efficient than other jamming operating modes because it has to constantly monitor the network to determine when it has to transmit, but it is also much more difficult to detect.

The jammer generates a blind area, whose extension and intensity may vary, and within which the communication between the nodes is affected. Therefore, whether the nodes move in the same



(a) Nodes moving in the same direction



(b) Nodes moving towards each other

Fig. 6.7. Attacker Model

direction or towards each other (see Fig. 6.7), when they reach the blind area, they may not be able to communicate so easily.

6.4.3 Working with real datasets

The proposed framework has been tested using 8 different datasets, which have been generated from tracesets [124], as a result of a feature extraction process. For sake of completeness, we remark that the reference tracesets consists of IEEE 802.11p raw packets exchanged among nodes of a VANET, with and without the presence of a jammer. The jammer can acts in two operation modes, constant and reactive, and the traces were collected in both indoor and outdoor scenarios.

The analysis is conducted by highlighting the values of *Accuracy*, *Precision*, *Recall* and *F-measure* gained by several classifiers operating in different network configurations.

In detail, we compare classifiers by varying: *i)* the classification algorithm, which can be a MLP or a Decision Tree; *ii)* the dataset used to fit the classifiers, which depends on the size of the time window, set at 2, 3, 4, 5 sec, and the presence of overlapping windows.

To evaluate the system performances with the aim of obtaining meaningful statistical results [90], we made 10 replications, repeating the test 10 times and averaging the obtained results; moreover, we set the confidence interval level to 0.95 and we excluded the first seconds of the reference tracesets from the statistical error computation in order to verify the correctness of the statistical analysis for the obtained results also reducing the system's transient effects.

In the following we analyze separately the quality of neural networks and decision trees, and then finally compare the two models.

MLP evaluation

By looking at the results we make two main points. First, by comparing each result obtained for the datasets with and without overlapping for a fixed window size, we observe that models induced from the former achieve higher accuracy values. It may be

due to the fact that datasets with overlapping contain a greater number of instances. Also, we can observe that the performance of the MLP classifier decreases as the size of the time window increases. Fixed the number of nodes, we notice that datasets having $window\ size = 2s$ always perform better that those having $window\ size = 5s$, in both with and without overlapping cases. These results are shown in Figure 6.8.a.

MLP classifiers reach precision values between about 70% and 75%, as shown in Figure 6.8.b. This is a symptom of the presence of False Positives in the classification. This is mostly due to the imbalance of the datasets, which present a majority of instances belonging to the Jamming class.

The recall values follow the same accuracy trend, as shown in Figure 6.8.c where the classifiers trained on datasets with overlapped windows perform better than their respective non-overlapped ones and the performance worsens as the number of nodes in the network increases. The best case is obtained by setting $window\ size = 2s$ with overlapping.

Usually, we refer to precision and recall separately when in the problem addressed, reducing the number of False Positives is more important than reducing that of False Negatives, or vice versa.

In our case, however, minimizing the number of False Negatives is important as reducing that of False Positives, since countermeasures must be taken in response to the presence of jamming. Therefore, we analyze the F-measure, which combines the two measures,

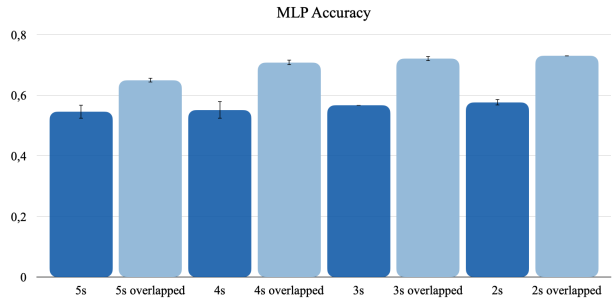
as defined in 6.4.1. Also regarding the F-measure, we observe the same trend: the performance of the classifiers worsens with the increasing of the size of the time window. As seen before, datasets with overlapping lead to a better behavior of the model.

In conclusion, by comparing all the accuracy values obtained for all the possible window sizes, we can assert that accuracy drops as the window size increase.

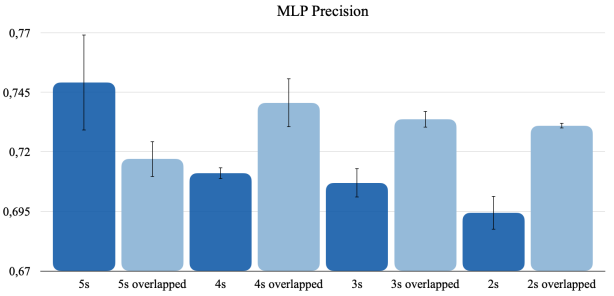
Decision Tree evaluation

As in the previous case, we start by comparing the two cases of maximum and minimum size of the time window.

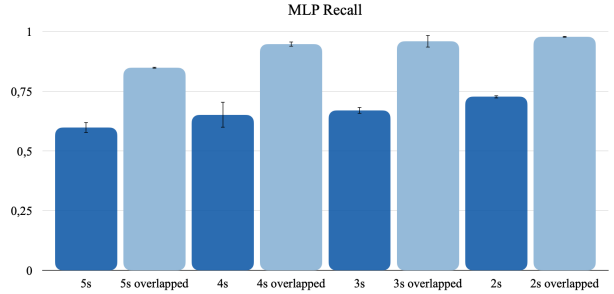
The accuracy assessment on DTs is very similar to that made for MLPs: it decreases as the size of the time window increase, and the datasets with overlapping perform better than the others. However, unlike what happens for MLP classifiers, the differences between the performances reached by the various decision trees are slighter, as shown in Figure 6.9.a. The same goes for the Precision and Recall values, and therefore for the F-measure, which reaches 95.36% for the $dataset = 2sec$ *with overlapping* case, as shown in Figure 6.9.c and 6.9.d respectively.



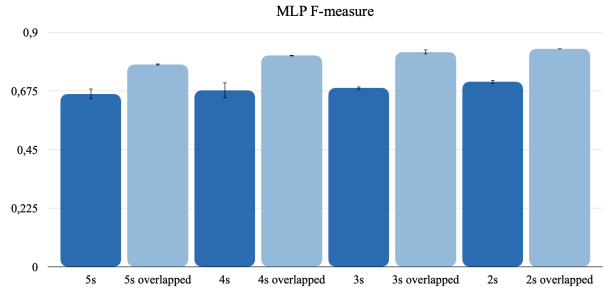
(a)



(b)

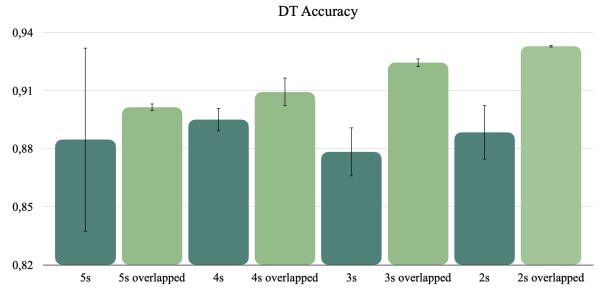


(c)

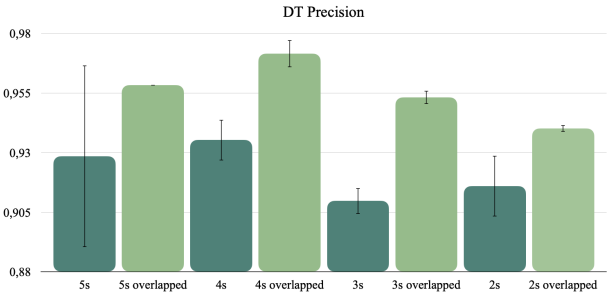


(d)

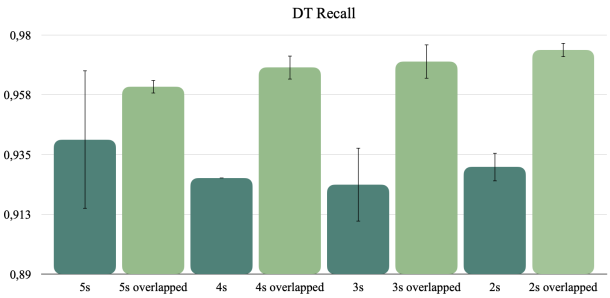
Fig. 6.8. MLP performances for *window size = 2 and 5 sec* - a) Accuracy; b) Precision; c) Recall; d) F-measure.



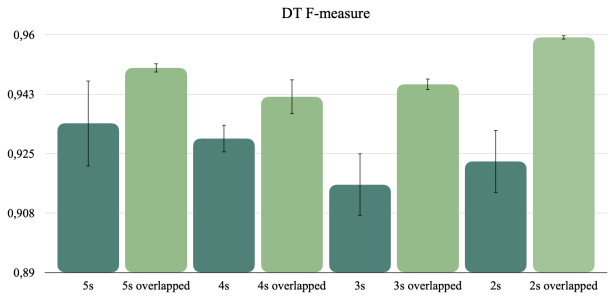
(a)



(b)



(c)



(d)

Fig. 6.9. TREE performances for *window size = 2 and 5 sec* - a) Accuracy; b) Precision; c) Recall; d) F-measure.

Classifiers comparison

By comparing the results of MLP and DT we observe that the latter achieves better results, in all the scenarios. However, we can see the same trend followed by both the models, which can be summarized as follows: i) the performances decrease as the size of the time window increases; ii) the datasets with overlapping induce more performing models than those without overlap.

Figure 6.10 shows a comparison between the two models and the accuracy values that they reach in the best case scenario, which is window size = 2 sec with overlapping.

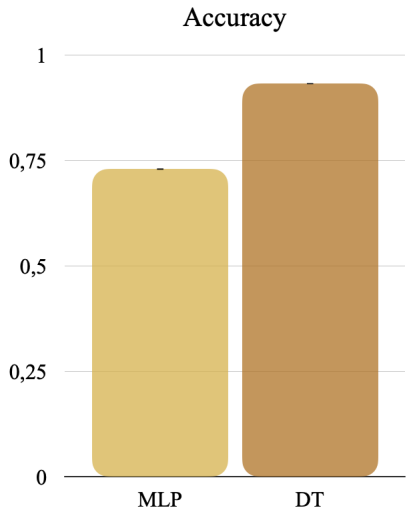


Fig. 6.10. MLP/DT Accuracy comparison for *window size = 2 sec with overlapping*

6.4.4 Working with simulated scenarios

It is important to emphasize that in all the scenarios in which the CRAWDAD dataset was collected, only two nodes, exchanging IEEE 802.11p packets, are involved in communication and therefore the network design does not take into account the possible presence of other devices that transmit legitimately. Moreover, those nodes only move in two directions being terrestrial vehicles and not flying drones. These simplifications could constitute a limitation and not a faithful representation of UAVs communications. However, according to the Federated Aviation Administration, Fact Sheet on Unmanned Aircraft Regulations [107] the UAVs usually maintain a fixed altitude because of which the third dimension of the UAVs in the FANET can be considered fixed. Therefore, given that no other standard UAV jamming attack datasets are currently available, the jamming attack dataset in the vehicular ad hoc network performs as the closest fit for the numerical analysis and it can be used as a proof-of-concept for jamming detection in FANET.

By following the previous considerations, aiming at generating and studying more realistic drone network communication scenarios to validate our proposal, we also simulated several ns-3 [4] based Flying Ad hoc Network topologies with three-dimensional mobility model in an adhoc setting, communicating over the WiFi physical standard 802.11n [9, 50]. A jammer node was introduced with reactive RF jamming signals which interferes with the communi-

cation between UAV nodes playing the roles of simple transmitters and an UAV equipped with long-range communication technology playing the role of a receiver server node, in the three-dimensional UAV ad hoc Gauss Markov mobility model [130]. We extracted 15786 instances with 12 features each consisting of PDR, RSSI, and Throughput considering a window size of 2 sec with overlapping.

Two network topologies were considered: linear and grid. The linear scenarios consist of nodes arranged in a straight line, while in the grid ones the transmitters surround the receiver, making up each side of a square. As an example, this last configuration can represent a real application scenario in which the receiver drone can work as an edge gateway collecting the information coming from the other neighboring drones in order to process data locally at the edge or to forward data to a remote server using long-range communication technologies. In any case, the jammer is going after the receiver, in a *follow-me* pattern. All the nodes are moving at similar speeds, between 25 and 35 km/h and at an altitude between 5 and 10 meters.

For each of the considered settings, the jammer was placed at two different distances from the receiver in order to emulate two types of attack: the closer it is, the stronger the attack.

Moreover, in order to test the proposed framework in different channel conditions, we varied the average interference values in terms of RSSI experienced by the drone acting as receiver. In this

Table 6.3. Simulation parameters

Parameter	Value
Drones speed	[25km/h - 35km/h] uniformly distributed
Drones height	[5m - 10m] uniformly distributed
Flying time	5 minutes
Radio propagation model	Friis
Communication technology	IEEE 802.11n
Average receiver interference	high, medium, low
Data rate	90 packets/sec
Jamming type	Reactive
Windows size	2 secs with overlapping

way we could collect and analyze communication data traces in which the effect of radio interference was considered. In particular, according to the work in [153] and the selected radio communication technology, we considered two opposite channel conditions (*i.e.*, good and poor) corresponding to mean RSSI values of $\sim$ -63dBm and $\sim$ -88dBm respectively; we also tested intermediate conditions of low interference ($\sim$ -74dBm) and medium interference ($\sim$ -81dBm). Table 6.3 summarizes the main simulation parameters.

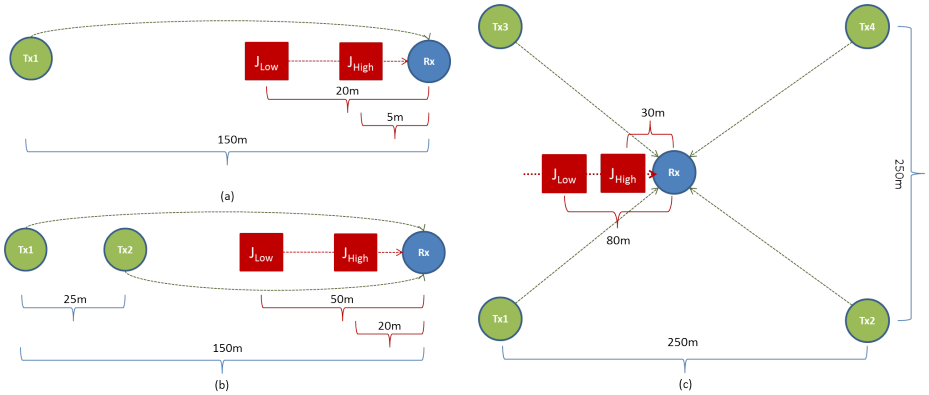


Fig. 6.11. Simulated drone network topology - a) Linear-1Tx; b) Linear-2Tx; c) Grid-4Tx;

We started by analyzing three basic settings, which are shown in Figure 6.11: two linear, with one and two transmitting nodes respectively, and a grid, with four transmitting nodes. In details, in order to emulate a fairly effective attack, the jammer was placed at 5m and at 20m in the linear scenario with one transmitter, at 20m and at 50m in the linear scenario with two jammers, and at 30m and at 80m in the grid scenario with four transmitters.

Later, we generated some more complex settings, still following the linear and grid network topologies, but with a larger number of transmitters: linear with 4 transmitters, linear with 8 transmitters, linear with 10 transmitters, grid with 8 transmitters, and grid with 10 transmitters.

Results

In this section we present the results obtained throughout the simulations in terms of classifiers accuracy.

First of all, we generated several datasets collecting the packets exchanged among nodes in each of the previously depicted basic scenarios, and we used them to build a series of classifiers, whose accuracy values are shown in table 6.4. It is worth to note that, in these basic scenarios, the DT classifiers always achieves better performances compared to the MLPs, regardless of the interference and jamming levels. Afterward, we combined the datasets related to the same scenario obtaining three new datasets comprising of different interference and jamming levels, and we trained new DTs and MLPs on these datasets. Then, these three datasets were merged into a single one, which we used to train two new classifiers: a DT and a MLP, reaching an accuracy of 93% and 86.7% respectively. These two classifiers were tested on several datasets corresponding to both the basic and more complex scenarios, aiming at making more general the conducted study. The resulting accuracy values are listed in Table 6.5.

By comparing the performances of the classifiers on each dataset individually, we can observe that DT performs better than MLP only for those datasets used to build the classifiers. In other words, MLP seems to be more capable of generalization, achieving higher accuracy values than DT when the testset is not included in the initial dataset.

Table 6.4. Decision Tree (DT) and Multi Layer Perceptron (MLP) Accuracy in reference communication scenarios.

Communication Scenario	Interference Level	Jamming Level	Accuracy	
			DT	MLP
Linear-1Tx	None	Low	0,9	0,87
		High	0,94	0,93
	Low	Low	0,87	0,85
		High	0,92	0,91
	Medium	Low	0,81	0,79
		High	0,87	0,85
	High	Low	0,71	0,67
		High	0,76	0,74
Linear-2Tx	None	Low	1	0,52
		High	1	0,51
	Low	Low	0,93	0,53
		High	0,92	0,507
	Medium	Low	0,91	0,52
		High	0,9	0,51
	High	Low	0,98	0,505
		High	0,976	0,5
Grid-4Tx	None	Low	0,998	0,73
		High	1	0,75
	Low	Low	0,998	0,73
		High	0,998	0,81
	Medium	Low	0,998	0,75
		High	0,998	0,79
	High	Low	0,998	0,75
		High	0,998	0,83

6.4.5 Framework execution time and memory occupation

The proposed framework allows to perform a jamming attack detection activity in wireless drone networks.

Table 6.5. Accuracy of trained classifiers on new and more complex communication scenarios.

Linear Scenarios	DT	MLP	Grid Scenarios	DT	MLP
1Tx	0.85	0.82	4Tx	1	0.94
2Tx	0.92	0.80	8Tx	0.5	0.9
4Tx	0.9	0.95	10Tx	0.5	0.96
8Tx	0.5	0.79	Training Set	0.93	0.86
10Tx	0.36	0.92	Linear 1Tx +2Tx +Grid 4Tx		

Since our goal is to integrate it into a multi-UAV system having limited computing powers, we carried out an analysis in terms of execution time and memory occupation in order to verify the feasible implementation of the framework.

The framework has been tested on a constrained device such as the Raspberry Pi 3, whose main hardware characteristics are:

- CPU: 4×ARM Cortex-A53, 1.2GHz
- GPU: Broadcom VideoCore IV
- RAM: 1GB LPDDR2 (900 MHz)
- Bluetooth: Bluetooth 4.1 Classic, Bluetooth Low Energy

These features are supported by most commercial drones, so it is realistic to think that the framework can also run on other UAVs having similar computing power.

By executing the framework we discovered that the memory space occupied by an MLP varies between 10256 and 16552 bytes. An instance of the Decision Tree occupies 2192 bytes, instead.

Table 6.6. Online Execution Times of MLP and DT

Window Size	**MLP** [s]	**DT** [s]
2 sec	0.399	0.378
3 sec	0.392	0.381
4 sec	0.383	0.376
5 sec	0.401	0.390

We also observed a time execution which ranges between 0.378 sec and 0.4 sec. The exact values are listed in Table 6.6. Clearly, this time refers to the online execution of the framework, that is to the actual detection activity, while the offline one varies between 5.02 sec and 7.5 sec for the MLP, and between 0.44 sec and 0.8 sec for the DT.

Ideally, execution times should vary according to the size of the window: the wider the window, the greater the number of traces that fall within it. This, however, is not true in the case of possible interference or jamming: during an attack, in a 5 sec window no packet could be sent (or received), while, in normal conditions, in a 2 sec window much more data could be transmitted.

In conclusion we verified that Off-the-Shelf flying drones such as the *Intel Aero Ready-to-Fly* [3] have hardware features that exceed those listed above, with a 2.4 GHz Quad Core CPU and 4GB of RAM; thus, these devices are easily able to run the framework even shorter times.

7

CSRF vulnerabilities detection

The rapid proliferation of web applications has significantly impacted the digital landscape, enabling efficient connectivity, limitless information sharing, and interactive user experiences. However, this impressive expansion has also led to new challenges in terms of web security. As web applications become increasingly integrated into various domains of daily life, they attract the attention of malicious actors who exploit vulnerabilities to execute advanced attacks. One of the most significant threats to web security is the Cross-Site Request Forgery attack. Cross-Site Request Forgery (CSRF) [92] is a malicious attack that exploits vulnerabilities in web applications to force an end user to perform unwanted actions on a web platform where they are currently authenticated. This type of attack takes advantage of the way web applications handle authentication requests, as they automatically include any relevant credentials associated with the site, such as session cookies, IP addresses, or domain-specific credentials. When a user is authenticated to a website, the site cannot differentiate between

a legitimate request initiated by the victim and a forged request initiated by an attacker sent through another compromised website visited by the user. This lack of distinction enables the attacker to deceive the server into treating the forged request as a genuine one. By exploiting the trust placed in an authenticated user, CSRF attacks can lead to unauthorized changes and potentially significant consequences for both the victim and the targeted web platform, including unauthorized alterations of the victim's email or password, or the submission of deceptive purchase requests. These attacks can result in various illicit activities, such as unauthorized financial transactions. Moreover, CSRF attacks can be employed to manipulate online systems, thereby compromising the integrity of sensitive data, such as medical records or election voting systems.

Although several security measures and mitigation approaches have been proposed in literature to prevent and detect CSRF attacks [38, 40, 69, 142, 86] - such as adding random tokens to HTTP requests, checking HTTP Referer headers or Origin headers, or using SameSite cookies - they all share a common challenge: they require meticulous and careful consideration regarding their placement within an application to ensure the necessary security controls do not compromise the user experience. Finding the optimal placement can be a daunting task for web developers, and although modern frameworks provide automatic support for this procedure,

CSRF vulnerabilities are still common even in high-level web applications.

In this Chapter the prevention of CSRF attacks is discussed and the rest of the article is organized as follows: Section 7.1 provides an overview of the current state of the art in scientific literature pertaining to CSRF attack detection techniques. In Section 7.2, a proposal framework for CSRF vulnerabilities detection is presented [17], commencing with a description of the dataset employed and culminating in an explanation of the neural network architectures adopted. Section 7.3 presents the results obtained employing the framework and compares them with those achieved in prior studies.

The framework that my co-authors and I proposed leverages advanced deep learning techniques to address this persistent security concern is presented. A fundamental aspect of such methodology involves the classification of HTTP requests, distinguishing them as either security-sensitive or non-security-sensitive. To conduct this study, a dataset comprising real-world HTTP requests was utilized. Such HTTP requestes were subsequently categorized as either *sensitive* or *insensitive* [2]. A sensitive request is defined as an HTTP request that induces a significant security-related state change in the web application and is processed within the valid authentication context of a registered user. The identification of sensitive requests holds paramount importance in the detection of CSRF vulnerabilities, as the exploitation of such vulnerabilities

hinges on the attacker's knowledge of the requisite parameters and values. However, the task of identifying sensitive requests is generally arduous due to the necessity to comprehend the semantics of the web application and monitor server-side state alterations.

7.1 State-of-the-art

Several works have been proposed in literature that leverage machine learning techniques to address security threats and attacks associated with both web and non-web domains [141, 53, 112, 75, 93]. Mitch [23] is a Machine Learning-based framework for classifying HTTP requests into sensitive or non-sensitive categories with respect to CSRF vulnerabilities. The authors of Mitch make a significant contribution in two aspects. Firstly, they provide a dataset of HTTP requests collected from popular websites. This dataset serves as the foundation for their research, enabling them to extract and analyze meaningful features. Secondly, through a meticulous process of feature selection, they employ this dataset to build a powerful machine learning model capable of distinguishing sensitive requests from non-sensitive ones. Their approach showcases the potential of leveraging machine learning techniques to tackle the challenge of identifying and protecting against CSRF attacks.

Other works can be found in the literature that focus on providing methods to detect CSRF vulnerabilities, but they achieve infe-

rior results [99, 39, 119, 129, 143, 150]. Browser-Enforced Authenticity Protection (BEAP) [99] counters CSRF attacks by evaluating user intent and token sensitivity, discarding incongruous tokens based on request attributes and real-world web app heuristics. De Rick et al. propose CeaseFire (CsFire) [39], a client-side browser extension for robust CSRF defense. Employing a predefined policy, it discerns restricted cross-domain requests, employing either request blocking or credential stripping. The extension integrates client-side and server-driven policies for heightened protection and adaptability. Deemon [119] is an automated security testing framework for detecting CSRF vulnerabilities. It adopts a novel modeling paradigm that unifies web application aspects, constructing a comprehensive property graph through execution traces, data flows, and architecture tiers. Deemon employs this model to automatically identify, validate, and conduct security tests for CSRF concerns. Rocchetto et al. [129] present a formal model-based approach for automated CSRF detection. The technique defines specific web app specifications to expose CSRF vulnerabilities. Utilizing an intruder model akin to Dolev-Yao, it explores how intruder-cryptographic protocol interactions yield CSRF attacks. Shahriar et al. [143] present a client-side CSRF attack detection mechanism, utilizing parameter-value matching and visibility checks against a webpage's input fields. Content type validation supplements defense against attacker evasion. Implemented as a Firefox browser plug-in tool, it's evaluated on real PHP programs and a test suite,

revealing effective detection of typical CSRF attacks, even with partial visible form fields. Sudhodanan et al [150] investigate Authentication CSRF attacks on web authentication and identity functions. They formulate 7 security testing strategies after analyzing reported attacks and developed a penetration testing tool extension "CSRF-checker".

7.2 Proposed framework

In this Section the deep learning-based framework that my co-authors and I designed to address the detection of Cross-Site Request Forgery (CSRF) vulnerabilities is presented. The framework centers on a neural network model meticulously trained using an extensive dataset of HTTP requests, with the primary objective of classifying these requests into two categories: security-*sensitive* or security-*insensitive*. Our model was trained using a sample of the dataset collected and used for the training of Mitch [2].

This original dataset comprised 5,828 HTTP requests, featuring an extensive array of 49 distinctive attributes, encompassing structural properties, textual descriptors, and the classification of HTTP methods (namely, GET and POST) associated with each request. However, a notable challenge surfaced due to a significant imbalance in the distribution of labels, with far more non-sensitive requests than sensitive ones. To address this imbalance, a data sampling strategy was applied, resulting in the retention of

1,000 insensitive instances while preserving the original sensitive instances. The outcome of this procedure yielded a more modestly sized dataset, yet one marked by substantially improved label balance. This transformation, along with other preprocessing steps such as standardization reduction, was undertaken with the objective of preparing the dataset optimally for subsequent stages of classification model training. The final dataset comprised 1,901 records, characterized by 12 distinct columns representing relevant features.

7.2.1 Model tuning

The constructed model is a neural network with Dense-type hidden layers. As illustrated in *Code* 7.1, the layers are configured with an initial increase and subsequent decrease in the number of units (neurons). This approach is designed to augment the dimensionality of the input data and subsequently reduce it until reaching the final layer, which consists of a single unit responsible for generating the model's prediction.

The model setup begins with the utilization of the *Sequential()* class, followed by the definition of the first input layer with a dimensionality of 11.Subsequently, additional layers are added, with meticulous consideration given to the number of units assigned to each layer.

```
def createModel(alpha_val=0.3,drop_val=0.1,lr=0.00001):
    model=Sequential()
    model.add(Dense(32,LeakyReLU(alpha=alpha_val), input_dim=11))
    model.add(Dense(64,LeakyReLU(alpha=alpha_val)))
    model.add(Dropout(dropout_val))
    model.add(Dense(128,LeakyReLU(alpha=alpha_val)))
    model.add(Dropout(dropout_val))
    model.add(Dense(64,LeakyReLU(alpha=alpha_val)))
    model.add(Dense(16,LeakyReLU(alpha=alpha_val)))
    model.add(Dense(1,activation='sigmoid'))
    model.compile(optimizer=Adam(learning_rate=lr),
    loss='binary_crossentropy',metrics=['accuracy'])
    return model
```

Listing 7.1. *Python method for model creation*

Moreover, the neural network architecture includes hidden layers with the Dropout mechanism. This particular layer type func-

tions by randomly dropping nodes with a probability denoted as p within the neural network. When a node is dropped, both its forward and backward connections are temporarily severed, resulting in an altered network architecture distinguishable from the original. In our configuration, we have integrated two Dropout layers, each employing a dropout rate of 0.1, which corresponds to a 10% probability of node deactivation during the training phase. This strategy is aimed at mitigating the occurrence of overfitting in the model. Concerning the choice of activation function employed in the layers of our model, we opted for the `LeakyReLU` (Leaky Rectified Linear Unit) function. This activation function serves as an extension of the conventional `ReLU` function, wherein negative values exhibit a minor slope rather than remaining flat. This modification introduces increased adaptability and precision in the weight updates throughout the training process, albeit at the cost of a slight increment in computational complexity. The slope of this function is governed by the parameter α, which is established prior to training. In our specific configuration, we set α to the value of 0.3. As for the activation function employed in the final layer, we selected the `Sigmoid` function, given the binary classification nature of our task. Furthermore, we incorporated the `Adam` algorithm as the optimizer for our model, representing an efficient extension of the Stochastic Gradient Descent algorithm. This optimization technique demonstrates exceptional efficiency, particularly in the context of large-scale problems involving ex-

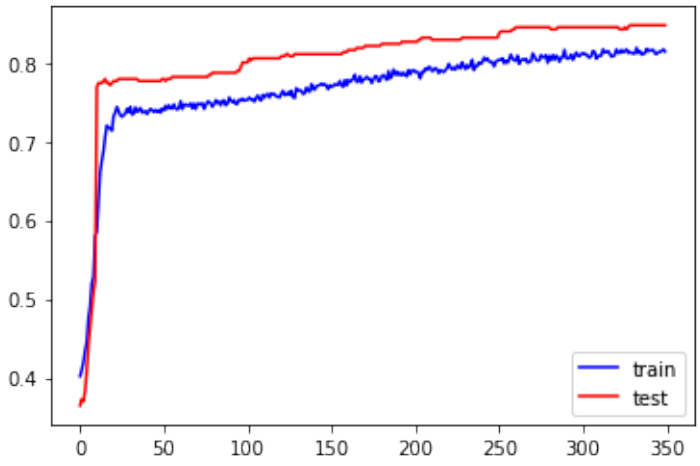
tensive datasets or numerous parameters, as it imposes a reduced memory demand. As for the choice of the loss function, we selected `Binary Crossentropy`. This loss function operates by comparing each predicted probability with the actual output, which can assume values of 0 or 1 in the context of binary classification. Essentially, it furnishes a metric that gauges the proximity between our model's predictions and the ground truth values.

7.2.2 Model training

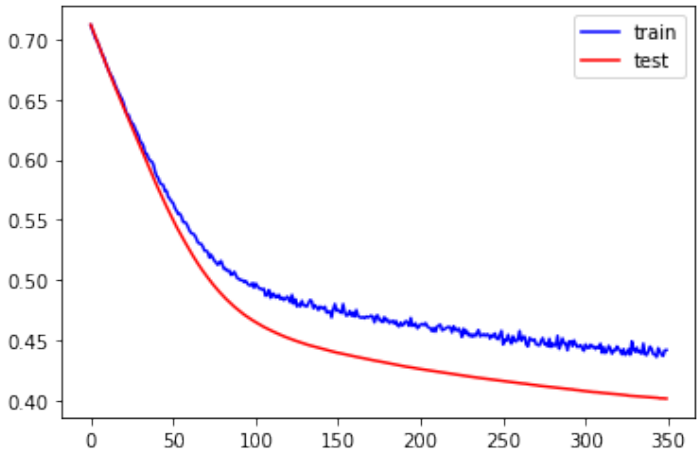
The model was trained for 350 *epoches* with a *learning rate* set to 10^{-5}. The number of epochs serves as a hyperparameter, denoting the number of times the learning algorithm iterates through the entire training dataset, while the learning rate defines the rate at which the model acquires knowledge. The two parameters are closely related, as the learning rate, a key factor in the stochastic gradient descent algorithm, influences the number of training epochs by adjusting the rate at which the model learns from the data. Additionally, the *batch size*, which is a hyperparameter defining the number of samples that are processed before updating the internal weights of the model, was set to 64.

The model's performance displayed a positive trajectory, consistently achieving an average accuracy of 0.85 across multiple evaluations.

The accuracy trend chart (Figure 7.1(a)), pertaining to one of the iterations within the 10-fold cross-validation process, il-



(a) Accuracy trend



(b) Loss trend

Fig. 7.1. Accuracy and loss trend plots

lustrates a stable and consistent growth pattern throughout the training epochs. Notably, there is no evidence of overfitting, as evidenced by the higher accuracy achieved on the test set compared to the training set. A similar trend is observed in the loss trend chart, as depicted in Figure 7.1(b), which showcases a well-defined

and consistent decrease. Once again, the absence of overfitting is evident when comparing the performance on the training set and the test set in these charts.

7.3 Experimental Evaluation

In this Section we present the results and performance obtained during the evaluation of the implemented system. Throughout this phase, we adopted various approaches and evaluation metrics in order to gain a comprehensive understanding of the model's performance.

To assess the effectiveness of our framework, we conducted a comprehensive evaluation using a 10-fold cross-validation procedure, thus mitigating the risk of overfitting and yielding a more robust evaluation. We computed several standard evaluation metrics, such as accuracy, precision, recall, and F1-score, for each fold and report the average values as well as standard deviations to provide a comprehensive understanding of the framework's stability and consistency across different subsets of the data. In paricular, our results indicate that the proposed framework achieved an accuracy score of 0.83, highlighting its capacity to correctly classify instances. Moreover, the framework exhibited a precision of 0.81, signifying its ability to accurately identify positive instances among the predicted positive cases. Additionally, the recall value of 0.85 demonstrates the framework's proficiency in correctly iden-

tifying positive instances among all actual positives. The F1-score, which balances precision and recall, yielded a score of 0.82.

Additionally, we provide an evaluation based on the AUC-ROC score, which is highly valuable for observing and quantifying the trained model's ability to differentiate between items belonging to the two distinct classes.

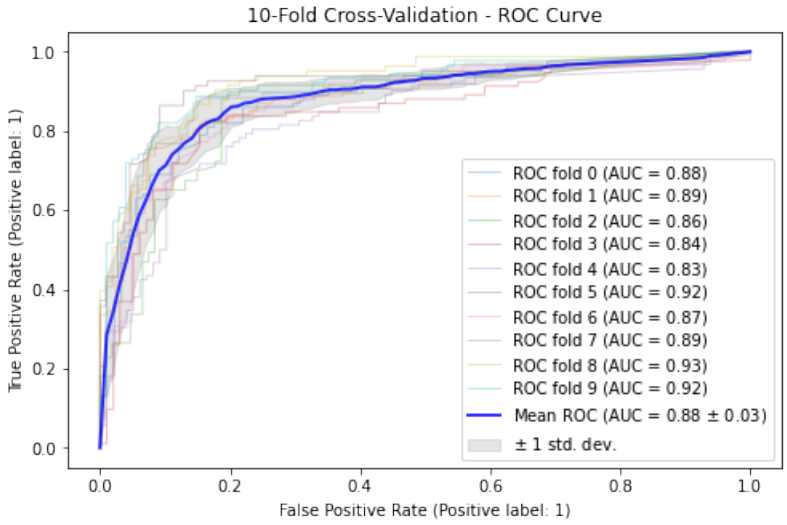


Fig. 7.2. *ROC Curve*

After an exhaustive analysis, we report that the mean of the ROC-AUC value obtained across the 10-fold cross validation was found to be 0.88. The ROC curve's performance trajectory throughout this evaluation is illustrated in Figure 7.2.

Model comparison

Given our development of a framework utilizing the dataset originally collected by the authors of Mitch and adopted to train their model, a comparative analysis with the Mitch framework was a natural course of action. By comparing the individual measured results with the evaluation metrics and taking into account the use of the same dataset for training the respective models, it can be inferred that our work performed successfully met the expected performance expectations. The metrics for *precision*, *recall*, and *f1-score* (equal to **0.81**, **0.83**, and **0.82**, respectively) outperformed those observed in the Mitch (**0.78**, **0.67**, and **0.72**). These findings underscore the improved capacity of the implemented neural network model to learn from the data during the training process, as evidenced in Table 7.1. Nevertheless, it is imperative to acknowledge that the system featured in the original study encompasses, in addition to the classification model, robust heuristics designed to conduct field checks of the classification results. In contrast, our approach exclusively depends on the classification capabilities of the neural network model.

Table 7.1. *Metrics comparison*

Classifier	**Precision**	**Recall**	**F1-Score**
RF (Mitch)	0.78	0.67	0.72
Neural Network	**0.81**	**0.85**	**0.82**

8

Conclusions

The Internet of Things (IoT) represents a revolutionary paradigm in the world of technology, offering unparalleled levels of connectivity and data exchange among smart devices. However, this interconnectedness brings forth a significant challenge in terms of security. As IoT systems become increasingly complex, incorporating a wide range of technologies, and operating with limited resources in terms of computational power, energy, and memory, the development and adaptation of standardized security solutions become a formidable task.

The importance of researching and implementing new methodologies and tools for IoT security cannot be overstated. This thesis has sought to address the shortcomings in methods and approaches within the IoT security paradigm, focusing on the protection of the software that runs on these devices, particularly through techniques such as code obfuscation. Chapter 2 provided an in-depth exploration of code obfuscation and its utility, emphasizing how it is also employed in the context of malware writing to evade tradi-

tional antivirus systems based on signature scanning mechanisms. In this context, the ability to detect obfuscated code is crucial for two primary reasons: it can serve as the first step toward identifying malicious code, and it is invaluable for assessing the protection applied to the code.

Chapter 3 expanded on the state-of-the-art analysis of the scientific literature concerning code obfuscation detection techniques and introduced a proposal for analyzing static properties of programs that are subject to changes following the application of specific obfuscating transformations.

In Chapter 4 delved into the topic of signature scanning and the limitations of this widely used technique in the context of malware detection when dealing with obfuscated code or code compiled for different architectures, as is often the case in the IoT domain. We discussed the state-of-the-art in the scientific literature regarding methods to enhance signature scanning.

Chapter 5 addressed the issue of firmware re-hosting and the challenges of applying traditional binary analysis techniques to firmware, which is particularly problematic due to the numerous and frequent interactions firmware has with hardware peripherals. In this chapter, we introduced a new methodology to enable binary analysis techniques that demand frequent and rapid executions, such as fuzzing, for testing firmware security, using binary rewriting techniques.

Chapter 6 examined the challenges associated with detecting jamming attacks within networks of devices, such as drones, and delved into the current state of research in this domain. Furthermore, this chapter introduced a novel methodology grounded in machine learning techniques that aims to overcome the current limitations in jamming attack detection, revealing promising results.

Throughout this thesis, we have underscored the critical importance of research in the field of IoT security. The ever-evolving landscape of IoT technology demands continuous innovation and adaptation in the realm of security to safeguard the increasing number of interconnected devices that have been deeply integrated into modern life. As we conclude this research, it is clear that the need for robust IoT security solutions will remain a vital pursuit, and the insights presented herein provide a foundation for future research and development in this crucial domain.

References

1. https://lcamtuf.coredump.cx/afl/
2. Dataset "mitch" github, https://github.com/alviser/mitch
3. Intel® aero ready to fly drone. https://www.intel.it/content/www/it/it/drones/aero-ready-to-fly-brief.html, [Online; accessed 04-January-2021]
4. Network simulator 3. http://www.nsnam.org/, [Online; accessed 04-January-2021]
5. AL-Taharwa, I.A., Lee, H.M., Jeng, A.B., Wu, K.P., Ho, C.S., Chen, S.M.: Jsod: Javascript obfuscation detector. Security and Communication Networks **8**(6), 1092–1107 (2015)
6. Alam, S., Horspool, R.N., Traore, I.: Mail: Malware analysis intermediate language: a step towards automating and optimizing malware detection. In: Proceedings of the 6th International Conference on Security of Information and Networks. pp. 233–240 (2013)
7. Aljehani, M., Inoue, M.: Communication and autonomous control of multi-uav system in disaster response tasks. In: Jezic, G., Kusek, M., Chen-Burger, Y.J., Howlett, R.J., Jain, L.C. (eds.) Agent and Multi-Agent Systems: Technology and Applications, 11th KES International Conference, KES-AMSTA 2017, Vilamoura, Algarve, Portugal, June 21-23, 2017, Proceedings. Smart Innovation, Systems and Technologies, vol. 74, pp. 123–132. Springer (2017). https://doi.org/10.1007/978-3-319-59394-4_12, https://doi.org/10.1007/978-3-319-59394-4_12
8. Alvarez, V.M.: Yara, https://virustotal.github.io/yara/

9. Asadpour, M., Giustiniano, D., Hummel, K.A., Heimlicher, S.: Characterizing 802.11n aerial communication. In: Kumar, P.R., Ghose, D., Namuduri, K., Wan, Y. (eds.) Proceedings of the second ACM MobiHoc workshop on Airborne networks and communications, ANC@MobiHoc 2013, Bangalore, India, July 29, 2013. pp. 7–12. ACM (2013). https://doi.org/10.1145/2491260.2491262, https://doi.org/10.1145/2491260.2491262
10. Bacci, A., Bartoli, A., Martinelli, F., Medvet, E., Mercaldo, F.: Detection of obfuscation techniques in android applications. In: Proceedings of the 13th International Conference on Availability, Reliability and Security. pp. 1–9 (2018)
11. Balakrishnan, A., Schulze, C.: Code obfuscation literature survey. CS701 Construction of compilers **19** (2005)
12. Baseca, C.C., Díaz, J.R., Lloret, J.: Communication ad hoc protocol for intelligent video sensing using ar drones. In: 2013 IEEE 9th International Conference on Mobile Ad-hoc and Sensor Networks. pp. 449–453 (2013). https://doi.org/10.1109/MSN.2013.115
13. Bat-Erdene, M., Kim, T., Li, H., Lee, H.: Dynamic classification of packing algorithms for inspecting executables using entropy analysis. In: 2013 8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE). pp. 19–26. IEEE (2013)
14. Bauman, E., Lin, Z., Hamlen, K.W., et al.: Superset disassembly: Statically rewriting x86 binaries without heuristics. In: NDSS (2018)
15. Bekmezci, I., Sahingoz, O.K., Temel, S.: Flying ad-hoc networks (fanets): A survey. Ad Hoc Networks **11**(3), 1254–1270 (2013). https://doi.org/10.1016/j.adhoc.2012.12.004, https://doi.org/10.1016/j.adhoc.2012.12.004
16. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX annual technical conference, FREENIX Track. vol. 41, pp. 10–5555. Califor-nia, USA (2005)
17. Beltrano, G., Greco, C., Ianni, M., Fortino, G.: Deep learning-based detection of csrf vulnerabilities in web applications. In: 2023 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology

Congress (DASC/PiCom/CBDCom/CyberSciTech). pp. 0916–0920 (2023). https://doi.org/10.1109/DASC/PiCom/CBDCom/Cy59711.2023.10361414

18. Bensalem, M., Singh, S.K., Jukan, A.: On detecting and preventing jamming attacks with machine learning in optical networks. In: 2019 IEEE Global Communications Conference, GLOBECOM 2019, Waikoloa, HI, USA, December 9-13, 2019. pp. 1–6. IEEE (2019). https://doi.org/10.1109/GLOBECOM38437.2019.9013238, https://doi.org/10.1109/GLOBECOM38437.2019.9013238
19. Blanc, G., Miyamoto, D., Akiyama, M., Kadobayashi, Y.: Characterizing obfuscated javascript using abstract syntax trees: Experimenting with malicious scripts. In: 2012 26th International Conference on Advanced Information Networking and Applications Workshops. pp. 344–351. IEEE (2012)
20. Bonfante, G., Kaczmarek, M., Marion, J.Y.: Control flow graphs as malware signatures. In: International workshop on the Theory of Computer Viruses (2007)
21. Bruschi, D., Martignoni, L., Monga, M.: Code normalization for self-mutating malware. IEEE Security & Privacy **5**(2), 46–54 (2007)
22. Buck, B., Hollingsworth, J.K.: An api for runtime code patching. The International Journal of High Performance Computing Applications **14**(4), 317–329 (2000)
23. Calzavara, S., Conti, M., Focardi, R., Rabitti, A., Tolomei, G.: Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. 2019 IEEE European Symposium on Security and Privacy (EuroS&P) (2019). https://doi.org/10.1109/eurosp.2019.00045
24. Cambra Baseca, C., Sendra, S., Lloret, J., Tomas, J.: A smart decision system for digital farming. Agronomy **9**(5), 216 (Apr 2019). https://doi.org/10.3390/agronomy9050216, http://dx.doi.org/10.3390/agronomy9050216
25. Cao, C., Guan, L., Ming, J., Liu, P.: Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In: Annual Computer Security Applications Conference. pp. 746–759 (2020)
26. Cesare, S., Xiang, Y.: Malware variant detection using similarity search over sets of control flow graphs. In: 2011IEEE 10th International Conference on

Trust, Security and Privacy in Computing and Communications. pp. 181–189 (2011). https://doi.org/10.1109/TrustCom.2011.26

27. Chaganti, R., Ravi, V., Pham, T.D.: Deep learning based cross architecture internet of things malware detection and classification. Computers & Security **120**, 102779 (2022)
28. Chao, H., Cao, Y., Chen, Y.: Autopilots for small fixed-wing unmanned air vehicles: A survey. In: 2007 International Conference on Mechatronics and Automation. pp. 3144–3149. IEEE (2007)
29. Chen, D.D., Woo, M., Brumley, D., Egele, M.: Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS. vol. 1, pp. 1–1 (2016)
30. Choi, Y., Kim, T., Choi, S., Lee, C.: Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In: International Conference on Future Generation Information Technology. pp. 160–172. Springer (2009)
31. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. Tech. rep., WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES (2006)
32. Cifuentes, C., Lewis, B., Ung, D.: Walkabout-a retargetable dynamic binary translation framework. In: Workshop on Binary Translation. pp. 22–25 (2002)
33. Clements, A.A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., Payer, M.: {HALucinator}: Firmware re-hosting through abstraction layer emulation. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1201–1218 (2020)
34. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand (1997)
35. Conti, M., Vinod, P., Vitella, A.: Obfuscation detection in android applications using deep learning. Journal of Information Security and Applications **70**, 103311 (2022)
36. Corteggiani, N., Camurati, G., Francillon, A.: Inception:{System-Wide} security testing of {Real-World} embedded systems software. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 309–326 (2018)

37. Costin, A., Zarras, A., Francillon, A.: Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. pp. 437–448 (2016)
38. Czeskis, A.: Lightweight server support for browser-based csrf protection. Proceedings of the 22nd international conference on World Wide Web (2013). https://doi.org/10.1145/2488388.2488413
39. De Ryck, P., Desmet, L., Heyman, T., Piessens, F., Joosen, W.: Csfire: Transparent client-side mitigation of malicious cross-domain requests. In: Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings 2. pp. 18–34. Springer (2010)
40. De Ryck, P., Desmet, L., Joosen, W., Piessens, F.: Automatic and precise client-side protection against csrf attacks. In: Computer Security–ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings 16. pp. 100–116. Springer (2011)
41. Di Felice, M., Trotta, A., Bedogni, L., Bononi, L., Panzieri, F., Ruggeri, G., Loscrì, V., Pace, P.: Stem-mesh: Self-organizing mobile cognitive radio network for disaster recovery operations. In: 9th International Wireless Communications and Mobile Computing Conference (IWCMC). pp. 602–608 (2013)
42. Djuraev, S., Choi, J., Sohn, K., Nam, S.Y.: Channel hopping scheme to mitigate jamming attacks in wireless lans. EURASIP J. Wirel. Commun. Netw. **2017**, 11 (2017). https://doi.org/10.1186/s13638-016-0785-z, `https://doi.org/10.1186/s13638-016-0785-z`
43. Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., Zhang, K.: Understanding android obfuscation techniques: A large-scale investigation in the wild. In: International conference on security and privacy in communication systems. pp. 172–192. Springer (2018)
44. Driller, M.: Metamorphism in practice. 29A Magazine **1** (2002)
45. Feng, B., Mera, A., Lu, L.: {P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1237–1254 (2020)
46. Fortino, G., Greco, C., Guzzo, A., Ianni, M.: Enabling faster security assessment of re-hosted firmware. In: 2022 IEEE Intl Conf

on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech). pp. 1–6 (2022). https://doi.org/10.1109/DASC/PiCom/CBDCom/Cy55231.2022.9927780

47. Fortino, G., Greco, C., Guzzo, A., Ianni, M.: Sigil: A signature-based approach of malware detection on intermediate language. In: The 6th International Workshop on Attacks and Defenses for Internet-of-Things (ADIoT 2023), ESORICS, TO APPEAR (2023)
48. Gao, N., Qin, Z., Jing, X., Ni, Q.: Anti-intelligent UAV jamming strategy via deep q-networks. In: 2019 IEEE International Conference on Communications, ICC 2019, Shanghai, China, May 20-24, 2019. pp. 1–6. IEEE (2019). https://doi.org/10.1109/ICC.2019.8762016, https://doi.org/10.1109/ICC.2019.8762016
49. Gardner, M.W., Dorling, S.: Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. Atmospheric environment **32**(14-15), 2627–2636 (1998)
50. Goddemeier, N., Rohde, S., Wietfeld, C.: Experimental validation of RSS driven UAV mobility behaviors in IEEE 802.11s networks. In: Workshops Proceedings of the Global Communications Conference, GLOBECOM 2012, 3-7 December 2012, Anaheim, California, USA. pp. 1550–1555. IEEE (2012). https://doi.org/10.1109/GLOCOMW.2012.6477816, https://doi.org/10.1109/GLOCOMW.2012.6477816
51. Greco, C., Fortino, G., Crispo, B., Choo, K.K.R.: Ai-enabled iot penetration testing: state-of-the-art and research challenges. Enterprise Information Systems **17**(9), 2130014 (2023)
52. Greco, C., Ianni, M., Guzzo, A., Fortino, G.: Explaining binary obfuscation. In: 2023 IEEE International Conference on Cyber Security and Resilience (CSR). pp. 22–27. IEEE (2023)
53. Greco, C., Pace, P., Basagni, S., Fortino, G.: Jamming detection at the edge of drone networks using multi-layer perceptrons and decision trees. Applied Soft Computing **111**, 107806 (2021)

54. Grover, K., Lim, A.S., Yang, Q.: Jamming and anti-jamming techniques in wireless networks: a survey. Int. J. Ad Hoc Ubiquitous Comput. **17**(4), 197–215 (2014). https://doi.org/10.1504/IJAHUC.2014.066419, https://doi.org/10.1504/IJAHUC.2014.066419
55. Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y.R., Kruegel, C., et al.: Toward the analysis of embedded firmware through automated re-hosting. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019). pp. 135–150 (2019)
56. Guzzo, A., Ianni, M., Pugliese, A., Saccà, D.: Modeling and efficiently detecting security-critical sequences of actions. Future Generation Computer Systems (2020). https://doi.org/https://doi.org/10.1016/j.future.2020.06.054, http://www.sciencedirect.com/science/article/pii/S0167739X19331528
57. Hentati, A.I., Fourati, L.C.: Comprehensive survey of uavs communication networks. Comput. Stand. Interfaces **72**, 103451 (2020). https://doi.org/10.1016/j.csi.2020.103451, https://doi.org/10.1016/j.csi.2020.103451
58. Hollingsworth, J.K., Miller, B.P., Cargille, J.: Dynamic program instrumentation for scalable performance tools. In: Proceedings of IEEE Scalable High Performance Computing Conference. pp. 841–850. IEEE (1994)
59. Ianni, M., Masciari, E.: A compact encoding of security logs for high performance activity detection. In: 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2021, Valladolid, Spain, March 10-12, 2021. pp. 240–244. IEEE (2021). https://doi.org/10.1109/PDP52278.2021.00045, https://doi.org/10.1109/PDP52278.2021.00045
60. Ianni, M., Masciari, E.: Scout: Security by computing outliers on activity logs. Computers & Security **132**, 103355 (2023). https://doi.org/https://doi.org/10.1016/j.cose.2023.103355, https://www.sciencedirect.com/science/article/pii/S0167404823002651
61. Ianni, M., Masciari, E., Saccà, D.: An overview of the endless battle between virus writers and detectors: How compilers can be used as an evasion technique. In: Proceedings of the 8th International Conference on Data Sci-

ence, Technology and Applications, DATA 2019, Prague, Czech Republic, July 26-28, 2019. pp. 203–208 (2019). https://doi.org/10.5220/0007922802030208, https://doi.org/10.5220/0007922802030208

62. Idika, N., Mathur, A.P.: A survey of malware detection techniques. Purdue University **48** (2007)
63. Jhaveri, R.H., Patel, S.J., Jinwala, D.C.: Dos attacks in mobile ad hoc networks: A survey. In: 2012 second international conference on advanced computing & communication technologies. pp. 535–541. IEEE (2012)
64. Jia, L., Yao, F., Sun, Y., Xu, Y., Feng, S., Anpalagan, A.: A hierarchical learning solution for anti-jamming stackelberg game with discrete power strategies. IEEE Wirel. Commun. Lett. **6**(6), 818–821 (2017). https://doi.org/10.1109/LWC.2017.2747543, https://doi.org/10.1109/LWC.2017.2747543
65. Jiang, S., Fu, C., Qian, Y., He, S., Lv, J., Han, L.: Ifattn: Binary code similarity analysis based on interpretable features with attention. Computers & Security p. 102804 (2022)
66. Jiang, S., Hong, Y., Fu, C., Qian, Y., Han, L.: Function-level obfuscation detection method based on graph convolutional networks. Journal of Information Security and Applications **61** (2021)
67. Jodavi, M., Abadi, M., Parhizkar, E.: Jsobfusdetector: A binary pso-based one-class classifier ensemble to detect obfuscated javascript code. In: 2015 The International Symposium on Artificial Intelligence and Signal Processing (AISP). pp. 322–327. IEEE (2015)
68. Johnson, E., Bland, M., Zhu, Y., Mason, J., Checkoway, S., Savage, S., Levchenko, K.: Jetset: Targeted firmware rehosting for embedded systems. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 321–338 (2021)
69. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing cross site request forgery attacks. In: 2006 Securecomm and Workshops. pp. 1–10 (2006). https://doi.org/10.1109/SECCOMW.2006.359531
70. Julus, L.: Metamorphism. 29A Magazine **1**(5) (2000)
71. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM – software protection for the masses. In: Wyseur, B. (ed.) Proceedings of the IEEE/ACM

1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015. pp. 3–9. IEEE (2015). https://doi.org/10.1109/SPRO.2015.10

72. Kammerstetter, M., Platzer, C., Kastner, W.: Prospect: peripheral proxying supported embedded code testing. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. pp. 329–340 (2014)
73. Kaplan, S., Livshits, B., Zorn, B., Siefert, C., Curtsinger, C.: " nofus: Automatically detecting"+ string. fromcharcode (32)+" obfuscated". tolowercase ()+" javascript code. Technical report, Technical Report MSR-TR 2011–57, Microsoft Research (2011)
74. Karagiannis, D., Argyriou, A.: Jamming attack detection in a pair of RF communicating vehicles using unsupervised machine learning. Veh. Commun. **13**, 56–63 (2018). https://doi.org/10.1016/j.vehcom.2018.05.001, `https://doi.org/10.1016/j.vehcom.2018.05.001`
75. Karimipour, H., Dehghantanha, A., Parizi, R.M., Choo, K.K.R., Leung, H.: A deep and scalable unsupervised machine learning system for cyber-attack detection in large-scale smart grids. IEEE Access **7**, 80778–80788 (2019)
76. Karlof, C., Wagner, D.A.: Secure routing in wireless sensor networks: attacks and countermeasures. Ad Hoc Networks **1**(2-3), 293–315 (2003). https://doi.org/10.1016/S1570-8705(03)00008-8, `https://doi.org/10.1016/S1570-8705(03)00008-8`
77. Khan, M.A., Safi, A., Qureshi, I.M., Khan, I.U.: Flying ad-hoc networks (fanets): A review of communication architectures, and routing protocols. In: First International Conference on Latest trends in Electrical Engineering and Computing Technologies, INTELLECT 2017, Karachi, Pakistan, November 15-16, 2017. pp. 1–9. IEEE (2017). https://doi.org/10.1109/INTELLECT.2017.8277614, `https://doi.org/10.1109/INTELLECT.2017.8277614`
78. Kim, D., Majlesi-Kupaei, A., Roy, J., Anand, K., ElWazeer, K., Buettner, D., Barua, R.: Dynodet: Detecting dynamic obfuscation in malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 97–118. Springer (2017)
79. Kim, D., Kim, E., Cha, S.K., Son, S., Kim, Y.: Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. IEEE

Transactions on Software Engineering (2022)

80. Konstantinou, E., Wolthusen, S.: Metamorphic virus: Analysis and detection. Royal Holloway University of London **15**, 15 (2008)
81. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digital investigation **3**, 91–97 (2006)
82. Koscher, K., Kohno, T., Molnar, D.: {SURROGATES}: Enabling {Near-Real-Time} dynamic analyses of embedded systems. In: 9th USENIX Workshop on Offensive Technologies (WOOT 15) (2015)
83. Kotz, D., Henderson, T., Abyzov, I., Yeo, J.: CRAWDAD dataset dartmouth/campus (v. 2009-09-09). Downloaded from https://crawdad.org/dartmouth/campus/20090909 (Sep 2009). https://doi.org/10.15783/C7F59T
84. Kumar, P., Sharma, P.: Artificial neural networks-a study. International Journal of Emerging Engineering Research and Technology **2**(2), 143–148 (2014)
85. Lakhotia, A., Mohammed, M.: Imposing order on program statements to assist anti-virus scanners. In: Reverse Engineering, 2004. Proceedings. 11th Working Conference on. pp. 161–170. IEEE (2004)
86. Lalia, S., Moustafa, K.: Implementation of web browser extension for mitigating csrf attack. In: Rocha, Á., Adeli, H., Reis, L.P., Costanzo, S. (eds.) New Knowledge in Information Systems and Technologies. pp. 867–880. Springer International Publishing, Cham (2019)
87. Larus, J.R., Ball, T.: Rewriting executable files to measure program behavior. Software: Practice and Experience **24**(2), 197–218 (1994)
88. László, T., Kiss, Á.: Obfuscating c++ programs via control flow flattening. Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **30**(1), 3–19 (2009)
89. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. pp. 75–86 (2004). https://doi.org/10.1109/CGO.2004.1281665
90. Law, A.M., Kelton, W.D., Kelton, W.D.: Simulation modeling and analysis, vol. 3. McGraw-Hill New York (2000)

91. Li, W., Guan, L., Lin, J., Shi, J., Li, F.: From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware. arXiv preprint arXiv:2107.12867 (2021)
92. Lin, X., Zavarsky, P., Ruhl, R., Lindskog, D.: Threat modeling for csrf attacks. In: 2009 International Conference on Computational Science and Engineering. vol. 3, pp. 486–491 (2009). https://doi.org/10.1109/CSE.2009.372
93. Lin, Y., Liu, R., Divakaran, D.M., Ng, J.Y., Chan, Q.Z., Lu, Y., Si, Y., Zhang, F., Dong, J.S.: Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3793–3810 (2021)
94. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices **40**(6), 190–200 (2005)
95. Lundberg, S.M., Erion, G., Chen, H., DeGrave, A., Prutkin, J.M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., Lee, S.I.: From local explanations to global understanding with explainable ai for trees. Nature Machine Intelligence **2**(1), 2522–5839 (2020)
96. Lundberg, S.M., Erion, G.G., Lee, S.: Consistent individualized feature attribution for tree ensembles. CoRR **abs/1802.03888** (2018)
97. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. Advances in neural information processing systems (2017)
98. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer **35**(2), 50–58 (2002)
99. Mao, Z., Li, N., Molloy, I.: Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In: Financial Cryptography and Data Security: 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers 13. pp. 238–255. Springer (2009)
100. Maxa, J.A., Mahmoud, M.S.B., Larrieu, N.: Secure routing protocol design for uav ad hoc networks. In: 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC). pp. 4A5–1. IEEE (2015)
101. McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In:

Singh, A., Zhu, X.J. (eds.) Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA. Proceedings of Machine Learning Research, vol. 54, pp. 1273–1282. PMLR (2017), http://proceedings.mlr.press/v54/mcmahan17a.html

102. Mirzaei, O., de Fuentes, J.M., Tapiador, J., Gonzalez-Manzano, L.: Androdet: An adaptive android obfuscation detector. Future Generation Computer Systems **90**, 240–261 (2019)
103. Mohammadinodooshan, A., Kargén, U., Shahmehri, N.: Robust detection of obfuscated strings in android apps. In: Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security. pp. 25–35 (2019)
104. Mohanty, D.: Anti-virus evasion techniques and countermeasures. Published online at http://www. hackingspirits. com/eth-hac/papers/whitepapers. asp. Last accessed on **18** (2005)
105. Moog, M., Demmel, M., Backes, M., Fass, A.: Statically detecting javascript obfuscation and minification techniques in the wild. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 569–580. IEEE (2021)
106. Mowla, N.I., Tran, N.H., Doh, I., Chae, K.: Federated learning-based cognitive detection of jamming attack in flying ad-hoc network. IEEE Access **8**, 4338–4350 (2020). https://doi.org/10.1109/ACCESS.2019.2962873, https://doi.org/10.1109/ACCESS.2019.2962873
107. Mozaffari, M., Saad, W., Bennis, M., Nam, Y., Debbah, M.: A tutorial on uavs for wireless networks: Applications, challenges, and open problems. IEEE Commun. Surv. Tutorials **21**(3), 2334–2360 (2019). https://doi.org/10.1109/COMST.2019.2902862, https://doi.org/10.1109/COMST.2019.2902862
108. Mpitziopoulos, A., Gavalas, D., Pantziou, G.E., Konstantopoulos, C.: Defending wireless sensor networks from jamming attacks. In: Proceedings of the IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC 2007, 3-7 September 2007, Athens, Greece. pp. 1–5. IEEE (2007). https://doi.org/10.1109/PIMRC.2007.4394775, https://doi.org/10.1109/PIMRC.2007.4394775

109. Muench, M., Nisi, D., Francillon, A., Balzarotti, D.: Avatar 2: A multi-target orchestration platform. In: Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.). vol. 18, pp. 1–11 (2018)
110. Navda, V., Bohra, A., Ganguly, S., Rubenstein, D.: Using channel hopping to increase 802.11 resilience to jamming attacks. In: INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA. pp. 2526–2530. IEEE (2007). https://doi.org/10.1109/INFCOM.2007.314, https://doi.org/10.1109/INFCOM.2007.314
111. Oliver, J., Cheng, C., Chen, Y.: Tlsh–a locality sensitive hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop. pp. 7–13. IEEE (2013)
112. Ozay, M., Esnaola, I., Vural, F.T.Y., Kulkarni, S.R., Poor, H.V.: Machine learning methods for attack detection in the smart grid. IEEE transactions on neural networks and learning systems **27**(8), 1773–1786 (2015)
113. Pal, S.K., Mitra, S.: Multilayer perceptron, fuzzy sets, and classification. IEEE Transactions on Neural Networks **3**(5), 683–697 (1992). https://doi.org/10.1109/72.159058
114. Papadimitratos, P., Haas, Z.: Secure routing for mobile ad hoc networks. In: Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002). No. CONF, SCS (2002)
115. Park, M., You, G., Cho, S.j., Park, M., Han, S.: A framework for identifying obfuscation techniques applied to android apps using machine learning. J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl. **10**(4), 22–30 (2019)
116. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in python. CoRR **abs/1201.0490** (2012), http://arxiv.org/abs/1201.0490
117. Pei, K., Xuan, Z., Yang, J., Jana, S., Ray, B.: Trex: Learning execution semantics from micro-traces for binary similarity. arXiv preprint arXiv:2012.08680 (2020)

118. Pelechrinis, K., Iliofotou, M., Krishnamurthy, S.V.: Denial of service attacks in wireless networks: The case of jammers. IEEE Commun. Surv. Tutorials **13**(2), 245–257 (2011). https://doi.org/10.1109/SURV.2011.041110.00022, https://doi.org/10.1109/SURV.2011.041110.00022

119. Pellegrino, G., Johns, M., Koch, S., Backes, M., Rossow, C.: Deemon: Detecting csrf with dynamic analysis and property graphs. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017). https://doi.org/10.1145/3133956.3133959

120. Perriot, F.: Defeating polymorphism through code optimization. In: Proc. of the 2003 Virus Bulletin Conference (VB2003).(September 2003). pp. 1–18 (2003)

121. Phu, T.N., Hoang, L.H., Toan, N.N., Tho, N.D., Binh, N.N.: Cfdvex: A novel feature extraction method for detecting cross-architecture iot malware. In: Proceedings of the 10th International Symposium on Information and Communication Technology. pp. 248–254 (2019)

122. Pucher, M., Kudera, C., Merzdovnik, G.: Detecting obfuscated function clones in binaries using machine learning (2022)

123. Puñal, O., Aguiar, A., Gross, J.: In vanets we trust?: characterizing RF jamming in vehicular networks. In: Kenney, J.B., Gozálvez, J., Bai, F., Kravets, R. (eds.) Proceedings of the ninth ACM international workshop on Vehicular inter-networking, systems, and applications, VANET '12, Low Wood Bay, Lake District, UK, June 25, 2012. pp. 83–92. ACM (2012). https://doi.org/10.1145/2307888.2307903, https://doi.org/10.1145/2307888.2307903

124. Puñal, O., Pereira, C., Aguiar, A., Gross, J.: CRAWDAD dataset uportorwthaachen/vanetjamming2012 (v. 2014-05-12). Downloaded from https://crawdad.org/uportorwthaachen/vanetjamming2012/20140512 (May 2014). https://doi.org/10.15783/C7TS35

125. Rajaat: Polimorphism. 29A Magazine **1**(3) (1999)

126. Ravipati, G., Bernat, A.R., Rosenblum, N., Miller, B.P., Hollingsworth, J.K.: Toward the deconstruction of dyninst. Univ. of Wisconsin, technical report **32** (2007)

127. Ribeiro, M.T., Singh, S., Guestrin, C.: " why should i trust you?" explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. pp. 1135–1144 (2016)
128. Ribeiro, M.T., Singh, S., Guestrin, C.: Anchors: High-precision model-agnostic explanations. In: Proceedings of the AAAI conference on artificial intelligence. vol. 32 (2018)
129. Rocchetto, M., Ochoa, M., Torabi Dashti, M.: Model-based detection of csrf. In: ICT Systems Security and Privacy Protection: 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, June 2-4, 2014. Proceedings 29. pp. 30–43. Springer (2014)
130. Rohrer, J.P., Çetinkaya, E.K., Narra, H., Broyles, D., Peters, K., Sterbenz, J.P.G.: Aerorp performance in highly-dynamic airborne networks using 3d gauss-markov mobility model. In: MILCOM 2011 - 2011 IEEE Military Communications Conference, Baltimore, MD, USA, November 7-10, 2011. pp. 834–841. IEEE (2011). https://doi.org/10.1109/MILCOM.2011.6127781, https://doi.org/10.1109/MILCOM.2011.6127781
131. Roussev, V.: Data fingerprinting with similarity digests. In: Advances in Digital Forensics VI: Sixth IFIP WG 11.9 International Conference on Digital Forensics, Hong Kong, China, January 4-6, 2010, Revised Selected Papers 6. pp. 207–226. Springer (2010)
132. Safavian, S.R., Landgrebe, D.: A survey of decision tree classifier methodology. IEEE transactions on systems, man, and cybernetics **21**(3), 660–674 (1991)
133. Sahingoz, O.K.: Networking models in flying ad-hoc networks (fanets): Concepts and challenges. J. Intell. Robotic Syst. **74**(1-2), 513–527 (2014). https://doi.org/10.1007/s10846-013-9959-7, https://doi.org/10.1007/s10846-013-9959-7
134. Salehi, M., Degani, L., Roveri, M., Hughes, D., Crispo, B.: Discovery and identification of memory corruption vulnerabilities on bare-metal embedded devices. IEEE Transactions on Dependable and Secure Computing **20**(2), 1124–1138 (2022)
135. Salem, A., Banescu, S.: Metadata recovery from obfuscated programs using machine learning. In: Proceedings of the 6th Workshop on Soft-

ware Security, Protection, and Reverse Engineering. SSPREW '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/3015135.3015136

136. Saxe, J., Berlin, K.: Deep neural network based malware detection using two dimensional binary program features. In: 2015 10th international conference on malicious and unwanted software (MALWARE). pp. 11–20. IEEE (2015)
137. Schiffman, M.: A brief history of malware obfuscation: Part 1 of 2. Published online at `https://blogs.cisco.com/security/a\_brief\_history\_of\_malware\_obfuscation\_part\_1\_of\_2`, accessed: 2018-11-13
138. Schiffman, M.: A brief history of malware obfuscation: Part 2 of 2. Published online at `https://blogs.cisco.com/security/a\_brief\_history\_of\_malware\_obfuscation\_part\_2\_of\_2`, accessed: 2018-11-13
139. Scott, K., Davidson, J.: Strata: A software dynamic translation infrastructure. In: IEEE Workshop on Binary Translation (2001)
140. Selvaraju, R.R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., Batra, D.: Grad-cam: Visual explanations from deep networks via gradient-based localization. In: Proceedings of the IEEE international conference on computer vision. pp. 618–626 (2017)
141. Shahid, M.: Machine learning for detection and mitigation of web vulnerabilities and web attacks. arXiv preprint arXiv:2304.14451 (2023)
142. Shahriar, H., Zulkernine, M.: Client-side detection of cross-site request forgery attacks. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering. pp. 358–367 (2010). https://doi.org/10.1109/ISSRE.2010.12
143. Shahriar, H., Zulkernine, M.: Client-side detection of cross-site request forgery attacks. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering. pp. 358–367. IEEE (2010)
144. Shakhatreh, H., Sawalmeh, A.H., Al-Fuqaha, A., Dou, Z., Almaita, E., Khalil, I., Othman, N.S., Khreishah, A., Guizani, M.: Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges. IEEE Access **7**, 48572–48634 (2019). https://doi.org/10.1109/ACCESS.2019.2909530
145. Shapley, L.S.: A Value for n-Person Games. Princeton University Press (1953)
146. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The

Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)

147. Sihag, V., Vardhan, M., Singh, P.: Blade: robust malware detection against obfuscation in android. Forensic Science International: Digital Investigation **38**, 301176 (2021)
148. Smith, A.J., Mills, R.F., Bryant, A.R., Peterson, G.L., Grimaila, M.R.: Redir: Automated static detection of obfuscated anti-debugging techniques. In: 2014 International Conference on Collaboration Technologies and Systems (CTS). pp. 173–180. IEEE (2014)
149. Spensky, C., Machiry, A., Redini, N., Unger, C., Foster, G., Blasband, E., Okhravi, H., Kruegel, C., Vigna, G.: Conware: Automated modeling of hardware peripherals. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. pp. 95–109 (2021)
150. Sudhodanan, A., Carbone, R., Compagna, L., Dolgin, N., Armando, A., Morelli, U.: Large-scale analysis & detection of authentication cross-site request forgeries. In: 2017 IEEE European symposium on security and privacy (EuroS&P). pp. 350–365. IEEE (2017)
151. Szor, P., Ferrie, P.: Hunting for metamorphic. In: Virus bulletin conference. Prague (2001)
152. Talebi, S.M.S., Tavakoli, H., Zhang, H., Zhang, Z., Sani, A.A., Qian, Z.: Charm: Facilitating dynamic analysis of device drivers of mobile systems. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 291–307 (2018)
153. Tauber, M., Bhatti, S.N.: Low RSSI in wlans: Impact on application-level performance. In: International Conference on Computing, Networking and Communications, ICNC 2013, San Diego, CA, USA, January 28-31, 2013. pp. 123–129. IEEE Computer Society (2013). https://doi.org/10.1109/ICCNC.2013.6504066, `https://doi.org/10.1109/ICCNC.2013.6504066`
154. Tofighi-Shirazi, R., Asăvoae, I.M., Elbaz-Vincent, P.: Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning. In: Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering. SSPREW9 '19, Association for Computing Machin-

ery, New York, NY, USA (2019). https://doi.org/10.1145/3371307.3371313, https://doi.org/10.1145/3371307.3371313

155. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: Diablo: a reliable, retargetable and extensible link-time rewriting framework. In: Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005. pp. 7–12. IEEE (2005)
156. Vishnevskiy, V.M., Samouylov, K.E., Kozyrev, D.V. (eds.): Distributed Computer and Communication Networks - 23rd International Conference, DCCN 2020, Moscow, Russia, September 14-18, 2020, Revised Selected Papers, Lecture Notes in Computer Science, vol. 12563. Springer (2020). https://doi.org/10.1007/978-3-030-66471-8, https://doi.org/10.1007/978-3-030-66471-8
157. Wall, D.W.: Systems for late code modification. In: Code Generation—Concepts, Tools, Techniques, pp. 275–293. Springer (1992)
158. Wang, C., Davidson, J., Hill, J., Knight, J.: Protection of software-based survivability mechanisms. In: 2001 International Conference on Dependable Systems and Networks. pp. 193–202. IEEE (2001)
159. Wang, Y., Rountev, A.: Who changed you? obfuscator identification for android. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). pp. 154–164. IEEE (2017)
160. Wenzl, M., Merzdovnik, G., Ullrich, J., Weippl, E.: From hack to elaborate technique—a survey on binary rewriting. ACM Computing Surveys (CSUR) **52**(3), 1–37 (2019)
161. Wermke, D., Huaman, N., Acar, Y., Reaves, B., Traynor, P., Fahl, S.: A large scale investigation of obfuscation use in google play. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 222–235 (2018)
162. Wikipedia contributors: Direct-sequence spread spectrum — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Direct-sequence_spread_spectrum&oldid=954223274 (2020), [Online; accessed 9-June-2020]
163. Wong, W., Stamp, M.: Hunting for metamorphic engines. Journal in Computer Virology **2**(3), 211–229 (2006)

164. Xiao, L., Chen, T., Liu, J., Dai, H.: Anti-jamming transmission stackelberg game with observation errors. IEEE Commun. Lett. **19**(6), 949–952 (2015). https://doi.org/10.1109/LCOMM.2015.2418776, https://doi.org/10.1109/LCOMM.2015.2418776

165. Xiao, L., Lu, X., Xu, D., Tang, Y., Wang, L., Zhuang, W.: UAV relay in vanets against smart jamming with reinforcement learning. IEEE Trans. Veh. Technol. **67**(5), 4087–4097 (2018). https://doi.org/10.1109/TVT.2018.2789466, https://doi.org/10.1109/TVT.2018.2789466

166. Xu, K.M., Zhang, M., Eitzen, Z.A., Ghan, S.J., Klein, S.A., Wu, X., Xie, S., Branson, M., Del Genio, A.D., Iacobellis, S.F., et al.: Modeling springtime shallow frontal clouds with cloud-resolving and single-column models. Journal of Geophysical Research: Atmospheres **110**(D15) (2005)

167. Xu, W., Ma, K., Trappe, W., Zhang, Y.: Jamming sensor networks: attack and defense strategies. IEEE Netw. **20**(3), 41–47 (2006). https://doi.org/10.1109/MNET.2006.1637931, https://doi.org/10.1109/MNET.2006.1637931

168. Xu, W., Trappe, W., Zhang, Y., Wood, T.: The feasibility of launching and detecting jamming attacks in wireless networks. In: Kumar, P.R., Campbell, A.T., Wattenhofer, R. (eds.) Proceedings of the 6th ACM Interational Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc 2005, Urbana-Champaign, IL, USA, May 25-27, 2005. pp. 46–57. ACM (2005). https://doi.org/10.1145/1062689.1062697, https://doi.org/10.1145/1062689.1062697

169. You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on. IEEE (2010)

170. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al.: Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: NDSS. vol. 14, pp. 1–16 (2014)

171. Zhao, B., Han, J., Meng, X.: A malware detection system based on intermediate language. In: 2017 4th International Conference on Systems and Informatics (ICSAI). pp. 824–830. IEEE (2017)

172. Zhou, W., Guan, L., Liu, P., Zhang, Y.: Automatic firmware emulation through invalidity-guided knowledge inference. In: USENIX Security Symposium. pp. 2007–2024 (2021)