# Backdoor Detection Systems for Embedded Devices

by

Sam Lloyd Thomas

A thesis submitted to
University of Birmingham
for the degree of
Doctor of Philosophy

School of Computer Science
College of Engineering & Physical Sciences
University of Birmingham
April 2018

# *Abstract*

A system is said to contain a backdoor when it intentionally includes a means to trigger the execution of functionality that serves to subvert its expected security. Unfortunately, such constructs are pervasive in software and systems today, particularly in the firmware of commodity embedded systems and "Internet of Things" devices.

The work presented in this thesis concerns itself with the problem of detecting backdoor-like constructs, specifically those present in embedded device firmware, which, as we show, presents additional challenges in devising detection methodologies. The term "backdoor", while used throughout the academic literature, by industry, and in the media, lacks a rigorous definition, which exacerbates the challenges in their detection. To this end, we present such a definition, as well as a framework, which serves as a basis for their discovery, devising new detection techniques and evaluating the current state-of-the-art.

Further, we present two backdoor detection methodologies, as well as corresponding tools which implement those approaches. Both of these methods serve to automate many of the currently manual aspects of backdoor identification and discovery. And, in both cases, we demonstrate that our approaches are capable of analysing device firmware at scale and can be used to discover previously undocumented real-world backdoors.

# *Acknowledgements*

Numerous people have contributed in one way or another to my "PhD experience", and the words written here do not reflect the true thanks owed to each of them.

I am grateful to those that kindly gave up their time to assess this manuscript: Cristiano Giuffrida and David Oswald. As well as Dave Parker and Eike Ritter, who took the time throughout my studies to provide guidance and a review of my progress.

I, of course, would not be in a position to write these words without the opportunities and support provided to me by my PhD supervisors, Tom Chothia and Flavio Garcia, who afforded me as much freedom as my thesis topic could permit – thank you. My thanks also go to Christophe Petit, who was a pleasure to work alongside.

My sincere thanks go to Aurélien Francillon, who was not only an excellent person to collaborate with, but someone I would consider a friend. The short time I spent visiting EURECOM working with you, and meeting your group affirmed my decision to remain in academia.

I would like to thank the many people I have met during my time at Birmingham, who have provided support in many forms: Chris Hicks, Jack Hargreaves, Andreea Radu, Chris McMahon-Stone, Jan van den Herrewegen, Keyhan Kouhkiloui, Tim Booth, and George Carpenter. In particular, I would like to thank Richard Thomas, who as well as being a reliable friend, donated large amounts of his time to read through the seemingly endless draft versions of this manuscript.

I would like to thank my family, in particular, my parents, Jeanette and Ian, and my brother, Joe, for providing love, support, and the encouragement I needed to follow my dreams. Finally, I would like to thank my best friend and partner, Lucy, for always being there for me, and for all the love I could wish for.

# *Contents*

# Chapter 1

## *Introduction*

Embedded devices are pervasive in almost every aspect of our existence. They are in our pockets, around our wrists, and are rapidly becoming a vital part of our daily lives – an extension of ourselves. Yet, rarely does the layman consider what actually makes these devices tick, the underlying software, the underlying hardware: which are both designed by processes that can be influenced by ruthlessness, greed and malice. To the Information Security community, however, these devices are a terrifying prospect. What we would have considered embedded devices in the early 2000's, such as the microcontroller in our dishwasher, or the board responsible for controlling our alarm clock, are now being redeveloped as part of what has been coined the "Internet of Things" (IoT) – a blanket term referring to so-called "smart-devices" that are connected to the public Internet. Previously, these everyday devices never needed to be considered as part of a network security policy, or potential attack surface. An intentional bug in such a device would serve no advantage to its manufacturer: they couldn't use the device to monitor us, record our conversations, or leak our personal information. Now, however, full trust is given to the developers of these smart-devices – that have the potential to do just that.

Unfortunately, this (often implicit) trust is misplaced; security researchers have exposed instances – through a number of high profile cases – of both powerful adversaries [188, 189] and consumer device manufacturers [102, 103] deliberately inserting flaws into software running on embedded devices. While some might argue that these flaws are inserted to protect us: giving state actors the ability to disrupt the heinous crimes that take place online, anyone with knowledge of such flaws can exploit them, which is how malware families such as Mirai (see, *e.g.,* [39]) – a botnet constructed from compromised IoT devices – come into existence.

Once backdoors are reported, many manufacturers "brush them off" as accidents (*e.g.,* [45]) or as so-called left-over *debug* functionality. Some see backdoor-like behaviour in software as useful to implement features – such as unattended device upgrades (*e.g.,* [103]). In other cases, device manufacturers are not entirely to blame: developers share code and use third-party libraries or software, which they themselves trust implicitly. And again, that trust is often misplaced. A backdoor coined TCP-32764 [36] – due to the port it listens on – compromised routers produced by many well-known device vendors, including, Cisco, Netgear and Linksys. The backdoor was introduced into their firmware by means of a code fragment shared amongst the affected devices, which originated from SerComm, a Taiwanese device manufacturer. The Chinese device manufacturer Huawei was similarly affected by a backdoor that compromised many of its Android-based devices, by use of a third-party application: Ad-Ups [123]. These problems manifest in part due to inadequate oversight, but mainly due to the fact that reverse engineering third-party libraries and software to check for flaws and backdoors is expensive and requires significant expertise. Moreover, these analysis processes – especially those to locate backdoors – are difficult to automate.

The term "backdoor" is used as a buzzword in the media – a sensationalist term to imply a flaw deliberately left within a system. Though it is not just the media that use the term in such a way: in both industry and academia, the term is used just as loosely.

We currently lack a rigorous definition as to what a backdoor is and an understanding of what common components one is made up of. This is in stark contrast to other types of malware and program flaws: which for the most part, have precise definitions as to how they manifest, and what characteristics we expect them to exhibit. This lack of specificity, coupled with the broad array of implementation possibilities for backdoors, makes it incredibly challenging to develop generalised tools and techniques for their detection. This exacerbates the problem of backdoors that are easy for attackers to find and exploit, as, without detection tools, those backdoors that manifest simply due to poor coding practices – such as hard-coded credential checks – are often viewed as easy, viable implementation strategies by developers. Further hampering the effort to detect such backdoors – especially those of a more complex or esoteric nature – is the sheer lack of real-world samples. Documented real-world backdoors are generally simplistic, where their trigger, or activation conditions rely upon an adversary inputting certain static data – the most obvious example being hard-coded credentials.

In order for a third-party analyst to assess the security of a particular embedded device, they require access to its firmware – which can range from a single monolithic piece of software, to a multi-component system containing an operating system, filesystem and associated system software. While many consumer device manufacturers provide firmware update files, which contain partial copies of such software, in some cases, these devices are not updateable, or employ proprietary automated update mechanisms, such as those often found in Internet Service Provider (ISP) supplied routers. In these cases, firmware must be obtained directly from the hardware: making the analysis process more challenging and expensive, as specialised tools and significantly more expertise are required.

## 1.1 Contributions

In the following sections, we give an overview of the contributions of this thesis.

### 1.1.1 Definitions & Fundamentals for Backdoor Detection

In this thesis, we provide, first and foremost, a much needed rigorous definition of the term backdoor, as well as an in-depth analysis and discussion of developer intention, deniability, and accountability for manufacturers that deliberately produce devices containing software with backdoors present. To do this, we propose a framework which allows us to decompose the abstract notion of a backdoor into its component parts, that is, the minimal set of components that it needs to function and exhibit backdoor-like behaviour. While our framework allows us to decompose constructs that are known to contain backdoor-like functionality effectively, alone, it does not provide a way of reasoning about developer intent; to overcome this, we provide a number of definitions to discern types of backdoor implementations: those that are intentional, those that are deniable and those that are accidental. To demonstrate the completeness of our framework, we provide a number of case-studies in which we explore concrete implementations of real-world backdoors; within these case-studies, we decompose each backdoor, and reason about its deniability and how it can be detected. Our framework is not only useful at aiding in identifying and reasoning about backdoor-like constructs, but also for developing and analysing backdoor detection methodologies. To this end, we provide an evaluation of four state-of-the-art backdoor detection approaches, and show none fully consider a complete model of what a backdoor is, and thus, their respective approaches are limited.

### 1.1.2 Detection of Unexpected & Undocumented Functionality

Many complex embedded devices, such as routers, and IoT devices, such as IP cameras and DVR systems, are built on top of simplified versions of Linux. Thus, they share many common services such as web-servers, Telnet-daemons, and so on. Web-servers are usually used to present the configuration interface for a device, while Telnet daemons are often present – sometimes in a usable state – as an artefact left over from development.

To reduce the effort of manually analysing firmware, we present a semi-automated approach to identify unexpected, undocumented functionality in such services. We base our approach on the following observation: in addition to being present in a large number of device firmware images, many common services tend to have well-known, well-defined behaviour. Through extensive manual analysis of instances of services that are present in Linux-based embedded devices, we were able to identify the services that are prevalent amongst a significant portion of devices, as well as their key functionality traits and behaviours. Using this information, to identify unexpected or undocumented functionality in a given service, we propose a two-stage process. In the first stage, we identify the type, or class of the service, and in the second, we check if the service exhibits the functionality we expect of the class it was identified as – which ultimately tells us if it contains unexpected or undocumented functionality. To perform the first stage of analysis, we use a classifier constructed using semi-supervised learning, which given an input binary program, is able to identify if it is a common service, and if so, provide a label for the type of service it is. Then, to perform the second stage, we leverage its assigned label, and the fact that all identifiable services have well defined behaviour by checking if the binary only exhibits the behaviour we expect of services associated with the label. To model our expectations of behaviour and functionality, we use a rule-based system. At the core of this system is our domain-specific language – Binary Functionality Description Language – which is able encode expected program behaviour in a high-level manner in what we call an *expected functionality profile*. We construct such a profile for each class of service our classifier is able to detect. Thus, in order to check if the program exhibits any unexpected functionality, we evaluate it against the previously defined expected functionality profile corresponding to its identified service type, and if it does not conform to that profile, we consider it to contain undocumented or unexpected functionality, and perform manual analysis to ascertain the nature of that functionality. To evaluate our approach and demonstrate its effectiveness, we use case-studies of both new backdoors we have found using our method, and existing backdoors that have been previously identified (by others) using manual analysis.

### 1.1.3 Detection of Undocumented Commands & Credentials

While detecting the means by which a service might diverge from its expected behaviour is effective in identifying backdoors and undocumented functionality when that expected behaviour is known *a priori*, it is not useful in identifying such functionality when this is not the case. To this end, we construct a system – again, with the intent of reducing the effort of manual analysis to detect the conditions required to trigger backdoor-like behaviour – which serves as a complementary method to our previously described technique. As a basis for our approach, we observe that backdoor-like constructs in programs that are guarded by static-data comparisons – such as hard-coded credential checks, or undocumented commands – are generally only reachable by successful comparison with the static data involved, and are not accessible by any other means. To automate the detection of backdoor "triggers" used in this way, we use a two-stage process. In the first stage, we automatically identify potential static data comparison functions, *e.g.,* functions such as `strcmp`, the C++ string equality operator and custom functions that implement related functionality. In the second stage, we analyse the call-sites of those functions, extract the static data used as arguments and assign that data a score. We attempt to maximise the scores assigned to static data that when compared successfully against, results in the execution of code paths that are not reachable without that successful comparison, *i.e.,* they are uniquely reachable via that comparison. These individual scores serve as a basis for assigning scores to functions, which we subsequently use to perform an ordering of all of the functions of a binary, relative to how much of their control-flow is influenced by comparisons with static data, and how much those comparisons impact the reachability of their functionality. Using this ordering, we generate a report, which lists functions and static data by their importance; this serves as a starting point for performing targeted manual analysis of functions that may contain hidden, undocumented commands, or hard-coded credential checks. We implement our techniques in a tool – Stringer and with it, we demonstrate how our approach is able to identify new backdoors in firmware from a number of devices, as well as detect previously (manually) identified backdoors.

## 1.2 Thesis Overview & Structure

We structure the thesis as follows:

**Chapter 2** In this chapter, we introduce the required technical background on analysis of embedded device firmware. We also introduce terms, definitions, and the required background in program analysis and machine learning needed for an understanding of the content presented in later chapters.

**Chapter 3** Building on the material presented in Chapter 2, in this chapter, we provide a study of related work and the current state-of-the-art. In particular, we provide a discussion of the key challenges surrounding analysis of embedded device firmware and backdoor detection, as well as an analysis of current backdoor detection methodologies and implementation approaches. More generally, we cover related work in program analysis and malware detection, which we see as sister fields to that of this work.

**Chapter 4** To remedy the deficiencies in current research outlined in Chapter 3, in this chapter, we provide a first rigorous definition of what a backdoor is and the process of backdoor detection. We provide a decomposition of an abstract backdoor's *anatomy* into a framework, which serves as a basis for identifying backdoor-like constructs and reasoning about them, as well as both a premise for devising new backdoor detection methodologies, and as a tool for evaluating current approaches. We additionally discuss different types of backdoors, their deniability and how they can be detected, which we demonstrate through a number of case-studies of both academic and real-world backdoor implementations. Finally, we provide an analysis of state-of-the-art backdoor detection methods, and in doing so, we show that none consider a complete model of what a backdoor is, and as a result, their effectiveness is limited. This chapter is based on the following publication:

*Backdoors: Definition, Deniability & Detection* by the author and Aurélien Francillon [175], presented at RAID'18.

**Chapter 5** In this chapter, we describe how the similarities in Linux-based embedded device firmware can be exploited in order to detect backdoors and undocumented functionality in services that are commonly shared amongst such firmware. We describe the implementation of our system, HumIDIFy, which enables semi-automated analysis of Linux-based firmware at scale. We demonstrate the effectiveness of our approach through a number of case-studies of artificial and real-world backdoors that we have detected using HumIDIFy. This chapter is based on the following publication:

*HumIDIFy: A Tool for Hidden Functionality Detection in Firmware* by the author, Flavio D. Garcia, and Tom Chothia [176], presented at DIMVA'17.

**Chapter 6** As a complementary approach to that described in Chapter 5, in this chapter, we demonstrate how backdoors can be effectively identified by identification of their "trigger" conditions. This chapter details how by identifying key static data comparisons – whose successful comparison leads to execution of uniquely reachable code sequences, backdoor triggers can be identified automatically, which substantially reduces the time taken to perform manual analysis of firmware binaries. We present an evaluation of our approach by use of our tool, Stringer, which we show – through a number of case-studies – is able to detect the backdoor trigger conditions of various real-world backdoors, some of which were previously undiscovered. The content of this chapter is based upon the following publication:

*Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality* by the author, Tom Chothia, and Flavio D. Garcia [174], presented at ESORICS'17.

**Chapter 7** To conclude the thesis, we provide a reflection on the work described in the previous chapters, draw conclusions and examine directions for future research.

# Part I

*Background, Preliminaries & Related Work*

# Chapter 2

*Background & Preliminaries*

## Contents

In this thesis, we concern ourselves with detecting backdoors, specifically those found within embedded device firmware. This chapter provides a number of things: the required background for understanding the composition of embedded device firmware, a high-level overview of the methodologies for analysing that firmware, and an explanation of terms and definitions used throughout the literature and this thesis for specifying and evaluating those methodologies.

## 2.1 Complex Embedded Devices

While complex embedded devices such as routers and Internet of Things (IoT) devices come in many forms, at the software-level, they share some commonalities; for example, the vast majority utilise variants of the Linux operating system. Similarly, at the hardware level, while sensors and other components may differ, the underlying CPU architectures of the devices fall into two major categories: ARM and MIPS (which reportedly cover 63% and 7% of the total market share [76]). The software component of these devices is referred to as firmware, which, for complex devices, is generally a collection of components: a bootloader, an operating system, compiled software, configuration files, and scripts.

## 2.2 Firmware Acquisition

To perform analysis of firmware, it must first be acquired, which can often be as simple as downloading it from a device vendor's website. In other cases, it can be as complex as directly lifting it from the hardware of a device. Both situations present difficulties, as noted in the literature [67, 76, 77].

For targeted analysis of a particular device, obtaining its firmware manually, for example, by downloading it from a vendor's website is relatively straightforward. However, for large-scale analysis of firmware, such an approach is obviously infeasible. Both the literature (*e.g.,* [67, 76]), and the work presented here (Chapters 5 and 6), have thus, found it desirable to automate this acquisition process. One approach to do this is through the use of a purpose-built web-crawler. To be effective, such a crawler needs to be adapted for each manufacturer's website, and employ appropriate heuristics to correctly identify links to firmware images, to avoid downloading unrelated files. While alleviating manual effort, obtaining firmware in such a manner has a number of disadvantages. For instance, many firmware images distributed online are packed using esoteric or proprietary algorithms, and therefore, cannot be unpacked automatically using off-the-shelf methods or tools. For those that do unpack using automated methods, a large majority of firmware images are often incomplete – either due to flaws in the process used to unpack them, or because they have been distributed as partial firmware updates, and thus, miss executables and configuration files, crucial for analysis.

The standard tool for unpacking firmware images is binwalk [6], which treats a firmware image as a collection of files concatenated into a single binary blob. To extract those files, it first scans the binary blob for so-called magic numbers, or file format identifiers. Upon encountering such an identifier, it records its offset, and then attempts to parse the data in the binary blob at that offset as a file-format header related to the identifier. Using the file-format header, it calculates the length of the embedded file, and proceeds to *carve* it from the binary blob. If the file has been packed or compressed using a known algorithm (*e.g.,* LZMA [34]), a standard utility can be invoked to unpack it; if successful, this procedure will produce further files, which may also require unpacking. To handle such a scenario, binwalk performs its extraction processes in a recursive manner. If binwalk finds a file or embedded file is an image of a file-system, for example, a SquashFS [27] or cramfs [9] file-system, it will attempt to extract and reconstruct it. Unfortunately, the extraction process performed by binwalk is far from perfect, and in many cases manufacturers have been known to use modified standard compression and file-system formats – potentially to thwart analysis efforts. These non-standard formats often lead the standard utilities used by binwalk to fail. There have been efforts to overcome these issues, such as the tool sasquatch [26], which attempts to unpack file-systems that are variants of the common SquashFS format. However, in general, as binwalk relies on detecting *known* file-formats and using mostly standard unpacking tools and methods, it is ineffective when firmware is itself or composed of proprietary, or non-standard firmware file format(s).

While automating the extraction of firmware components from suspected firmware images is a significant challenge in itself, so too is the process of identifying if the extracted files actually constitute firmware components – as opposed to something else entirely. Since binwalk will only report if it has successfully been able to extract files, and that none of the invocations of its sub-utilities have failed, further analysis has to be performed on the extracted files and directories to perform that identification process. Both [67] and [77] utilise heuristics in order to perform this identification. They attempt to detect well-known file paths included in the standard Linux file-system layout, such as the `/bin` and `/etc` directories. However, as with the other processes involved in firmware acquisition, identification is not perfect. And from both our own experience, and that reported in the literature, not all firmware conforms to a standard structure. This is especially true of firmware upgrade files, where often the only files included in those firmware images are those that will be modified as part of the upgrade process. Unfortunately, without human intervention, the identification of firmware images must rely on heuristics, which will obviously miss edge-cases and must be continuously updated to keep up with changes in firmware, but are a necessary trade-off to facilitate large-scale analysis.

### 2.2.1 Firmware Acquisition via Hardware Interaction

An alternative to downloading firmware from a device manufacturer – which is required in cases where vendors restrict access to firmware updates, or perform automated updates of their devices using device-specific methods – is to extract the firmware from a device directly. Unlike conventional personal computers, software for embedded devices, *i.e.,* firmware, is usually stored in flash memory chips, which are generally soldered onto the board of a device. In order to access the contents of these chips, specialised tools and techniques are required.

For the vast majority of complex devices, *i.e.,* that have Linux-based firmware, access to flash chips (from the software-side) is provided by an abstraction layer called the

Memory Technology Device (MTD) sub-system [18]; this sub-system allows for generalised low-level file-system operations such as read, write and erase. A more conventional file-system provides further abstraction above this basic layer, which might, for example, be JFFS2 [182]. While many devices deploy a file-system directly on top of a MTD block device, another possibility is that an Unsorted Block Image (UBI) layer [30] is used in-between, which acts as a logical volume manager over the underlying MTD block device sub-system. A UBIFS [107] file-system is most commonly used on top of this abstraction layer, but other file-systems are possible, such as SquashFS [27]. Due to this variability, the process of obtaining firmware from a physical device often encounters the same problems as accessing file-system data from downloaded firmware images, and in many cases, binwalk, or manual approaches are required to access the file-system data.

Irrespective of the file-system used, or device configuration above the hardware, a number of (hardware-based) physical interfaces exist on devices that allow varying levels of access to firmware. In some cases, these can be as high-level as accessing a shell over a serial connection, in other cases, they can be as laborious as reading firmware block-by-block directly from the flash chip. The proceeding paragraphs cover the different interfaces and associated tools for accessing firmware using a hardware-based interface.

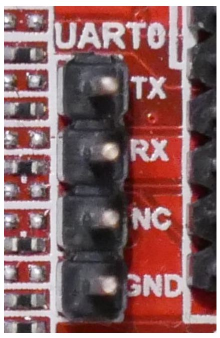


Figure 2.1: UART pinout.

Universal Asynchronous Receiver-Transmitter (UART) [140] is perhaps the simplest means of gaining access to a device's firmware. The UART interface of a device essentially acts as a serial port and allows interaction with the device via a terminal emulator such as screen [13] or minicom [20]. Physically, a UART interface manifests as a set of three, possibly four (including VCC, for power) pads or pins (Figure 2.1[1]), these are labelled as follows: TX (transmission line), RX (receiver line) and GND (ground). A device such as the Bus Pirate [7] can be used to access this interface by connecting to these pins. The serial interface can then be accessed using a conventional personal computer connected to the Bus Pirate. An attractive alternative to the Bus Pirate is the JTAGulator [15] (Figure 2.2), which can automatically infer the correct configuration required to interface with the TX and RX pins/pads, irrespective of how they are physically connected to the JTAGulator; it can then perform UART passthrough, which enables interaction with the device in the same manner as the Bus Pirate.

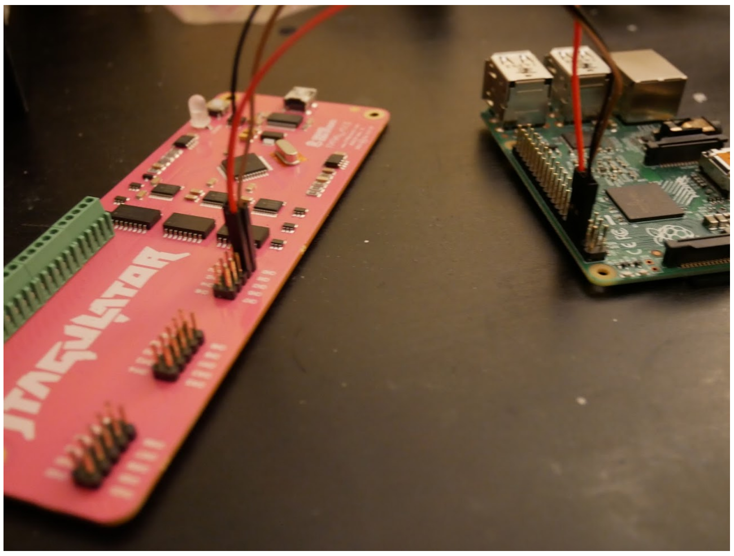


Figure 2.2: JTAGulator connected to UART pins.

Firmware can be obtained via UART in a number of ways. As discussed in [76], many devices have weak administrative passwords, or from our anecdotal experience, no passwords; thus, once connected via a serial terminal, it is possible to log-in as an administrative user *i.e.,* root. Firmware can then be dumped by directly copying from a

[1]The NC pin in Figure 2.1 stands for "No Connection" or "Not Connected".

MTD block device, or by transferring individual files using a protocol such as XMODEM [70], YMODEM [89], or ZMODEM [90], all of which provide varying degrees of integrity protection for the transfer.

Serial Peripheral Interface (SPI) bus is another interface that can be used to obtain firmware, though it is less reliable than UART, due to its lack of integrity protection capabilities. SPI is a synchronous serial communication protocol that allows data transfer between a Master Control Unit (MCU) and slave peripheral devices (*i.e.,* a flash chip). In a similar way to UART, the Bus Pirate can be used to interface with a device over SPI, which acts as a MCU. Communication is carried out over four pins located on the board of a device: MOSI (for master to slave transfer), MISO (for slave to master transfer), SS (the select signal, used by the MCU to specify which slave to communicate with), and SCLK (which allows for synchronisation of data sequences between the devices).

In some cases, even with direct access to the hardware of a device, obtaining its firmware can be extremely difficult. In these cases, a more holistic approach is often required. An example of this might be locating a vulnerable service running on a given device, exploiting it, and using the access obtained as a vector to facilitate dumping the device's firmware. For the work carried out in this thesis, we restrict ourselves to software-based approaches, which are significantly more effective for firmware analysis at scale.

## 2.3 Anatomy of Firmware

Once a firmware image has been obtained and unpacked (if required), the result will be in one of two forms: a single monolithic binary, or an operating system kernel, and associated file-system. In both cases, a bootloader will often be used to facilitate the loading of the initial software component(s) of the firmware.

**An operating system kernel and associated file-system:**

The bootloader loads the operating system kernel first, which is responsible for initialisation of drivers, which in turn, facilitate interaction with hardware, such as sensors. The initial userland processes will then be loaded, which will typically be stored in (and loaded from) the file-system. The file-system will store all (generally in a read-only manner) software, configuration files and scripts used to operate the device.

**A single monolithic binary:**

The software loaded by the bootloader will typically be bespoke and dedicated to performing only domain-specific functionality, as well as routines for interacting with hardware. As a result, performing generalised analysis on devices with this configuration is much more difficult than for the previous case, as the software on each different type of device will be radically different.

This thesis focuses on firmware that falls into the former category, specifically Linux-based firmware, due, in part, to its homogeneity, and because, at the time of writing, its use is prevalent in the majority of devices on the market (86% according to Costin et al. [76]). Thus, hereafter, when we refer to firmware, unless explicitly noted, we imply that which is Linux-based.

During firmware boot, the operating system kernel will initialise the hardware peripherals of a given device and then subsequently pass control to the initial userland processes. In turn, these processes will execute start-up scripts, which will finally start the processes responsible for the advertised (software-based) functionality of the device. Therefore, if access to a complete firmware image or corresponding device is not possible, analysing any available boot scripts – *if* they are available – can give key insight into what software may be running on a device. To this end, we performed extensive manual analysis of firmware file-systems and boot-processes; we found that from a small-scale sample of 92 firmware images obtained from manufacturer's websites, only 23 (25%) contained boot scripts.

```
mount -t proc proc /proc
mount -t ramfs ramfs /var
mkdir /var/tmp
mkdir /var/ppp/
mkdir /var/log
mkdir /var/run
mkdir /var/lock
mkdir /var/flash
#iwcontrol is required for RTL8185 Wireless driver
#iwcontrol auth &

#busybox insmod /lib/modules/2.4.26-uc0/kernel/drivers/usb/quickcam.o

/bin/webs -u root -d /www -i /var/run/thttpd.pid &
#ifconfig wlan0 up promisc
```

Figure 2.3: Example of a boot script taken from an IP camera.

Thus, performing a large-scale, meaningful analysis based on the presence of boot scripts would not produce usable results for the majority of firmware. In the firmware images that did contain boot scripts, we found that those boot scripts were located in a number of places: `/etc/inittab`, `/etc_ro/inittab`, `/etc/rc`, `/etc/rcS`, and those scripts often referenced additional supporting scripts located in the `/etc/init.d` and `/etc_ro/init.d` directories. Figure 2.3 shows an example of a boot script (found in `/etc/rcS`), taken from an IP camera. What should be noted, is that while a firmware image may contain many different executables, a large majority of them will never be executed, thus, knowing, or estimating those that will be started aids in reducing the time taken to analyse a complete firmware image.

### 2.3.1 Device Configuration

On many embedded devices, user-configurable information is often not stored within the file-system of the device; rather, it is stored within a region of flash memory called Non-Volatile Random Access Memory (NVRAM) – which retains its state between power cycles. Meanwhile, the operating system, bootloader and default file-system are generally stored within Read-Only Memory (ROM). Many devices treat NVRAM as a key-value store and include utilities such as `nvram-get` and `nvram-set` to get and set values stored there. On a router, for example, the current Wi-Fi passphrase, or administrative interface credentials, might be stored within the NVRAM, which will be queried by software to facilitate authentication to the device and its services.

All other device configuration, without performing a firmware upgrade, will be static. As a result of this, any, *e.g.,* hard-coded passwords or certificates, can be leveraged by an adversary to compromise a device. A further problem, as noted in the literature [76], is the use of user accounts with passwords stored in plain-text, weak passwords or, no password at all. In these cases, should a shell service, such as Telnet or the Secure Shell daemon be running on a device, where that service is reachable on an unfirewalled, Internet-facing interface, that device can be accessed, and compromised by a remote adversary with very little effort. In this way, the combination of a shell service coupled with hard-coded, weak or no credentials, acts as a backdoor into a device. From experience, we note that while some firmware developers do not explicitly insert hard-coded user accounts into `/etc/passwd` and `/etc/shadow`, often, scripts started at boot utilise utilities such as `adduser` to create user accounts with such credentials. Thus, searching for such vulnerabilities is not as simple as examining the contents of *e.g.,* `/etc/passwd`.

### 2.3.2 Device Upgrade Mechanisms

Exacerbating the security threats previously discussed is the fact that, while a portion of devices feature either user-controlled or automated upgrade mechanisms, a portion of devices do not feature any upgrade mechanism at all (commonly those where their firmware must be extracted from the device itself). Thus, any security vulnerabilities that cause a device to compromise the network it is attached to, cannot be addressed by any other means than detaching the device from that network, often rendering the device useless to the end-user. Moreover, while a small portion of technologically proficient device owners might diligently patch their devices manually, a much larger proportion of users will not. Thus, even devices that could be patched (assuming firmware is available that addresses the security concerns) will be left largely vulnerable.

## 2.4 (Firmware-Oriented) Program Analysis

As previously noted, Linux-based firmware contains executables, which interact – together with hardware – to provide a device with its advertised capabilities. To perform any type of analysis of these executables, their program instructions must first be recovered from a so-called container format, which, in the case of Linux-based firmware, is the Executable Linking Format (ELF) [33]. This container format contains segments, and instructions for how those segments should be loaded into memory; this might be, for example, the pre-initialised data to be mapped into memory starting from a particular address, or the program instructions, and where they should be mapped into memory. This format additionally contains the entry point of the program *i.e.,* the address that execution starts at. A program's instructions are recovered by the process of *disassembly*, which is performed by a tool called a *disassembler*; the process of *disassembly* describes the translation of binary data into human-readable assembly language instructions.

As with many parts of firmware analysis, unfortunately, the process of disassembly is not perfect, and the quality of its results greatly impact the quality of any analysis performed upon the instructions and program structure it helps recover [48]. This is particularly an issue in the case of malware analysis and backdoor detection: for example, if areas of code that contain malicious behaviour are left undiscovered by the disassembly process, an analysis performed on that disassembly will be incomplete and therefore ineffective. Thus, selecting optimal tooling and libraries to recover instructions and related information is crucial; such tools and libraries are reviewed in detail in Chapter 3.

### 2.4.1 Program Representation

Prior to translation to machine-code, software written in a high-level language is constructed using functions, and a program is executed by means of these functions transferring control back-and-forth between each-other: *i.e.,* the processor's control-flow moves from instruction-to-instruction, and, thus, function-to-function. Recovering this high-level structure is useful for program analysis: on the one hand, it allows visualisation of the results of disassembly in a more comprehensible way, and, on the other, it enables the computation of program properties in a more tractable manner (*i.e.,* intraprocedurally rather interprocedurally). The following standard notational conventions (inspired by the literature and [117]) and related definitions are used to represent and express programs, and their constructs:

- A *basic block* is a maximal, non-empty, consecutive sequence of instructions, such that no instruction modifies local control-flow and there is a strict sequential transition between each such instruction. The last instruction of a block may (conditionally) alter control flow (*i.e.,* it is a conditional or unconditional branch instruction), and the first instruction shall be the target of some intra- or inter- (in the case of a call) procedural branch instruction.
- A *control flow graph* (CFG) is a directed (possibly cyclic) graph, expressed as $G = V \times E$, where each node $v \in V$ represents a *basic block*, and each edge $e \in E$ represents a program branch condition.
- A *function* $f$ is represented by a CFG with the first instruction of one of its basic blocks designated as the function entry point $f_{entry}$.
- A *program* $P$ is represented as a set of its functions.

Aside from notational conventions, a number of definitions and graph-theoretic properties are used when referring to CFGs and their basic blocks:

- For a *basic block* $b$, the *degree* of the block refers to the number of edges connected to the block. We use the notation $deg_{in}(b)$ to refer to the number of incoming edges into the block, and $deg_{out}(b)$ to refer to the number of outgoing edges from the block.
- A block contains instructions; we thus use the notation $b_{insns}$ to refer to those instructions. We use $b_{addr}$ to refer to the address of the first instruction of a block.
- Within a CFG, for a function $f$, a block $b_i$ is said to *dominate* a block $b_j$ if every path from the block containing $f_{entry}$ to $b_j$ contains $b_i$. A block is said to strictly dominate another block if $b_i \neq b_j$. The immediate dominator of a block $b$ is the block that strictly dominates $b$, where it does not strictly dominate another block that also strictly dominates $b$.
- A *dominator tree* for a CFG is a tree where each node is a block, the start node is the block containing $f_{entry}$, and the child nodes of a block are the blocks it immediately dominates.

As previously stated, processor instruction set architectures used in embedded devices are not consistent amongst all devices. This creates a problem when we wish to write generalised analysis passes over the CFGs recovered from the disassembly process – as a variant of that analysis must be written for each different architecture we wish to support. Further, each instruction set tends to have many instructions, which are often complex in nature and there is normally no one-to-one mapping between different instruction sets. Thus, to simplify analysis, an *intermediate language* (IL) representation can be used. An intermediate representation is itself essentially an instruction set that contains far fewer, more high-level instruction-like constructs than a typical processor instruction set. To obtain a representation of a processor's instruction set in an IL, a process called *lifting* is performed; the lifting process takes as input a processor-specific instruction and translates it to one or more intermediate language constructs, which generally model not only the localised behaviour of the instruction (*e.g.,* adding two registers and saving the result), but also the side-effects it may produce (*e.g.,* modifications to the internal processor flags).

### 2.4.2 Analysis Methods

Complex firmware contains software, scripts and configuration files. Two main methodologies exist to observe and analyse the effects of those components, and, both of which have various advantages and disadvantages, specifically with respect to analysis of embedded device firmware.

**Static Analysis** views a piece of software or script without state, often without consideration of an underlying execution environment. Analysis is performed by making predictions by looking at a so-called dead-code listing: *i.e.,* the instructions, or higher-level language constructs of a script. It makes assumptions about what code *will* be executed, and, thus, will often provide an overestimate of what will eventually be executed.

**Dynamic Analysis** considers the live execution state of a running program or interpreted script. Thus, analysis is performed upon what has *already* been executed up until the *currently* observed state of execution. In addition to the instructions or high-level constructs that have been or will be executed, dynamic analysis also provides a means to view the execution environment, as well as the overall impact of the program's execution on the system the program is being run from. For example, this might include environment variables modified, files written to, and network traffic generated. However, dynamic analysis restricts analysis to program paths executed under a particular program run: which may be an under-estimate of the overall functionality that could be executed. A further disadvantage of dynamic analysis is that inputs to a program are often required to *drive* execution, which, of course, limit its scalability significantly.

Even under ideal conditions, both methods have various advantages and disadvantages; however, for embedded device firmware, where we often have an incomplete firmware image to analyse, dynamic analysis is comparatively more difficult, and often not possible. Both [67, 77] demonstrate it is possible to use system emulation software such as QEMU [23] to emulate *enough* of a device's firmware to facilitate running some programs, but neither approach extends to full system emulation. Where access to a device is available, differing degrees of analysis are possible: we, of course, can perform static analysis on the software obtained from the firmware of the device, and, dynamic analysis, in this case, can generally be made possible via the use of so-called debug stubs, which are special purpose programs injected into a running system, which facilitate on-device debugging. Dynamic analysis can be performed via accessing on-board debug functionality through technologies such OpenOCD [21], which provides a software interface to on-board debug interfaces such as the JTAG [114] interface, or its two-wire variant, the SWD [3] interface.

## 2.5 Backdoors as a form of Malware

Malware is a term used to refer to software that, from an end user's perspective, performs malicious or unwanted functionality, often without the user's knowledge. The colloquial definition often used to describe a backdoor is a modification to a system, which allows an adversary to access that system in an unauthorised manner, often without the end user's knowledge. Thus, using these definitions, a backdoor is most definitely a form of malware. However, unlike the definitions for malware families such as viruses and worms, which have distinct properties – such as how they propagate and how they maintain a presence on a system – the definition of a backdoor is much more open. Therefore, it is much more challenging to create generalised methods for their detection: in part due to their open definition, but mainly due to the unbounded ways in which they can manifest. This lack of definition is addressed in Chapter 4, while Chapters 5 and 6 address the problem of backdoor detection.

### 2.5.1 A Taxonomy for (Firmware) Backdoors

Although backdoors can be implemented in numerous ways, the means by which they provide an adversary with some level of privileged access to a system generally fall into a small number of common categories; the following taxonomy provides an overview of the features of backdoors representative of the most common of those categories.

The first class we consider is that of so-called *authentication bypass vulnerabilities*, which is a blanket term used to refer to backdoors that make it possible for a would-be adversary to bypass the standard authentication mechanism of a program, or system. The privileged access obtained via the use of such a backdoor, in many cases, will be equivalent to that of a standard authenticated user, in others, the backdoor may provide a greater level of access, *e.g.,* through an interface reserved for backdoor-authenticated users. Concretely, such a vulnerability may manifest in a number of ways; the most prevalent classes of such implementations are those that rely on static, hard-coded data.

An authentication bypass vulnerability that relies on the use of static program properties facilitates backdoor access by giving an adversary who is aware of those properties (and how to exploit them) the ability to bypass standard authentication. From a practical point-of-view, an adversary exploiting such a backdoor is able to pass from a program state that is considered *unauthenticated* to one which is considered *authenticated*, by a computation relying on, or comparison with, static-data. On an embedded device, this static-data might be stored within a program, or the non-volatile storage of the device. The most common, and blatant type of backdoor covered by this definition, is a hard-coded credential check. In the simplest case of such a backdoor, user-input to the standard authentication routine will be compared against one or more static strings; if these comparisons match, then the routine will act as if those inputs were valid credentials. Often, in practice, however, less obvious examples exist. One such example is "Joel's backdoor" [103] – a backdoor found in many D-Link products, which enables an adversary to bypass the standard authentication routine of the web-based configuration portal of those devices, by specifying a certain user-agent when accessing them – a property not usually associated with authentication.

A less straight-forward type of authentication bypass vulnerability is one that most often manifests due to an intentional programming "error". In this case, such an "error" might be caused by incorrect handling of input passed to a credential verification routine. While this definition is broad, it is necessarily so, and implies the inclusion of standard, but *intentionally* placed memory corruption vulnerabilities – such as buffer overflows. However, it also covers more explicit cases in which services implement their handling of protocol messages in such a way that facilitates authentication bypass. Concretely this might be due to the incorrect handling of global state or permitting invalid transitions within the underlying state-machine logic that models a protocol.

Backdoors of the aforementioned types generally serve to provide an adversary with a means to bypass standard authentication and then grant them the same level of access as a legitimate user of a backdoored program or system. Another category of backdoors, which manifest as *undocumented commands and features* often provide an adversary with additional functionality not available to a legitimate user. In practice, such undocumented features are commonly accessed through the use of additional, non-standard protocol commands. In other cases, a system, or program, might provide a *hidden* interface, which an attacker is able to interact with that serves to facilitate backdoor access. In embedded device firmware, a backdoor of this kind might take form as an unauthenticated shell running, *e.g.,* via a Telnet daemon. In other cases, bespoke or dedicated services may provide backdoor access; for instance, a service that implements a custom, unauthenticated protocol for executing commands on the device. In practice, many of these bespoke services are referred to (by their implementer's) as *debug* interfaces, which are reportedly left over as a by-product of development.

The final class we consider (for the work in this thesis) we coin *information leakage* backdoors. Backdoors of this kind enable an attacker to *leak*, or access otherwise privileged information from a given system. The data leaked may be, for example, the contents of an arbitrary file or buffer in memory. In embedded device firmware, such a backdoor might manifest as a dedicated *debug* service – in a way similar to the previous class of backdoor. A real-world example of a backdoor of this kind is the TCP-32764 backdoor [36], which affects devices from many device manufacturers, including Cisco, Linksys and Netgear. The backdoor manifests as a dedicated service that listens for connections via TCP port 32764, that when provided with specific input, will leak device configuration values – such as the password for the administrative interface of a device. What distinguishes this class of backdoor from the previous, is that backdoors of this class do not directly enable command execution, rather they only facilitate reading privileged information.

An additional class of backdoors – so-called *cryptographic backdoors* also exist. They are able to compromise a device through weakening cryptographic primitives, *e.g.,* random number generators. While we acknowledge that such threats pose a significant problem, their detection is beyond the scope of the work presented in this thesis.

### 2.5.2 Backdoors in Firmware

Backdoors, like most malware, can be introduced into a device at various points – both before the device reaches an end user, such as during development, or afterwards, by an attack performed by an active adversary. For embedded devices, due to the aforementioned lack of a persistent file-system, backdoors are often introduced either through firmware updates, or shipped with the original device firmware. In the proceeding descriptions, we account only for software-based backdoors, but do acknowledge that hardware-based backdoors (*e.g.,* [184]) can just as easily compromise the security of a device, such backdoors, however, are not in the scope of this thesis. We consider three possible points for a device to be compromised:

1. **Compromised at source** In this case, the firmware of the device contains either a legitimate service, with a backdoor added to it (*e.g.,* [102]), or a dedicated service, often irrelevant to the advertised functionality of the device that acts as a backdoor (*e.g.,* [36]).

2. **Compromised in transit** The device, in this case, is assumed to be secure or uncompromised in leaving the manufacturer or vendor, and then intercepted by a third-party whilst in transit and subsequently tampered with before finally reaching the end-user.

3. **Compromised in use** The device is compromised due to the exploitation of either, an operating system kernel, or application-level vulnerability, in which a malware payload allows a backdoor to persist on the device following a successful exploitation attempt. There are no end of examples of such malware – targeted-or-not – high-profile instances include, Flame [126], Stuxnet [116] and that developed by the so-called Equation group [115].

## 2.6 Machine Learning for Malware Detection

Detecting if a device is compromised by a backdoor can be modelled as a (binary) classification problem; *i.e.,* a device is in one of two states: either it is *compromised*, or *not compromised*. Machine learning is a blanket term used to describe generalised techniques and algorithms that "learn" *classification*, *regression* or *clustering* models, given a collection of input-data, often referred to as *training data*. For the purposes of this thesis, we consider only *classification* and *clustering* models. Given a model as input, a machine learning algorithm is able to *classify* or *cluster* further instances of data presented to it, that are of the same format as the data used to create it. The term *attribute* or *feature* is used to refer to a property of an instance of that data, where such an instance is represented as a collection, or vector of those properties, commonly named a *feature vector*. The process of constructing a model is referred to as *training*. Different algorithms require varying degrees of user intervention during this *training* process, such as algorithm-specific parameters, which can be used to *fine-tune* the output model produced. Each algorithm will make certain assumptions about the data used to *train* it, specifically in influencing, for instance, how to produce a corresponding output for a given input. Thus, for a given problem, the choice of algorithm, as well as the type of output, will be strongly dependent upon the data available, what format it is in, how much is known about it *a priori*, and what we wish to learn from it. This, in turn, will influence the choice of the so-called learning methodology; for our purposes, we consider three such methodologies:

**Unsupervised learning** denotes a methodology that will attempt to learn patterns in the provided input-data, with no specific guidance as to what to learn. This methodology requires the least manual effort perform training. The most common task of algorithms supporting unsupervised learning is to cluster data: that is, learn how to partition that data into clusters that are *similar*, or *related* in some way, often according to some distance metric.

**Supervised learning** requires each instance of input-data be a pair containing an *input* component and a corresponding *output* value, or label. An algorithm supporting supervised learning will learn some function of the *input* component domain as a means to map them to values, or labels in the corresponding output domain.

**Semi-supervised learning** is essentially a hybrid methodology of supervised learning and unsupervised learning; it requires a portion of its input-data be labelled – as with supervised learning, while the rest of its input-data can simply be input components. A semi-supervised learning algorithm attempts to learn a function mapping data from the input-component domain to a label, or value in the output-component domain.

In order to evaluate the quality of a model learned by a machine learning algorithm, a number of techniques can be used; here, we outline the two most common. In the case of a clustering algorithm, this would be a classes to clusters evaluative approach, which assigns known classes/expected cluster labels to clusters generated within a learned model. Such mappings are derived from the majority value of known class label/expected cluster label instances mapped to each cluster. The quality of a model, in this case, is computed using the error in these mappings. While for a classification algorithm, the quality is computed based on the error between the assigned class labels and the actual class labels. These measures can be computed either using the *training data*, or a dedicated set of so-called *test data*, the two approaches are detailed below:

***k*-fold cross-validation** refers to when *training data* is used both to *train* a model and evaluate it; this is advantageous if the amount of data is small, or the process of labelling data is manually labourious. However, it has the downside of potentially causing *overfitting* of the input data; this is where the model learned is too specific to the input data, and not general enough to model the entire input domain. This situation arises if there is not sufficient variety representative of the entire input domain within the *training data*. $k$-fold cross-validation is performed by first dividing the input data into $k$ partitions; one of those $k$ partitions is held back as test or validation data, and the remaining $k-1$ partitions are used as *training data*. This process is repeated $k$ times, where each partition is used once as *test data*. The $k$ results are averaged to produce a single estimation of the performance of the model.

**Use of dedicated test data** refers to a methodology that utilises a separate set of labelled data *i.e., test data*, independent of the *training data*, to perform evaluation. Thus, gives the most clear estimate of the (extrapolated) performance of a model – assuming of course, that the *test data* is representative of the entire input domain. To perform evaluation, each instance of *test data* is used as input to the model under evaluation, the output of this process is then compared against an expected value, or label, which is then used to produce an estimate of the performance of the model.

In evaluating a model, a number of properties can be computed, which relate to what the model classified or clustered, correctly or incorrectly:

**True Positive (TP)** is the measure of data that is identified as positive that is actually positive; *i.e.,* a program is classified as containing a backdoor, and it does contain a backdoor.

**False Positive (FP)** is the measure of data that is identified as positive that is actually negative: *i.e.,* a program is classified as containing a backdoor, and it does not contain a backdoor.

**True Negative (TN)** is the measure of data that is identified as negative that is actually negative: *i.e.,* a program does not contain a backdoor, and is classified as not containing a backdoor.

**False Negative (FN)** is the measure of data that is identified as negative that is actually positive: *i.e.,* a program does contain a backdoor, but is classified as not containing a backdoor.

The above definitions are easily understood for binary classification problems (*i.e.,* where there is a choice of two outcomes, which can be abstractly labelled as the positive and negative cases), for non-binary classification problems, it is not immediately obvious how these measures can be used. For a classification problem such as assigning a data point a label from a set of labels *i.e.,* a program could be a web-server, FTP-server, Telnet-daemon, and so on, the properties are computed for each label, for example, for a web-server, we would consider a positive label of *web-server* and negative label which would be any label that is not *web-server.* From these basic properties, the following standard measures can be used to quantify the performance of a model:

**Precision (P)** is the ratio between data correctly identified as positive and the total amount of data identified as positive: *i.e.,* $P = \frac{TP}{TP+FP}$.

**Recall (R) / True Positive Rate (TPR)** is the ratio between the data correctly identified as positive and the total amount of data that should have been identified as positive: *i.e.,* $R = \frac{TP}{TP+FN}$.

**False Positive Rate (FPR)** is the ratio between the data incorrectly identified as positive and the total amount of data that should have been identified as negative: *i.e.,* $FPR = \frac{FP}{FP+TN}$

**Accuracy (ACC)** is the ratio between the data correctly identified and all identifications made: *i.e.,* $ACC = \frac{TP+TN}{TP+FP+TN+FN}$.

**Error (ERR)** is the ratio between the data incorrectly identified and all identifications made: *i.e.,* $ERR = \frac{FP+FN}{TP+FP+TN+FN}$.

## 2.7 Chapter Summary

In this chapter, we have outlined the necessary background required for understanding the remainder of this thesis, as well as provided an insight into existing (manual) approaches and techniques for binary firmware acquisition and analysis. Furthermore, we have outlined a number of strategies and metrics for measuring the performance of systems constructed using machine learning techniques, which are used by both the related literature and the work we present in Chapter 5.

# Chapter 3

## *Related & Previous Work*

**Contents**

As noted in Chapter 2, the process of detecting backdoors within embedded device firmware draws on techniques and inspired methodologies from a variety of distinct fields. In this chapter, we explore the literature related to those fields, as well as the small body of work directly related to backdoor detection.

Since the primary target of backdoor implementations are programs, we investigate the literature related to program analysis, or more specifically, binary program analysis. Indeed, the deficiencies in this field are particularly useful to consider as they directly impact the ability of any detection methodology based upon analysing a program's code, either statically or dynamically. We investigate general tools, libraries, and intermediate languages that have been developed for binary program analysis which can be considered a base for building more specific analysis tools, for example, to detect backdoors. Alongside more general program analysis approaches, we review more niche state-of-the-art techniques for vulnerability discovery, malware detection, and ascertaining semantic properties of programs – which are all fields intrinsically related to backdoor detection; the literature here serves as inspiration for the techniques we discuss in detail in Chapters 5 and 6. Further, we discuss techniques explicitly developed to detect security vulnerabilities within embedded device firmware, as well as review approaches taken to large-scale analysis of those devices. In order to ascertain a complete understanding of the state of backdoor implementations, we look at the inverse problem to that which we address, *i.e.,* backdoor implementation strategies. This literature serves to highlight more complex and esoteric backdoor implementations, compared to real-world backdoors that have been publicly documented – which are often more simplistic in nature, this, in turn, aids in providing a more suitable definition to the term backdoor and their detection, which we present in Chapter 4.

## 3.1 Program Analysis

Program analysis is a broad field of research; in Chapter 2, we discussed how (binary) program analysis is an essential technique for backdoor detection. In this section, we outline the critical problems in program analysis which impose limitations on the practicality of implemented backdoor detection approaches.

### 3.1.1 Recovering Program Structure

When attempting to perform any kind of program analysis, the first and perhaps most fundamental stage involved, is the recovery of an accurate-as-possible representation of the program under analysis. By this, we mean the recovery of its functions, their associated CFGs and any referenced data. Unfortunately, the process of this recovery, *i.e.,* disassembly, is an undecidable problem [149]. Having an incomplete view of a program – especially when attempting to detect backdoors or bugs – is particularly problematic for building practical tools. To understand why, consider the following scenario: suppose a code structure in which a particular backdoor manifests is not correctly recovered and what is recovered is used as input to a backdoor detection tool which will output if a given program is benign or malicious. In such a case, the program analysed will likely be identified as benign, when in fact it contains something malicious – not due to a flaw in the backdoor detection method, but due to the incomplete structure recovered and supplied to it as input. Andriesse et al. [48] examine the difficulties in recovering program structure, in particular, they address the process of disassembly and highlight the impact the challenges examined upon program analyses subsequently performed upon the results of disassembly.

The fundamentals of program analysis are given an in-depth treatment in numerous works, such as [138] and [117]. In this section, we review related work in this area, which serves to address specific challenges in disassembly and program structure recovery. Cojocar et al. [73] for instance, attempt to tackle the problem of switch-case recovery. Switch-case constructs pose a challenge for disassemblers as, often, their jump targets are dependent upon program input, and thus, will be dynamically computed at program run-time. Disassemblers often employ heuristics to estimate these jump targets, which can lead to the incomplete recovery of program instructions in the case of under-estimation. Consider the switch-case construct depicted in Figure 3.1, in this case, the branch target is dependent upon the variable `v`. Here, a compiler may generate code that utilises a so-called jump-table, as opposed to translating the construct into a series of cascaded if-then-else-like branches, such a table will often be embedded (as data) within the code section of a binary. In doing so, the generated code will violate the assumptions made by many

```
switch (v) {
  case 10:
    // ...
  case 20:
    // ...
  ...
  default:
    // ...
}
```

Figure 3.1: Switch-case construct.

static analysis methodologies (*e.g.,* code and data are not mixed), and therefore result in incomplete recovery of program structure (and instructions). The authors attempt to recover the jump targets of such constructs using a tailored Value Set Analysis (VSA) [51]; they demonstrate the effectiveness of their approach by evaluating it on a number of binaries from various public datasets, compiled using both Clang [8] and GCC [12]. For Clang compiled binaries from the SPEC dataset [28], which contains 828 switch-case constructs implemented as jump tables, their technique is able to recover a total of 763 jump tables, outperforming IDA Pro [14] – the industry standard tool for disassembly, which recovers only 11 instances successfully. While for GCC compiled binaries, their approach achieves comparable performance.

A further problem faced by disassemblers is function start identification, *i.e.,* given a series of disassembled instructions, identifying where a function starts. Such identification is critical for performing many types of analysis effectively, for example, intraprocedural analyses. Traditionally, to perform identification, tools such as IDA Pro have relied on sophisticated heuristics, which have to be continuously updated to incorporate changes introduced through different compiler optimisations and versions. To improve upon such heuristic-based approaches, Bao et al. [53] propose the use of machine learning to classify function start patterns. For their system, Byteweight, they train a classifier using supervised learning on a dataset of instruction sequences indicative of function prologues, *i.e.,* function start patterns. On the dataset they use to evaluate their classifier, they show Byteweight is able to consistently outperform IDA Pro on ELF binaries (for both the x86 and x86-64 architectures) and performs comparably to IDA Pro on PE executables. Shin et al. [165] similarly attempt to tackle the problem using a machine learning approach; they note deficiencies in the approach and results presented by the authors of Byteweight – both in their stated accuracy and run-time performance, and ultimate feasibility for real-world application. They propose using recurrent neural networks (see, *e.g.,* [110]), which when trained and evaluated on the same dataset as the aforementioned approach, trains faster (by an order of magnitude) and has a reduced error-rate on six out of eight benchmarks, while remaining comparable on the remainder.

Andriesse et al. [49] propose an approach, as well as an open-source implementation of that approach, Nucleus [44], which addresses function start detection by an alternative means. As with [53] and [165], they aim to identify function starts in a compiler and architecture agnostic manner, but additionally attempt to mitigate the need for things such as classifier retraining – required by both of those approaches in order to account for new compiler optimisations, and conventions. In place of machine learning, Nucleus gradually refines a so-called Interprocedural CFG (ICFG). An ICFG is generated from a disassembly produced through the use of the linear sweep algorithm (see, *e.g.,* [162]); recent work finds such an approach to be effective [48], despite its apparent simplicity. They then refine the ICFG by pruning edges generated by `call`-type instructions, *i.e.,* interprocedural control-flow, which enables them to discover the entry-points of functions that are called directly. These entry-points are subsequently expanded by following control-flow (without consideration of flow direction). Remaining function starts are identified via connected components analysis (see, *e.g.,* [101]) combined with intraprocedural control-flow analysis. The authors evaluate Nucleus upon a different dataset of binaries compared to the previously described approaches – citing concerns that the dataset used by those approaches is biased due to many binaries sharing common functions. In their evaluation, Nucleus is shown to outperform Byteweight, IDA Pro and Dyninst [55] (a binary analysis framework implementing an alternate approach to function start identification, discussed further in §3.1.4).

### 3.1.2 Cross Platform Analysis

Since embedded device manufacturers build their devices on multiple, differing hardware architectures (*e.g.,* ARM and MIPS), techniques applicable for performing cross-platform analysis are necessary for effective analysis. This requirement poses a problem: if an analysis pass targets the native processor instruction set of a given device, to support devices of other architectures that analysis must be reimplemented for each additional architecture it needs to support. To overcome this, the native processor instruction set can be *lifted* to an intermediate language (IL), for which the analysis pass needs only to be implemented once (as described in Chapter 2).

#### 3.1.2.1 Intermediate Representations

Intermediate language representations not only simplify the implementation of analysis techniques by creating a uniform program representation, irrespective of the original architecture a program was written for, they are also able to reduce the number of language constructs to consider when performing that analysis. That is to say, IL representations can reduce an extensive, complex instruction set to a small number of simple IL constructs, which model the functionality of more complex instructions in the source instruction set. Of course, the quality of analysis possible on an IL representation depends upon the quality of the lifting procedure and the expressiveness of the IL. When assessing this quality, we can ask the following questions: are all instructions from the source instruction set architecture (ISA) liftable into the IL? Are those instructions that are lifted modelled correctly (including their side effects)? Kim et al. [118] evaluate the correctness of three state-of-the-art binary lifters: PyVEX [167], BAP [63] and BINSEC [81]. In their evaluation, they find bugs in all of the lifters and demonstrate that none of the lifters are able to correctly lift all of the instructions they were tested upon from the source ISAs evaluated (x86 and x86-64). Of the three lifters, BAP was able to correctly lift the highest number of instructions, 206,118 for x86 and 632,035 for x86-64, compared to 135,172 and 516,974 for PyVEX, and 202,652 (x86 instructions) for BINSEC.

A large number of ILs have been proposed by both academia and industry. These ILs can be divided into two categories: those that are suitable for writing general analyses, and those that are designed specifically for implementing particular types of analysis. BIL [97] is a general purpose IL and is part of the BAP framework [63]. BIL provides high-level constructs such as if statements and while loops; at the time of writing, BAP provides lifters for ARM (including Thumb extensions), x86 and x86-64. The authors provide a thorough assessment of their lifters [42], which on their dataset boasts 98.3% correctness for the ARM ISA, evaluated with a QEMU-based tracer, 99.8% and 99.8% for x86-64 with binaries built with GCC and Clang, respectively, with evaluation performed using an Intel Pin tracer [153]. When evaluated with a QEMU-based tracer, they obtain 96.1% correctness for x86-64 binaries built with GCC and 96.3% for binaries built with Clang. For x86 binaries built with GCC and evaluated with a QEMU-based tracer, they achieve 99.1% correctness.

VEX [32], while originally an IL designed specifically for use with tooling from the Valgrind project [31], has been used in isolation – notably in the angr binary analysis framework [167] as PyVEX. Lifters with VEX as the target IL exist for a significantly larger number of architectures compared to BIL (AArch64, ARM, MIPS, PPC, x86, x86-64). However, VEX is unable to directly represent certain instruction side-effects, such as flag calculations, that are required for correctly modelling some ARM and x86 instructions, this leads to a loss of semantic information, which can be problematic for certain types of static analysis. To represent these computations, so-called helper functions are used, which are implemented in C in the Valgrind reference implementation, while in PyVEX, they are reimplemented in Python. Djoudi et al. [81] introduce yet another IL with their tool BINSEC which – at the time of writing – lifts only a subset of x86 instructions. Dullien et al. [84] propose REIL as a platform-independent IL for static analysis, which has a number of open-source lifter implementations such as [4] and [22]. ESIL is another IL, which has been developed as part of the Radare2 project [24]; it is a stack-based language, which simplifies evaluation and automated analysis due to utilising a uniform representation for all of its constructs. However, like VEX, it handles flag operations as a special case. Other research [17, 10, 68] has proposed using the LLVM Intermediate Representation [16] as an IL target for analysis; a number of projects (*e.g.,* [25]) implement lifters for it from various source architectures.

### 3.1.3 Discovering Bugs & Program Properties

As discussed in Chapter 4, the process of detecting certain classes of backdoor often involves detecting program bugs, while computing program properties such as reachability satisfaction can be used to detect others. In this section, we discuss the state-of-the-art in automated vulnerability detection and related work on detecting program properties – both in the context of binary executables. The notion of bug search is adaptable for backdoor detection, as backdoor implementations, like vulnerabilities, can be extracted and searched for.

#### 3.1.3.1 Program & Code Fragment Similarity

One way to automate backdoor detection is to search for a known backdoor, or a component thereof. This search may yield binaries that are derivatives of the original program containing the backdoor (*e.g.,* due to recompilation), different versions of the program, or programs containing a construct considered a component of the backdoor in the original program. Performing this search effectively requires being able to represent a construct or binary in a generalised, meaningful way suitable for comparison, ideally such that semantic information is preserved. Scalability of a search is directly related to the complexity of performing comparison; thus, the choice of representation is inherently a trade-off between attempting to optimise in regard to scalability, or in regard to the accuracy of comparison.

Dullien et al. [85] propose a graph-based representation for executables, which they utilise to perform binary executable comparison. Their comparison methodology assumes pair-wise comparisons, where to compare two binaries, they construct a bijective mapping between the functions of each binary. This mapping is constructed by iterative improvement of a partial graph isomorphism (see, *e.g.,* [57]) on the call graphs of each binary. Using that mapping as a basis, a further mapping between basic blocks and then functions is computed. They measure the similarity of basic blocks using the similarity of their instruction sequences, which, in turn, is used to compute a measure of function CFG similarity. In order to be sensitive to differing compilers and optimisations, their similarity measure attempts to account for different register allocation, instruction reordering and branch inversion. To handle instruction reordering, each unique instruction within a block is assigned the value of a small prime number. Two sets of instruction sequences, or blocks, are said to be permutations of each other if their Small Primes Product (SPP) is equal (*i.e.,* the product of each prime number assigned to each instruction within a block). However, for heavily optimised code – such as when a compiler applies function inlining – constructing such a mapping between two executables is less than effective; in such a case, both the basic blocks and control-flow graphs they are part of can differ significantly enough to render matching on syntactic properties (*i.e.,* disassembled instructions) ineffective. Gao et al. [96] show that the situation is, in fact, worse – they find that in practice, differences in code generated overall, even due to a single modification of one function, is rarely isolated just to that function. Using a representation similar to Dullien et al. [85], Bourquin et al. [58], employ a technique based upon bi-partite graph matching (see, *e.g.,* [57]) to measure similarity; however, their computation of similarity relies on computing a so-called graph edit difference.

Neither of the aforementioned approaches take into account instruction semantics in their similarity computation. In contrast, Gao et al. [96] devise a technique for finding differences in binary executables, that does take into account such semantic information. They utilise a novel graph-based isomorphism technique, which they augment with symbolic execution and theorem proving. Previous techniques such as that by Dullien et al. [85] use a greedy approach in their construction of graph-based isomorphisms, and so erroneous matches are be propagated through their isomorphisms. In the proof-of-concept tool presented by Gao et al. [96], BinHunt, they utilise backtracking when computing graph-based isomorphisms, which serves as a means to replace erroneous matches. For a given binary, BinHunt first performs disassembly and then lifts the disassembled instructions into an IL representation. Following this, it constructs a call graph for the entire binary and recovers control-flow graphs for the functions it is able to identify. To determine if two basic blocks within those CFGs are equivalent, the IR of the blocks are evaluated using symbolic execution and a theorem prover is used to check whether their effects are equivalent. The similarity between two functions is computed based upon the recovered CFGs and the similarity between their basic blocks. The output of the isomorphism construction is two sets of triples: matching function pairs and their matching strength, and matching basic blocks and their matching strength. The matching strength of function pairs is determined by the size of the maximum common induced subgraph (see, *e.g.,* [57]) between their control-flow graphs, which, in turn, is used to compute an overall similarity measure between two given binaries.

Ming et al. [130] propose a derivative tool, iBinHunt, which extends the work by Gao et al. [96]. As with that work, they compute a graph-based isomorphism to aid in measuring similarity. To reduce the number of basic blocks to compare, they perform a preprocessing step. This step involves monitoring the execution of the two binaries being compared, where each is supplied with a common input. They use taint analysis to record all of the basic blocks involved in processing the supplied input; they additionally augment this analysis by assigning different tags to different components of the input – a technique they call *deep taint*, which allows them to match basic blocks that are marked with the same taint tags, as opposed to all basic blocks tainted. This reportedly reduces the number of block matchings by up to 74%.

Graph-based representations by definition use graphs, such as function-level CFGs directly. Directly comparing two graph-based representations, by nature, must be performed in a pair-wise manner. When using standard methods, these comparisons will not produce results that are transitive, that is, computing the similarity between binaries $A$ and $B$, will tell us nothing about the similarity between binaries $B$ and $C$. Additionally, while graph-representations are structure preserving, they are computationally inefficient to compare. The combination of these issues makes graph-based representations largely impractical for large-scale similarity queries. To attempt to overcome these problems, other approaches perform similarity computations upon high-level features derived either from graph-based representations or other information present in program binaries. One such approach, which uses a combination of CFG structural information and high-level numeric features, is explored by the authors of discovRE [87]. They demonstrate their methods are effective in locating the presence of a number of high-profile TLS implementation bugs, such as Heartbleed [37] and POODLE [132] within a large dataset. In their approach, they represent a binary's functions using numeric feature vectors, where the features represent meta-information related to a given function, for example, the number of instructions within the function and the total number of basic blocks it contains. Their proposed system operates in two stages; in the first stage, an input dataset is queried for functions that are similar to an input function based upon its computed numeric feature vectors. In the second stage, the functions that were reportedly similar to that queried for are compared for structural similarity; this similarity comparison serves to produce the final output of the system. In order to find similar feature vectors, the authors embed the feature vectors into a vector space and use an unsupervised learning approach to cluster them. They locate functions similar to a given input function by using it to query the learned model, which will output a cluster assignment; this cluster will contain the functions the queried function is most similar to.

Feng et al. [88] tackle the same problem of (known) bug search, applied specifically to IoT firmware images. Similar to the approach taken by the authors of discovRE [87], they use numerical feature vectors to represent program constructs. Specifically, they lift program CFGs into numerical feature vectors. To do this, they first construct a so-called Attribute CFG (ACFG) for each function, which is a CFG where each basic block is labelled with a set of attributes. These attributes are both statistical and structural. For statistical attributes, the authors use information related to the use of string and numeric constants, the number of instructions it contains, and counts of instructions with certain properties *e.g.,* arithmetic instructions and transfer instructions (*e.g.,* calls). The structural attributes used are the out-degree of a block and its betweenness centrality [94] (a measure of the block's centrality within the overall CFG). They use bipartite graph matching (see, *e.g.,* [57]) to compute the similarity between two ACFGs; the structural information incorporated into each block's attributes is used to remedy the fact structure is not taken into account when performing bipartite matching. They then construct feature vectors by learning a codebook for the ACFGs (performed through unsupervised clustering) and then learn a quantisation function over the codebook. The resulting function maps from ACFGs to high dimensional numerical feature vectors representing ACFGs. Prior to searching for similar ACFGs, the feature vectors are hashed using

Locality Sensitive Hashing (LSH) (see, *e.g.,* [148]) and then projected onto a so-called hashing space. They perform the final search using a nearest neighbour query. Genius, the author's proof-of-concept tool implementing their approach, outperforms discovRE in both query time and accuracy.

A further alternative to using graph-based representations for comparison is to encode a binary, or a component of a binary using a so-called *signature.* Cesare et al. [65] use such an approach as a basis for comparing malware binaries. Pewny et al. [146] utilise a signature-based representation for performing known bug search. They devise a similarity metric, Tree Edit Distance Based Equational Matching (TEDEM), which provides a means to identify regions of code within a given binary that are *similar* to those of a reference bug. In their approach, a signature is defined as a sub-graph of the control-flow graph of a function containing a particular bug, which is composed of complete or partial basic blocks and the transitions between them. To test a given binary for a particular reference bug, the binary and bug signature's basic blocks are translated into so-called expression trees, which summarise the effects of the computation performed by their blocks. They then use TEDEM to compute a block-centric measure of similarity between expression trees. TEDEM quantifies the minimum cost required to transform one block into the other, based upon node replacement and/or insertion/deletion of sub-trees. They use the blocks most closely matching those from the reference bug as starting points for matching the remainder of the blocks in the signature. In their approach, signatures must be specified manually. While this has the disadvantage of requiring an expert program analyst perform manual reverse-engineering to extract and generate a given signature, it allows them to specify arbitrary program bugs – which would not be possible in a completely automated approach.

Pewny et al. [145] propose another method for cross-architecture bug search. They define so-called *bug signatures*, which attempt to capture a unified representation of binaries, irrespective of their target architecture. To do this – as with the previous approach – they first disassemble the input binary and lift instructions into an intermediate representation. From that representation, they construct so-called assignment formulas for each basic block. These assignment formulas capture the behaviour of each block in terms of input and output variables, which serve to represent their semantics. The authors generate a so-called *bug signature* from a fragment of a *known* vulnerable binary program, which itself is specified as a set of semantic hashes, which encode the assignment formulas representing its basic blocks. These signatures are used as a basis for computing block-level similarity, which, in turn, are used to compute a measure of function-level similarity.

Lakhotia et al. [125] discuss a further means for identifying semantic differences between binaries through a notion they coin "semantic juice". Semantic juice is an abstract representation of a basic block that encodes its semantics. They use this representation in a similar manner to the previously described works to perform scalable search for similar code fragments.

Other research considers variants of semantic signatures; for instance, Jin et al. [112] utilise semantic hashes with clustering to locate functions that are similar based on those hashes. Ng et al. [137] present Exposé, which aims to identify binary code reuse between application and library code. They use a number of properties to determine matches, including semantic information, syntactic representation and a novel numeric distance measure.

#### 3.1.3.2 Heuristic-based approaches to identifying program properties

Thus far we have explored work which focuses on finding previously *known* bugs. While these approaches are adaptable for locating *known* backdoors, the amount of publicly documented backdoors is small. Moreover, such approaches do not address the initial effort in identifying *bugs*, or in the case of this work, components of backdoors. As opposed to searching for *known* constructs that are indicative of a program property, heuristic-based approaches work on the notion that particular program behaviour can be characteristic of a given program property, and attempt to find constructs that exhibit that behaviour. A buffer overflow vulnerability, for instance, will manifest as a memory corruption, which, can, in turn, lead a program to crash if control-flow can be hijacked. Checking for a backdoor, or component thereof in this context, requires identifying program behaviour that is indicative of backdoor-like behaviour, which is not a well-explored area; we address this issue in Chapter 4.

Automatic vulnerability detection is closely related to backdoor detection: firstly, backdoor components can manifest as *intentional* program bugs; secondly, identifying inputs used to induce a particular program state (*i.e.,* a crash for control-flow hijack vulnerabilities) can be used to locate, for example, hard-coded credential checks.

Avgerinos et al. [50] present AEG, reportedly the first *end-to-end* system for fully automated exploit generation. Their approach is divided into two phases: vulnerability identification and exploit generation. They first show that vulnerabilities that result in control-flow hijacks can be modelled as a formal verification problem and as a result, are able to demonstrate that targeted symbolic execution is effective in identifying those vulnerabilities. To overcome deficiencies in symbolic execution – that would otherwise make it prohibitive for use in such a way (due to the problem of path explosion) – they devise heuristics for optimal path selection. To do this, they use a priority queue which relies upon heuristics to decide which paths to check first. These heuristics target paths which are more likely to be exploitable and attempt to eliminate other paths. Their system requires access to both the source code and compiled representation of the program they are analysing.

Cha et al. [66] propose another automated exploit generation system, which, in contrast to AEG, works without access to the source code of the program under analysis.

They introduce a technique called *hybrid execution*, which combines both offline and online symbolic execution. Offline symbolic execution, or concolic execution, involves concretely executing a target binary with an initial *seed* input and tracing that execution, symbolic execution is then performed on the execution trace. Offline symbolic execution requires re-evaluation of entire paths in order to examine different program branches. Online execution overcomes this problem by forking two symbolic interpreters at branch points, as a trade-off by utilising more system resources. In evaluating their system, Mayhem, the authors show that of the 29 applications they tested, they are able to identify 29 exploitable vulnerabilities. Other work utilising symbolic execution – targeting other analysis domains – includes, angr [167] for reverse-engineering, S$^2$E [69] for program analysis, Firmalice [166] for backdoor detection, [80, 144] for bug-search, KLEE [64] for software testing, and [170] for software verification.

Fuzzing is a method traditionally used to generate input to a program, in the hope that that input forces execution down a path that induces a program crash state. Automated techniques for input generation and monitoring the result of a program processing that input are applicable for identifying, *e.g.,* hard-coded credential checks. AFL [2] is the de-facto tool for performing fuzzing. A significant limitation of the approach taken by AFL is that its generated inputs are based upon randomised byte-strings, thus, satisfying a string comparison against a specific hard-coded value amounts to blindly *guessing* that particular input via brute force.

Stephens et al. [169] present Driller, which introduces a hybrid approach to vulnerability discovery based on fuzzing and selective symbolic execution. Both fuzzing and symbolic execution have drawbacks; the former can get "stuck" when it fails to produce inputs to explore new paths, the latter can succumb to path explosion. In their approach, they utilise fuzzing and symbolic execution in a complementary manner. First, fuzzing is used to explore a so-called initial program "compartment", that is, all paths that can be explored at that point. When the fuzzing stage runs out of such paths, Driller attempts to find new paths by performing concolic execution – solving input constraints to satisfy branch conditions that fuzzing is unable to; once new paths have been identified, fuzzing is resumed; this process is used in a cyclic manner to drive analysis deeper and deeper into a program's code. The authors report that their hybrid approach is more effective than either fuzzing or symbolic execution used in isolation.

VUzzer, a system developed by Rawat et al. [150], employs a similarly augmented approach to fuzzing. Their method first extracts both control- and data-flow properties obtained via static and dynamic analysis and uses them to infer characteristics of the program under analysis. These characteristics are used to facilitate generation of more *interesting* or *application-aware* fuzzing inputs, compared to more traditional target-agnostic approaches. VUzzer does not require any prior knowledge of the binary it is analysing or specific information regarding the format of its expected input. The authors show their approach is able to provide a competitive improvement over a related tool, the AFL-based AFLPIN [1], without the use of symbolic execution as used in Driller [169], and as a result, can achieve greater scalability.

### 3.1.4 Frameworks, Libraries & Tools

A number of frameworks, libraries and tools exist for performing binary analysis. Which, to varying degrees, can facilitate the creation of more specific binary analysis tools, for example, for backdoor detection.

IDA Pro [14] is a state-of-the-art commercial, cross-platform disassembler supporting multiple architectures including, ARM, MIPS, PPC, x86, and x86-64. It can be extended using plugins and scripts developed in a number of languages: C++, IDC (a language designed specifically for scripting with IDA Pro) and Python. While plugins and scripts are primarily used to extend its capabilities, IDA Pro can also be executed in a *headless* configuration, which enables it to be used (unintrusively) by other programs and scripts via its scripting interface, *e.g.,* to access information related to the results of its disassembly process. Binary Ninja [5] is another commercial cross-platform disassembler; it supports a comparable feature set to IDA Pro, though natively supports fewer architectures. As with IDA Pro, it can be extended through a plugin interface. Binary Ninja also supports native lifting of assembly language instructions to a number of ILs, which provide varying levels of abstraction over the underlying disassembly. Radare2 [24] is an open-source reverse engineering framework, which, among other features, provides a disassembler, which supports a large number of architectures. It can (as with IDA Pro and Binary Ninja) be scripted from a variety of languages, supports lifting to its own IL, ESIL, and provides an interpreter for instructions lifted into that representation. BinCAT [56] is a framework developed in OCaml, which supports three architectures (x86, ARMv7, AArch64), and provides tight integration with IDA Pro; it features a number of analyses: value analysis for both memory and registers, taint analysis, and type reconstruction.

While IDA Pro, Binary Ninja and Radare2, provide what amount to front-ends that aid in reverse-engineering and support scripting and automation as a secondary feature, other frameworks and libraries exist, which have a greater emphasis for use in tool development. angr [167] is one such framework; it is written in Python and has been used as a basis for developing numerous academic research tools, notably Firmalice [166]. angr provides a means of performing both static and dynamic symbolic (concolic) analysis, as well as convenient instruction lifting to the VEX IL [32]. Amoco [179] is a Python framework for binary static analysis. It supports a number of strategies for performing disassembly, including: linear sweep, recursive traversal and path-predicate based disassembly – which utilises SAT/SMT solvers to aid in CFG recovery. As with angr, it also provides facilities for performing symbolic execution of instructions. BINSEC [81] is a platform for writing binary analysis tools developed in OCaml. At the time of writing, it supports x86 ELF binaries and has experimental support for PE executables. It can perform basic disassembly, symbolic execution and lifting to its own IR format. Dyninst [55] is a binary instrumentation framework supporting the AArch64, PPC and x86 architectures. It enables binaries to be both analysed and dynamically patched; to support development of derivative tools, it provides an API facilitating the insertion of code into running programs. Triton [154] is a dynamic binary analysis framework written in C++: it can be interfaced with both C++ and Python. It provides a dynamic symbolic execution engine, taint analysis, and an AST representation for the architectures it supports (x86 and x86-64). Binary Analysis Platform (BAP) [63] is a binary analysis framework written in OCaml and can be seen as the successor to the BitBlaze project [168]. It supports the x86 and ARM architectures and has partial support for MIPS and PPC, amongst others. It has usable APIs for C, OCaml, Python, and Rust. Analysis passes written for BAP generally operate upon the BAP specific IL, BIL; and the framework provides lifters to that IR as plugins; in its default configuration, it provides lifters for both x86 and ARM instructions. Adding lifters for other architectures is possible and BAP provides an API for their development. In earlier versions of BAP – which the tools described in Chapters 5 and 6 were built with – BAP was generally used as a standalone library, such that its components could be used in isolation to develop analysis tools. However, in the current version of BAP, developers are encouraged to implement their tools as analysis passes in the form of plugins, with the intention that these plugins can then be chained together to realise more complex passes and analysis tooling.

## 3.2 Embedded Device Analysis & Security

The previous section outlined general frameworks and tools that can be used as a basis for creating binary analysis tools; for embedded device firmware, various limitations exist which prevent the unconstrained use of some of these frameworks directly: for example, those that rely on dynamic execution (as discussed in Chapter 2). Avatar [186] is a framework designed to facilitate developing dynamic analysis tools that operate on binary firmware. Avatar enables such analysis by directly incorporating the physical device into the analysis process; this allows the framework to overcome difficulties that would otherwise make dynamic analysis impossible, *i.e.,* requiring access to peripherals connected to the device that cannot be reasonably emulated. To perform analysis, Avatar takes a hybrid approach: essentially acting as an orchestrator between the physical device and an external emulator. By doing this, Avatar is able to execute the majority of the firmware's instructions within an emulator and when I/O operations (that cannot be emulated) need to be performed, performs them on the physical hardware. Using this scheme as a basis, Avatar can facilitate the use of other analysis methods that would otherwise not be possible, without complete firmware emulation – such as dynamic symbolic execution. The current iteration of the Avatar framework, Avatar$^2$ [135], supports interfacing with a number of other analysis tools such as QEMU [23], frameworks such as PANDA [83] – an architectural agnostic dynamic analysis platform, and angr [167].

Research into the security of embedded device firmware presents a number of difficulties when compared to analysis of software for commodity PCs. These difficulties arise, in part due to the challenges in adapting current analysis methodologies for use with such devices, and the fact research into their security has only recently become of immediate relevance – largely as a result of the pervasiveness of IoT devices. Costin et al. [75] highlight a number of these difficulties in their analysis of networked CCTV and video surveillance systems firmware. The first large-scale analysis of embedded device firmware was performed by Costin et al. [76]. To facilitate their analysis, they built a system[1], which performs automated static analysis of firmware. They demonstrate its effectiveness on a large dataset (~32,000 firmware images), in which it is able to identify 38

[1]Provided as a service at `http://firmware.re`.

previously unknown vulnerabilities over 693 distinct firmware images. To facilitate their analyses, their system performs automated acquisition, extraction (using Binary Analysis Toolkit (BAT) [29]) and identification of firmware images from a number of vendors. In analysing the security of the firmware, they perform analysis not only of binary-level features, but also search for shared, hard-coded credentials, and hard-coded self-signed certificates within extracted file-systems. To compare firmware components, they utilise fuzzy hashing [121] which facilitates clustering and correlation of components, as well as provides a means to detect previously identified vulnerabilities within their dataset. Amongst the firmware analysed, they were able to discover backdoors in firmware images from multiple vendors – simply by searching firmware images for backdoor-related keywords.

While the approach taken in Avatar [186] generalises for many types of embedded device, Chen et al. [67] focus on analysing devices with Linux-based firmware. Their system, FIRMADYNE supports partial emulation and dynamic analysis of firmware components using QEMU – without access to the physical device under analysis. FIRMADYNE's architecture is separated into four distinct stages. The first provides automated firmware acquisition, which utilises vendor-specific web-crawlers. In the second stage, they extract the firmware obtained in the first stage; to do this, they use a custom utility based upon binwalk [6]. The latter two stages are novel to their approach. In these stages, they perform emulation of the firmware. To do this, they first identify the architecture and endianness of the firmware, and use this information to select a pre-modified Linux kernel to boot it using QEMU. They instrument their pre-modified kernels to enable the interception of syscalls. Their emulation proceeds by first *learning* a suitable network configuration for the firmware by monitoring networking syscalls; using this *learned* configuration, they configure the system appropriately and boot it for subsequent analysis. As discussed in Chapter 2, many Linux-based devices utilise NVRAM as a key-value store to save and restore device configuration; to handle this, FIRMADYNE uses a custom version of `libnvram.so`, which intercepts accesses to NVRAM, that would otherwise fail during emulation. In the final stage of FIRMADYNE's architecture, the authors perform automated vulnerability analyses using previously known vulnerabilities and corresponding exploits from the Metasploit framework [19].

As noted in Chapter 2, many embedded devices utilise web-based interfaces for user-configuration. Costin et al. [77], explore the security of such web interfaces by performing large-scale dynamic analysis using off-the-shelf vulnerability scanners. To perform their analysis, they use firmware obtained from crawling device manufacturer's websites. Without access to the devices for the firmware under analysis, the authors resort to emulation. As discussed in [186] and [67], full system emulation presents difficulties, even for Linux-based systems. Thus, they emulate "enough" of the firmware to execute the firmware's web-server; in many cases, their approach is able to serve both scripts (*e.g.,* via CGI or PHP) and static web content. As in [67], they utilise QEMU to perform the emulation, and likewise opt not to use the kernel (if any) included with the firmware image, which they found to be present in only 5% of the firmware they gathered. Their emulation process proceeds by performing a *chroot* into an unpacked firmware image's filesystem from an emulated Linux system with a generic kernel; within this chroot environment, they execute either a shell (*e.g.,* `/bin/sh`) or the init binary (*e.g.,* `/sbin/init`) and proceed to start a web-server process. They then use automated vulnerability scanners to analyse the web-server and the scripts it serves. Their analysis discovers 225 "high impact" vulnerabilities in the 246 web-interfaces they were able to emulate successfully.

Since there is no universal distribution source for firmware updates, multiple different firmware update mechanisms, and no standard way of storing firmware updates, when presented with a firmware image, identifying the firmware's manufacturer or device type is a challenging problem. Costin et al. [78] address this problem; their method utilises supervised machine learning to classify firmware images: first to differentiate them from standard files, then to attribute them to *known* device vendors and devices types. They achieve 93.5% accuracy in identifying a given file as a firmware image and 89.4% in correctly identifying a firmware's device type.

On most embedded devices, the services that are networked are Internet-facing, hence potentially remotely exploitable, or otherwise vulnerable should they contain, for example, a backdoor. The vast majority of these services implement their interaction with a client (at some level) through the use of protocol parsers. Automatically identifying binaries that contain parsers, and the functions within those binaries responsible for implementing those parsers is useful for targeted analysis of specific types of network services (*e.g.,* web-servers). Cojocar et al. [74], propose a solution to address locating such components through the use of machine learning. They train a supervised learning classifier on a number of simple features of the (LLVM) IL [16] representation of binaries containing protocol parsers. They perform both control- and data-flow normalisation passes over the IL representations and subsequently extract feature vectors, which include properties, such as, basic block count, total number of callers, and number of incident edges to blocks. To optimise the contribution of specific features, they apply a weighting to each feature. They evaluate their classifier on software from various embedded devices (a GPS receiver, power meter, hard drive, and programmable logic controller), and show their system is effective in locating the parser routines in those examples.

## 3.3 Backdoor Implementation

Zaddach et al. [187] discuss the implementation of a "stealth" hard drive backdoor and describe how such a backdoor can be introduced through the use of a hard drive manufacturer's own firmware update tool. An attack of this kind is therefore feasible by a remote attacker who is first able to compromise a system with a vulnerable hard drive attached.

Their backdoor is implemented as a modification to a hard drive's existing firmware. Its trigger mechanism is implemented by hooking the hard drive's write routines and performing monitoring of the buffers supplied to them for the presence of certain so-called *magic* values. When these special values are present, the backdoor functionality – data exfiltration – is performed. The authors report that the overheads induced by their backdoor are negligible – such that attempting to detect it by monitoring changes in device throughput is infeasible. In addition to the implementation of their backdoor in physical hardware, the authors present an implementation of a real-world scenario where their backdoor is leveraged to exfiltrate data via specially crafted requests to a web-service interacting with a database (which is stored on the hard drive containing backdoored firmware), which they demonstrate on emulated hardware.

Following successfully triggering a backdoor, an adversary will have some level of *privileged* access to a compromised device, which is similar to the result of a successful exploitation attempt. Cui et al. [79] detail a case-study on the viability of so-called firmware modification attacks that can be performed as a result of first compromising a device. In contrast to "modifications" induced by malware such as Mirai (see, *e.g.,* [39]) that do not survive device reboots, they consider permanent modifications. Their case-study details an attack upon the HP remote firmware update functionality present within HP LaserJet printers. The vulnerability enables firmware updates to be triggered on the printer by sending specific commands as a print job. To perform their firmware modification, the authors first obtain the original firmware of the device by extracting it from the device's SPI flash chip; they then reverse-engineer and modify it. Finally, they show the device's firmware can be reflashed with their modifications through the use of the aforementioned vulnerability. While in this case-study the authors show that such a modification is possible due to the exploitation of a firmware vulnerability, such a modification attack could just as easily be facilitated by a device backdoor.

Brocker et al. [62] perform a firmware modification attack upon the integrated webcam present within MacBooks. In this case, they modify the firmware to covertly monitor users. The process of implementing their modifications is similar to the attack described by Cui et al. [79]: they first locate a vulnerability that permits code execution with sufficient privileges to reflash the device's firmware and then perform exploitation of that vulnerability in order to reflash the firmware. Halperin et al. [100] investigate the security of implantable cardioverter-defibrillators (ICDs) and show that it is possible to modify the firmware of such devices remotely.

### 3.3.1 Weird Machines

Andriesse et al. [47] propose a general method for devising covert trigger-based malware. They describe trigger-based malware as a construct that is designed to execute only when a specific condition is satisfied – in essence, a backdoor. They design their technique with the aim that malware implemented using their methods will be difficult to analyse using both static and dynamic analysis. In order to thwart static analysis, they propose dividing the malware *payload* into fragments and distributing them within the binary. More specifically, they embed them at offsets such that through the standard disassembly process, they will be interpreted as other instructions. In order to execute such malware, the authors propose using a program bug, that when exploited causes control-flow to divert to the first fragment, *i.e.,* its *entry-point*; the remainder of its execution then proceeds by jumping from fragment to fragment. Their use of a program bug in this way serves to thwart dynamic analysis. The authors assert that even if the bug is found, an analyst without knowledge of the correct *trigger*, *i.e.,* the initial address of the first fragment of the *payload*, will be unable to state that the bug is intentional, or locate the *payload* fragments.

Wang et al. [181] suggest a similar scheme to [47]. That is, they deliberately insert vulnerabilities into otherwise benign applications in order to facilitate the *delayed* execution of malicious payloads. Their proposal differs in that they do not hard-code their *payload* component into the binary. They instead reuse components of the binary as in a more traditional vulnerability exploitation scenario. Unfortunately, such backdoors (without any knowledge of the developer's intention) are difficult to reason about and define concretely. While a binary containing a deliberately inserted bug is indeed a backdoor, without knowledge of the intent of the application developer, proving this is indeed the case is impossible. We address the issues presented in this work and that by Andriesse et al. [47] in detail in Chapter 4.

Tan et al. [172] present a technique for encoding so-called "bugdoor" functionality within embedded device firmware. They first describe a *new* programming model – Interrupt Oriented Programming (IOP) – which is based upon the side-effects produced by triggering interrupts. They demonstrate how these side-effects can serve as primitives for basic computation, and then demonstrate how they can be combined to implement a "bugdoor" program. By the definitions we propose in Chapter 4, a "bugdoor" is essentially a backdoor, which, as a result of its implementation, is arguably deniable. Deniability is achieved by using program bugs to implement the backdoor components, which serve to mask the intention of the implementer. This work and the two previously mentioned approaches represent types of backdoor implementations using so-called *weird machines* as a basis.

A further body of literature considers the re-use of application components for various purposes: these relate to the techniques explored within [47], when considered in a malicious context. Therefore, they can be seen as means of implementing backdoor-like software components, and we cover them for a thorough treatment of the area. Lu et al. [127] employ steganography based upon Return Oriented Programming (ROP) (see, *e.g.,* [152]). Their system, RopSteg, takes as input a program and a subset of its instructions which should be hidden. They hide the instructions by first completely removing them from the program, they then locate byte-string prefix matches of their byte representations within the modified program and use the offsets of those matches as starting points for inserting so-called *ineffective* instruction sequences. These sequences form ROP-gadgets, that when combined, are semantically equivalent to the instructions removed. To transition to the embedded ROP chain implementing their "hidden" functionality, they insert additional control-flow instructions. Ma et al. [128] also use embedded ROP-gadgets that preserve semantic meaning, but for the purpose of software watermarking. While Tang et al. [173] assess the effectiveness of code-reuse techniques to distribute "hidden code" amongst a binary in the form of ROP-gadgets for the purpose of code obfuscation.

Bratus et al. [60], describe the notion of *weird machines.* They describe a weird machine as something that amounts to a programmable machine embedded within another system (often a program). Using the case of program exploitation as an example, they state that an exploit serves as a proof of existence of such a weird machine. Such a machine is executed by specially crafted input data; in the situation where an exploit is created through ROP, the exploit makes a *processor* out of the fragments of the program it reuses, and these fragments act as the *instructions* of that processor. The authors detail how a weird machine can be constructed generically; this involves first identifying components of a target platform that allow manipulation of its internal state via its input; then forming primitives out of those components, such that programs can be formed by chaining them together. As outlined in the preceding paragraphs, weird machines can provide primitives for constructing components of backdoors.

Oakley et al. [139] explore embedding latent computation within the DWARF exception handling mechanism, which can be found in all GCC compiled executables that are exception-aware. The DWARF mechanism is essentially a bytecode-driven virtual machine which is invoked when an exception is triggered. The authors show that bytecode for that machine is capable of general computation and as the bytecode is stored in neither the data nor code sections of a binary, conventional static and dynamic analysis tools will be ineffective in analysing it. To demonstrate the effectiveness and expressiveness of the DWARF mechanism, the authors demonstrate a tool, Katana, which is able to automatically embed programs written in so-called “Dwarfscript” into standard ELF executables. They demonstrate a backdoor implemented using their tool, in which an ELF binary is modified such that when an exception is triggered, it launches a shell, as opposed to running an exception handler.

Shapiro et al. [164] present a similar mechanism for hidden computation, in this case, embedded within ELF meta-data. The programs they implement in this computation model are interpreted by the runtime loader (RTLD), and are formed by purposefully crafted relocation entries and symbol information. The authors demonstrate a tool, Cobbler, which is able to automatically modify an ELF binary to contain arbitrary programs that can be interpreted by the RTLD. As with the implementation strategy discussed in [139], if malicious code is embedded through ELF meta-data, standard analysis tools will be unable to analyse it. Additionally, a program implemented using this mechanism does not need to be transitioned to from standard code, rather it will automatically be executed by the RTLD. The authors describe the high-level implementation of a backdoor embedded within a standard networking utility (`ping`); their implementation uses a total of nine relocations entries, one symbol table entry, and makes no changes to the code sections of the modified binary.

Bangert et al. [52] demonstrate another so-called weird-machine, which is constructed through specifically crafted page-faults. They demonstrate how such a weird machine can be realised without executing any CPU instructions. The authors show that a Turing-complete execution environment is present within the IA32 architecture’s interrupt handling and memory transition tables. During a page-fault the processor becomes “trapped”, and in order to resolve the fault, hard-coded logic is performed; by chaining together multiple faults, it is possible to utilise that logic in order to perform arbitrary computation. Schuster et al. [161] present another computation model, Counterfeit Object-oriented Programming (COOP), which is similar to that of ROP. Similar to ROP, COOP performs code reuse, however, does so by chaining C++ virtual function calls. The authors demonstrate that in realistic attack scenarios, COOP is Turing complete and therefore forms a weird machine capable of general computation. Dolan [82], demonstrates a similarly Turing-complete weird machine, which can be programmed using just the `mov` instruction found in x86-based ISAs.

Vanegue [180] makes steps towards a formal definition of a weird machine. They use the notion of weird machines to model untrusted code execution. The existence of such weird machines and the possibilities they hold for implementing components of backdoors, emphasises that detecting backdoors is a problem bigger than analysing any single program or system using conventional methods. Similarly, it suggests that no single solution exists, and while components of backdoors can be identified individually, they are often not enough to definitively constitute what is informally agreed upon as a backdoor.

While Bratus et al. [60] discuss the relationship between exploits weird machines in an informal manner and Vanegue [180] makes an attempt at a formal definition for a weird machine, Dullien [86] addresses the problem of formalising the term *weird machine*, the relationship between exploitation and weird machines, and introduces the concept of provable exploitability. Dullien argues that it is possible to model a program or system using a so-called Intended Finite-State Machine (IFSM) and in doing so, view a piece of software as an emulator for a specific IFSM. He then shows that it is possible to create security games to argue about the security properties of a program or system by reasoning about it at the level of the states and state transitions of its IFSM. Since an IFSM only models an ideal system and is not a program that can be executed, Dullien proposes a mapping from states of the IFSM to states of a concrete CPU. As this mapping from IFSM states to CPU states (and hence transitions) may not be one-to-one, the CPU's possible states can be thought of as: sane states – which correspond to IFSM states, transition states – which represent the emulator transitioning between valid IFSM states, and weird states, which fall into neither of the previous two definitions. If the emulator for a program can somehow be forced to transition to a weird state by program input, then a new computational device, or weird machine can be realised. This weird machine can then be programmed using instructions which are formed as result of the IFSM and the emulator.

## 3.4 Malware Detection

As well as being seen as intentional program vulnerabilities, backdoors can be viewed as a form of trigger-based malware [47]. Malware has traditionally been detected using signature-based approaches. These signatures are often derived from a program's instruction sequences [157], or based on so-called *n-grams* [158] – which are contiguous sequences of features, such as API calls. While these signatures are generally scalable and as a result can be used for large-scale analysis, they tend not to be resistant to binary modifications [141], such as obfuscation or recompilation. Approaches for state-of-the-art malware detection use similar techniques to those for large-scale bug-search (*i.e.,* those described in §3.1.3), for instance, signature-based approaches combined with machine learning to achieve automated, scalable classification and clustering, and graph-based approaches combined with similarity metrics. A key difference between traditional malware and backdoors is how they compromise a system. A backdoor is almost always (intentionally) embedded within an otherwise legitimate program binary, whereas (binary-based) malware tends to manifest as a dedicated program or as a modification to third-party software (without the original developer's intent). Moreover, by definition, backdoors are activated by a trigger condition through interaction from a third-party, whereas traditional malware is typically always active.

Schultz et al. [159] were the first to apply machine learning to the problem of malware detection. They utilise features derived from both program code and data, such as properties related to shared library usage, for example, the number of function calls to imported functions and the number of functions in each imported library called, and string and byte-sequence usage. Rieck et al. [151] propose a framework for automated analysis of malware behaviours using machine learning. They use a combination of clustering to identify novel classes of malware that exhibit similar behaviour, and classification to associate malware to those classes. To obtain the behavioural properties of a given malware sample, they perform dynamic analysis to monitor the system calls it uses, and the arguments passed to those calls; they then use those properties to construct feature vectors. Using a combination of standard classification and clustering algorithms, they automatically generate behavioural classes and map malware to those classes.

Ye et al. [185] utilise a so-called hierarchical associative classifier (HAC) (see, *e.g.,* [129]) to detect malware. The authors use API features to train their classifier; they show that their classifier achieves a precision of 96.2405% and recall of 36.2606% on binaries it assigns a "confident" label (*i.e.,* the classifier is confident the binary contains malware, and it does actually contain malware), and 61.5% precision and 34.8442% recall on binaries it assigns a "candidate" label (*i.e.,* it may contain malware, and actually contains malware). Bayer et al. [54] use clustering to group malware samples whose behaviour is similar. They perform dynamic analysis to first obtain execution traces, then using those traces, generate so-called behavioural profiles which characterise the activity of the malware abstractly. Their machine learning approach uses hierarchical clustering (see, *e.g.,* [129]) to group malware which have similar behavioural profiles.

Nataraj et al. [136] use image processing techniques to classify malware. They observe that when visualised as greyscale images, malware of the same family share common visual properties. They use standard clustering techniques ($k$-nearest neighbours, with a Euclidean distance metric) to construct a set of clusters for *known* malware families, based on the properties of the greyscale image representations of the malware. On a large dataset of malware from known malware families, the authors achieve a classification accuracy of 97.18%. Shabtai et al. [163] propose using opcode byte n-gram patterns as features for classifying (and hence detection) of previously unknown malicious code. Moskovitch et al. [134] and Kolter et al. [120] employ related approaches, both with a differing feature representation – using just *n-grams* as opposed to *n-gram* patterns.

Tian et al. [177] perform automated malware classification using the printable strings extracted from a number of malware families. They construct a classifier by combining the outputs of a number of standard classification algorithms (to form a meta-classifier). This classifier is trained using only string data extracted from the binaries of their training set. The resultant classifier is shown to be effective: achieving an accuracy of 97% from 5-fold cross-validation. Tian et al. [178] use function length information as features to address the same classification problem; they find their classifier performs with an accuracy of 87%. Islam et al. [109] use a combination of strings and function information (function lengths) to form features; as in [178] and [177], they perform classification of malware families; they find the combination of both strings and function lengths improves the classification accuracy over both of those approaches, yielding a classifier with an accuracy of 98.8%. Sami et al. [155] train a classifier to distinguish between malicious and benign binaries. Their classifier is trained using API call sets as features, where an API call set is essentially the set of all API functions that are called by a binary. To reduce the size (and hence dimensionality) of these sets, feature selection is applied to keep only those API calls that are most discriminating. On evaluation, they find their classifier achieves a detection rate of 99.7%.

Aside from machine learning, other approaches have been proposed for malware detection and identification. For instance, Park et al. [143] use maximal common sub-graph detection (see, *e.g.,* [57]) to perform malware classification. They use a so-called behavioural graph to represent a binary, which is obtained by executing the binary and observing the system calls it makes, as well as the arguments passed to those calls. Within the graphs, nodes represent system calls, and edges represent dependence relations between the arguments of those calls, for example, if two nodes, $n_1$ and $n_2$ are connected, then the arguments passed to $n_2$ have a dependence upon the arguments passed to $n_1$. To compute the similarity between two malware samples, they compute the maximal common sub-graph from the behavioural graph representations and use a distance metric (as a function of the maximal common sub-graph) to quantify the similarity between the two samples. Hu et al. [105] propose using function-call graphs to perform malware classification. They devise a graph similarity metric based upon graph-edit distance, which they approximate using graph matching implemented as a modification of the Hungarian algorithm (see, *e.g.,* [124]). They use a multi-level index to facilitate searching for similar malware over a large dataset.

Fredrikson et al. [93] propose a means to distinguish between classes of programs and apply it to malware behaviours. They derive so-called optimally discriminative specifications – which are constructed by extracting what the authors coin significant malicious behaviours from *known* malware. These behaviours are mined from sets of (pre-labelled) similar binaries (malware families). Specifications are essentially collections of behaviours with a characteristic function that defines one or more subsets of the binaries used to extract those behaviours. An optimal specification is one that discriminates the malicious and benign binaries of a set of samples with respect to a given threshold. Their tool, Holmes, which implements a detection scheme based upon optimally discriminative specifications, achieves a 86% detection rate against new, previously unknown malware, with no false positives.

## 3.5 Backdoor Detection

While techniques proposed for detecting both malware and vulnerabilities are strongly related to those for detecting backdoors, methods for specifically addressing the problem of backdoor detection are scarce in the academic literature. Zhang et al. [190] explore the notion of backdoor detection and give a first informal definition of the term backdoor. They define a backdoor as "a mechanism surreptitiously introduced into a computer system to facilitate unauthorised access to the system". This definition was made in their paper entitled *Detecting Backdoors*, published in 2000, and while the definition they provide is largely agreeable with current usage of the term backdoor, they consider backdoors only from a network security perspective. They propose an algorithm for detecting so-called *interactive* backdoors, which model a scenario in which an attacker interacts with a backdoor through inputting keystrokes. Their algorithm uses network monitoring as a basis for detection and analyses three types of network traffic characteristics: directionality, packet sizes and packet interval times – where directionality refers to the initiator of a network connection. Their algorithm attempts to detect keystrokes – which the authors argue are symptomatic of backdoor behaviour under certain circumstances. The authors further propose a number of protocol-specific algorithms for detecting backdoors in network traffic for a number of common services, such as, Telnet, SSH, Rlogin, and FTP.

Wysopal et al. [183] provide a taxonomy for backdoors, in which they detail three major types of backdoor. The first type, they call system backdoors, which involve either a single dedicated process which serves to compromise a system, *e.g.,* a rootkit, or an intentional misconfiguration of a legitimate service. The second type are so-called cryptographic backdoors, which compromise cryptographic primitives through deliberate selection of weak parameters or bad sources of randomness. The final class of backdoor they describe are application backdoors, which they state are versions of otherwise legitimate software modified to bypass security mechanisms under certain conditions. The authors provide a number of source-code oriented detection strategies focusing on specific types of application backdoors, which fall into the following categories: special credentials, hidden functionality, unintended network activity, and modification of security-critical parameters. To detect the use of special credentials (*i.e.,* hard-coded credential backdoors), the authors propose searching for static variables that *look* like usernames or passwords, for instance, variables that contain strings that are only made up of characters from the printable ASCII character set. To detect hidden functionality, they propose identifying static variables that *look* like application commands. For unintended network activity backdoors, the authors suggest identifying common network API usage, specifically where hard-coded IP addresses are used and analysing code-flows that include that usage for possible information leaks, *i.e.,* where information is sent over a network connection that should not be. To detect backdoors that manipulate security-critical parameters, they propose first identifying variables used to store such parameters and then suggest performing a systematic assessment of their usage. The detection methods the authors propose, while generally applicable for use by a human analyst reviewing source code, are high-level and cannot be automated easily. Moreover, they require expert domain knowledge and human intuition to perform effectively, *e.g.,* to identify strings that *look* like usernames. While machine learning can certainly attempt to mimic such intuition (*e.g.,* StringFighter [156]), the results of such approaches are neither well studied, nor considered by the authors.

Schuster et al. [160] address the problem of backdoor detection in binaries through the use of dynamic analysis. They define a backdoor as a “hidden, undocumented, and unwanted program or program modification/manipulation that on certain triggers bypasses security mechanisms or performs unwanted/undocumented malicious actions". Their detection method addresses two classes of backdoor: flawed authentication routines (*e.g.,* those that contain hard-coded credentials) and hidden commands and services. They restrict their detection approach to a subset of programs, specifically server applications, where the protocols handled by those applications are known and can be modelled prior to analysis. This limits their approach to binaries that perform relatively domain-specific functionality and handle only *known* protocols. To detect backdoors, they propose an algorithm, A-WEASEL, which learns a decision tree that models the possible execution paths a program takes in response to particular inputs. This algorithm works by interacting with a binary using a model of the protocol it implements, recording the execution traces of multiple, differing protocol runs, and using those traces to construct call graphs. These call graphs are combined to form a so-called combined call graph, which is, in turn, used to construct a decision tree. The authors use heuristics to identify crucial decisions within a decision tree (*i.e.,* those that handle authentication), these decisions are used to locate suspicious program paths, which are subsequently checked for backdoor-like properties using standard static and dynamic analysis approaches. The prototype implementation of their approach is able to identify a number of "artificial" and previously identified backdoors in programs for multiple device architectures.

Firmalice [166] is a tool developed using the angr framework [167] to detect authentication bypass vulnerabilities (backdoors) within embedded device firmware. The authors propose a model for a class of backdoors they coin authentication bypass vulnerabilities. They note that such backdoors tend to manifest in three distinct ways: intentionally hard-coded credentials, intentionally *hidden* authentication interfaces and unintended bugs (*e.g.,* shell-injection bugs through incorrect input handling). Firmalice uses the notion of a security policy, which is a definition of a state that a program can reach that is considered privileged. An authentication bypass vulnerability exists when it is possible to reach such a state without performing proper authentication. Firmalice uses symbolic execution aided by various simplification techniques (for efficiency) to attempt to prove it is possible to reach a privileged state – specified by a given security policy – from a supplied input state using constraint satisfaction. This proof will be a concrete program input, that satisfies all of the conditions required in order to reach the privileged state. In the case of a hard-coded credential backdoor, this would mean finding concrete values for the hard-coded credentials. To evaluate Firmalice, the authors use three embedded device firmware images known to contain backdoors; in all cases, Firmalice is able to find inputs that trigger those backdoors. It should be noted that Firmalice requires some level of manual intervention to perform its analysis: namely specifying the security policy – which can potentially require a degree of manual reverse engineering to achieve.

Papp et al. [142] propose techniques for semi-automated detection of trigger-based behaviours (*i.e.,* backdoor-like behaviours) in source code. They use concolic execution to identify code paths that can reach (potentially) malicious functionality, which they define as calls to `system`, `exec` and `send`. The prototype implementation of their approach is able to detect three out of the five backdoors they evaluate against.

BinPro [131] is a tool that attempts to reduce the manual effort required to perform binary audits of security-critical code. It operates on applications where both a binary implementation and its source code are available. The primary objective of BinPro is to identify where backdoors have been inserted into applications, assuming they do not exist within the available source code for the applications. BinPro uses graph-based matching to identify additional function call, and control-flow graph features present in a binary application, that are not present within its source code. BinPro is able to correctly match 74% of functions over a test set of 10 open source applications, which the authors claim reduces the effort of binary auditing by an average of 25% in relation to the number of functions to manually analyse.

## 3.6 Chapter Summary

In this chapter, we have explored works from both academia and the broader Information Security community, directly related to the topic of this thesis, *i.e.,* backdoor detection, and related fields: program analysis, vulnerability discovery, and malware detection. In doing so, we have found the need for a clear, rigorous definition of the term backdoor, as well as highlighted the distinct lack of methodologies for detecting backdoors. To serve as inspiration for developing new detection techniques, we have surveyed the state-of-the-art in the related fields of bug search and malware detection, which we find to largely use either heuristics or machine learning as the basis for large-scale analyses, or symbolic execution for more targeted, small-scale analyses. Finally, we have reviewed the fundamental difficulties in binary program analysis, as well as examined the implementations of complex, esoteric backdoors, both of which serve to expose the limitations of practical approaches for backdoor detection and the potential trade-offs required for their implementation.

# Chapter 4

## *Definitions, Deniability & Detection*

**Contents**

In this chapter, we present rigorous definitions for the terms backdoor and backdoor detection. We provide a framework for decomposing backdoors into their component parts, which allows us to model them concisely, and aids in both their identification and detection. Moreover, we provide definitions for intentional, deniable and accidental backdoors, and provide a means for reasoning about their implementations. Further, in order to demonstrate our framework and definitions, we provide an analysis of a number of real-world backdoors and a more complex backdoor from the literature. Finally, we show how our framework can be used as both a premise for developing backdoor detection methodologies, and to analyse existing approaches.

## 4.1 Motivation

Although the term backdoor is generally understood as something that intentionally compromises a platform, there has been little to no effort to give a definition that is more rigorous. To give such a definition is difficult, as backdoors can take many forms, and can compromise a platform by almost any means (see, *e.g.,* [183]); for example, a hardware component, a dedicated program or a malicious program fragment. This lack of a concrete definition prohibits reasoning about backdoors in a generalised way that is a premise to developing methods to detect them. Further hampering that reasoning – especially in the case of backdoors of a more complex, or esoteric nature – is the sheer lack of real-world samples. Documented real-world backdoors are generally simplistic, where their trigger conditions rely upon a user inputting certain static data: the most obvious example being hard-coded credentials. These have been studied in the literature with various tools providing solutions relying on varying degrees of user interaction [166, 174].

## 4.2 Contributions

The work presented in this chapter, provides first and foremost, a much needed rigorous definition of the term backdoor: which we view as an intentional construct inserted into a system, known to the system's implementer, unknown to its end-user, that serves to compromise its perceived security. We propose a framework to decompose and componentise the abstract notion of such a backdoor, which serves as a means to both identify backdoor-like constructs, and reason about their detection.

Many backdoors found in the real-world fall into a grey area as to whom is accountable for their presence; to address this, we define the notion of deniability. We model deniability by considering different views of a system: that of the implementer, the actual system, and the end-user; this allows us to – depending on where backdoor-like functionality has been introduced – reason about if that functionality is a deniable backdoor, accidental vulnerability, or intentional backdoor. In many cases, attempting to model this intention, or the lack thereof, is something that is social or political; thus, we do not address such cases in this work. Instead, we focus on the technical aspects of a backdoor-like functionality.

We show that under our definitions, many backdoors publicly identified are not deniable and thus, their manufacturers should be held accountable for their presence. Aside from manual analysis, little work has been performed to address the detection of backdoors. We perform a study of both academic and real-world backdoors and consider existing methods that can be used to locate backdoor components, as well as how those methods can be improved. To summarise, the contributions of this work are as follows:

1. We provide a rigorous definition for the term backdoor and the process of backdoor detection.

2. We provide a framework for decomposition of backdoor-like functionality, which serves as a basis for identifying such constructs, and reasoning about their detection.

3. We express the notion of deniable backdoors by considering different views of a system: the developer's perspective, the actual system, the end-user, and a user analysing the system.

4. We show examples of both academic and real-world backdoors expressed in terms of our definitions, and reason about their deniability and detectability.

5. We demonstrate how our framework can be used to reason about backdoor detection methodologies, which we use to show that current state-of-the-art tools do not

consider a complete model of what a backdoor is, and as a result, we are able to identify limitations in their respective approaches.

## 4.3 Nomenclature & Preliminaries

In the remainder of this chapter, we reference terms and initialisms described in this section. A *platform* represents the highest level of abstraction of a device that a given backdoor targets. We define a *system*, which we say is the highest level of abstraction required to model a given backdoor, within a *platform*. More specifically, a *system* – for the purposes of our discussion – is the implementation target of a given backdoor. Since a backdoor can be implemented at any level of abstraction of the *platform* it is designed to compromise – for example, as a dedicated program, a hardware component, or embedded as part of another program – we attempt to abstract away from such details. To do this, we instead focus on an abstract *system*, which we model as a finite state machine (FSM).

When considering a backdoor, there are two perspectives to consider a *system* from: that of the entity that implements a backdoor, and that of the end-user – which could, for example, be a general consumer, or a security consultant analysing the *platform*. To model this situation, we consider four versions of the FSM; for any given *system*, the *Developer* FSM (DFSM) refers to the developer's view of the system, the *Actual* FSM (AFSM) refers to the FSM that models a real manifestation of the system, *i.e.,* a program, the *Expected* FSM (EFSM) refers to the end-user's expectations of the *system*, and finally, the *Reverse-engineered* FSM (RFSM), represents a refinement of the EFSM obtained by reverse-engineering the *actual* system.

Each state of the FSM describing a *system* can be viewed as an abstraction of a particular functionality – which, in turn, can be modelled using a FSM. Thus, we view an entire *system* as a collection of sub-systems, which can be visualised in a layered manner – with each layer representing a view of a part of the *system* at an increasing level detail,

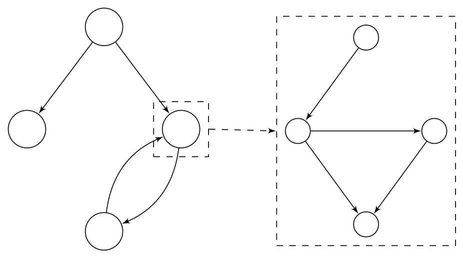


Figure 4.1: Multi-layered *system* FSM.

as in Figure 4.1.

As a concrete example, if, for instance, a given backdoor compromised a router, then we would refer to the router as the *platform*. If the backdoor was implemented in software, as a dedicated program, we would then view the highest level of abstraction, referred to as the *system*, as the interactions between the processes of the operating system, modelled as a FSM. Each individual running program, or process, would, in turn, be modelled in further detail by arbitrary levels of FSMs.

### 4.3.1 Analysis and Formalisation of FSMs

We specify a FSM as a quintuple: $\theta = (S, i, F, \Sigma, \delta)$, where: $S$ is the set of its states, $i$ is its initial state, $F$ is the set of its final states, $\Sigma$ is the set of its state transition conditions, *e.g.,* conditional statements that when satisfied cause transitions, and $\delta : S \times \Sigma \to S$ represents its state transitions, by means of a transition labelling function.

Inspired by the approach taken by Dullien [86], we view the implemented, or *real* system modelled by a FSM as an emulator for the actual FSM (AFSM) of the system. Thus, when the user's EFSM and the AFSM are not equivalent, *e.g.,* the user assumes there is no backdoor present, when there is, specific interactions with the real system will yield unexpected behaviour. How this unexpected behaviour manifests is what determines if that unexpected behaviour means that the system contains a backdoor. Different users of the system will assume different EFSMs. In order to analyse a system, a program analyst, for example, will derive a RFSM – which, for notational ease, we refer to as

$\theta_R$ – by reverse-engineering the *real* system; they do this by making perceptions and observations of its concrete implementation, *i.e.,* the emulator for $\theta_A$. What the analyst will observe is a set of states and state transitions, which are a subset of all those possible within the *platform*, *e.g.,* CPU states. To analyse these states, and hence, derive $\theta_R$, the analyst will require a means of mapping concrete states and state transitions of the *platform* to the level of abstraction modelled by the states and state transitions of their FSM. To perform analysis, we assume that an analyst has the following capabilities:

1. They have access to the emulator for the actual FSM ($\theta_A$) – in the case of software, this would be the program binary.

2. They are able to perform static analysis upon the emulator, *i.e.,* using a tool such as IDA Pro, and hence, *perceive* a set of system states and state transitions between those states of the real system.

3. They are able to perform dynamic analysis of the system, *i.e.,* with a debugger, and hence, *observe* a set of system states and state transitions of the real system.

The perceptions and observations of the analyst, along with a means to map concrete states and transitions to abstract FSM states and transitions, allows them to construct a RFSM ($\theta_R$) from the emulator for a given AFSM.

### 4.3.2 Backdoor Definition

The implementation strategies of backdoor implementers can vary widely; therefore, we consider the notion of an abstract backdoor, which we then decompose into component parts. In order to do this, we attempt to answer a number of questions: what is it that makes a collection of functionalities, when interacting together, manifest as a backdoor? What abstract component parts can be found in all such backdoors? To what extent do we need to abstract to identify all such components?

A distinguishing feature of all backdoors is that they must be triggered. Thus, a pivotal component of any backdoor is its *trigger* mechanism. However, this *trigger* mechanism alone does not constitute a backdoor: what causes it to become active? Another component is needed to account for the satisfaction of the *trigger* condition: *i.e.,* a type of *input source.* Upon *trigger* activation an eventual system state is reached that can be considered the *backdoor-activated* state, *i.e.,* a *privileged state.* To reach this final state, an intermediate component that facilitates the transition from the *normal* system state upon satisfaction of the backdoor *trigger* to the *backdoor-activated* state is required: we refer to this as the backdoor *payload.* By this reasoning, there are four components (and the transitions between them) required to capture the notion of a backdoor. Thus, we define a backdoor as:

*Definition 4.1.* ***Backdoor*** An *intentional* construct contained within a system that serves to compromise its expected security by facilitating access to otherwise privileged functionality or information. Its implementation is identifiable by its decomposition into four components: *input source*, *trigger*, *payload*, and *privileged state*, and the intention of that implementation is reflected in its presence within the DFSM and AFSM, but not the EFSM of the system containing it.

### 4.3.3 Backdoor Detection

Using Definition 4.1 as a basis, a backdoor can be modelled as two related FSMs: $\theta_{trigger}$, which represents the *trigger* without a state transition to the *payload*, and $\theta_{payload}$, which represents the *payload* and $F_{payload}$, the set of possible *privileged states.* Thus, we can define backdoor detection as:

*Definition 4.2.* ***Backdoor Detection*** A backdoor is detected by obtaining:

$$\theta_R = (S_R, i_R, F_R, \Sigma_R, \delta_R)$$

Where, within $\theta_R$, the states and state transitions of both the *trigger* and *payload* must exist:

$$\Sigma_{trigger} \cup \Sigma_{payload} \subseteq \Sigma_R$$
$$\forall s \in S_{trigger}, \forall \sigma \in \Sigma_{trigger}.\ \delta_{trigger}(s, \sigma) \neq \bot \implies \delta_R(s, \sigma) = \delta_{trigger}(s, \sigma)$$
$$\forall s \in S_{payload}, \forall \sigma \in \Sigma_{payload}.\ \delta_{payload}(s, \sigma) \neq \bot \implies \delta_R(s, \sigma) = \delta_{payload}(s, \sigma)$$

The *privileged states* reachable as a result of the *payload* are either final states of $\theta_R$, or states that can be transitioned from to some state of $\theta_R$:

$$S_{trigger} \cup S_{payload} \subseteq S_R$$
$$\forall f \in F_{payload}.\ f \in F_R \lor (f \notin F_R \implies \exists \sigma \in \Sigma_R.\ \delta_R(f, \sigma) \in S_R)$$

The *payload* must be reachable from the *trigger*, and there must exist a transition to the *trigger* within $\theta_R$:

$$\forall f \in F_{trigger}.\ \exists \sigma \in \Sigma_{trigger}.\ \delta_R(f, \sigma) = i_{payload}$$
$$\exists s \in S_R, \exists \sigma \in \Sigma_R.\ \delta_R(s, \sigma) = i_{trigger}$$

## 4.4 A Framework for Modelling Backdoors

In this section, we detail a framework for decomposing a backdoor into the four components defined in §4.3.2; we exhaustively enumerate the types of these components, which allows us to both identify and reason about them.

In addition to locating a construct consisting of an *input source*, *trigger*, *payload*, and *privileged state*, to detect a backdoor, an analyst must demonstrate that the construct would be part of the DFSM of the system. For open-source software, this could be done by analysing the source code version control logs, or in closed-source software, analysing the differences between software versions. In other cases, where such analysis is not possible, the following framework can serve as a basis for reasoning about how a backdoor's components can indicate an implementer's intent.

In the proceeding framework, we refer to the RFSM of an end-user that has analysed a particular *system*. Initially, that user will expect functionality that can be modelled by one FSM (their EFSM), and through their analysis, they will learn, or derive another FSM (RFSM) that matches what they have learnt about the *system*. Therefore, to discover a backdoor, the user will perform a process of refinement on their original EFSM (pre-analysis), such that their RFSM (post-analysis) will contain a backdoor if there is one present in the AFSM, and they are able to identify it.

From the process of refinement, new states and state transitions will be added to the RFSM. We divide these states and state transitions into two categories: those that are explicit, which we say are *discovered* (and always exist within the AFSM) and those that are not explicit, which we say are *created* (and may not exist within the AFSM). To serve this distinction with an example, suppose we have a RFSM that models a program. The explicit states and state transitions that are added to it through analysis are those that represent basic blocks and branches that are *explicitly* part of the program's code (and will always be part of the DFSM and AFSM). Those that are added that are not explicit are in a sense *weird* states and state transitions, which might, for example, be the states representing some shellcode.

## 4.4.1 Input Source

If we model the satisfaction of a backdoor *trigger* as a function – `trigger_decision` – as in the state machine diagram in Figure 4.2, then we can view it as a function that takes at least one parameter (implicit or otherwise) – an *input source* – which is used to decide which state transition that is made as a result of executing that function.

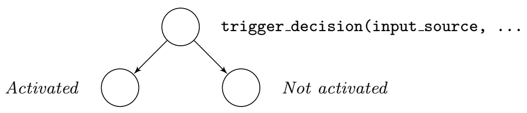


Figure 4.2: Idealised Backdoor Trigger.

The value yielded by the *input source* may be derived from any number of inputs to the FSM: it could be a string input by an attacker wishing to activate the backdoor *trigger*, or it could be the value of the system clock such that during a specific time period the backdoor *trigger* becomes active. For this reason, we choose to abstract away from the exact implementation details, and use the term "*input source*" to represent this component of the backdoor. Note, that the *input source* is not the value that causes the activation of the backdoor *trigger*, but rather describes the origin of that input: *e.g.,* a network socket, the standard input, the system clock, and so on.

## 4.4.2 Trigger Mechanism

The backdoor *trigger*, under the correct conditions, will cause the execution of the backdoor *payload*, which will subsequently elevate the privileges of the attacker. We model the backdoor *trigger* as a boolean function where its positive outcome, *i.e.,* when it outputs *true*, will cause a state transition to the backdoor *payload*. The way the FSM transitions to the *payload* as a result of the satisfaction of the *trigger* conditions can be modelled exhaustively with two cases:

1. The state transition is explicit, hence, will always exist within the backdoor implementer's DFSM. The backdoor *trigger* is added to the RFSM by adding the explicit states and transitions related to satisfying the backdoor *trigger* conditions, and adding one or more transitions which transition to the *payload*, where those transitions are *discovered* (not newly created) as part of the analysis.

```
_Bool vulnerable_auth_check( \
  const char *user, const char *pass) {
  char buf[80], hash[32];

  strcpy(buf, user); strcat(buf, pass);
  create_user_pass_hash(hash, buf);

  return check_valid_hash(hash);
}
```

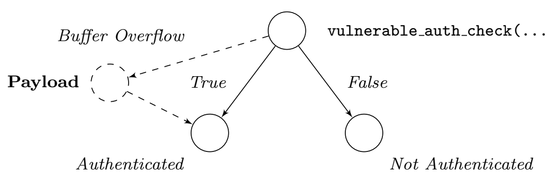


Figure 4.3: Bug-based backdoor *trigger*.

2. The state transition is not explicit. The *trigger* is added to the RFSM by adding explicit states and state transitions related to satisfying the backdoor *trigger* conditions, and by *adding* one or more state transitions that transition to the *payload*, where those transitions are newly *created* as part of the analysis, *i.e.,* they are not explicit.

To visualise these cases, we use concrete examples in which we use a *system* that is a single program, where the backdoor is embedded as part of the program.

In the first case, we view a *trigger* that is obvious and explicit, where the backdoor is encoded within a single function of the program. This case is shown in Figure 4.2. The backdoor *trigger* is comprised of the single state required to satisfy the backdoor *trigger* conditions, *i.e.,* the one labelled with `trigger_decision(...)`, and the state transition to the *Activated* state. In a more realistic scenario, the backdoor *trigger* mechanism may require satisfaction of multiple branch conditions and/or execution of multiple basic blocks and might be obfuscated. Irrespective of these implementation details, the core concept is the same: the collection of checks can be viewed as a single function, whose outcome is used to decide if the backdoor *payload* is transitioned to and hence, executed or not, where the transition – a CFG edge in this example – is explicitly part of the FSM.

While the first case considers conditions that are satisfied within a *valid* function CFG, and a transition to the *payload* which is contained entirely within that same *valid* CFG, and thus, constitutes *normal* control-flow, the second case of backdoor *trigger* manifests as *abnormal* control-flow. Within a program, we can think of such a construct as akin to

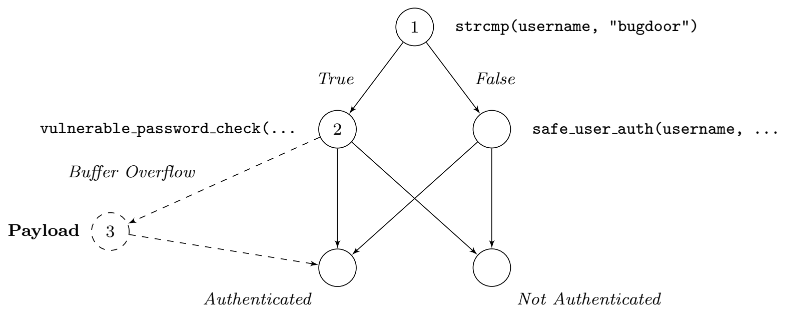

1
strcmp(username, "bugdoor")
True
False
vulnerable_password_check(...
2
safe_user_auth(username, ...
Buffer Overflow
Payload
3
Authenticated
Not Authenticated

Figure 4.4: Hybrid bug-based backdoor *trigger*.

a program bug that allows control-flow hijacking. One can conjecture a simple case for this being a buffer overflow vulnerability, that when exploited correctly, causes a program to transition to a backdoor *payload*, shown in Figure 4.3.

Alongside these basic cases, a more complex example of a backdoor *trigger* is one that relies both on explicit checks and a bug, as visualised in Figure 4.4. In this case, a hard-coded credential check against a specific username (`bugdoor`) is used to *guard* access to a vulnerable password check (`vulnerable_password_check`). A username other than `bugdoor` will cause the standard authentication routine (`safe_user_auth`) to be executed, and only a password with a long enough length (and specific content) will lead to the execution of the backdoor *payload*. In this example, the backdoor *trigger* is comprised of the explicit states 1 and 2, and the non-explicit state transition between states 2 and 3, *i.e.,* the *payload* state.

Note, that to make the case that all vulnerabilities are backdoor *trigger* mechanisms is an oversimplification, and obviously false claim, as such a simplification does not differentiate between accidental and intentional program bugs. We discuss the implications of bug-based backdoors in §4.5.

### 4.4.3 Payload

A backdoor *payload* can be viewed as the solution to a puzzle: *i.e.,* how to reach a *privileged state* from successfully satisfying the conditions of a *backdoor trigger*. In our model, we represent this by the state transition taken in order to reach a *privileged state*, and any additional states and state transitions that perform prerequisite computation following activation of the backdoor *trigger*. In practice, a *payload* component can take many forms; however, we can exhaustively categorise all types of *payload* by how they are modelled as part of a RFSM, and how they are transitioned to:

1. The transition to the *payload* is explicit, and does not permit the creation of new states and state transitions (see Figure 4.5). The *payload* is added to the RFSM by adding explicit states and transitions required to reach a *privileged state*, where those states and transitions are *discovered* by analysis (explicit). They will be contained in the backdoor implementer's DFSM.

2. The transition to the *payload* is explicit, but state(s) reachable due to this transition permit the creation of new states and transitions, *e.g.,* a system that contains an intentional interpreter which can be accessed via a backdoor (see Figure 4.6). The *payload* is added to the RFSM by adding *discovered* (explicit) states and transitions – which exist in the backdoor implementer's DFSM – from which both newly *created* (non-explicit) and *discovered* (explicit) states and transitions can be reached, which facilitate the eventual transition to a *privileged state.* The non-explicit states and transitions added will not exist within the backdoor implementer's DFSM.

3. The transition to the *payload* is not explicit (bug-based), and the *payload's* states and transitions will either be explicit or non-explicit, *e.g.,* a ROP-based construct. The *payload* is added to the RFSM by adding both newly *created* (non-explicit) and *discovered* (explicit) states and transitions, which facilitate the transition to a *privileged state.* The non-explicit states and transitions added will not exist within the backdoor implementer's DFSM.

#### 4.4.3.1 Payload examples

To give concrete examples of the variants of backdoor *payload*, we once again demonstrate backdoors that are implemented within programs.

```
/* Trigger; if active then: (1) -> (2) */
if (strcmp(user._name, "backdoor") == 0) {
  /* Payload */
  user._is_admin = true;  // (2)

  /* Transition to privileged state */
  open_shell(&user); // (3) -> (4)
}
```

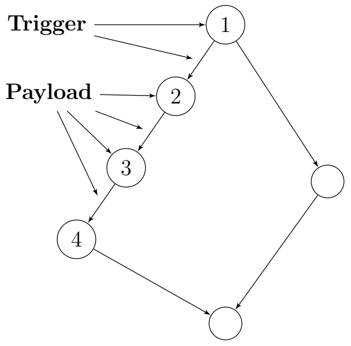


Figure 4.5: Explicit transition to *payload*, where *payload* has explicit components.

**Explicit transition to payload with explicit payload components** This class of *payload* (case 1 above) is inherently an intentional construct and requires no abnormal control flow for it to be executed. An example of a backdoor with such *payload* is shown in Figure 4.5. The backdoor *trigger* condition (state 1) is a hard-coded credential check, which if satisfied, will transition to the backdoor *payload* (transition from state 1 to 2). In the *payload*, the backdoor user's permissions are first elevated (state 2), and then a shell is opened for that user (state 3), which allows them to transition to the *privileged state* (state 4).

```
/* Trigger; if active then: (1) -> (2) */
if (strcmp(req._path, "/BKDRLDR") == 0) {
  /* Payload; req._data == payload input */
  run(&req._data); // (2) -> (3)
}
```

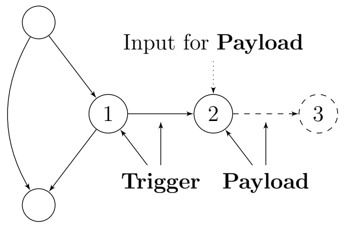


Figure 4.6: Explicit transition to *payload* comprised of explicit and non-explicit components.

**Explicit transition to payload with explicit and non-explicit payload components** In this case (case 2 above), we model a backdoor that enables an attacker to perform computation not part of the developer's DFSM, without being in a state that is bug-induced. An example of such a backdoor is shown in Figure 4.6; if the backdoor *trigger* is satisfied, the program will interpret and execute an input supplied by the user of the backdoor. The *trigger* condition is a check to see if a user is requesting access to a specific path (state 1), if it is, then the *payload* is transitioned to (state 1 to 2), where the data sent with the request (`req._data`) is used as input to an interpreter (state 2, via `run`). In this case, the *privileged state* (state 3) transitioned to is dynamically constructed as a result of the input to the interpreter executed in state 2.

```
void some_function() {
  char buf[80];
  /* ... */
  /* Backdoor activated if len(input)
     causes buffer overflow */
  strcpy(buf, input); // (1) -> (2)
  return;
}

void other_function() {
  /* ... */
  /* Payload reaches via (2) -> (3) */
  g_user._is_admin = true; // (3)
  open_control_panel(); // (4)
}
```

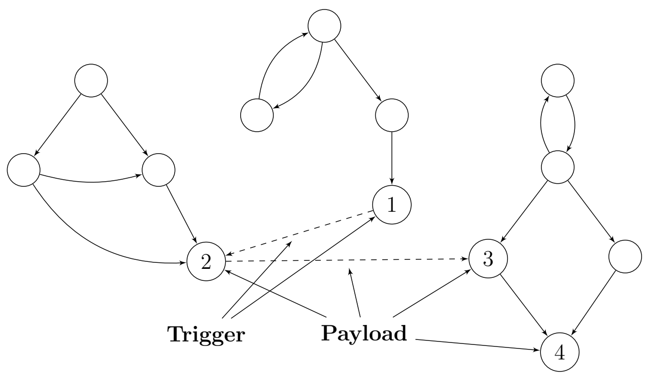


Figure 4.7: Non-explicit transition to *payload*, where *payload* has both explicit and non-explicit components.

**Non-explicit transition to payload with explicit and non-explicit payload components** In the final case (case 3 above), we model backdoors that have a *trigger* mechanism that is bug-based, *i.e.,* allows an attacker to perform computation not part of the developer's DFSM. We visualise such a case in Figure 4.7; here the *trigger* consists of an *intentional* buffer overflow bug in `some_function` (state 1), which if exploited – in this case with a ROP-based *payload* – transitions (via 1 to 2) to the *payload*. The *payload* consists of states 2 and 3, and the transitions from states 2 to 3, and 3 to 4. As a result of the *payload*, the user is granted administrative privileges (state 3), and entered into a (privileged) control panel via `open_control_panel` in `other_function` (state 4).

```
if (strcmp(password, "_BACKDOOR_") == 0 \
    || is_valid_password(password)) {
  // Authenticated
} else {
  // Not authenticated
}
```

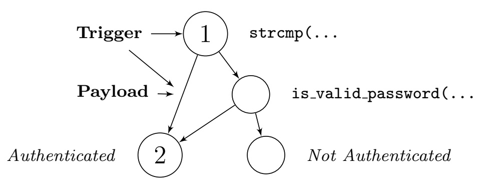


Figure 4.8: A backdoor *payload* composed solely of a state transition.

**Single transition payloads** We note there is a special case for both cases 1 and 3, namely, where the *payload* is composed of only a single state transition. That is, no additional computation is undertaken as part of the *payload*, rather the *payload* shares its state transition with the backdoor *trigger*, as shown in Figure 4.8. This special case accounts for situations where the backdoor *trigger* acts like a trapdoor (state 1), allowing an attacker to bypass a (potentially) more complex check for user-authentication, and rather provides a direct transition to a *privileged state* (the transition from state 1 to 2). The form of the *payload* is identical for cases 1 and 3, other than the explicitness of the state transition (the *payload*) between the *trigger* and the *privileged state*.

#### 4.4.3.2 Payload obfuscation

So far, we have not considered how a backdoor implementer might hide a backdoor's presence – other than by using a bug-based *trigger* mechanism. While such a trigger is simple to implement, it offers the implementer no control over how the backdoor will eventually be used; this control can be regained, by, for example, limiting the computational freedom of newly created states. In this section, we explore the means by which a backdoor implementer can permit obfuscated *payload* components.

Since backdoor *payloads* that contain only explicit states and state transitions are obvious and thus, intentional constructs, an obfuscated *payload* by nature must be implemented through the use of some degree of abnormal control flow, *i.e.,* non-explicit states and state transitions. An example of such a *payload* is one *derived* by reusing components of the system it is implemented within to obscure its execution, *e.g.,* for a program, from static analysis methods. From an attacker's perspective, the only way to execute such a backdoor is either to have prior knowledge of the *payload*, or *solve a puzzle* and derive it from the original system. Andriesse et al. [47] describe such a backdoor (examined in further detail in §4.6), whereby its *payload* component is composed of multiple code fragments embedded and distributed throughout a binary, which execute in sequence upon the backdoor being triggered. Figure 4.7 shows a naïve example of such a *payload*.

Another example is that where a *payload* can be derived from attacker controlled data. In the simplest case, this is akin to shellcode often executed as a result of the successful exploitation of a buffer overflow vulnerability: it shares a commonality that it does not rely upon any existing program components. In more sophisticated cases, such a backdoor *payload* might take a hybrid approach: where either user-data is interpreted by the program itself, or components of the program are used alongside the user input. Figure 4.6 shows a simple example of such a *payload*. In both of these examples, the *payload* components are implemented in a so-called *weird machine*, as defined by Oakley and Bratus [139].

### 4.4.4 Privileged State

Following successful activation of the backdoor *trigger* and subsequent transitioning from the associated *payload*, the system will enter into a *privileged state*. There are two possibilities for this state: either it can be reached under *normal* system execution, or it can only be reached through activation of the backdoor. If we consider *privileged states* by how they are added to a RFSM, then one that is newly created, *i.e.,* is non-explicit, will not be reachable under normal system execution, meanwhile, one that is explicit, may or may not be reachable under normal execution: for example, while the *privileged state* might be explicit, the only way to reach it might be via the backdoor *trigger*.

In the case of a *privileged state* reachable through normal execution, consider the backdoor presented in Figure 4.8 – which models a hard-coded credential check. The *privileged state* (state 2) of the backdoor is both reachable via the backdoor *trigger* (from state 1), and the state labelled `is_valid_password`.

For the other case, where the *privileged state* is not reachable by a *legitimate* user, it is essentially *guarded* by the activation of the backdoor. This case can further be subcategorised. The first variant is where the *privileged state* is explicit, as in Figure 4.5; the *privileged state* (state 4) is only reachable through activation of the backdoor *trigger* (state 1 and the transition from state 1 to state 2). In this example, the *privileged state* manifests as an undocumented *backdoor* shell, where after entering a specific username, an attacker is able to perform additional functionality, not otherwise possible. The other variant is a *privileged state* that provides an attacker access to functionality that is not available to a *legitimate* user, where that functionality does not explicitly exist within the system – as shown in Figure 4.6. Here the *privileged state* (state 3) is some function of attacker input, *i.e.,* the result of `run(&req._data)`.

## 4.5 Practical Detection & Deniability

While the previous definitions and framework focus on identifying the structure of backdoors, backdoor detection in practice will happen through, for example, manually reverse-engineering a program binary or observing a backdoor's usage through suspicious system events, such as anomalous network traffic. As is, our proposed framework oversimplifies as it does not model intention. If we knew that a particular vulnerability was placed *intentionally*, then there would be no question that the vulnerability was placed deliberately to act as a backdoor. Thus, in this section we answer the question: *if we have identified a backdoor-like construct, can we distinguish it from a vulnerability, and if so, how deniable is it?*

In order to make such a distinction, recall that we can view a system from four perspectives: its DFSM, AFSM, EFSM, and RFSM. If a backdoor-like construct has been identified, then it will be present in both the emulator for the AFSM and the RFSM. To state that the construct is a backdoor – and was placed intentionally – we must show that it, or some part of it, was present within the DFSM. In some cases, the intent is explicit and hard-coded in the implementation – *i.e.,* it leaves no ambiguity. The most obvious example of this is a hard-coded credential check which serves to bypass standard authentication. Indeed, all cases of backdoor that transition explicitly – *i.e.,* are discoverable by analysis – from the satisfaction of their *trigger* conditions to their *payload* can be considered intentional.

In the other case, where that transition is non-explicit, *i.e.,* bug-based, various approaches can be taken. For instance, in the case of software, where version control logs are available, it is possible to identify the exact point where a backdoor has been inserted, as well as its author. For binary-only software, where there exist multiple versions of that software, it is possible to identify the version the backdoor was introduced in, and reason about its presence by asking the question *was there a legitimate reason for making such a change to the software?* Further, we can consider the explicitness of the backdoor components: for example, if a code fragment exists within a binary that does nothing more than facilitate privilege escalation, and it is unreachable by normal program control-flow, then there is an indication of intent. A similar case can be made if the satisfaction of the *trigger* conditions rely on checks discoverable by analysis as well as a bug. Unfortunately, all of these approaches have non-technical aspects and rely on human intuition – thus, do not provide a concrete proof of intent. We are therefore left with three possible ways to classify backdoor-like constructs:

*Definition 4.3.* ***Intentional backdoor*** Those constructs that can be unambiguously identified as backdoors: the transition from their *trigger* satisfaction to their *payload* is explicit. Will be present in the DFSM, AFSM, and if found, the RFSM, but not the

EFSM.

*Definition 4.4.* ***Deniable backdoor*** Those constructs that fall into a grey area, where the transition from their *trigger* satisfaction to their *payload* is non-explicit, but from a non-technical perspective can be argued to be intentional. Will be present in the AFSM, if found, the RFSM, but not the EFSM; we cannot definitively tell if it is in the DFSM.

*Definition 4.5.* ***Accidental vulnerability*** Those constructs where there is no evidence – technical, or otherwise – to suggest any intent, and the transition from their *trigger* satisfaction to their *payload* is non-explicit. Will be present in the AFSM, and if found, the RFSM, but not the DFSM or EFSM.

From a purely technical perspective, a *deniable* backdoor will be indistinguishable from an *accidental* vulnerability. Consider a minimal example – a simple buffer overflow vulnerability and its corresponding exploit. If this vulnerability was deliberately placed, then it is a backdoor; otherwise, it is just a vulnerability coupled with an exploit. As we do not know anything about the implementer's intention, we are unable to discern between the two. Thus, a vulnerability can be seen as an unintentional way to add new state transitions or states to a system's FSM, while an exploit is a set of states and state transitions, such that when combined with a vulnerability within a given FSM, provides a means to compromise the believed security of the system modelled by that FSM. In contrast to backdoors and vulnerabilities, a construct providing standard privileged access will be intentional and manifest within the DFSM, AFSM, EFSM, and RFSM of a system.

## 4.6 Discussion & Case-studies

In order to demonstrate our framework, we provide a number of case studies. We show examples from both the literature, as well as real-world backdoors, which have been detected manually. For each backdoor, we reason about if and why its implementation

```
ngx_int_t ngx_http_parse_header_line(/* ... */) {
  u_char badc; /* last bad character */
  ngx_uint_t hash; /* hash of header, same size as pointer */
  /* ... */
}

void ngx_http_finalize_request(ngx_http_request_t *r, ngx_int_t rc) {
  uint8_t have_err; /* overlaps badc */
  void (*err_handler)(ngx_http_request_t *r); /* overlaps hash */
  /* ... */
  if(rc == NGX_HTTP_BAD_REQUEST && have_err == 1 && err_handler) {
    err_handler(r); /* points to hidden code, set by trigger */
  }
}

void ngx_http_process_request_headers(/* ... */) {
  rc = ngx_http_parse_header_line(/* ... */);
  /* ... */
  ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST); /* bad header */
}
```

Figure 4.9: Source-code listing for Nginx backdoor *trigger*.

can be considered deniable in respect to our definitions, and analyse it by performing a complete decomposition of its implementation using our framework. Finally, we provide a discussion of how our framework can be used to reason about methods for detecting backdoors.

Table 4.1 shows eleven real-world backdoors, each decomposed using our framework. As each backdoor can be modelled with explicit states and state transitions, by our definitions, none are not deniable. Thus, their implementers should be held accountable. The remainder of this section provides three in-depth case-studies of backdoors of varying complexity.

**Nginx Bug-Based Backdoor** Andriesse and Bos [47] describe a general method for embedding a backdoor within a program binary. Their technique utilises a backdoor *trigger* based upon an intentional program bug combined with a hard-coded *payload* composed of intentionally misaligned instruction sequence fragments. Their *payload* is, in a sense, obfuscated, yet fixed; its implementation exploits the nature of the x86 instruction set, where byte-sequences that represent instructions can be interpreted differently if they are accessed at different alignments, or offsets.

Table 4.1: Real-world backdoors modelled using our proposed framework.

| **Backdoor description** | **Input source** | **Trigger** | **Payload** | **Privileged State** |
|---|---|---|---|---|
| D-Link router backdoor "Joel's backdoor" [103]; bypass standard authentication by setting a specific user-agent when accessing web-configuration interface. | Network socket | `strcmp(ua, "xmlset_roodkcableoj28840ybtide") == 0` | Trigger conditions satisfied: explicit transition by matching with user-agent | Authenticated as legitimate user |
| Tenda router backdoor [102]; additional UDP server embedded within web-server allowing remote command execution. | Network socket; UDP port 7329 | Correct packet format: `strcmp("w302r_mfg", packet->magic) == 0` and `packet->command_byte == 'x'` | Trigger conditions satisfied; arbitrary command executed via: `popen(packet->command, "r")` | Dependent upon `packet->command` payload input |
| TCP-32764 router backdoor [36]; multiple vendors (Netgear, Cisco, Belkin, ...); remotely accessible undocumented service allows modification of configuration (*e.g.,* device credentials), and command execution. | Network socket; TCP port 32764 | Correct packet format: `"ScMM" \| 7 \| cmd_len \| cmd` | Trigger conditions satisfied (correct packet format with); executes `cmd` via `popen` | Dependent upon `cmd` payload input |
| Quanta LTE router backdoor [40]; dedicated UDP service "appmgr" if sent specific string enables an unauthenticated root shell via Telnet. | Network socket; UDP port 39889 | `strncmp("HELODBG", data, 7) == 0` | Trigger conditions satisfied: `data` matches `HELODBG`; starts Telnetd as root using: `system("/sbin/telnetd -l /bin/sh")` | Shell accessible via TCP port 23 |
| Sony IP camera backdoor [38]; combination of HTTP request with specific values set as parameters and hard-coded HTTP authentication credentials, can start Telnet service on device remotely via web-server. | Network socket; TCP port 80/443 | HTTP request to `/command/prima-factory.cgi` with parameters that satisfy `strcmp("cPoq2fi4cFk", param1) == 0` and `strcmp("zKw2hEr9", param2) == 0`; hard-coded username and password for HTTP authentication: `strcmp("primana", username)` == 0 and `strcmp("primana", password) == 0` | Trigger conditions satisfied; performs `system("/usr/sbin/inetd")` which starts `telnetd` using configuration in `/etc/inetd.conf` | Shell accessible via TCP port 23 |
| Ray Sharp DVR backdoor [174]; hard-coded credentials bypass standard authentication in web-interface. | Network socket; TCP port 80/443 | `strcmp("root", username) == 0` and `strcmp("519070", password) == 0` | Trigger conditions satisfied: explicit transition by matching credentials | Authenticated as root/administrative user |
| Western Digital My Cloud NAS backdoor [43]; setting specific value in (unencrypted) cookie when accessing web-interface allows user to login as administrator. | Network socket; TCP port 80/443 | `$_COOKIE["isAdmin"] == 1` | Trigger conditions satisfied; explicit transition via value being set correctly | Authenticated as administrative user |
| Q-See DVR backdoor [174]; multiple hard-coded username/password combinations, each combination gives access to additional functionality, as well as bypassing standard authentication. | Network socket/virtual keyboard | `strcmp(username, "admin") == 0` and `strcmp("6036huanyuan", password) == 0`; multiple possible passwords | Explicit transitions and states dependent upon password | Depends on password; authenticated as administrative user with greater privileges: alternate control-panel |
| D-Link router backdoor [35]; execution of arbitrary operating system commands through unauthenticated PHP script. | Network socket; TCP port 80/443 | POST request to `command.php`, with `cmd` parameter equal to command to run | Trigger conditions satisfied; explicit transition if parameter is set; executes command specified by `cmd` | Dependent upon payload input |
| Netis router backdoor [41]; custom network service (`igdmptd`) protected with a hard-coded password; enables (among other functionality) execution of arbitrary commands. | Network socket; TCP port 53413 | Authenticate to service using hard-coded credentials: `strcmp("netcore", password) == 0` | Trigger conditions satisfied; explicit transition to custom command shell; input arbitrary commands executed using `popen` | Dependent upon commands entered for payload input |
| 3S Vision N5072 camera backdoor [166]; hard-coded credential check for HTTP authentication, bypasses standard authentication. | Network socket; TCP port 80/443 | Authenticate to service using hard-coded credentials: `strcmp(username, "3sadmin") == 0` and `strcmp(password, "27988303") == 0` | Trigger conditions satisfied; explicit transition to authenticated state | Authenticated as legitimate user of device |

The authors demonstrate their approach by modifying the popular web-server, Nginx, and embedding a remotely exploitable backdoor. In their implementation, a would-be attacker provides a crafted input, which serves to satisfy the backdoor's *trigger* conditions; this input is provided as a malformed HTTP packet – its *input source* is thus, a network socket. Figure 4.9 provides a code listing adapted from [47] which contains the backdoor *trigger* conditions. Those conditions are: `have_err == 1`, and `err_handler != NULL`, which are set as a result of the use of uninitialised variables `have_err` and `err_handler` in the `ngx_http_finalize_request` function, which take the values of `badc` and `hash` in `ngx_http_parse_header_line`. The bug manifests due to the fact the two functions' stack frames overlap between their invocations. The intended *payload* states are meant to be those embedded as *weird* states, however additional states are possible, for example, if an attacker provides a different input packet to that expected by the implementers. The *privileged state* depends on the backdoor *payload.* We visualise the backdoor in Figure 4.10; the *trigger* is captured by state 1 and the non-explicit, bug-based transition to state 2; the *payload* consists of state 2 and the transition between state 2 and 3; while state 3 is the *privileged state.*

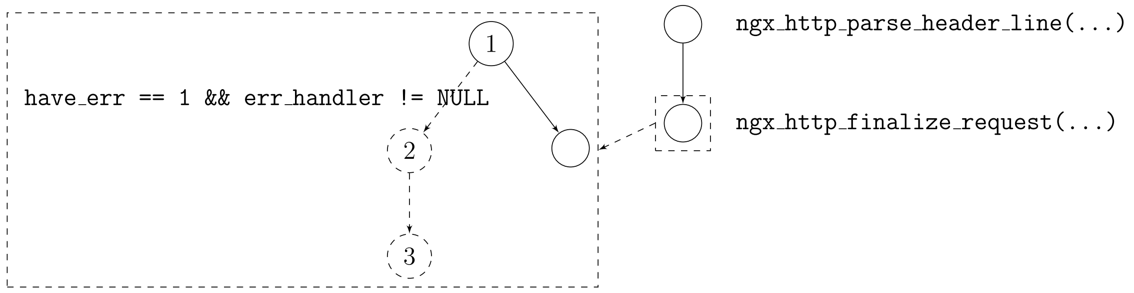
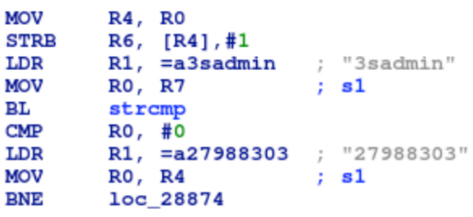

1
have_err == 1 && err_handler != NULL
2
3
ngx_http_parse_header_line(...)
ngx_http_finalize_request(...)

Figure 4.10: Multi-layered FSM for Nginx backdoor.

From a technical standpoint, the backdoor is deniable, *i.e.,* by definition 4.4 in §4.5; this is due to the backdoor *trigger* transition being bug-based, whilst the backdoor *payload*, if discovered, is arguably intentional. The componentisation using our framework allows us to visualise a complex backdoor succinctly, which would otherwise be buried across multiple functions in thousands of lines of source code. Further, its componentisation allows us to reason about how such a backdoor can be detected: for example, we could attempt to detect its bug-based trigger condition using symbolic execution; alternatively, we could heuristically attempt to identify its *payload* by scanning for misaligned instruction sequences that branch to other instruction sequences of the same kind, where the combination of those sequences would serve to elevate an attacker's privileges.

**3S Vision N5072 Camera Backdoor** The 3SVision N5072 IP camera contains a hard-coded credential backdoor [166]. To trigger the backdoor, an attacker must use specific credentials during HTTP authentication. In our framework, the *input source*, in this case, is a network socket. The backdoor *trigger* condition is composed of two checks: a comparison with the string `3sadmin` (for the username), and a comparison with the string `27988303` (for the password). If both of these checks are satisfied, then the attacker becomes an authenticated user of the device; Figure 4.11 shows a snippet of its disassembly. Figure 4.12 visualises the backdoor as a FSM; the backdoor *trigger* is represented by states 1 and 2 and state transitions between states 1 and 2, and 2 and 3; the backdoor *payload* is represented by the transition between states 2 and 3 (which manifests as a special case of an explicit *payload*, as described in §4.4.3), and the *privileged state* is represented by state 3 (which in this case is explicit).



```
MOV     R4, R0
STRB    R6, [R4],#1
LDR     R1, =a3sadmin     ; "3sadmin"
MOV     R0, R7            ; s1
BL      strcmp
CMP     R0, #0
LDR     R1, =a27988303    ; "27988303"
MOV     R0, R4            ; s1
BNE     loc_28874
```

Figure 4.11: IDA code snippet of N5072 credential backdoor.

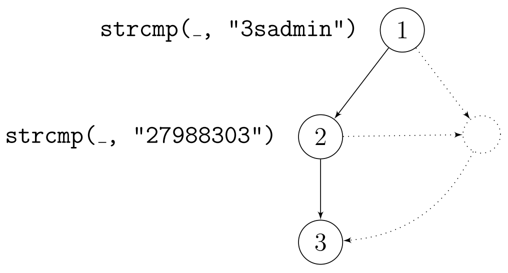


Figure 4.12: N5072 backdoor FSM.

The backdoor is not deniable; it is intentional – by definition 4.3 in §4.5 – as it can be modelled entirely by explicit states and state transitions. In this case, the backdoor implementer's intent is evident within the program binary, and they make no attempt to hide the presence of the backdoor, and so should be held accountable.

**Q-See DVR Firmware Backdoor** Thomas et al. [174] identify a hard-coded credential backdoor within the firmware of a Q-See DVR device. The device receives input either via its own virtual on-screen keyboard, or via a remotely accessible web-interface. In the latter case, the *input source* is a network socket. The IDA code snippet in Figure 4.13 shows part of the hard-coded credential authentication routine; the backdoor *trigger* consists of a comparison with the username `admin`, and a number of different hard-coded passwords; each password provides an attacker with a different type of backdoor access, not available to a legitimate user of the device. We can model each username/password combination as a separate backdoor, each with its own *payload* and *privileged state*. Figure 4.14 models one backdoor case. The backdoor *trigger* is modelled by states 1 and 2, as well as the transition between them, and the explicit transition to state 3; the *payload* is modelled by state 3 and the explicit transition between state 3 and 4, as the backdoor performs additional computation prior to reaching state 4, the *privileged state*.

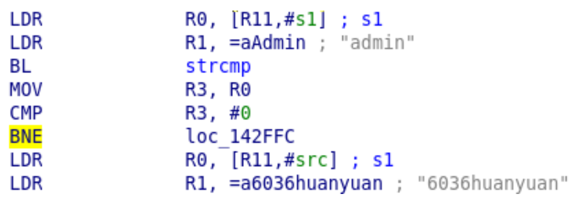

```
LDR     R0, [R11,#s1] ; s1
LDR     R1, =aAdmin ; "admin"
BL      strcmp
MOV     R3, R0
CMP     R3, #0
BNE     loc_142FFC
LDR     R0, [R11,#src] ; s1
LDR     R1, =a6036huanyuan ; "6036huanyuan"
```

Figure 4.13: IDA code snippet of Q-See credential backdoor.

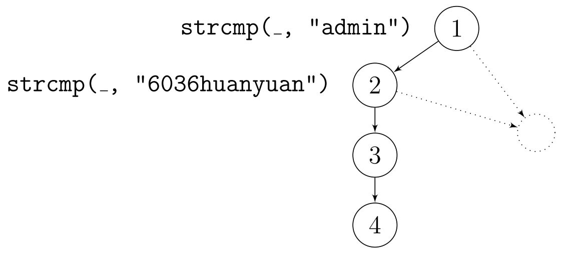


Figure 4.14: Q-See credential backdoor FSM.

As in the previous case, none of the backdoors in this device are deniable: they can all be modelled through explicit states and state transitions, and thus, the device manufacturer should be held accountable. Stringer – the tool described in Chapter 6 – is able to detect the backdoor by identifying the input required to satisfy the backdoor's *trigger* conditions.

### 4.6.1 Backdoor Detection Methodologies

Table 4.2: Tool detection methodologies decomposed using our framework.

| **Tool** | **Input source** | **Trigger** | **Payload** | **Privileged State** |
|---|---|---|---|---|
| Firmalice [166] | Partial | Partial | No | Partial |
| HumIDIFy [176] | Partial | No | Partial | No |
| Stringer [174] | No | Partial | Partial | No |
| Weasel [47] | No | Partial | Partial | Partial |

Our framework provides not only a means to reason about backdoors, but also backdoor detection techniques. Table 4.2 shows the decomposition of the detection methodologies of four state-of-the-art backdoor detection tools. Each tool claims to detect a particular subset of backdoor types. However, while these tools are all effective, none consider a complete model of backdoors, and, as a result, are limited in their effectiveness.

Firmalice [166] is designed to detect authentication bypass vulnerabilities. It uses a so-called security policy to define the observable side-effects of a program being in a *privileged state*. Using a specified *input source*, it attempts to find data provided via this *input source* that satisfies the conditions – *i.e.,* akin to a backdoor *trigger* – required to observe the side-effects specified by the security policy. Firmalice has no notion of a *payload state*; when entered, a *payload state* might leave a program in a *privileged state* that is not captured by a given security policy, for instance, where the *privileged state* reached by a backdoor user is different from that of a legitimate user reaches, *e.g.,* the Q-See DVR backdoor from the previous case-studies. Firmalice is able to detect such a *privileged state* by modification of the input security policy, however, to do so will require the same amount of manual analysis to detect the entire backdoor as it would to identify the *privileged state.*

HumIDIFy [176] – discussed in detail in Chapter 5 – aims to detect if a program can execute functionality it should never execute under normal circumstances. This might be the establishment of a suspicious *input source*, or the execution of API that is considered anomalous, *i.e.,* what might be part of a backdoor *payload*. However, since it does not consider the notion of a *trigger*, it is unable to distinguish between *abnormal* program behaviour that is benign – because it can only be performed by a legitimate user, and behaviour that is genuinely anomalous – that is part of a backdoor. Again, this is due to its approach not considering a complete model of a backdoor.

Stringer [174] – discussed in detail in Chapter 6 – attempts to detect static data used as program input that is responsible for either enabling authentication bypass vulnerabilities, or used for triggering the execution of undocumented functionality. To do this, it uses a scoring metric, which ranks static data, that when matched against, leads to the execution of unique functionality, *i.e.,* functionality not reachable by other program paths. Stringer considers the partial notion of a backdoor *trigger* and uses heuristics for identifying *payload-like* constructs. It does not consider the notion of *input source*, or *privileged states*, and as a result of the latter is unable to meaningfully score data that leads to states that are actually privileged higher than those that are not.

Weasel [47] detects both authentication bypass vulnerabilities and undocumented commands in server-like program binaries. It works by attempting to automatically identify so-called deciders (akin to backdoor *triggers*) and handlers (akin to the combination of a backdoor *payload* and *privileged state*), which then serve to aid in the detection of backdoors. Its approach does not fully model the notion of a backdoor; it does not consider an *input source* at all, rather, the approach models a single input for the program, and data from that source, when processed, is assumed to reveal all deciders and handlers. The Tenda web-server backdoor in Table 4.1 acts as an undocumented command interface; its *input source* is a UDP port; in this case, the backdoor uses a separate *input source* from the standard input to the program, *i.e.,* TCP port 80 or 443. Since Weasel does not capture the notion of an *input source*, it is unable to detect such a backdoor – not due to a deficiency in its detection method, but because it does not consider a complete model of a backdoor.

## 4.7 Limitations & Scope

While our formalisations attempt to capture any backdoor-like functionality, backdoors introduced into a system by, for example, a deliberate side-channel vulnerability or hardware-based component – that would not even be obvious to detect by analysing an execution trace of such a backdoor being used (*i.e.,* the backdoor occurs as a passive, unobservable side-effect of program execution) would prove difficult to model using our FSM-based abstraction and thus, will likely require more specialised models to handle effectively. Similarly, “backdoors” introduced into machine-learning models (*e.g.,* [111]) are beyond the scope of our formalisms.

## 4.8 Chapter Summary

In this chapter, we have provided a much-needed definition for the term backdoor, as well as definitions for backdoor detection, deniable backdoors, and a means to discern between intentional backdoors and accidental vulnerabilities. Furthermore, we have presented a framework to aid in identifying backdoors based upon their structure, which additionally serves as a means to compare existing backdoor detection approaches, and as a basis for developing new techniques. To demonstrate the effectiveness of our approach, we have analysed twelve different backdoors of varying complexity. In each case, we have been able to concisely model those backdoors, which previously, might have manifested as hundreds or thousands of assembly language instructions in a disassembler. We have used our framework to evaluate four state-of-the-art backdoor detection approaches, and in all cases, have shown that none consider a complete model of backdoors, and, as a result, their potential effectiveness is limited.

# Part II

*Anomalous & Undocumented Program Behaviour*

# Chapter 5

## *Undocumented Functionality Detection in Firmware*

**Contents**

In this chapter, we present a methodology for detecting undocumented functionality, specifically where it has been added to the standard software found within Linux-based embedded device firmware. To demonstrate our approach, we present our proof-of-concept tool, HumIDIFy, and show its effectiveness through a number of case-studies and experiments.

## 5.1 Motivation

In recent years there have been a number of incidents where hidden, unexpected functionality has been identified in both the software (firmware) [59, 122] and hardware [184] components of embedded devices. In some cases, it is referred to as a backdoor, in others, it is considered undocumented functionality; though in both cases, such functionality can be abused, and thus, compromise the security of a given device or network it is attached to. When this kind of threat is present within the hardware of a device, it is not only notoriously difficult to detect, but also requires an extremely capable adversary, such as a nation-state actor or chip manufacturer. Conversely, inserting such functionality into firmware is comparatively simple, and while still challenging to detect, requires a much less powerful adversary, which provides a backdoor implementer with greater deniability.

As discussed in Chapter 2, the most prevalent classes of this functionality documented in the real-world, are authentication bypass vulnerabilities and additional, undocumented features added to common services, that serve to compromise a device's security. In such cases, when reported, manufacturers often claim these features are present by accident: that they are so-called *debugging interfaces*, left over from development.

When we say a piece of software contains a backdoor or unexpected functionality, we require context to make such a statement. That is to say, certain behaviour found in one piece of software that we consider *abnormal*, we might consider standard functionality in another. The realisation of this notion of expected functionality inevitably requires a degree of human intervention, however, aside from this, analysis of firmware can, to a large extent, be automated. One major challenge in developing techniques to perform this automation is the huge diversity in the binaries contained within device firmware, such variation – as discussed in Chapters 2 and 3 – arises from manufacturers using different embedded architectures, operating system versions, and compiler options and optimisation levels. A further challenge lies in the fact that a large portion of the firmware that is readily available online is only partially complete, *i.e.,* many are partial firmware updates, which contain just files that will be modified, and those necessary to perform the update process. Compounding these difficulties is the sheer quantity of firmware and devices available.

## 5.2 Contributions

In this work, we demonstrate an approach to automate as much of the process of finding hidden, unexpected functionality as possible. We demonstrate our approach is capable of overcoming the challenges presented by varying firmware implementation architectures and compiler optimisations, and is lightweight enough to scale to analyse large amounts of firmware in reasonable time. We implement our approach in a system, HumIDIFy, whose operation is semi-automated, that is, it requires manual intervention in order to confirm the *abnormalities* it identifies. Despite this manual intervention, compared to manual analysis alone, we find the overall time taken to analyse firmware as a result of using HumIDIFy is significantly reduced.

Our proposed techniques utilise a hybrid of machine learning and human knowledge, which serve to support an expert analyst in a semi-automated fashion by automatically detecting where common binaries from Linux-based embedded device firmware deviate from their expected functionality. The techniques we propose, while implemented here for a specific kind of firmware, can easily be generalised to support other systems. Our methodology is composed of the following components:

- A classifier for common types of program binaries contained within Linux-based embedded device firmware that is resilient to both the heterogeneity of device architectures, and cases where analysed binaries contain unwanted data due to the deficiencies in off-the-shelf firmware extraction tools.

- A domain-specific language and a corresponding interpreter for specifying and evaluating *functionality profiles*, which encode “expert” knowledge in such a way as to facilitate the identification of hidden/unexpected functionality within program binaries.

### 5.2.1 Overview of Our Approach

To conduct the research presented in this chapter, we use a dataset composed of 15,438 Linux-based firmware images from 30 different device vendors. To obtain these images, we built custom web- and FTP-crawlers, modified for each device vendor. Our system takes as input such a firmware image, unpacks it to extract its file-system, and then identifies all of the ELF program binaries (see, *e.g.,* [33]) within the extracted file-system. It then classifies each binary based on what kind of *well-known* service it provides, for example, *web-server*, *ftp-server*, *ssh-daemon*, or *telnet-daemon*. In addition to identifying its service type (which we call a *functionality category* label), it also assigns an associated confidence value, which represents the degree of certainty that the binary contains the functionality associated with its assigned label.

Following classification, we subject each binary to targeted static analysis passes, which are predefined in so-called *functionality profiles*. Such a profile corresponds to the functionality category assigned to the binary, and we define one for each possible functionality label output by our classifier. These profiles are manually specified using a domain-specific language, which provides a high-level means to encode expected program behaviour. By using such an approach, our implementation is both flexible in terms of the functionality it can capture, such that a wide range of abnormalities can be identified, and it can be refined and adapted to support detection of future threats. The output of our defined static analysis passes is a judgement as to whether a given binary contains unexpected functionality for the functionality class it was assigned. If unexpected functionality is identified, manual intervention is required to ascertain the nature of that functionality; this might be performed by, for example, a tool such as IDA Pro [14].

In order to train our classifier, we use binaries taken from 800 firmware images, selected at random from our dataset. Additionally, we sample a further 100 images to construct a *hold-out* test set to evaluate the performance of our trained classifier. To construct our classifier, we first performed evaluation (on a small-scale) of an extensive suite of 17 supervised learning algorithms on our problem domain (*i.e.,* identifying program functionality classes). We combined the best performing supervised learning classifier with an adaptation of the semi-supervised self-training algorithm (see, *e.g.,* [191]), which, as we demonstrate, produces a resultant classifier that exhibits significantly better performance than one using supervised learning alone.

To evaluate the effectiveness of our approach, we use both real-world samples and artificially generated samples, known to contain backdoor-like functionality. The artificial samples were produced using the methodology proposed in [160]. They manifest as backdoors in the `mini_httpd` web-server and the `utelnetd` Telnet daemon, which are programs commonly found within Linux-based firmware. We demonstrate that our approach is able to identify that the modified binaries contain unexpected, hidden functionality – even when compiled for different architectures and using varying compiler optimisation levels. To provide evidence of our method's applicability in detecting such functionality in real-world programs, we analyse binaries taken from 50 firmware images, randomly sampled from our dataset. From this small-scale evaluation, we show our approach is able to detect nine binaries containing potentially unexpected functionality, and in one of those cases, discovers a previously documented backdoor [102].

### 5.2.2 Expectations of Our Approach

In this work, we focus on identifying a class of threat found in common applications, where those applications contain hidden, additional functionality that deviates from their expected functionality. In this way, our approach does not claim completely to solve the problem of automating the identification of unexpected, hidden functionality within firmware. Instead, it serves to reduce the manual effort of doing so by automating parts of the process that can be practically automated. Moreover, we do not claim to detect all kinds of hidden functionality, for example, authentication bypass vulnerabilities (like those detected by Firmalice [166]), cryptographic backdoors, highly complex backdoors [47], or functionality that is hidden due to obfuscation. In embedded device firmware, from backdoors publicly documented and from our own manual analysis, complex and cryptographic backdoors are very rare, and conversely, as we show in our experimental analysis, undocumented, unexpected functionality is common.

As stated in Chapter 2, for many devices, the mere presence of a Telnet or SSH daemon should signify a real threat – a large portion of firmware does not contain firewall rules to protect such services, many of which are Internet-facing. Additionally, the user accounts on many devices generally have weak passwords – with some not even protected using cryptographic hashing, and on almost all devices, the only user available has privileges equivalent to the `root` user on UNIX-like systems. These problems present a tangential threat that we do not attempt to address using the techniques we propose in this chapter.

Finally, we note that we do not evaluate the effectiveness of our approach in cases where an adversary has introduced hidden functionality in transit (*i.e.,* point 2 in §2.5.2); rather, we address the problem of detecting if a device vendor, deliberately or otherwise, has inserted unexpected functionality into common firmware services.

## 5.3 HumIDIFy System Architecture

Figure 5.1 provides an overview of our system architecture, which we implement in our tool, HumIDIFy. Our system takes as input either a firmware image obtained directly from a device vendor, or a compressed file-system extracted from a device. It then:

1. Unpacks and extracts the input firmware image. Our unpacking engine provides a unified wrapper around several off-the-shelf firmware extraction utilities, notably: binwalk [6], Firmware Mod Kit [11] and Binary Analysis Toolkit [29]. If the unpacking process is successful and a file-system is recovered, it is subsequently scanned

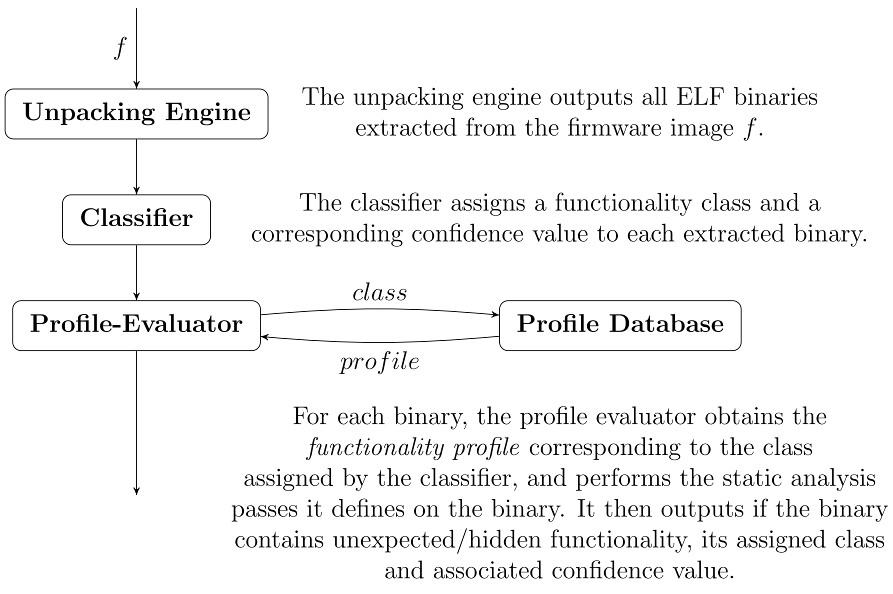

$f$
Unpacking Engine
The unpacking engine outputs all ELF binaries extracted from the firmware image $f$.
Classifier
The classifier assigns a functionality class and a corresponding confidence value to each extracted binary.
Profile-Evaluator
$class$
Profile Database
$profile$
For each binary, the profile evaluator obtains the *functionality profile* corresponding to the class assigned by the classifier, and performs the static analysis passes it defines on the binary. It then outputs if the binary contains unexpected/hidden functionality, its assigned class and associated confidence value.

Figure 5.1: HumIDIFy system architecture.

for ELF program binaries, which are used as input to our trained classifier.

2. Classifies each input binary. The classifier component takes as input a binary executable and outputs a corresponding functionality category label and confidence value. As previously noted, the set of categories match one-to-one with possible functionality profiles of the profile evaluator and represent general functionality classes, such as *web-server* or *ssh-daemon*. The input binary, its assigned label and confidence value are used as input to the profile evaluator.

3. Performs static analysis of each binary and outputs a report. For each binary, the profile evaluator first locates the appropriate functionality profile corresponding to its assigned functionality category from the profile database. It then performs targeted static analysis passes upon the binary, as defined in its functionality profile (see Figure 5.1). It generates a report containing the binary path, if it contains unexpected functionality, and its assigned label and confidence value, which serve as a basis for subsequent manual analysis.

For a given firmware image, the output of HumIDIFy is a report for each of its binaries that details if it found them to contain potential unexpected functionality, their assigned classification labels and corresponding confidence values.

## 5.4 Classification of Binaries

### 5.4.1 Dataset Composition

In order to train and evaluate our classifier, we require a set of binaries taken from a representative sample of firmware, that is balanced in the sense that it contains programs implementing various functionalities. As noted earlier, to obtain these binaries we built custom HTTP- and FTP- crawlers, which we used to download firmware from 30 different device vendors.

When processing a given firmware image, if our unpacking engine is unable to successfully unpack it, either due to an external tool failure, or exertion of a reasonable threshold of system resources (*i.e.,* execution time or disk-space), the firmware is discarded. The dataset of firmware we obtained via our crawlers consists of a total of 15,438 firmware images, which after processing by our unpacking engine, results in 7,590 successfully extracted file-systems, corresponding to a total of 2,451,532 (non-unique) binaries.

### 5.4.2 Scope and Device Functionality

Through the manual analysis of a random sample of 100 firmware images from our dataset, we observe that, in general, the firmware obtained (although targeted at performing a number of domain-specific functions) tends to adhere to a common structure: device configuration is usually performed via a web-interface, and firmware upgrades are often also integrated into this same interface. Other common services present include file-servers and remote access services, such as Telnet and SSH daemons. These observations are consistent with the literature examined in Chapter 3.

As noted in Chapter 2, a major problem in the analysis of binaries within embedded device firmware is the heterogeneousness of the architectures they are compiled for. Our approach targets the predominant architectures: ARM and MIPS and restricts itself to Linux-based firmware. Although these choices may appear as limitations, we observe that the vast majority of firmware falls within these boundaries, as highlighted in [76]. In addition, our techniques are general enough to be adapted to other architectures and operating system targets.

### 5.4.3 Choice of classification domain

A naïve approach to classify a binary's functionality would be to assign a functionality category based upon its filename: for example, an FTP daemon might be called `ftpd` or `vsftpd`. Such an approach, however, has several drawbacks. Firstly, we observe that a portion of the firmware in our dataset does not unpack cleanly, that is, while we are able to recover program binaries, we are often unable to recover their filenames. Additionally, for firmware that we are able to unpack cleanly, we observe that a range of names are used for binaries that provide the same service. For example, we found web-servers with filenames, such as `webs`, `httpd`, `mini_httpd`, `goahead`, and `web_server`. Creating a database of these names, while feasible, would not scale, or handle firmware from new vendors. Furthermore, having attempted to use just filenames as a basis for a classifier, on evaluation, we found it to over-fit the binaries from its training set (*i.e.,* the set of files used to build its filename database) with a bias towards services from particular vendors.

To overcome the deficiencies of the aforementioned approach, we consider two further methods for classifier construction: supervised learning and semi-supervised learning. As detailed in Chapter 2, both learning methodologies demand at least a subset of their input dataset be labelled, and a reasonable number of examples be collected for each such label to avoid over-fitting. In this case, the labels chosen for classification inherently cannot cover all possible binary types found within firmware; we, therefore, focus only on those services common to most firmware images. To this end, from manual analysis of 100 firmware images (the same sample used in §5.4.2), we selected 24 such labels, and from those firmware images, we formed an initial training set of 419 unique binaries. In order to establish uniqueness, we used cryptographic hashing to compare binaries. A number of labels selected serve as meta-labels, that is, they encode a particular set of functionalities: one such label is *web-server*, which covers a number of distinct example binaries within our dataset: from very simple web-servers, such as `uhttpd`, to more complex examples, such as `lighttpd`. We use such labels in order to ensure our constructed classifier is robust to different, not seen before examples of labels.

While we acknowledge our initial training set is relatively small in proportion to the overall number of firmware images collected, manual analysis of binaries is very time-consuming for a human analyst and one of the problems we attempt to address with this work.

Traditionally, for malware detection, approaches taken by the literature, such as [151] have utilised supervised learning to train a binary classifier to distinguish if a binary is malicious or not. While such approaches are applicable for binaries on commodity systems – due to the existence of large, balanced datasets of malicious binaries – large datasets do not exist for binaries found in embedded systems firmware. Moreover, supervised learning requires roughly equal sized input-sets for each label; in the case of binaries for embedded device firmware, this is also not possible to construct, due to the relatively small number of binaries known to contain hidden functionality or backdoors.

Therefore, we take an approach that overcomes the difficulties presented by supervised learning – we construct a general functionality classifier using semi-supervised learning, which instead of directly detecting if a binary is malicious or not, provides a classification of common, well-known program functionality. To detect malicious functionality, we use our classifier as an initial filtering mechanism, in order to facilitate targeted static analyses, which we then use to detect anomalous or malicious functionality in program binaries, irrespective of the initial number of anomalous binaries identified in our input training set.

### 5.4.4 Attribute selection

To train our classifier, as detailed in Chapter 4, we require a set of attributes that can be used to represent input instances, *i.e.,* in this case, program binaries. In order to select the most discriminating attributes – to improve classifier performance and training time – we perform a process of attribute selection. To perform training and attribute selection, we utilise the open-source machine learning toolkit WEKA [99].

Since our system aims to facilitate analysis of program binaries targeting multiple device architectures, we restrict the possible attributes to those that are homogeneous among binaries across different device architectures. As program instructions are architecture specific, we use high-level meta-information, *i.e.,* strings and the contents of function import and export tables. As evidenced in our evaluation in §5.6, this meta-information is sufficient to derive a classifier capable of inferring the general class of an arbitrary binary taken from a firmware image with high precision and accuracy.

Although the number of possible attribute types that we consider for constructing our classifier is small, the number of distinct values associated with each class of attributes is impractically large to use directly for classifier construction. To overcome this, we apply attribute selection methods to remove needless, non-discriminating attributes that do not characterise a general category.

The representation of a program binary as an attribute vector used as input to our classifier consists of nominal attributes representing if it contains a given API name or string. That is, for each attribute $a_i$ within an attribute vector: $a_i \in \{0, 1\}$, where a value of 1 represents that attribute is present in the binary and 0 represents it is not. As a specific example, suppose the API names: `socket`, `bind` and `puts` are selected as attributes, and a given training instance is given the label *web-server* and imports only the first two API names, then we would represent it using the following attribute vector: $\langle 1, 1, 0, \textit{web-server} \rangle$.

We use two stages of attribute selection. Our first pass filters attributes based on their association with a given functionality class. That is to say, for each binary of a given class, if an attribute is to be included in the set of all possible attributes, it must be present in a relatively high proportion of instances of that class. For example, for web-servers, the string `GET / HTTP/1.1` is included in a large proportion of examples whereas, in those same binaries there exist unique compiler strings which are irrelevant. In the second-stage, we use standard feature selection algorithms found in WEKA.

To perform the first stage of feature selection, we define a threshold delta to filter the set of possible attributes. To select this value, we construct a number of representations of our input training set. In each of these representations, we represent a particular training instance using the attributes remaining after filtering them using a given delta. For each of these representations, we perform our second-stage of attribute selection, in which we apply all standard attribute selection algorithms from WEKA to generate further representations. To select the optimal delta and attribute selection algorithm combination, we use each of the representations following second-stage attribute selection to construct a supervised learning classifier and evaluate it using 10-fold cross-validation. In this case, the optimal combination will be that which produces the best performing classifier in respect to maximising the number of correctly classified instances. For each combination, to construct a classifier, we use the `BayesNet` algorithm, chosen arbitrarily to provide a uniform measure of performance for each algorithm-delta combination. We note that we did not evaluate the performance of those deltas where API attributes were not considered, thus, our evaluation was performed with deltas between 0.1 and 0.7 inclusive, and the following feature selection algorithms: `CfsSubsetEval` with the `BestFirst` ranker, `CorrelationAttributeEval`, `GainRatioAttributeEval`, `InfoGainAttributeEval`, `OneRAttributeEval`, `ReliefAttributeEval`, `SymmetricUncertAttributeEval`, all with `Ranker`. From this process, we find `CfsSubsetEval` combined with the `BestFirst` ranker (using default parameters) and a delta of 0.6 performs best. We detail the number of attributes left after filtering for each delta in Table 5.1, and the result of evaluating a classifier trained using the second-stage representations for `CfsSubsetEval` with `BestFirst` in Table 5.2. Appendix A shows the results for the other attribute selection algorithms.

Table 5.1: First stage attribute filtering.

| Threshold | API count | String count |
|---|---|---|
| 0.0 | 2391 | 38040 |
| 0.1 | 1688 | 14328 |
| 0.2 | 1513 | 11074 |
| 0.3 | 1209 | 8522 |
| 0.4 | 442 | 5624 |
| 0.5 | 442 | 4843 |
| 0.6 | 231 | 3001 |
| 0.7 | 14 | 2020 |
| 0.8 | 0 | 1920 |
| 0.9 | 0 | 1830 |
| 1.0 | 0 | 1790 |

Table 5.2: Second stage attribute filtering with `CfsSubsetEval`.

| Threshold | Correct (%) |
|---|---|
| 0.1 | 87.8788 |
| 0.2 | 87.8788 |
| 0.3 | 87.8788 |
| 0.4 | 87.8788 |
| 0.5 | 85.4545 |
| **0.6** | **88.4848** |
| 0.7 | 85.4545 |

The `CfsSubsetEval` algorithm (outlined in [98]) evaluates the merit of subsets of attributes by correlating the predictive nature of individual attributes with respect to their relative redundancy amongst the subset. Those subsets that are highly correlated with a given class, whilst maintaining a low degree of intercorrelation are considered the most useful. The `BestFirst` ranking algorithm searches the subsets of attributes by hill climbing; that is, starting from an initial solution, attempts to find a better solution incrementally by changing a single element upon each iteration until a fix-point is reached.

Table 5.3: Supervised learning algorithm evaluation.

| Classifier | Correct (%) | Time (s) |
|---|---|---|
| BayesNet [95] | 88.4848 | 0.00 |
| NaiveBayes [113] | 79.3939 | 0.01 |
| IBk [46] | 84.2424 | 0.00 |
| KStar [71] | 84.2424 | 0.00 |
| LWL [91] | 51.5152 | 0.00 |
| DecisionTable [119] | 68.4848 | 0.58 |
| JRip [72] | 66.6667 | 0.08 |
| OneR [104] | 21.2121 | 0.00 |
| PART [92] | 77.5758 | 0.04 |
| ZeroR | 10.9091 | 0.00 |
| DecisionStump [108] | 20.6061 | 0.00 |
| HoeffdingTree [106] | 79.3939 | 0.00 |
| J48 [147] | 76.9697 | 0.00 |
| LMT [171] | 85.4545 | 0.90 |
| RandomForest [61] | 88.4848 | 0.11 |
| RandomTree | 78.7879 | 0.00 |
| REPTree | 64.8485 | 0.03 |

For `BestFirst`, hill climbing is performed in a greedy manner with backtracking.

### 5.4.5 Construction of the classifier

Following attribute selection, in order to construct our classifier, as a premise to semi-supervised learning, we evaluated an extensive set of supervised learning algorithms. We sought to maximise the accuracy of the classifier (*i.e.,* maximise the number of correctly classified instances, and minimise the number of incorrectly classified instances), whilst attempting to minimise the time taken to train the classifier. We note that minimisation of the training time for semi-supervised learning is particularly important as training is performed as an iterative process. To perform our evaluation, we trained each classifier using the same labelled dataset, and evaluated it using 10-fold cross-validation. We detail the results in Table 5.3.

Amongst the possible choices for classification algorithm, we found the two best performing in terms of optimising the number of correctly/incorrectly classified instances were `BayesNet` and `RandomForest`. Of those, the time taken to train the `BayesNet` classifier was less than that using `RandomForest`: 0.00$s$ compared to 0.11$s$.

**Algorithm 5.1** Bounded self-training.

```
1: function boundedSelfTraining(labelledData, unlabelledData, v, bound)
2:     L ← labelledData, U ← unlabelledData, k ← 0
3:     loop
4:         train f from L using supervised learning
5:         (k', L', U') ← apply f to unlabelled instances in U where u ∈ U' if confidence(f(u)) ≥ v
6:         if U = U' ∨ k' − k ≤ bound then return f
7:         end if
8:         k ← k', L ← L', U ← U'
9:     end loop
10: end function
```

In order to perform semi-supervised learning, we adapted the self-training algorithm outlined in [191], by introducing a bound on its iteration. We show our modified algorithm in Algorithm 5.1. After training an initial classifier (*i.e.,* performing the first iteration of Algorithm 5.1), we used binaries from a further 700 firmware images as input to construct the final classifier, all of which were previously unlabelled. We performed additional evaluation of that classifier using a further hold-out test set of (manually) labelled binaries from 100 firmware images.

To train our classifier using Algorithm 5.1, we use the `BayesNet` classifier to perform its supervised learning stage, we set a threshold bound of 0.05% and use a value of 0.9 as the required confidence bound (*i.e., bound*) to move an instance from the set of unclassified data ($U$) to the set of classified data ($L$). Using a value less than 1.0 as the confidence bound is required to avoid over-fitting the training-data. A value of 1.0 would produce a classifier that after being trained over multiple iterations would learn to only correctly classify instances that were of extremely high similarity to those used to initially train it. After running the first stage of semi-supervised learning using a range of values, we found 0.9 to be the most suitable – lower values produced classifiers that performed worse when using 10-fold cross-validation.

Table 5.4: Semi-supervised iterations.

| Iteration | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Correct (%) | 88.4848 | 95.4819 | 97.0760 | 97.9021 | 98.5462 | 99.2366 | 99.3256 | 99.3691 |

The number of iterations required to reach our selected threshold was eight iterations; that is, between the 7th and 8th iterations the percentage difference in the classifier's performance was less than 0.05%. Table 5.4 demonstrates the monotonic nature of the number of correctly classified instances at each iteration of training. The final classifier achieved a correct classification rate of 99.3691% when evaluated using 10-fold cross-validation, while evaluating it using the hold-out test set resulted in that rate dropping marginally to 96.4523%. This drop can be attributed to a number of instances being mislabelled; of those incorrectly labelled, the maximum confidence the classifier gave was 0.65, which was to a *dhcp-daemon* mislabelled as a *upnp-daemon*.

#### 5.4.5.1 Avoiding over-fitting

As noted in Chapter 2, over-fitting can become a problem when a classifier is biased to the data it was trained on, and thus, the chance of introducing such a bias needs to be minimised in order to produce a useful classifier. In the case of identifying classes of binaries, we identify two sources of bias:

- Using firmware from a small subset of vendors. As many vendors tend to share code between the most common services found in their firmware, *i.e.,* for web-servers, Telnet daemons, and so on, a classifier trained on a dataset consisting of a limited number of vendors will be biased towards identifying a restricted set of functionality for each functionality class.

- Using particular types of firmware, *e.g.,* router or IP camera firmware. These devices – as they have similar specifications and requirements – tend to use the same programs to perform the same tasks, irrespective of their manufacturer. Therefore, a classifier trained on a dataset restricted by device type will be biased towards identifying a restricted set of program functionality for each type of service.

In order to ensure our training- and test sets, are free from bias and are thus, representative samples of device firmware from the entire device market, our overall dataset includes firmware from 30 different device vendors, on which we perform random sampling in order to construct our training- and test sets.

#### 5.4.5.2 Overcoming limitations in the classification method

A limitation of our classification approach is that a label must be assigned to every input instance. Therefore, if a binary contains functionality never seen before, the classifier, rather than returning an *unknown* classification label, must assign a known label. We handle this deficiency by assigning a confidence value to each label. This confidence value allows us to discern between binaries that are assigned a given label with low confidence, that do not match their functionality profiles – which are less likely to contain unexpected functionality and require further manual analysis – and those labelled with high confidence not matching their expected functionality profiles – which are likely to contain additional functionality.

#### 5.4.5.3 Overcoming limitations in data collection

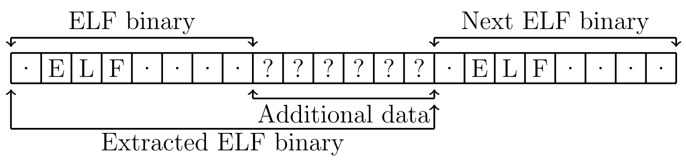


Figure 5.2: ELF binary carving.

We observe that binwalk [6] – one of the tools used to construct our unpacking engine – fails to correctly extract binaries in cases such as that shown in Figure 5.2. As described in Chapter 2, binwalk operates by identifying contiguous files by locating so-called "magic numbers". Unfortunately, if it happens that an ELF binary is followed by a chunk of data that does not contain a "magic number", that data will be considered part of the binary it extracts. Thus, if we attempt to extract strings from such a binary, strings found within its "additional data" can potentially interfere with the results of our classifier. To overcome this, when performing firmware unpacking and file-system recovery, our system parses the ELF file header of binaries it finds and uses it to compute their correct file-size. If the calculated size is smaller than the extracted binary, then we remove the additional data.

## 5.5 Unexpected Functionality Detection

The output of our classifier is a functionality class label and a corresponding confidence value. In our system, in order to check if a binary assigned a particular functionality class contains unexpected or anomalous functionality, we perform targeted static analysis passes. The types of analyses performed in these passes are specified in a so-called functionality profile, which describes how to check the expected functionality of a particular program binary. Within our system, the component responsible for performing those analyses is the profile evaluator. For each functionality class our classifier identifies, we have manually defined a corresponding functionality profile, which we store in a profile database (as in Figure 5.1). To extend our system to handle additional functionality classes, a user must first define a label for that functionality, and then create an appropriate functionality profile to describe their expectations of that functionality class. For example, in the case of a *web-server*, a user might expect that it will only perform TCP-based networking, if instead it also performs UDP-based networking, then it should be flagged as containing unexpected functionality, and further analysed to ascertain if that functionality is malicious or benign. In this case, the user would encode that a *web-server* is expected to perform only TCP-based networking.

### 5.5.1 Binary Functionality Description Language

In order to define functionality profiles, we have created a domain-specific language, Binary Functionality Description Language (BFDL), detailed in Figure 5.3. The language is statically-typed, expression-based and features a ML-inspired syntax. For a given functionality class, a corresponding functionality profile contains a top-level **rule** directive, whose name corresponds to that class, *i.e.,* `web_server` for the *web-server* class. The body of that rule should evaluate to true if the binary satisfies the checks performed in that rule, *i.e.,* it matches its expected functionality profile, and false otherwise, *i.e.,* it contains unexpected functionality. In addition a single *main* rule, other rules may also be defined, which can be parameterised to allow the values returned by different analyses and rules to be used to influence their control-flow or the parameters of subsequent analyses performed. In this way, rules can be used to define isolated, reusable functionality, in essence acting as building-blocks for defining more complex rules. The **import** keyword allows for further rule reuse by permitting rules to be defined in separate files, thus, providing a facility to implement libraries of predefined analyses for common tasks. In addition to boolean values, BFDL also supports strings and integers, as primitive types.

At the core of BFDL are its built-in rules, which serve as a basis for defining more complex analyses to check for specific program properties and functionality. The most basic of these are: **import_exists**, **export_exists** and **string_exists**. Each of these rules derive their results from parsing the binary under analysis in different ways. Both **import_exists** and **export_exists** check for the existence of strings representing imported or exported function names within the import and export tables of the binary's container format, while the **string_exists** rule checks for the existence of a specified byte-string in the binary. The **architecture** rule takes a case-insensitive string argument representing a

```
   ⟨top-level⟩  ::=  rule ⟨ident⟩(⟨arg-list⟩) = ⟨expr⟩
                 |   import ⟨string⟩

        ⟨expr⟩  ::=  ⟨rule⟩(⟨values⟩)
                 |   let ⟨ident⟩ = ⟨expr⟩ in ⟨expr⟩
                 |   if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩
                 |   ! ⟨expr⟩
                 |   ⟨expr⟩ ⟨logic-op⟩ ⟨expr⟩
                 |   ⟨value⟩ ⟨comp-op⟩ ⟨value⟩
                 |   forall ⟨ident⟩(⟨arg-list⟩) ⇒ ⟨expr⟩
                 |   exists ⟨ident⟩(⟨arg-list⟩) ⇒ ⟨expr⟩

   ⟨type-name⟩  ::=  bool | int | string

         ⟨arg⟩  ::=  ⟨ident⟩ : ⟨type-name⟩
    ⟨arg-list⟩  ::=  ε | ⟨arg-list1⟩
   ⟨arg-list1⟩  ::=  ⟨arg⟩ | ⟨arg⟩, ⟨arg-list1⟩

        ⟨narg⟩  ::=  _ | ⟨arg⟩
   ⟨narg-list⟩  ::=  ε | ⟨narg-list1⟩
  ⟨narg-list1⟩  ::=  ⟨narg⟩ | ⟨narg⟩, ⟨narg-list1⟩

       ⟨value⟩  ::=  ⟨const⟩
                 |   ⟨variable⟩
                 |   ⟨value⟩ ⟨arith-op⟩ ⟨value⟩

        ⟨rule⟩  ::=  ⟨base-rule⟩
                 |   ⟨ident⟩

   ⟨base-rule⟩  ::=  import_exists
                 |   export_exists
                 |   string_exists
                 |   function_ref
                 |   string_ref
                 |   architecture
                 |   endianness

       ⟨const⟩  ::=  ⟨bool⟩
                 |   ⟨int⟩
                 |   ⟨string⟩
                 |   error

    ⟨arith-op⟩  ::=  + | − | × | ÷ | % | & | ˆ | | | ∼ | << | >>
     ⟨comp-op⟩  ::=  == | != | < | > | <= | >=
    ⟨logic-op⟩  ::=  || | &&
```

Figure 5.3: BFDL language specification.

device architecture (*e.g.,* `ARM` or `MIPS`), and evaluates to true if the analysed binary is of that architecture, and false otherwise; this rule serves as a basis for creating architecture-dependent analysis passes.

In order to implement our more complex built-in rules, we leverage the BAP [63] framework to provide code-lifting and disassembly, and IDA Pro [14] to provide CFG recovery, which are both described in further detail in Chapter 3. To ascertain if a function is called within the binary, we provide a rule named **function_ref**. It operates first by inspection of the call-graph of the binary, and attempts to verify the existence of an incoming edge to the node representing the function name being searched for; if such an edge does not exist, then we search for data references to the function, which could indicate the use of the function as a callback or its indirect use, such as via a function pointer. As an example, the expression **function_ref**("`listen`") can be used to check if the binary makes a call to the `listen` function, which is associated with opening an incoming network socket. Similar to **function_ref**, **string_ref** searches for data references to a given byte-string within the binary.

We provide the **forall** and **exists** keywords, which allow us to quantify over the parameters of function calls that we are able to identify in the analysed binary. They enable us to define constraints over those parameters, which can be used to enforce restrictions on how particular functions should be called (in the case of **forall**), or ensure a function is called in a particular way (in the case of **exists**). For both **exists** and **forall**, the inputs into the underlying analysis pass are a function name to search for, a list of arguments, qualified with the type of data they should reference, and an expression. The expression specified defines a constraint over each instantiation of the arguments identified for each call-site corresponding to the function name. As an example, to check if a binary makes a call to the `socket` function with the *type* argument equal to 2, the following expression could be used:

$$\textbf{exists}\ \mathsf{socket}(\mathit{domain{:}\ int},\ \mathit{type{:}\ int},\ \mathit{protocol{:}\ int}) \Rightarrow \mathit{type} == 2$$

We use components of the BAP framework as a basis for writing analysis routines to implement the functionality of the **forall** and **exists** keywords. A core component of these analyses involves statically estimating the arguments passed to functions. To do this, we construct CFGs for each function of the binary and then perform call-site identification. For a given function, we identify its call-sites by searching for blocks containing *call*-type instructions (*i.e.,* `bl` and `blx` for the ARM architecture) and check if the function is the target of one of those instructions. In order to recover the arguments passed to the function, we use the prototype specified with the **forall** or **exists** keyword to obtain both its expected number of arguments and their types. In the interest of maintaining reasonably lightweight analysis, we make the assumption that the basic block containing the call to the function searched for contains all of the argument loading instructions for that particular call-site. We also assume that any argument loading instructions related to the call in parent blocks will be conditional, and thus, cannot be determined without further, more heavy-weight analysis. Since in both the ARM and MIPS instruction sets, argument passing is implemented by passing values in registers, we estimate integer constants and string references to the data section of the binary by examination of load operations into registers. For integer-based constants, argument estimation is trivial as both instruction sets have instructions for loading constant integers directly into registers. For strings, estimation is slightly more complex; for both instruction sets, we identify loads into registers that are address references to the data section of the binary, we then verify the data at those addresses is string-like by checking for a consecutive sequence of ASCII characters followed by a terminating NULL byte (*i.e.,* a C-style string). From manual analysis, we find that C-style strings are the most pervasive type of text-based data, and are found in an overwhelming majority of binaries from Linux-based device firmware. If the arguments of a call-site are computed by some complex calculation (*i.e.,* something that cannot be resolved using constant propagation and folding), our approach will be unable to recover their values. To represent this failure at the language level, we augment each type with an additional value, $\perp$, which we represent using the error keyword. A comparison with error that is not itself an error value will always result in false. In a boolean context, when used as part of a logical expression, error will automatically be coerced into the value false.

To compose expressions, BFDL supports all of C's logical and equality operators. It implements conditionals by way of an **if** expression, and allows for binding names to values through the **let** keyword; the semantics of which follows that of ML-like languages. For a given functionality profile, its expected functionality may be encoded in a number of ways: some rules make it possible to estimate "behaviour" in a manner that has a bias towards minimising the execution time of the profile evaluator, while others trade execution time and resources for greater precision.

```
rule uses_udp() = exists socket(domain, type, protocol) ⇒
      if architecture("MIPS") then type == 2 else type == 1

rule may_read_files() = exists fopen(filename, mode) ⇒
      (mode == "r" || mode == "r+" || mode == "w+" || mode == "a+")
```

Figure 5.4: An excerpt of BFDL rules from our standard library.

Figure 5.4 details an excerpt from the BFDL standard library. It shows how checks for both certain socket and file (stream) behaviour can be encoded within the language. We note that these rules do not provide an absolute check of the behaviour being tested – for example, uses_udp() checks if the socket API is used with an appropriate parameter (2 for MIPS, 1 for other architectures). It is possible that a developer may implement a wrapper around the socket API, which this rule would not detect; it is also possible that a program might generate the socket type parameter as a result of a complex calculation, which this rule would also not detect. Therefore, this rule rather tests if UDP-networking was used in a standard, expected way, than if UDP-networking was used at all.

```
import "prelude.bfdl"

rule web_server() = uses_tcp() && !uses_udp() && may_read_write_files()
                    && !outgoing_socket()

rule telnet_daemon() = uses_tcp() && !(read_write_files() || uses_udp())
```

Figure 5.5: Simplified example profiles for web-servers and Telnet-daemons.

Figure 5.5 shows simple examples of how one might encode the functionality profiles for a *web-server* and a *telnet-daemon*, and Appendix B details an extended example. Within functionality profiles, we are primarily concerned with outlining expected functionality; thus, these rules focus on checking that binaries conform to their expected network and file behaviours. We use such simple examples to emphasise how basic rules may be composed to implement more complex analyses.

## 5.6 Experimental Results

In this section, we show the evaluation of the separate components of our approach, according to the points outlined in §5.2. First, we examine the performance of our classifier on a new hold-out set of manually labelled binaries. We then evaluate the entire system (*i.e.,* HumIDIFy) using a set of binaries known to have hidden functionality embedded within them. Following this, we evaluate our tool on a sample of binaries taken from real-world firmware images. We then examine the run-time performance of our tool and demonstrate its applicability to large-scale analysis. Finally, we look at how one might attempt to evade our techniques – within the limitations outlined in §5.2.2, and discuss possible ways to mitigate such attempts.

### 5.6.1 Evaluation of Classifier

As outlined in §5.4.5, our classifier was trained on a dataset consisting of binaries from 800 firmware images and subsequently tested against an additional (separate, manually labelled) dataset of binaries from 100 firmware images. It achieves a correct classification rate of 99.3691% on its training set using 10-fold cross-validation, and a correct classification rate of 96.4523% on the independent test set, which, in total, consisted of 451 individual binaries that exactly match their assigned functionality labels. The overall TPR (true positive rate) over all 24 classes identified by the classifier on the test set was 0.965, while the FPR (false positive rate) was 0.002. Of the instances that were incorrectly classified, seven labels were involved. Table 5.5 outlines the TP/FP rates for those instances, as well as their precision, accuracy and error rates.

In the test set gathered, we found only a single binary that corresponded to the label

Table 5.5: Statistics for labels that were misclassified.

| Label | TPR | FPR | Precision | Accuracy | Error |
|---|---|---|---|---|---|
| *cron-daemon* | 0.000 | 0.002 | 0.000 | 0.996 | 0.004 |
| *dhcp-daemon* | 0.636 | 0.002 | 0.875 | 0.989 | 0.011 |
| *ftp-daemon* | 1.000 | 0.002 | 0.929 | 0.998 | 0.002 |
| *ntp-client* | 1.000 | 0.002 | 0.933 | 0.998 | 0.002 |
| *nvram-get-set* | 1.000 | 0.011 | 0.750 | 0.989 | 0.011 |
| *ping* | 0.667 | 0.002 | 0.667 | 0.996 | 0.004 |
| *tcp-daemon* | 0.000 | 0.000 | 0.000 | 0.998 | 0.002 |
| *telnet-daemon* | 0.800 | 0.000 | 1.000 | 0.998 | 0.002 |
| *upnp-daemon* | 0.739 | 0.005 | 0.895 | 0.983 | 0.017 |
| *web-server* | 0.939 | 0.010 | 0.886 | 0.988 | 0.012 |

*cron-daemon*, the remainder of binaries that implemented such functionality we labelled as *busybox*. This can be explained by the existence of `busybox` in an overwhelming majority of firmware images in our dataset, which includes the functionality implemented by the `cron` daemon. Our classifier mislabelled the *cron-daemon* instance as a *dhcp-daemon*. We found the same pattern for instances of both the *tcp-daemon* label, of which we found none of in our test set, and the *ping* label, of which we found three, one of which was mislabelled as *nvram-get-set*. Of the incorrectly labelled instances of the *telnet-daemon* label, one was labelled as *nvram-get-set*. We found four instances of the *dhcp-daemon* label (of 11) to be mislabelled; they received the labels: *ftp-daemon*, *nvram-get-set*, *ping* and *upnp-daemon*. Of the two (of 33) mislabelled *web-server* instances, one was labelled as a *upnp-daemon*, while the other was labelled as *cron-daemon*. In this case, we observed similarity in the API used by the instances of these services, which was the source for the mislabelling. The *upnp-daemon* label was mislabelled in six (of 23) cases, four of those were assigned the *web-server* label (for the reasons previously described); the remaining two were mislabelled as *nvram-get-set*.

The *nvram-get-set* label represents binaries that include general functionality to access and modify non-volatile storage (NVRAM, explained in detail in Chapter 2), often found in Linux-based embedded devices. Many devices contain binaries specifically for NVRAM interaction (commonly called `nvram-get` and `nvram-set`); however, we have found that some program binaries implement NVRAM interaction directly, as opposed to relying on these dedicated utilities. Thus, of all labels, we would expect it to induce the highest FP rate. For all instances in our test set where an instance was incorrectly mislabelled as *nvram-get-set*, we identified this behaviour as the cause.

Overall, while our classifier exhibits a number of false positive and false negative results, for the most pervasive services found within firmware, it is highly successful in assigning the correct label to binaries – irrespective of their origin (*i.e.,* even if they are new instances of common services).

### 5.6.2 Performance on New Artificial Instances

In this section, we demonstrate how we assess the ability of our system to recognise hidden functionality in well-known applications, modified by ourselves to contain additional, unexpected functionality.

To perform the evaluation, we modified the source-code of two services, `mini_httpd` and `utelnetd` – two of the most common services found in embedded device firmware from all device vendors. We inserted "unexpected functionality" into each of these services by implementing a remote-control backdoor, using the same methodology as that proposed in [160]. We compiled modified and unmodified versions of both services, for both ARM and MIPS targets, using various GCC compiler optimisations (-O0, -O1, -O2, and -O3).

Our tests consisted first of using the unmodified versions of the two services as input to our system (to act as a baseline). In each case, our system assigned the correct functionality label with a confidence value of 1.000, and reported each as not containing any unexpected functionality. We then used the modified binaries as input to our system, and in all cases, each binary was assigned the correct classification label with a confidence of

1.000. Furthermore, their feature vectors remained unchanged compared to their baseline versions – indicating the features chosen to define the binary functionality for those classes were discriminating enough to represent the core functionality for those classes. Additionally, in all cases, our system was able to correctly identify the unexpected functionality inserted into the modified binaries. From this, we observe both the effectiveness of our system in identifying hidden functionality, and its generality to handle program binaries of multiple device architectures compiled using different optimisation levels.

### 5.6.3 Real-world Performance Using Sampling

In this section, we evaluate the performance of our system using real-world data. The number of binaries within our overall dataset is too large to feasibly evaluate our approach (as manual analysis is required); therefore we use binaries taken from a random sample of 50 firmware images from that dataset. Of those 50 firmware images, a total of 15,507 program binaries were identified and used as input to the classification component of HumIDIFy. To make evaluation practical, we set a confidence value threshold of 0.9 to determine if a binary should be evaluated by the functionality profiler component. We selected this value for two reasons: it maintains consistency with the value chosen to train the classifier, and those binaries that were classified with confidence above this threshold value would be likely to match the functionality of their assigned label with a (known) high probability *i.e.,* 96.4523%. For the purposes of our experiment, the binaries processed that were assigned a label with a confidence value below 0.9 we considered to be classified as *unknown*.

From the 15,507 binaries, 4,012 were classified with a confidence value of 0.9 or greater. After removing duplicates, 425 unique binaries were classified with a confidence value equal to or above 0.9. From manual analysis, 392 were classified correctly, and of those classified correctly, nine were flagged by HumIDIFy as potentially containing unexpected functionality.

Of those nine binaries, six of them were found within the *web-server* class, one within the *ssh-daemon* class, one within the *telnet-daemon* class, and one within the *tcp-daemon* class. HumIDIFy identified a *web-server* binary that contained a previously documented backdoor – an embedded *management* interface which provides shell execution – which has been reported to be present in a number of devices by the device manufacturer Tenda [102]. Another contained an unexpected built-in DNS resolver. Two instances contained the same unexpected feature: an undocumented internal interface for device configuration listening on a non-standard port, which provides privileged access to anyone with shell access to an affected device. A further *web-server* was found to interact with the `Syslog` daemon over UDP to perform logging, and hence failed to match its expected functionality profile which assumes only TCP-based networking. Another example was a custom application implementing HTTP proxy functionality, which was actually middleware for the Trend Micro kernel engine. It was classified as containing unexpected functionality, as not only does it implement HTTP request processing using TCP, it also provides additional functionality via UDP. The *telnet-daemon* identified was implemented in a non-standard manner and thus, was flagged as containing unexpected functionality. A binary appearing as a *ssh-daemon* in the first stage of classification mismatched the second stage of processing due to being statically linked. The first stage of classification was correct as the classifier was able to correctly label the instance based upon string features alone. Another custom service detected was an Internet telephony proxy. In this case, HumIDIFy classified it as a *tcp-daemon*; from manual analysis of the service, we found it to support both TCP and UDP as a means of data transmission – thus, leading it to be classified as containing unexpected functionality.

In this small-scale evaluation, we observe that our method not only supports finding instances of services that are strictly adhering to the original set of functionality labels, but also those services that share the same core functionality with additional features. This is useful for an analyst, as it allows them to filter services that are known, but contain unexpected functionality, and those services that may be of interest that contain additional functionality unknown to HumIDIFy. Furthermore, this evaluation also demonstrates the flexibility and effectiveness of our system in a practical scenario: an analyst wishing to evaluate a firmware image in a more “paranoid” mindset can set the confidence threshold for classifier label assignment to a low value to have the system identify a larger amount of potential hidden, unexpected functionality, whereas an analyst wishing to analyse a large amount of firmware quickly can set this confidence threshold to a high value to limit the amount of manual analysis required. Moreover, as shown, on real-world data, our system, with a modest confidence threshold, is able to successfully identify binaries containing unexpected functionality – some of which representing real-world threats.

### 5.6.4 Performance

In this section, we examine the run-time performance of our analysis approach. For a single binary, the average time taken for our system to perform feature extraction is 1.31s. The average time it takes to classify a single binary is 0.291s (not including the time taken to invoke the Java virtual machine in order to run WEKA). The time taken to execute a profile is dependent upon the complexity of that profile. In the worst case (where we reconstruct function CFGs), the average time our system takes is 1.53s; we find the time taken where CFGs are reconstructed is proportional to the number of functions present in the binary being analysed.

From analysis of our dataset, we find an average firmware image to contain 310 binaries; thus, the average time to process a single firmware image – assuming a worst-case scenario, whereby our classifier assumes a confidence threshold of 0.0 (resulting in every binary passing through each stage of analysis) is 970.61s. We note that this evaluation does not take into account the time taken to perform the final stage of analysis, *i.e.,* that performed by a human analyst, who will manually analyse the binaries containing unexpected functionality.

In contrast to other approaches, such as that taken by the authors of Firmalice [166], which has similar goals, but identifies binaries containing authentication bypass vulnerabilities as opposed to hidden, unexpected functionality, HumIDIFy performs comparatively well. It is able to process an entire firmware image – on average – in roughly the same time taken to process a single binary using Firmalice. This performance evaluation demonstrates the feasibility for our techniques to be used for large-scale analysis.

### 5.6.5 Required Manual Analysis

As noted in §5.2, our implemented approach requires a degree of manual analysis to confirm its findings; first, to verify that what it has identified as potentially anomalous is indeed anomalous, and, second, to ascertain what that functionality is. This manual analysis can be guided by the output of our tool: namely, using the functionality label assigned to the binary by its classifier component and the contents of the corresponding functionality profile. Thus, the effort of manual analysis on a single binary using the output of our tool compared to performing manual analysis of the same binary alone, *i.e.,* without any knowledge of what functionality it is expected to exhibit, will be reduced.

When considering the analysis of an entire firmware image – manually compared to when using our tool – based upon our findings in §5.6.3, if only common services are checked for anomalous functionality, it is possible that the use of our techniques negates the need for any manual analysis, as in general, all binaries that can be checked by our approach conform to their expected functionality profiles. In the case where a binary is identified as anomalous, from our findings, in an overwhelming majority of cases, that binary will be the only binary identifed as anomalous within the associated firmware image, thus, in practice, the manual effort of analysing a firmware image will be reduced to that of checking the functionality of a single binary, as opposed to checking (on average) 310 binaries.

### 5.6.6 Limitations & Discussion

HumIDIFy relies on certain meta-data: both strings and imported symbol names. While strings are present within all binaries, imported symbol names are only present within dynamically-linked binaries. Thus, when classifying a binary that does not contain all of the required meta-data, incorrect labelling will occur, and thus, lead to false positives (*i.e.,* the binary will be reported as containing unexpected functionality, when it does not). Since our techniques are intended to reduce the time taken for manual analysis, as opposed to being completely automated, we view reporting a binary as potentially containing unexpected functionality, and therefore prompting manual analysis as the correct behaviour for our approach. Moreover, from manual analysis of a large number of firmware images, we have found that an overwhelming majority use dynamic-linking; we attribute this to the general lack of storage space available on embedded devices and the space savings afforded by utilising dynamic-linking.

If an attempt was made to evade our classifier with, for example, a binary that is a web-server manifesting as say, a Telnet daemon, our system would still detect the binary as containing unexpected functionality due to its two-stage classification mechanism. The expected profile of a Telnet daemon would obviously be quite different from that of a web-server, which the modified binary would fail to match. Thus, our overall approach is arguably robust despite the potential limitations of its individual components.

## 5.7 Chapter Summary

In this chapter, we have presented a semi-automated framework for detecting hidden and unexpected functionality in firmware. At the heart of our approach is a hybrid of machine learning and human knowledge encoding within our domain specific language, BFDL. As we have shown, this combination is a highly effective method for detecting unexpected functionality and (in some cases) backdoors in firmware.

# Part III

*Undocumented Commands & Credentials*

# Chapter 6

*Measuring the Importance of Static Data Comparisons*

**Contents**

In this chapter, we present techniques to aid in identifying hard-coded credential checks and hidden, undocumented commands in program binaries from Linux-based embedded device firmware. In order to demonstrate the effectiveness of our approach, we present a proof-of-concept tool, Stringer, which, using our techniques, is able to identify a number of real-world backdoors, as well as aid in the recovery of undocumented protocol command sets.

## 6.1 Motivation

The motivation behind the work presented in this chapter is similar to that presented in Chapter 5. Though, whereas HumIDIFy, the tool presented in that chapter, attempts to aid in the detection of unexpected, potential backdoor functionality in common services, Stringer, the tool presented in this chapter, attempts to aid in the detection of the *triggers* required to *activate* such functionality.

The current state of embedded device security needs much improvement – as evidenced in previous chapters: from manufacturers deploying outdated, vulnerable software components within device firmware, to so-called *debug* interfaces being *accidentally* enabled within production versions of firmware (*e.g.,* [36]). Numerous backdoors – all *activated* using undocumented commands or features – have been reported, *e.g.,* [38, 102, 103]. The impact of these malicious, or simply bad practices is exacerbated by the sheer number of devices available, with each device potentially having multiple firmware versions. Organisations handling sensitive data or critical infrastructure require a means to determine the trustworthiness of a device before deploying it as part of their infrastructure. Such analysis is currently either simply not done, or is carried out manually by an expert. This is obviously a very costly process that does not scale. Moreover, because the evaluation process is so expensive and needs to be done for each firmware version, it has the negative consequence of motivating organisations not to update a device's firmware, leaving them exposed to known security vulnerabilities.

The techniques we present in this chapter aim to reduce the effort of performing certain aspects of this manual analysis – primarily by automatically identifying *interesting* code structures and the functions they reside in. Our techniques are targeted at performing analysis in a lightweight, scalable manner and, are thus, applicable to processing large collections of binary executables found in Linux-based embedded device firmware, while being general enough to be applicable for use on binaries for more commodity hardware.

We say that a section of code is *interesting* when it exhibits unexpected behaviour, where that behaviour will generally only be executed when certain conditions are met – such as on the input of a *special* keyword. Using the terms we have defined in Chapter 4, the unexpected behaviour corresponds to a backdoor *payload*, and the conditions to be met – *i.e.,* successful comparison with a *special* keyword corresponds to a backdoor *trigger* condition. The code executed as a result of successful comparison with a *special* keyword is often not accessible by any other means, and is thus, uniquely *guarded* by that keyword. Our approach automates much of the process of identifying functions that contain functionality that is guarded by such keywords.

By applying our techniques to real-world firmware, we are able to find three backdoors, which manifest as hard-coded credential checks. Furthermore, we are able to demonstrate our approach is equally effective in recovering the protocol messages of both known and proprietary text-based protocols. In the case of such protocols, our method allows an analyst with knowledge of known protocols to isolate the function responsible for parsing the protocol and subsequently identify any superfluous protocol messages it utilises (which are often indicative of additional, undocumented functionality), with relative ease compared to manual analysis with tools such as IDA Pro [14], `strings` and `grep`.

### 6.1.1 Contributions

In this work, we demonstrate our tool, Stringer, which implements our aforementioned methods for performing lightweight, large-scale static analysis of commodity embedded device firmware. It serves as a complementary approach to standard manual analysis techniques, by reducing the time taken to identify constructs containing backdoor-like functionality – as defined in our framework described in Chapter 4. We demonstrate the effectiveness of our tool through identification of three backdoors, which we later present as case-studies in §6.5. The overall contributions presented in this chapter are:

- A set of heuristics for automatically identifying static data comparison functions.
- A metric for measuring the degree a binary's functions' branching behaviour is influenced by comparisons with static data, where that static data guards access to

unique functionality.

- The result of applying Stringer to a set of 7,590 firmware images, which exposes a number of backdoors. Additionally, we demonstrate how our methods can automatically identify static data processing routines. Concretely:
  - We demonstrate the recovery of the full FTP command set handled by a variant of `vsftpd` from Linksys firmware and the recovery of the SOAP-based RPC command set from a Netgear firmware's web-server.
  - We identify two previously undiscovered authentication backdoors relying on hard-coded credentials: one in a Q-See DVR device and the other in a TRENDnet router.
  - We identify a third (previously reported) backdoor in DVR firmware from Ray Sharp.

## 6.2 Methodology

In order to identify a program's functions whose branching behaviour is influenced by comparisons with static data, where that static data guards access to unique functionality, we use the following methodology:

1. We first automatically identify all possible static data comparison functions (e.g., `strcmp`) used within the analysed binary.
2. We perform analysis of the call-sites of those functions and extract the static data they compare against.
3. For each function containing those call-sites, using the extracted static data, we construct sets of sequences of that data that must be successfully compared against to reach each basic block of the function.

4. Using those sets as a basis, we compute a score for each function, which provides a measure of how much of its conditional processing is dependent on comparisons with static data that guards unique functionality.

5. Finally, we use these scores to impose an ordering over all of the functions – *i.e.,* to help an analyst choose which functions to analyse first.

When attempting to perform the first step of our methodology, *i.e.,* identifying a program's static data comparison functions, a naïve approach might rely on using the symbol names of the functions imported by a program and look for well-known comparison functions such as `strcmp` or `strncmp`. Such an approach, however, has a number of drawbacks. To reduce the space occupied by program binaries, firmware developers often opt to strip such symbol information. Additionally, if the binary is statically linked, then there will be no list of imported functions to extract such information. In order to overcome these problems, we instead use a number of heuristics to identify such functions, which can be applied in an automated manner. Our approach relies on first identifying call-sites where at least one argument passed is static data and where its return value is used in a boolean context to influence future control-flow. By analysing all call-sites of this nature, we are able to identify function usage patterns, which allow us to estimate the set of functions that are most likely static data comparison functions.

Once the static data comparison functions have been identified, to perform the remaining steps, we first label the basic blocks of each function with a set of static data sequences. These sets dictate the sequences of static data that must be matched to reach each block. Based on these sets, we calculate a score for each static data item relative to how it influences the branching behaviour of the function. Finally, we calculate a score for each function based on the scores assigned to the static data. These scores are used to impose an ordering of functions where those that score highly are those that contain decision logic that is dependent on comparisons with static data, where that data guards uniquely accessible functionality.

This process scores functions that implement protocol handling or contain parsing functionality the highest. Further analysis of those functions and their associated static data enables us to identify additional, undocumented functionality and (possible) back-door functionality.

For the proof of concept tool we have developed, Stringer, which implements our methodology, we leverage components of the BAP framework [63] and IDA Pro [14]. We rely on a number of useful components provided by BAP; in particular, the IL (intermediate language) it uses: BIL [97], its code-lifting components and the extensive set of algorithms implemented for handling graphs. We use IDA Pro to aid in CFG recovery.

### 6.2.1 Notation

In this section, we outline the notation used in the remainder of this chapter. In addition to the notation defined in Chapter 2, we denote a function of a program $P$ as $f \in P$. We use $f_{blocks}$ to denote the set of basic blocks of a function $f$, for a given block we use $b_{addr}$ to represent its entry address, and $b_{insns}$ to represent the sequence of its instructions lifted to BIL instructions. We use $succs(b)$ and $preds(b)$ to represent the set of successor and predecessor blocks of a block $b$. Additionally, we use the abstract notion of "sections" to denote regions of program's memory that have particular properties, *i.e.,* :

- $section_{data}$ which corresponds to the section holding data that can be both read and written.
- $section_{rodata}$ which corresponds to the section containing constant, read-only data.
- $section_{*}$ which corresponds to the union of the previous two sections.

Finally, we use the notation $m_k$ (where $m$ is a map) to evaluate to the value corresponding to the key $k$ within $m$. While to associate the value $v$ with the key $k$ within $m$

we use the notation $m_k \leftarrow v$.

## 6.3 Identifying Static Data Comparisons

Due to the composition of binaries found in Linux-based firmware – and indeed most commodity binary programs – it is both unreliable and restrictive to rely on symbol names to indicate functions used for static data comparison. In general, firmware images contain a mix of binaries that use either static or dynamic linking to call external library routines (such as the C standard library). Furthermore, a significant portion of those binaries are often stripped of symbol names. To more reliably identify static data comparison functions, we propose a collection of heuristics, which, together, are able to identify these types of functions, based on their call-site properties, and overall usage within a binary.

### 6.3.1 Code & Function Properties

Our heuristics are derived from a number of code and function properties, which we observe from manual analysis of the call-sites of static data comparison functions in binaries from both ARM and MIPS Linux-based device firmware.

**Argument references** Since static data comparison functions compare to hard-coded data, at least one function argument either points to, or is a direct reference to read-only program memory or the initialised data section. From manual analysis, we find that in many cases, those arguments are generally unique (*i.e.,* compared to only once) in functions that perform a substantial amount of static data processing.

**Function arity** Comparisons, by their nature, are made between at least two items; therefore, the arity, or the number of arguments passed to a data comparison function we find to be at least two, and three in length-bounded comparisons (*i.e.,* `strncmp`).

**Branching properties** From our observations, the result of a call to a data comparison function generally influences a branching condition. Thus, one of the variables influencing the branch will be tainted by the return value of that comparison function. In most cases, we found a literal value of 0 compared against in order to compute branching conditions – which, in a boolean (*i.e.,* matched/not matched) context represents *true* or *false*.

**Local call frequency** We observe that data processing routines, such as protocol parsers, generally utilise the same comparison function many times with different static data arguments, as opposed to different comparison functions for each element of static data to be compared against. Therefore, if a function contains many call-sites involving the same static data comparison function, in most cases, we would expect it to perform some degree of parsing.

### 6.3.2 Data Properties

While the previous properties describe attributes of call-sites, the following properties describe features of static data we observe used by those call-sites. With the premise that hidden commands and undocumented protocol messages are often found in parsing routines, we focus analysis of call-sites of static data comparisons found in protocol or message-based parsing routines. In these cases, we observe that the static data is contained in either $section_{rodata}$ or $section_{data}$ and is generally ASCII-based and `NULL` terminated (*i.e.,* C-strings). In addition, in general, we find the static data exhibits the following properties:

- It does not contain any characters (or combination of characters) that are indicative of it being a format string. Format strings can be identified by scanning a string for the '%' character, and checking if that character is followed by common format directives such as 'd', 's', or 'c'.
- It does not contain certain whitespace characters other than new line, line feed and space, *i.e.,* does not contain tab ('\t') or vertical tab ('\v'), nor does it contain characters that are used as control characters.

### 6.3.3 An Algorithm for Finding Static Data Comparisons

In this section, we detail our algorithm for identifying static data comparison functions, which is based on the observations outlined in the previous two sections. Its process can be decomposed into the following steps:

- For each function in a binary, we identify all basic blocks that contain function calls.
- Of those function calls, we filter out those whose return value does not influence branching conditions, and those that do where the branching condition they influence is not a comparison against 0.
- We analyse the call-sites of the remaining function calls based on their arguments, and categorise them using two cases: an "ideal" case and a "catch-all" case.
- We associate a score to each case, and use it to compute an overall score for the function's usage throughout the binary.

We categorise a call-site as the "ideal" case when there are two or more arguments passed, and one of those arguments is a reference to static data that conforms to the properties we observed in §6.3. Additionally, that argument and one of the other arguments should not be register-based constants, such as integers or floating point numbers that are not also (address) references to $section_*$. This case models the common scenario where a comparison function takes two references to data: one static, and the other dynamic. We categorise a call-site using the general "catch-all" case when at least two of the arguments identified do not reference constant data.

The result of applying our algorithm provides us with a set of functions that are likely to be static data comparison functions, as well as a relative score for each such function that indicates the likelihood it is a static data comparison function. Algorithm 6.1

shows the computations we perform to calculate the scores for each possible static data comparison function within a given binary $B$.

**Algorithm 6.1** Algorithm to compute heuristic scores.

```
function COMPUTEHEURISTICSCORES($\varsigma, \delta, \mu_+, \alpha_-, \alpha_*, B$)
    $\nu \leftarrow \{\}$
    for each $f \in B$ do
        $\nu' \leftarrow \{\}$
        for each $b \in \{b \mid b \in f_{blocks} \wedge branchesOnCall(b) \wedge branchesOnZCmp(b)\}$ do
            $args_{data} \leftarrow \emptyset,\ args_{rodata} \leftarrow \emptyset,\ args_{other} \leftarrow \emptyset$
            for each $arg \in dependentArgs(b, 3)$ do
                if $arg \in section_{rodata}$ then
                    $arg_{rodata} \leftarrow arg_{rodata} \cup \{arg\}$
                else if $arg \in section_{data}$ then
                    $arg_{data} \leftarrow arg_{data} \cup \{arg\}$
                else if $arg \notin section_*$ then
                    $arg_{other} \leftarrow arg_{other} \cup \{arg\}$
                end if
            end for
            $addr \leftarrow f_{addr}$
            if $|arg_{rodata}| + |arg_{data}| + |arg_{other}| \geq 2 \wedge containsIdealSD(args)$ then
                $\nu'_{addr} \leftarrow \nu'_{addr} + \varsigma$
            else if $|arg_{data}| + |arg_{other}| \geq 2$ then
                $\nu'_{addr} \leftarrow \nu'_{addr} + \delta \cdot \varsigma$
            end if
        end for
        $\nu \leftarrow mergeScores(\nu, applyRewards(applyPenalties(\nu', \alpha_-, \alpha_*), \mu_+))$
    end for
    return $\nu$
end function
```

In Algorithm 6.1, we use some notational shorthand. We use $\nu_{addr}$ to represent the heuristic score we assign to the function whose entry point is the address $addr$. We use $\varsigma$ to represent the value we increase $\nu_{addr}$ by when a call-site satisfies the "ideal" case; when the "ideal" case is not encountered, we use a multiplier $\delta$ to scale $\varsigma$ such that $0 < \delta \leqslant 1$, prior to incrementing. After processing the function $f$ (the inner loop in Algorithm 6.1), the function-local scores for each possible static data comparison function $\nu'$ are merged into a global map of scores $\nu$. Prior to this merge, we apply two modifiers as *rewards* and *penalties*: we scale up the score of the suspected static data comparison function with the highest number of call-site occurrences within $f$ by a constant $\mu_+$, where $\mu_+ \geq 1$

(accounting for local call frequency), we scale down the score of every function $h$ that references the same static data multiple times by $\alpha_{\text{-}}$, where $0 < \alpha_{\text{-}} \leq 1$ (accounting for argument references). We apply further scaling of $\alpha_{\text{-}}$ by $\alpha_*$ which is raised to the number of non-unique data references $n$, used as arguments to $h$. That is, if $h$ has the address $h_{addr}$, $h_{addr} \leftarrow h_{addr} \cdot \alpha_{\text{-}} \cdot \alpha_*^n$.

In addition to notational shorthand, for brevity, we also reference a number of functions:

- *containsIdealSD* evaluates to *true* if at least one of the expressions in the set passed as an argument satisfy the aforementioned data constraints.
- *branchesOnCall* evaluates to *true* if any variable in the conditional expression of the block is tainted by the last function invocation within the block.
- *branchesOnZCmp* evaluates to *true* if the conditional expression of the block depends on a comparison with 0 (or a semantically equivalent boolean comparison).
- $deg_{in}/deg_{out}$ evaluate to the number of incoming/outgoing edges from the block (as detailed previously in Chapter 2).
- *dependentCall* evaluates to the function that would cause *branchesOnCall* to evaluate to *true*.
- *dependentArgs* evaluates to a map of at most $n$ expressions that correspond to the arguments passed to the function call that *dependentCall* evaluates to.
- *applyPenalties* and *applyRewards* perform the previously outlined score modifications.
- *mergeScores* merges the locally computed scores (on a function-level basis) into the global map of scores.

Table 6.1: Concrete assignments to heuristic variables.

| **Variable** | **Assignment** |
|---|---|
| $\varsigma$ (ideal case increment) | 10 |
| $\delta$ (general case down-scaling) | 0.1 |
| $\mu_+$ (highest frequency reward) | 1.2 |
| $\alpha_{\text{-}}$ (duplicate argument penalty) | 0.5 |
| $\alpha_*$ (duplicate argument scaling) | 0.95 |

To select the concrete assignments of the variables: $\varsigma$, $\delta$, $\mu_*$, $\alpha_{\text{-}}$, and $\alpha_*$, used in the implementation of Stringer, we performed a number of small-scale experiments. In those experiments, from a dataset of 50 firmware images, we manually identified the comparison functions present in each binary, this enabled us to create a list of actual static data comparison functions. We then applied computeHeuristicScores to each of those same binaries in order to extract a list of *expected* comparison functions – corresponding to a particular variable assignment. To find an optimal assignment, we iteratively tested different variable assignments, which we modified according to the constraints outlined in §6.3.3. The assignments listed in Figure 6.1 were those that maximised the number of correctly identified comparison functions over all of the binaries in the dataset. On a test set of 50 binaries, these assignments were able to identify all of the standard C string comparison functions used by each binary.

## 6.4 A Metric for Scoring the Importance of Code

In this section, we discuss the design and implementation of our metric, which using the list of possible static data comparison functions computed using Algorithm 6.1, provides a measure of the degree to which a function's execution is influenced by comparisons with static data, where that data guards the execution of uniquely reachable functionality. More specifically:

1. It discovers branches within each function that are influenced by call-sites of static

data comparison functions.

2. For each of the call-sites, it assigns a block-level score to the static data provided as its arguments. This score gives a measure of how important the comparison the static data was used in is, relative to the other comparisons identified within the function, based upon how much unique functionality a successful comparison with it guards.

3. It computes a function-level score based on the block-level/static data score assignments, which provides a measure of how much a function's decision logic is influenced by comparisons with static data, where that static data guards the execution of uniquely reachable functionality.

For a given binary, the functions assigned the highest scores will be those with a relatively high density of decision logic that depends on comparison with static data, where those comparisons lead to execution of functionality that cannot be reached by other means.

### 6.4.1 Requirements of the Metric

As discussed above, our metric's goal is to maximise the score it assigns to functions that contain decision logic that depends upon static data comparisons, where those comparisons tend to uniquely isolate functionality.

In order to assign block-level scores to static data items (step 2 above), we assign them a score based on the properties of the successor blocks of the block containing the call-site it is used in and their reachability. A high score should be assigned if the only way to reach those successor blocks is due to the comparison with it succeeding. In this case, it will essentially isolate the functionality performed by those successors. There are a number of ways to assign a base score to a given block, and how to use that score to calculate the score for an item of static data that *guards* its execution. We base our calculations on the following observations:

**Number of incident blocks** We consider a block that has many incident edges a so-called *join-point*. In this case, its functionality can be considered to be of less importance than the functionality of a single isolated code path as it is reachable by many paths. Therefore, as its functionality is not uniquely reachable, we calculate the influence it has on the scores assigned to the static data compared in its predecessor blocks as a function of the number of incident edges it has (*i.e.,* $deg_{in}(b)$).

**Branches as "guards" of functionality** The functionality *guarded* by a comparison with static data is that of the immediate successor block associated with the branch condition it influences by evaluating to *true*, and its successor blocks. Where that functionality is uniquely reachable, the block containing the static data comparison will dominate (see Chapter 2) those blocks. We observe that for any given static data comparison, the degree it divides the overall CFG along the branch which is followed when the comparison is *true* is a general indicator of the importance of the static data involved. Therefore, we assign a relatively higher score to static data that guards large amounts of functionality in this manner.

### 6.4.2 Definition of the Metric

We divide the computation of our block-level metric score (*i.e.,* that calculated in step 2 in §6.4) into two stages:

1. For a given function, we first construct sets of static data sequences for each block within its CFG.

2. For each block, using the computed static data sequences, we assign a score to the static data used as part of static data comparisons that influence its reachability.

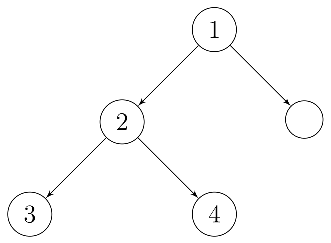


Figure 6.1: Example CFG with static data comparisons.

Table 6.2: Computed string sequence sets for Figure 6.1.

| Label | Computed string sequence set |
|---|---|
| 1 | $\{[]\}$ |
| 2 | $\{[s_1]\}$ |
| 3 | $\{[s_2, s_1]\}$ |
| 4 | $\{[s_1]\}$ |

The computed static data sequence set for a block represents all combinations of possible static data comparisons that must be successful in order to reach that block. For instance, in Figure 6.1, if we consider both nodes 1 and 2 static data comparisons, where the branches from $1 \rightarrow 2$ and $2 \rightarrow 3$ are taken if the comparisons at 1 and 2 evaluate to *true*, then the sets we compute are those shown in Table 6.2. In which we use the notation $s_i$ to represent the static data compared against at node $i$.

**Algorithm 6.2** Algorithm to compute static data sequences.

```
function COMPUTESDS(b)
    for each $p \in preds(b)$ do
        if $branchesOnStaticData(p)$ then
            $s_p \leftarrow branchData(p)$
            $b_s \leftarrow b_s \cup \{S_i +\!\!+ s_p | S_i \in p_s\}$
        else
            $b_s \leftarrow b_s \cup p_s$
        end if
    end for
    if $b_s = \emptyset$ then
        $b_s \leftarrow \{[]\}$
    end if
end function
```

Algorithm 6.2 details our algorithm for computing sets of static data sequences, which we apply to each block until the computed static data sequences for all blocks reach a fix-point. We use the notation $+\!\!+$ to denote the concatenation operator on sequences. Additionally, we make reference the following functions:

- *branchData*($b$) evaluates to the static data compared to at block $b$.

- $branchesOnStaticData(b)$ evaluates to $true$ if the branching of $b$ is influenced by a comparison with static data.

We ignore loops when determining if a fix-point is reached over the sets computed by COMPUTESDS($b$); this ensures termination and avoids the construction of sequences with repeated sub-sequences. If after iterating over all of $preds(b)$, $b_s$ is equivalent to $\emptyset$ (*i.e.,* no paths contain static data that needs to be matched to reach $b_s$), then we set $b_s$ to $\{[]\}$. We use this to represent the situation where there is no known path to reach $b$ that is dependent upon successful comparison with static data.

**Algorithm 6.3** Algorithm to compute metric scores.

```
function computeScores($\omega$, $f$)
    $M \leftarrow \{\}$
    for each $b \in f_{blocks}$ do
        $S \leftarrow b_S$, $baseScore \leftarrow \omega(b)$, $numChains \leftarrow |S|$, $countMap \leftarrow \{\}$
        for each $S_i \in S$ do
            for each $s \in S_i$ do
                $countMap_s \leftarrow countMap_s + 1$
            end for
        end for
        for each $s \in countMap$ do
            $occScale \leftarrow \frac{countMap_s}{numChains}$
            $M_s \leftarrow M_s + baseScore \times \ln\left(1 + occScale \times \frac{1}{deg_{in}(b)}\right)$
        end for
    end for
    return $M$
end function
```

We perform step 2 of the algorithm, *i.e.,* computation of the scores for each element of static data, using Algorithm 6.3. We first assign a base score to each block, which we call basic block complexity; we represent its computation by $\omega(b)$, which evaluates to $|b_{insns}|$ – *i.e.,* the number of lifted (BIL) instructions for the block $b$. We then take each static data sequence computed using Algorithm 6.2 and compute a score for each item of static data used in a static data comparison that influences the reachability of a block. For each block, we take its set of static data sequences $S$ and iteratively compute a score for each element of static data found within the static data sequences. The final result is a map

$M$ of static data and their associated scores.

We approach the computation for static data scores in two sub-phases: for each block, we first construct a mapping of static data to the number of times that data, *i.e.,* $s$, occurs within the static data sequences associated with that block $S$. This count is used to determine a scaling factor as a fraction of the total number of sequences and the count of those that $s$ is present in. We use this value to represent how much influence a successful comparison with a particular element of static data has on the reachability of a block. If in all cases, the data $s$ has to be matched to reach a block, then the fraction will be equivalent to 1. Using these fractions, we update the scores assigned to each element of static data within the map $M$. We compute the update as a function of the sum of the base score assigned to the block – computed by $\omega(b)$ – and perform scaling based on its influence on the reachability of the block and the number of incident edges into the block, *i.e.,* $\frac{1}{deg_{in}(b)}$.

In order to compute the score for a function, we sum the scores assigned to its static data (computed using Algorithms 6.2 and 6.3). Our function-level score, therefore, provides a measure of how much of a function's decision logic is influenced by its static data comparisons, where those comparisons lead to the execution of uniquely reachable functionality.

## 6.5 Experimental Evaluation

Our tool, Stringer, implements our techniques detailed in §6.3 and $6.4, as well as automated firmware acquisition, unpacking and report generation. In this section, we discuss the outcomes of running Stringer on a firmware collection totalling 7,590 successfully unpacked firmware images, corresponding to a total of 2,451,532 non-unique binaries. We performed our tests on a machine with a $3^{rd}$ generation Intel i5 dual-core CPU clocked at 2.6GHz with 16GB of DDR3 RAM.

### 6.5.1 Methodology

To perform firmware acquisition in Stringer, we have implemented web and FTP crawlers for downloading firmware images from 30 different firmware vendors (using the same methodology as §5.4.1). We implemented its firmware unpacking component using the same approach described in §5.3. Of the firmware downloaded, it was able to successfully extract 7,590 usable file-systems out of 15,438 images. Following firmware unpacking, Stringer performs a search for ELF binaries in the extracted file-systems, and using our metric, produces a report containing an ordered list of functions deemed to be most *important* (*i.e.,* that should be analysed first). Figure 6.2 depicts an excerpt of such a report: it lists the function name and corresponding score, the static data that influences its branching behaviour, and the individual scores assigned to items of static data.

```
[f] 37.66: sub_60118
    34.89: 664225 (via: strcmp)
     2.77: root (via: strcmp)
```

Figure 6.2: Excerpt of a report produced by Stringer.

While Stringer automates the majority of the analysis process, a degree of manual intervention is required to discern the most *interesting* binaries from the processed dataset. To find these cases, we perform semi-automated analysis of the reports Stringer generates. We perform this analysis using two methodologies:

1. To discover routines handling common protocols that contain additional, undocumented functionality, we devise simple models of what static data we expect to be grouped together within protocol handler functions. For instance, for a web-server, we expect the terms `GET` and `POST` and possibly, `PUT`, `HEAD` and `DELETE` to be grouped together. We search for cases, where we find such terms, as well as other static data that does not exist within our models. We then perform manual analysis using IDA Pro to ascertain the functionality associated with the unexpected static data.

2. To discover hard-coded credential checks, we search our reports for functions containing static data indicative of authentication, *e.g.,* `admin`, `Administrator`, and `root`. As in case 1, we use IDA Pro to perform manual analysis of the functionality associated with the identified static data comparisons.

In both cases, our manual analysis process is aided by the fact the functions and strings of interest are available from the generated reports and so anomalous functionality, such as backdoors or undocumented commands can quickly be checked for, and confirmed.

We note, that due to the modest amount of backdoors publicly available, that are both present in embedded device firmware, and also of the class that can be detected by Stringer, calculating the TP and FP rates of our technique is infeasible.

### 6.5.2 Case-studies

Due to the large number of firmware binaries processed as part of our analysis, in this section, we present just highlights of that analysis, in the form of case-studies. Each case-study follows a similar structure: we first present the scores and ranking of possible comparison functions as computed by application of our heuristics (from §6.3). We then detail *interesting* functions and static data identified by application of our metric and the results of manually analysing them.

#### 6.5.2.1 Identification of the FTP command set

The `vsftpd` FTP server, shared amongst numerous Linksys device firmware images provides a clear example of the effectiveness of our approach. The binary analysed contains a total of 600 functions, uses static linking and is stripped of symbol information. Our heuristic identifies 44 potential static data comparison functions. Those ranked highest are: `sub_10814` (394.84), `sub_1622C` (35.00), `sub_10754` (27.20), and `sub_139FC` (12.20). As `vsftpd` is open-source software, we are able to discover that `sub_10814` corresponds to the function `str_equal_text` – a string equality check for the `vsftpd`'s custom string implementation.

Our metric finds the highest ranking function to be the main protocol parsing routine: `sub_C4F0` which is assigned a score of 942.08 (and corresponds to the `process_post_login` function). The FTP command set handled by `vsftpd` is extensive; we, therefore, omit the specific output of the tool and associated CFG due to its size.

The uniformity of the scores our metric assigns to the static data used in `sub_C4F0` essentially groups the data associated with FTP protocol message processing. The scores reflect the function's use of a state-machine to handle connections; which following matching its input with a protocol message, transfers control to a secondary message-specific handler function, which performs further input processing or executes the functionality associated with the message. The group of highest scoring protocol messages (`HELP`, ...) are assigned scores of 16.00, while the lowest (such as `PROT`) are assigned 8.03. The largest group of commands – assigned uniform scores of 10.00, contains the core FTP command set (*i.e.,* `STOR`, `RETR`, `PASV`, `PORT`, `LIST`, `QUIT`, ...).

#### 6.5.2.2 Q-See DVR Hard-coded Credential Backdoor

Table 6.3: Scores for `_ZN9CLoginDlg5LogInEPKcS1_b`.

| Label | Score | Static Data | Function | Depends |
|---|---|---|---|---|
| 1 | 171.39 | `admin` | `strcmp` | {[]} |
| 2 | 58.92 | `ppttzz51shezhi` | `strcmp` | {[`admin`]} |
| 3 | 45.13 | `6036logo` | `strcmp` | {[`admin`]} |
| 4 | 42.14 | `6036adws` | `strcmp` | {[`admin`]} |
| 5 | 37.54 | `6036huanyuan` | `strcmp` | {[`admin`]} |
| 6 | 35.21 | `6036market` | `strcmp` | {[`admin`]} |
| 7 | 31.05 | `jiamijiami6036` | `strcmp` | {[`admin`]} |

In a number of firmware images for Q-See DVR products, Stringer is able to identify numerous hard-coded credentials[1] which provide differing *backdoor* functionalities,

[1]To the best of our knowledge, these credentials constitute a previously undiscovered backdoor.

and access to the device. The binary used in this case study, td3520, contains a total of 15,669 functions and is statically linked. Our heuristics for identifying static data comparison functions identify 911 possible functions; those that are ranked highest are: strcmp (1464.70), strncmp (779.33), CRYPTO_malloc (685.10) (from the statically linked OpenSSL library), _ZNKSs7compareEPKc (C++'s string equality operator) (376.20), strstr (306.00), and strcasecmp (196.00). All but one of these functions is a static data comparison function; the single false positive, CRYPTO_malloc, is identified due to its usage patterns being almost identical to that of an expected comparison function, as shown in Figure 6.3.

```
ADD         R0, R11, #0x14
LDR         R1, =0x6C8610 ; "pem_lib.c"
MOV         R2, #0x136
BL          CRYPTO_malloc
SUBS        R10, R0, #0
BNE         0x5E8804
```

Figure 6.3: Example usage of CRYPTO_malloc.

We identify the third highest ranked function by our metric as _ZN9CLoginDlg5LogIn-EPKcS1_b (scoring 421.38), which contains the hard-coded credential checking routine. Table 6.3 shows the scores and sets of static data sequences of the static data extracted from that function, while Figure 6.4 shows a simplified CFG representation of the function with static data labelled as in Table 6.3.

We observe that successor nodes that are dependent on the highest ranked static data (admin) follow from the left branch of the comparison node. All other static data comparisons are dependent upon a successful comparison with admin. The static data ranked as second most important isolates the most unique functionality relative to the other identified static data.

We discovered the report of the binary using methodology 2 (as described in §6.5.1). More precisely, we located it by searching the reports generated by Stringer for common

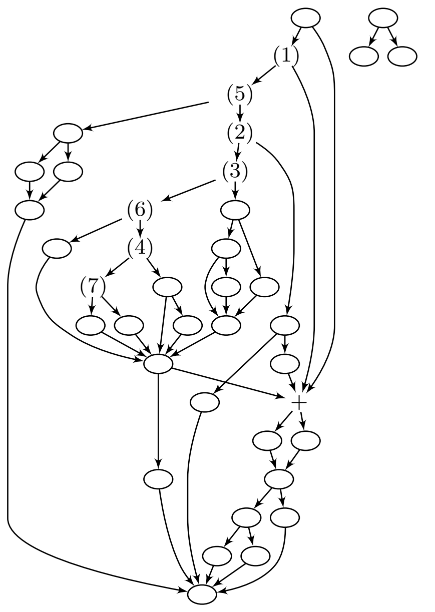


Figure 6.4: CFG for `_ZN9CLoginDlg5LogInEPKcS1_b`.

privileged usernames – in this case, `admin` matched. We verified the existence of the backdoor manually using IDA Pro.

#### 6.5.2.3 Ray Sharp DVR Hard-coded Credential Backdoor

We were able to use reports generated by Stringer to find a further hard-coded credential checking routine in firmware from another DVR vendor, Ray Sharp. The binary containing the backdoor, `raysharp_dvr` contains a total of 7,605 functions, is dynamically linked and stripped of local symbol names. Our heuristics reveal the highest ranked comparison functions to be those from the C standard library: `strcmp` (ranked highest) (5170.30), `strncmp` (1109.73), `strstr` (353.93) and `memcmp` (222.00). It also found `sub_1C7EC` (1351.96), which it ranked second; we were able to identify it as a wrapper around `strcmp` by manual analysis using IDA Pro.

The functions our metric scores highest consist of complex parsing routines – indicated by their relatively high scores compared to other *interesting* functions identified. We found `sub_60118` to contain backdoor-like functionality. Figure 6.5 details the CFG of the function along with the scores assigned to the username and password combination, and Table 6.4 shows the scores as assigned by our metric, as well as the corresponding computed static data sequence sets. Figure 6.6 shows an IDA Pro CFG snippet of the backdoor.

Table 6.4: Scores for `sub_60118`.

| Label | Score | Static Data | Function | Depends |
|---|---|---|---|---|
| 1 | 30.23 | `664225` | `strcmp` | {[]} |
| 2 | 2.77 | `root` | `strcmp` | {[664225]} |

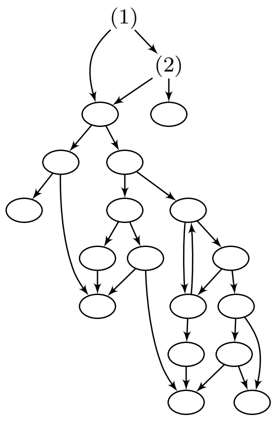
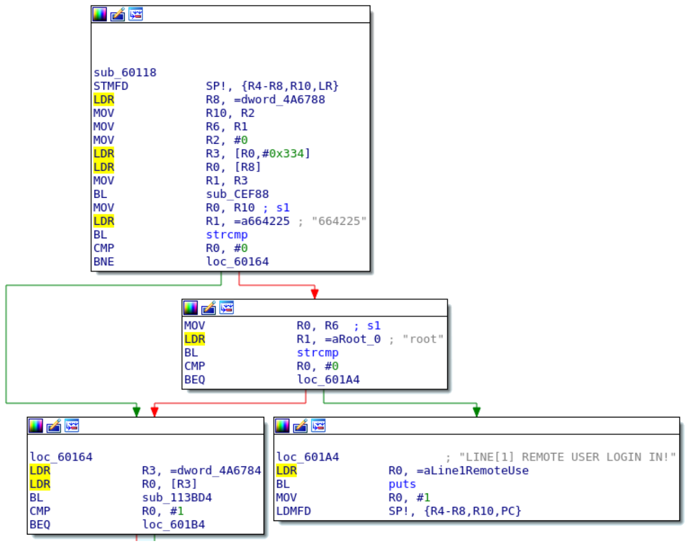


Figure 6.5: Ray Sharp hard-coded credential check.



```
sub_60118
STMFD           SP!, {R4-R8,R10,LR}
LDR             R8, =dword_4A6788
MOV             R10, R2
MOV             R6, R1
MOV             R2, #0
LDR             R3, [R0,#0x334]
LDR             R0, [R8]
MOV             R1, R3
BL              sub_CEF88
MOV             R0, R10 ; s1
LDR             R1, =a664225 ; "664225"
BL              strcmp
CMP             R0, #0
BNE             loc_60164
```

```
MOV             R0, R6  ; s1
LDR             R1, =aRoot_0 ; "root"
BL              strcmp
CMP             R0, #0
BEQ             loc_601A4
```

```
loc_60164
LDR             R3, =dword_4A6784
LDR             R0, [R3]
BL              sub_113BD4
CMP             R0, #1
BEQ             loc_601B4
```

```
loc_601A4                       ; "LINE[1] REMOTE USER LOGIN IN!"
LDR             R0, =aLine1RemoteUse
BL              puts
MOV             R0, #1
LDMFD           SP!, {R4-R8,R10,PC}
```

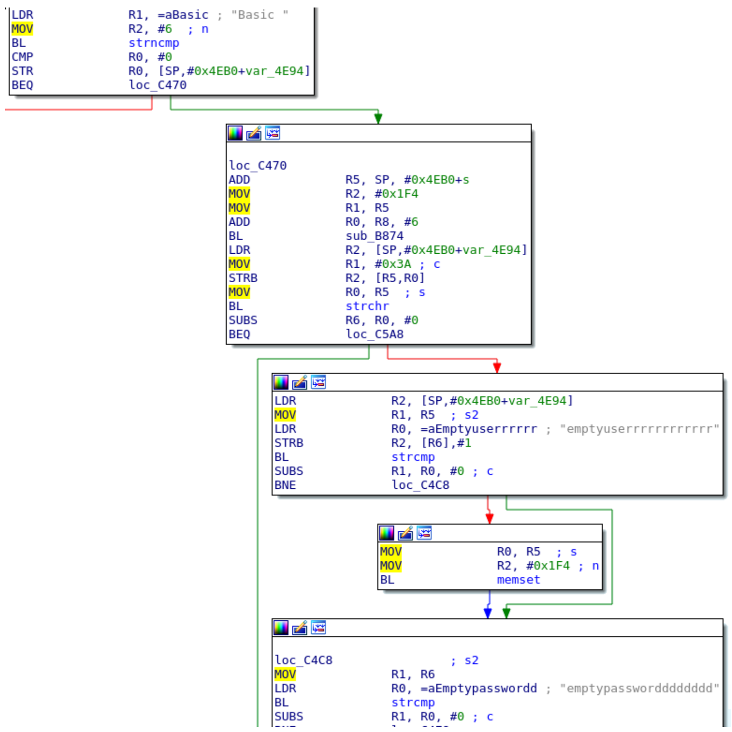
Figure 6.6: IDA Pro CFG snippet of the Ray Sharp backdoor.

We identified this binary using methodology 1 from §6.5.1; we searched the logs for common usernames that are associated with privileged user accounts: in this case, `root` matched. Our *a posteriori* research online revealed that (in contrast to our other case studies), we were not the first to discover this backdoor in Ray Sharp devices. The backdoor has been previously documented [133] and is found to be present in a multitude of devices from many vendors, including: Swann, Lorex, URMET, KGuard, Defender, DEAPA/DSP Cop, SVAT, Zmodo, BCS, Bolide, EyeForce, Atlantis, Protectron, Greatek, Soyo, Hi-View, Cosmos, and J2000.

#### 6.5.2.4 TRENDnet Hard-coded Credential in HTTP Authentication

Through the use of Stringer, we were able to find another hard-coded credential pair, in this case, in the bundled web-server found in the firmware of a number of TRENDnet devices. The hard-coded credentials were located in the routine handling processing of basic HTTP authentication. The credentials are compared against using the standard string comparison function, `strcmp`, which our heuristics successfully rank the most probable static data comparison function. Overall, it identifies 40 such functions, out of a total of 391 within the binary. `strcmp` is ranked highest by a large margin, with a score of 1635.01, followed by `strstr` (481.20), `nvram_get` (413.10), `strncmp` (265.45) and `sub_A2D0` (131.00). Through manual analysis, we identified that `sub_A2D0` provides a wrapper around `hsearch_r` – a lookup function for hash tables, evaluating to 0 on failure – and is hence a false positive. Similarly, `nvram_get` is a false positive, which serves to provide a lookup of the embedded device's NVRAM (see Chapter 2). In both of these cases, the call-sites of the functions exhibit features consistent with static data comparison functions, hence the false identification.

Our metric ranked the function containing the hard-coded credential pair as the eighth most important function – `sub_B958`, assigning it with a score of 827.99. From manual analysis of the function, we found that whilst validating the credentials passed for HTTP basic authentication, an additional code path checks for the hard-coded username/password pair: `emptyuserrrrrrrrrrr/emptypassworddddddd` using `strcmp`. The credentials are assigned scores of 106.00 and 103.47 respectively, and rank as the second and fourth most important items of static data used in the function. The most important is the string used to detect if basic HTTP authentication is being used, which is assigned a score of 151.84. We omit a complete diagrammatic representation of the CFG due to space considerations, and instead show a CFG snippet captured using IDA Pro in Figure 6.7.



```
LDR             R1, =aBasic ; "Basic "
MOV             R2, #6  ; n
BL              strncmp
CMP             R0, #0
STR             R0, [SP,#0x4EB0+var_4E94]
BEQ             loc_C470
```

```
loc_C470
ADD             R5, SP, #0x4EB0+s
MOV             R2, #0x1F4
MOV             R1, R5
ADD             R0, R8, #6
BL              sub_B874
LDR             R2, [SP,#0x4EB0+var_4E94]
MOV             R1, #0x3A ; c
STRB            R2, [R5,R0]
MOV             R0, R5  ; s
BL              strchr
SUBS            R6, R0, #0
BEQ             loc_C5A8
```

```
LDR             R2, [SP,#0x4EB0+var_4E94]
MOV             R1, R5  ; s2
LDR             R0, =aEmptyuserrrrrr ; "emptyuserrrrrrrrrrrr"
STRB            R2, [R6],#1
BL              strcmp
SUBS            R1, R0, #0 ; c
BNE             loc_C4C8
```

```
MOV             R0, R5  ; s
MOV             R2, #0x1F4 ; n
BL              memset
```

```
loc_C4C8                ; s2
MOV             R1, R6
LDR             R0, =aEmptypasswordd ; "emptypassworddddddddd"
BL              strcmp
SUBS            R1, R0, #0 ; c
```

Figure 6.7: IDA Pro CFG snippet of the TRENDnet backdoor.

We located this binary by using methodology 1 from §6.5.1; we constructed a model of the basic HTTP authentication method, and searched the reports produced by Stringer for the expected static data associated with that model. To the best of our knowledge, the hard-coded credentials we found have not been previously documented.

### 6.5.2.5 Recovery of a Netgear SOAP-based Protocol Command Set

We were able to use Stringer to extract a proprietary SOAP-based RPC command set from a web-server, found to be used by a number of Netgear devices. We discovered the RPC functionality in an otherwise standard web-server, `mini_httpd`. The binary we analysed contains 331 functions in total; of those, our heuristics identified 60 as possible static data comparison functions. Those that it ranked highest were a combination of functions from the C standard library: `strcmp`, `strstr` and `strcasecmp`, which were assigned scores of 380.52, 185.00 and 184.00, and a custom static data comparison function (ranked second),

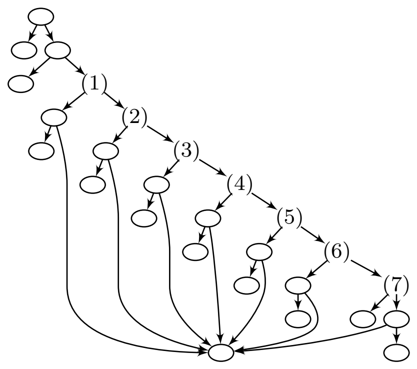


Figure 6.8: CFG fragment for `soap_parent_ctrl_handle`.

`safestrcmp`, which was assigned a score of 221.00.

Our metric ranks `handle_request` (scoring 952.91) as the most important function, which handles parsing and processing of the HTTP protocol. It ranks `do_file` (assigned a score of 486.47) the second most important, and the `main` function third (scoring 449.55) – which provides argument parsing for the binary. It ranks `soap_parent_ctrl_handle` as the fourth most important function, which it assigns a score of 328.75. `soap_parent_-ctrl_handle` handles the processing of a SOAP-based RPC command set. The output of Stringer for this function demonstrates its effectiveness in extracting protocol command sets that are previously unknown to an analyst. The scores it assigns to individual command strings within the function are uniform. Figure 6.8 shows a fragment of the CFG for `soap_parent_ctrl_handle`, while Table 6.5 shows the scores assigned to the static data compared against in that fragment.

We discovered this command set by searching the reports generated by Stringer for web-server related protocol strings, in this case: `GET`. We found this string – amongst other HTTP commands – in the higher scoring function `handle_request`, and `soap_-parent_ctrl_handle` was subsequently found by looking at other high ranking functions within the binary.

Table 6.5: Selection of static data from `soap_parent_ctrl_handle`.

| Label | Score | Static Data | Function | Depends |
|---|---|---|---|---|
| 1 | 7.64 | `EnableTrafficMeter` | `strcmp` | {[]} |
| 2 | 7.64 | `SetTrafficMeterOptions` | `strcmp` | {[]} |
| 3 | 7.64 | `SetGuestAccessEnabled` | `strcmp` | {[]} |
| 4 | 7.64 | `SetGuestAccessEnabled2` | `strcmp` | {[]} |
| 5 | 7.64 | `SetGuestAccessNetwork` | `strcmp` | {[]} |
| 6 | 7.64 | `SetWLANNoSecurity` | `strcmp` | {[]} |
| 7 | 7.64 | `SetWLANWPAPSKByPassphrase` | `strcmp` | {[]} |

### 6.5.3 Performance

In this section, we assess the run-time performance of Stringer. Stringer takes on average 0.573s to process a given binary; however, for larger binaries, with a greater number of functions, or more complex CFGs, this process can take considerably longer. As a concrete example, Stringer takes 44.966 seconds to process the binary `td3520` from the case study in §6.5.2.2, which contains a total of 15,669 functions. Such a binary is a rare case, particularly in Linux-based embedded device firmware, and even with such performance, Stringer is clearly suitable for use in large-scale analysis. A significant portion of the total runtime for Stringer can be attributed to the invocation of IDA Pro, which it uses to export data required for CFG recovery. The total time taken to invoke IDA Pro takes on average 11.26% of its execution time, while processing the exported data for use takes on average 0.63% of the total time. The remainder of the time is due to computation of our heuristic and metric scoring algorithms.

### 6.5.4 Comparison with Existing Techniques

Our techniques substantially improve upon existing techniques for identification of interesting static data. From our research, past work has generally used:

- A combination of the UNIX facility `strings` to first extract stings from binaries, and `grep` to process its output and locate interesting keywords.

- A more robust approach – revolving around the use of IDA Pro and IDAPython to export static data coupled with function names, which is then subsequently manually analysed with, for example, `grep`.

Neither of these existing tool combinations, however, provide any indication of the importance of a given piece of static data in relation to any other. Furthermore, neither provide a means of ranking the importance of functions in relation to how much of their conditional processing is influenced by static data. The lack of both of these properties limits the effectiveness of these methods, meaning that a large amount of manual analysis, and some luck is required when analysing large binaries. Moreover, these techniques only scale to locate functionality based upon *known* protocols or easily recognisable strings.

### 6.5.5 Required Manual Analysis

As noted in the previous section, our approach improves upon current augmented manual analysis techniques. When compared to manual analysis alone, using the case study in §6.5.2.2 as an example, manually checking the binary for the backdoor discovered, in the worst case, amounts to checking a total of 15,669 functions, compared to just three when using Stringer to first analyse the binary. Further, the analysis provided by Stringer not only identifies the function of interest, but also the static data used as part of the hard-coded credential checking routine: such static data can be searched for extremely quickly when performing manual analysis using a tool such as IDA Pro, thus, use of the output of Stringer in this way is able to further reduce the overall effort of manual analysis, compared to an entirely manual approach.

In summary, the manual analysis required when using Stringer amounts to:

- Interpreting the report output by running Stringer: this implies manually analysing each function in order of their scores as assigned by Stringer's metric.

- Using the static data identified within each function analysed as a basis for locating basic blocks that contain comparison functions and performing analysis of the functionality they guard access to.

### 6.5.6 Limitations & Discussion

The analysis approach taken by Stringer relies on heuristics, and, as noted in Chapter 3, such approaches are often negatively impacted when they encounter conditions not accounted for by the observations used to devise their heuristics. Such conditions can be exploited by a backdoor implementer in order to evade our approach and although from our small-scale manual analysis efforts we have not encountered such conditions, we discuss them here for completeness and a thorough consideration of the limitations of our approach.

As Stringer relies on detecting calls-sites of comparison functions, an effective means of thwarting its analysis would be to perform inlining of static data comparison functions. In such a case, if a hard-coded credential check was implemented using inline comparisons, then Stringer would be unable to detect the usage of the hard-coded credentials (*i.e.,* would produce a false-negative result). Similarly, if a backdoor implementer were to devise non-standard comparison functions that were constructed specifically to violate the assumptions in §6.3.1, Stringer would be unable to identify the call-sites of such functions, and, thus, miss any, *e.g.,* hard-coded credential check implemented using those functions, again resulting in false negative results. In both cases, the heuristics Stringer relies upon could be extended to handle such cases, however, as previously noted, from our manual analysis of Linux-based embedded device firmware, we have not encountered a single binary that utilises static data comparisons in a way that violates the assumptions used to devise our heuristics. We attribute that, especially in the case of lack of inlined static data comparisons, to the fact that the vast majority of (current) Linux-based firmware is composed of binaries that are compiled using extremely old compiler toolchains that do not (at least by default) perform such optimisations.

## 6.6 Chapter Summary

In this chapter, we have presented a novel approach to identify static data comparison functions within binaries, which when combined with our function-level scoring metric, as demonstrated, is effective in discovering undocumented functionality and recovering text-based protocol messages and commands. In the case of undocumented functionality, we have identified three instances of authentication backdoors in commodity firmware images from a number of different vendors. Furthermore, we have demonstrated our approach is suitable for large-scale analysis, and have detailed two methodologies for reducing the effort required by a human analyst processing the output of our techniques.

The key utility of our techniques lies in their ability to isolate functions of interest, ranking them within the first tens of functions for an analyst to manually analyse. This is in stark contrast to standard manual analysis, where an analyst often will have to trawl through (potentially) thousands of functions in order to locate that same functionality. A concrete example of this is demonstrated in our case-study of Q-See DVR firmware in §6.5.2.2, whereby the most interesting function for an analyst – which contains a backdoor – is ranked third most important out of 15,669 functions by use of our techniques.

Our approach improves on existing large-scale analysis methods targeted at embedded device firmware by performing more complex static analyses, which consider the control-flow properties of code, as opposed to propagating known bit-string patterns over a large dataset of firmware (*i.e.,* as in the approach taken by Costin et al. [76]). Moreover, our techniques introduce a new means of identifying potential program functionality, which can aid in other program analysis domains; thus, our techniques are applicable beyond merely finding backdoor-like functionality within Linux-based embedded device firmware.

# Part IV

*Closing Statements*

# Chapter 7

*Conclusion & Future Directions*

In the final part of this thesis, we draw conclusions from the work presented in previous chapters, and provide future directions for research.

## 7.1 Contributions & Future Directions

The research presented in this thesis aims to provide a means of detecting backdoor-like functionality within embedded device firmware. Due to the large number of devices available, each having different compositions, we focus our efforts on devices constituting the largest market share, which at the time of writing, are Linux-based devices, running on ARM and MIPS processors.

### 7.1.1 A Definition for Backdoors

Detecting backdoors is a difficult task; automating that detection process is equally challenging. Evidence for these claims lies in the fact that while threats coined that term have been known since at least 2000 [190], there is a distinct lack of tooling for their detection, and the vast majority of them that are publicly documented, are still detected by labourious manual analysis. In Chapter 4, we noted that this is, at least in part, due to the term backdoor, while casually used in both the literature and by the media, previously not having a concrete or rigorous definition. To overcome this, we provided such a definition, *i.e.,* definition 4.1 in Chapter 4. Moreover, to better understand, reason about, and identify backdoor-like functionality, we developed a framework for their componentisation, which serves the dual role of reasoning about both their implementations, and methodologies for their detection. Further, we introduced the notion of deniable backdoors in §4.5, and discussed their implications when attempting to attribute accountability to backdoor implementers. To build on this research, we consider the following directions:

### A Backdoor Detection Approach Based on Framework from Chapter 4

As stated in Chapter 4, the framework we proposed allows us to reason both about current backdoor detection methodologies, such as those we describe in Chapters 5 and 6, and provides a premise for developing new backdoor detection methodologies, that consider a more complete model of what a backdoor is. A clear direction for future research would, therefore, be to investigate how to develop a backdoor detection methodology that utilises both our proposed framework and corresponding model as a basis for its approach. Such an approach could potentially be achieved by a combination of existing methods, for example, using the output of Stringer [174] to guide the symbolic execution of Firmalice [166].

### Quantitative Evaluation of Backdoor Detection Methodologies

While development of new backdoor detection techniques is of course, fundamental, so too is the means by which we evaluate those techniques and compare them to the current state-of-the-art. In its current form, our framework provides a basis for comparing *how*, and *what* components of backdoors a particular approach detects, but it does not provide a quantifiable metric for measuring or comparing their performance, as is required for a more rigorous experimental evaluation. This is a challenging task, as different methodologies will detect different types of backdoors, and there may be little cross-over between the two sets of backdoors two approaches aim to detect. Thus, future research could investigate how to compare methodologies more rigorously; this could be done by, for instance, constructing common datasets of different classes of backdoors, and using them to measure the detection rates of new and existing approaches.

### 7.1.2 Anomalous & Undocumented Program Behaviour

In Chapter 5, we presented a semi-automated approach to detect hidden, undocumented functionality (such as backdoors) within Linux-based embedded device firmware. We implemented our approach in a tool called HumIDIFy [176], which, as we demonstrated, is able to detect a number of backdoors. Additionally, it boasts a correct classification rate of 96.45% in successfully identifying common program functionalities. Our techniques implemented in HumIDIFy focus on identifying undocumented functionality in common program types found prevalently in Linux-based firmware. To perform that identification, we utilise a two-stage process. The first stage takes as input a program binary and produces a classification label and a confidence value: the classification label tells us what *type* of binary – *e.g.,* web-server – it is and the confidence provides a measure of how certain it is in assigning that label to the binary. In the second stage, this label is used to select a (manually created) expected *functionality profile*, which is used to check to see if the binary deviates from the expected properties defined within that functionality profile (*e.g.,* does it use API not common to binaries of its assigned class), which results in an overall judgement: benign or anomalous.

Since our approach is restricted to only being able to classify commonly found binaries, it faces the following problem: if HumIDIFy is presented with a binary that doesn't fall into any of the common classes it is able to identify, it still must assign a label. In the ideal case, this label would be assigned with a low confidence value, and the binary would fail to match its unexpected functionality profile, in the worst case it would be misclassified with a high confidence value and match its expected functionality profile. To overcome this, future development of our techniques could instead attempt to classify binaries based on more general properties, for example, higher-level meta-classes (*e.g.,* this might include a class that detects TCP-only services, or a class that detects services that start other programs). Alternatively, HumIDIFy could provide easier extensibility with respect to adding new program classes. In its current form, to add a new program class, four steps are required:

1. Manually find example binaries that would be classified as the new program class.
2. Perform attribute extraction on those examples (*i.e.,* map the input binaries to attribute vectors).
3. Retrain the classifier component.
4. Manually write an expected functionality profile for the new program class.

In our current system, steps 2 and 3 are automated processes, while steps 1 and 4 are manual. To automate step 4, we would require a system that can find similarities in how functionality manifests within each class of binary. For example, for the web-server class, we would expect that process to identify that web-servers utilise TCP networking, and a web-server that additionally employs UDP-based networking should be considered anomalous. To automate the first step we would require an automated process to take as input an arbitrary collection of binaries, and as output, assign those binaries *meaningful* labels that are indicative of their functionality. New binaries presented to that classifier would be assigned one of these classes as in the previous system. However, given the labels assigned to binaries would no longer be human meaningful (at least in the same way as labels generated and applied manually), substantial effort would be required to create functionality profiles by hand for the generated classes; thus, if the first step were to be automated in this way, the final step would ideally also require automation. We see that automation of this step points to classifying based on general program properties, rather than concrete program types. Based on these observations, we see the following as directions for future research:

### Automated Generation of Program Classes

To address the first part of the automation process, one way of automatically deriving program classes would be to use unsupervised learning. This would partition the input training set into classes, based on the classifier algorithm, its parameters and the input data. The approach we take in Chapter 5 uses API and strings to construct feature vectors to represent programs, which, from small-scale experiments using unsupervised learning, produced machine-generated program classes that had little meaning to a human analyst. Thus, a future direction might be to extract program features based on meaningful functionalities, for example, networking behaviours, file system behaviours and so on, and use those features for unsupervised learning.

### Automated Generation of Functionality Profiles

For automatic generation of functionality profiles, future research could investigate the use of a standard technique often used in data mining and statistics: hierarchical cluster analysis, specifically agglomerative hierarchical clustering (see, *e.g.,* [129]). This (effectively) unsupervised learning technique takes as input a dataset, and a means to compute the distance between instances of that data (their dissimilarity). From this, it works in a bottom-up manner: merging pairs of clusters (each instance is initially in its own cluster) with the minimal distance between them: the new cluster's position in the hierarchy will be relative to the dissimilarity of the instances within it. Clustering data in this way would allow us to identify collections of instances that are most similar. Applying this methodology to generating functionality profiles could work as follows:

- For a set of binaries that should be classified as the same program class where they all contain no unexpected functionality, we could represent each of them as a set of the API they use.
- We could use these sets as input to an agglomerative hierarchical clustering algorithm, with the view of finding a cluster containing instances of programs that share the same API, which we can use to represent the *expected* functionality of their program class.

For the distance metric, we could use Jaccard distance, which computes the dissimilarity between two sets as:

$$d_J(P_i, P_j) = 1 - J(P_i, P_j) = \frac{|P_i \cup P_j| - |P_i \cap P_j|}{|P_i \cup P_j|}$$

From the output of this process, we could synthesise a BFDL representation of the expected functionality representing the set of API functions.

### 7.1.3 Undocumented Commands & Credentials

In Chapter 6, we proposed a new static analysis method that measures the individual influence pieces of static data (such as strings) have upon the control-flow of a program's functions. We implemented our techniques in a tool, Stringer, which we demonstrate is able to aid in identifying a number of backdoors in various Linux-based embedded device firmwares. Our method automatically identifies static data comparison functions within binaries, then labels each function's basic blocks with the set of sequences of static data that must be matched against to reach them. Using these sets, it then assigns a score to each function, which measures the extent to which its branching behaviour (and execution of isolated functionality) is influenced by comparisons with static data. We see the following as a possible future direction for this work:

### Augmenting Existing Tools

As noted in Chapter 4, Stringer takes into account neither the *input source*, nor the *privileged state* components of a backdoor, as modelled by our framework. As future research, in an effort to extend its utility, and provide higher scoring to static data that *guards* access to *privileged* functionality, we could, as taken in the approach of Firmalice, manually identify the *privileged states* of the program being analysed, and incorporate them into the calculation of our metric. In doing this, Stringer would score static data that guards access to unique, *privileged* functionality highly, thus, better matching its requirements outlined in §6.4.1.

## 7.2 Conclusion

In this thesis, we have not only addressed the problem of backdoor detection in embedded device firmware, but provided a more fundamental definition to the term backdoor and a corresponding model, which can be viewed as a premise for the development of new backdoor techniques, applicable to both embedded device firmware and systems in general.

In this chapter, we have discussed avenues for future research building on the work presented in this thesis. All of those avenues take a technical approach, however, we note that, while developing further technical approaches to detect backdoors is, of course, important, what is of equal importance – as highlighted in Chapter 4 – is the need to be able to hold developers and manufacturers accountable for intentional vulnerabilities they insert into their products. This effort requires more than a technical approach, and its solution rather lies in the realms of policy and law. Though, as a community, encouraging projects and organisations to provide greater transparency in their development processes, such that individual developers and contributors can be held accountable for deliberate attempts to compromise the security of programs and devices, will in the long-term, reduce the prevalence of backdoors. This is especially important for binary-only software products, such as PC and embedded device firmware, where currently, there is a staggering amount of products “backdoored on arrival”, without their developers being held properly accountable.

# Appendices

# Appendix A

## *Evaluation of Attribute Selection Algorithms*

Assessment of attribute selection algorithms for deltas 0.1 to 0.7:

Table A.1: `CorrelationAttributeEval`.

| Threshold | Correct (%) |
|---|---|
| 0.1 | 58.1818% |
| 0.2 | 58.1818% |
| 0.3 | 58.1818% |
| 0.4 | 67.8788% |
| 0.5 | 62.4242% |
| 0.6 | 63.6364% |
| 0.7 | 59.3939% |

Table A.2: `GainRatioAttributeEval`.

| Threshold | Correct (%) |
|---|---|
| 0.1 | 58.1818% |
| 0.2 | 58.1818% |
| 0.3 | 58.1818% |
| 0.4 | 67.8788% |
| 0.5 | 62.4242% |
| 0.6 | 63.6364% |
| 0.7 | 59.3939% |

Table A.3: `InfoGainAttributeEval`.

| Threshold | Correct (%) |
|---|---|
| 0.1 | 58.1818% |
| 0.2 | 58.1818% |
| 0.3 | 58.1818% |
| 0.4 | 67.8788% |
| 0.5 | 62.4242% |
| 0.6 | 63.6364% |
| 0.7 | 59.3939% |

Table A.4: `OneRAttributeEval`.

| Threshold | Correct (%) |
|---|---|
| 0.1 | 58.1818% |
| 0.2 | 58.1818% |
| 0.3 | 58.1818% |
| 0.4 | 67.8788% |
| 0.5 | 62.4242% |
| 0.6 | 63.6364% |
| 0.7 | 59.3939% |

Table A.5: `ReliefFAttributeEval`.

| Threshold | Correct (%) |
|---|---|
| 0.1 | 58.1818% |
| 0.2 | 58.1818% |
| 0.3 | 58.1818% |
| 0.4 | 67.8788% |
| 0.5 | 62.4242% |
| 0.6 | 63.6364% |
| 0.7 | 59.3939% |

Table A.6: `SymmetricalUncertAttributeEval`.

| Threshold | Correct (%) |
|---|---|
| 0.1 | 58.1818% |
| 0.2 | 58.1818% |
| 0.3 | 58.1818% |
| 0.4 | 67.8788% |
| 0.5 | 62.4242% |
| 0.6 | 63.6364% |
| 0.7 | 59.3939% |

# Appendix B

## *Example BFDL Functionality Profile*

```
rule handles_socket() =
      function_ref("socket")

rule handles_tcp() =
      handles_socket() && (function_ref("recv")
                           || function_ref("send"))

rule handles_udp() =
      handles_socket() && (function_ref("recvfrom")
                           || function_ref("sendto"))

rule telnetd() =
      handles_tcp() &&
      !handles_udp()
```

Figure B.1: Functionality profile for Telnet daemon.

# List of References

[1] AFLPIN. Accessed: 2018/03/28. [Online]. Available: https://github.com/mothran/aflpin

[2] american fuzzy lop. Accessed: 2018/03/28. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[3] ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2. Accessed: 2018/03/28. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0031c/index.html

[4] BARF: A multiplatform open source Binary Analysis and Reverse engineering Framework. Accessed: 2018/03/28. [Online]. Available: https://github.com/programa-stic/barf-project

[5] Binary Ninja. Accessed: 2018/03/28. [Online]. Available: https://binary.ninja

[6] Binwalk. Accessed: 2018/03/28. [Online]. Available: https://github.com/ReFirmLabs/binwalk

[7] BusPirate. Accessed: 2018/03/28. [Online]. Available: http://dangerousprototypes.com/docs/Bus_Pirate

[8] clang: a C language family frontend for LLVM. Accessed: 2018/03/28. [Online]. Available: https://clang.llvm.org/

[9] CramFS. Accessed: 2018/03/28. [Online]. Available: https://sourceforge.net/projects/cramfs/

[10] Dagger. Accessed: 2018/03/28. [Online]. Available: https://github.com/repzret/dagger

[11] Firmware Mod Kit. Accessed: 2018/03/28. [Online]. Available: https://code.google.com/archive/p/firmware-mod-kit/

[12] GCC, the GNU Compiler Collection. Accessed: 2018/03/28. [Online]. Available: https://gcc.gnu.org/

[13] GNU Screen. Accessed: 2018/03/28. [Online]. Available: https://www.gnu.org/software/screen/

[14] IDA Pro. Accessed: 2018/03/28. [Online]. Available: https://www.hex-rays.com/products/ida/

[15] JTAGulator. Accessed: 2018/03/28. [Online]. Available: http://www.grandideastudio.com/jtagulator/

[16] LLVM Language Reference Manual. Accessed: 2018/03/28. [Online]. Available: https://llvm.org/docs/LangRef.html

[17] McSema. Accessed: 2018/03/28. [Online]. Available: https://github.com/trailofbits/mcsema

[18] Memory Technology Device (MTD) Subsystem for Linux. Accessed: 2018/03/28. [Online]. Available: http://www.linux-mtd.infradead.org/doc/general.html

[19] Metasploit Framework. Accessed: 2018/03/28. [Online]. Available: https://www.metasploit.com/

[20] Minicom. Accessed: 2018/03/28. [Online]. Available: https://alioth.debian.org/projects/minicom

[21] Open On-Chip Debugger. Accessed: 2018/03/28. [Online]. Available: http://openocd.org/

[22] OpenREIL. Accessed: 2018/03/28. [Online]. Available: https://github.com/Cr4sh/openreil

[23] QEMU. Accessed: 2018/03/28. [Online]. Available: https://www.qemu.org/

[24] radare. Accessed: 2018/03/28. [Online]. Available: https://rada.re/r/

[25] Remill. Accessed: 2018/03/28. [Online]. Available: https://github.com/trailofbits/remill

[26] Sasquatch. Accessed: 2018/03/28. [Online]. Available: https://github.com/devttys0/sasquatch

[27] SquashFS. Accessed: 2018/03/28. [Online]. Available: http://squashfs.sourceforge.net/

[28] Standard Performance Evaluation Corporation. Accessed: 2018/03/28. [Online]. Available: https://www.spec.org/

[29] The Binary Analysis Tool (BAT). Accessed: 2018/03/28. [Online]. Available: http://www.binaryanalysis.org/

[30] UBI – Unsorted Block Images. Accessed: 2018/03/28. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubi.html

[31] Valgrind. Accessed: 2018/03/28. [Online]. Available: http://www.valgrind.org/

[32] VEX IR. Accessed: 2018/03/28. [Online]. Available: https://sourceware.org/git/?p=valgrind.git;a=blob;f=VEX/pub/libvex_ir.h;hb=HEAD

[33] (1997) System V Application Binary Interface. Accessed: 2018/03/28. [Online]. Available: http://www.sco.com/developers/devspecs/gabi41.pdf

[34] (2005) LZMA Reference Implementation. Accessed: 2018/03/28. [Online]. Available: https://sourceforge.net/projects/sevenzip/files/LZMA%20SDK/4.23/

[35] (2013) Multiple Vulnerabilities in D-Link DIR-600 and DIR-300 (rev B). Accessed: 2018/03/28. [Online]. Available: http://www.s3cur1ty.de/node/672

[36] (2013) TCP-32764 Backdoor. Accessed: 2018/03/28. [Online]. Available: https://github.com/elvanderb/TCP-32764

[37] (2014) CVE-2014-0160 (Heartbleed). Accessed: 2018/03/28. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160

[38] (2016) Backdoor in Sony IPELA Engine IP Cameras. Accessed: 2018/03/28. [Online]. Available: https://sec-consult.com/en/blog/2016/12/backdoor-in-sony-ipela-engine-ip-cameras/

[39] (2016) MMD-0056-2016 - Linux/Mirai, how an old ELF malcode is recycled. Accessed: 2018/03/28. [Online]. Available: http://blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxmirai-just.html

[40] (2016) Multiple vulnerabilities found in Quanta LTE routers. Accessed: 2018/03/28. [Online]. Available: http://pierrekim.github.io/blog/2016-04-04-quanta-lte-routers-vulnerabilities.html

[41] (2016) Netis Router Backdoor Update. Accessed: 2018/03/28. [Online]. Available: https://blog.trendmicro.com/netis-router-backdoor-update/

[42] (2017) BIL Accuracy. Accessed: 2018/03/28. [Online]. Available: https://github.com/BinaryAnalysisPlatform/bap/wiki/Bil-accuracy

[43] (2017) Hacking the Western Digital MyClound NAS. Accessed: 2018/03/28. [Online]. Available: https://blog.exploitee.rs/2017/hacking_wd_mycloud/

[44] (2017) Nucleus. Accessed: 2018/03/28. [Online]. Available: https://bitbucket.org/vusec/nucleus

[45] (2018) Cisco IOS XE Software Static Credential Vulnerability. Accessed: 2018/03/28. [Online]. Available: https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20180328-xesc

[46] D. Aha and D. Kibler, "Instance-based learning algorithms," *Machine Learning*, vol. 6, 1991.

[47] D. Andriesse and H. Bos, "Instruction-Level Steganography for Covert Trigger-Based Malware," in *Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, ser. DIMVA '14, 2014.

[48] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proceedings of the 25th USENIX Security Symposium*, ser. SEC '16, 2016.

[49] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-Agnostic Function Detection in Binaries," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, ser. EuroS&P '17, 2017.

[50] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *Proceedings of the 2011 Network and Distributed System Security Symposium*, ser. NDSS '11, 2011.

[51] G. Balakrishnan and T. Reps, "WYSINWYX: What You See is Not What You eXecute," *ACM Transactions on Programming Languages and Systems*, 2010.

[52] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, "The Page-fault Weird Machine: Lessons in Instruction-less Computation," in *Proceedings of the 7th USENIX Conference on Offensive Technologies*, ser. WOOT '13, 2013.

[53] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "ByteWeight: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium*, ser. SEC '14, 2014.

[54] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *Proceedings of the 2009 Network and Distributed System Security Symposium*, ser. NDSS '09, 2009.

[55] A. R. Bernat and B. P. Miller, "Anywhere, Any-time Binary Instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, ser. PASTE '11, 2011.

[56] P. Biondi, R. Rigo, S. Zennou, and X. Mehrenberger, "BinCAT: purrfecting binary static analysis," in *Proceedings of the 2017 Symposium sur la Securite des Technologies de l'Information et des Communications*, ser. SSTIC '17, 2017.

[57] A. Bondy and M. R. Murty, *Graph Theory.* Springer-Verlag London, 2008.

[58] M. Bourquin, A. King, and E. Robbins, "BinSlayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, ser. PPREW '13, 2013.

[59] D. Bradbury, "SCADA: a critical vulnerability," *Computer Fraud & Security*, vol. 2012, no. 4, 2012.

[60] S. Bratus, M. Locasto, M. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation," {*USENIX; login:*}, vol. 36, no. 6, 2011.

[61] L. Breiman, “Random Forests,” *Machine Learning*, vol. 45, no. 1, 2001.

[62] M. Brocker and S. Checkoway, “iSeeYou: Disabling the MacBook Webcam Indicator LED,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC ’14, 2014.

[63] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “BAP: A Binary Analysis Platform,” in *Proceedings of the 2011 International Conference on Computer Aided Verification*, ser. CAV ’11, 2011.

[64] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI ’08, 2008.

[65] S. Cesare and Y. Xiang, “Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs,” in *Proceedings of the IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, ser. TrustCom ’11, 2011.

[66] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing Mayhem on Binary Code,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP ’12, 2012.

[67] D. D. Chen, M. Woo, D. Brumley, and M. Egele, “Towards Automated Dynamic Analysis for Linux-based Embedded Firmware,” in *Proceedings of the 2016 Network and Distributed System Security Symposium*, ser. NDSS ’16, 2016.

[68] V. Chipounov and G. Candea, “Enabling Sophisticated Analyses of x86 Binaries with RevGen,” in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, ser. DSNW ’11, 2011.

[69] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A Platform for In-vivo Multipath Analysis of Software Systems,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011.

[70] W. Christensen. (1982) XMODEM. Accessed: 2018/03/28. [Online]. Available: http://techheap.packetizer.com/communication/modems/xmodem.html

[71] J. G. Cleary and L. E. Trigg, “K*: An Instance-based Learner Using an Entropic Distance Measure,” in *Proceedings of the 12th International Conference on Machine Learning*, 1995.

[72] W. W. Cohen, “Fast Effective Rule Induction,” in *Proceedings of the 12th International Conference on Machine Learning*, 1995.

[73] L. Cojocar, T. Kroes, and H. Bos, "JTR: A Binary Solution for Switch-Case Recovery," in *Proceedings of the 2017 International Symposium on Engineering Secure Software and Systems*, ser. ESSoS '17, 2017.

[74] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "PIE: Parser Identification in Embedded Systems," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC '15, 2015.

[75] A. Costin, "Security of CCTV and Video Surveillance Systems: Threats, Vulnerabilities, Attacks, and Mitigations," in *Proceedings of the 6th International Workshop on Trustworthy Embedded Devices*, ser. TrustED '16, 2016.

[76] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC '14, 2014.

[77] A. Costin, A. Zarras, and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces," in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*, ser. ASIACCS '16, 2016.

[78] A. Costin, A. Zarras, and A. Francillon, "Towards Automated Classification of Firmware Images and Identification of Embedded Devices," in *Proceedings of the 32nd International Conference on ICT Systems Security and Privacy Protection*, ser. IFIP SEC '17, 2017.

[79] A. Cui, M. Costello, and S. J. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," in *Proceedings of the 2013 Network and Distributed System Security Symposium*, ser. NDSS '13, 2013.

[80] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *Proceedings of the 22nd USENIX Security Symposium*, ser. SEC '13, 2013.

[81] A. Djoudi and S. Bardin, "BINSEC: Binary Code Analysis with Low-Level Regions," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '15, 2015.

[82] S. Dolan. (2013) `mov` is Turing-complete. Accessed: 2018/03/28. [Online]. Available: http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf

[83] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable Reverse Engineering with PANDA," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, ser. PPREW '15, 2015.

[84] T. Dullien and S. Porst, "REIL : A platform-independent intermediate representation of disassembled code for static code analysis," in *Proceedings of CanSecWest 2009*, ser. CSW '09, 2009, accessed: 2018/03/28. [Online]. Available: http://blog.zynamics.com/2010/08/24/the-reil-language-part-iv/

[85] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," in *Proceedings of the 2005 Symposium sur la Securite des Technologies de l'Information et des Communications*, ser. SSTIC '05, 2005.

[86] T. F. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. Preprint, 2017.

[87] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *Proceedings of the 2016 Network and Distributed System Security Symposium*, ser. NDSS '16, 2016.

[88] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable Graph-based Bug Search for Firmware Images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[89] C. Forsberg. (1985) YMODEM. Accessed: 2018/03/28. [Online]. Available: http://textfiles.com/programming/ymodem.txt

[90] C. Forsberg. (1988) ZMODEM. Accessed: 2018/03/28. [Online]. Available: http://gallium.inria.fr/~doligez/zmodem/zmodem.txt

[91] E. Frank, M. Hall, and B. Pfahringer, "Locally Weighted Naive Bayes," in *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*, 2003.

[92] E. Frank and I. H. Witten, "Generating Accurate Rule Sets Without Global Optimization," in *Proceedings of the 15th International Conference on Machine Learning*, 1998.

[93] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10, 2010.

[94] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, 1977.

[95] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian Network Classifiers," *Machine Learning*, vol. 29, 1997.

[96] D. Gao, M. K. Reiter, and D. X. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 10th International Conference on Information and Communications Security*, ser. ICICS '08, 2008.

[97] I. Gotovchits. (2015) A formal specification for BIL: BIL Instruction Language. Accessed: 2018/03/28. [Online]. Available: https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.1/bil.pdf

[98] M. A. Hall, "Correlation-based Feature Subset Selection for Machine Learning," Ph.D. dissertation, University of Waikato, 1998.

[99] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations Newsletter*, vol. 11, no. 1, 2009.

[100] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, "Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08, 2008.

[101] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao, "The connected-component labeling problem: A review of state-of-the-art algorithms," *Pattern Recognition*, vol. 70, 2017.

[102] C. Heffner. (2013) From China with Love. Accessed: 2018/03/28. [Online]. Available: http://www.devttys0.com/2013/10/from-china-with-love/

[103] C. Heffner. (2013) Reverse Engineering a D-Link Backdoor. Accessed: 2018/03/28. [Online]. Available: http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/

[104] R. C. Holte, "Very simple classification rules perform well on most commonly used datasets," *Machine Learning*, vol. 11, 1993.

[105] X. Hu, T. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security*, ser. CCS '09, 2009.

[106] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *Proceedings of the 2001 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2001.

[107] A. Hunter. (2008) A Brief Introduction to the Design of UBIFS. Accessed: 2018/03/28. [Online]. Available: http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf

[108] W. Iba and P. Langley, "Induction of One-Level Decision Trees," in *Proceedings of the 9th International Workshop on Machine Learning*, ser. ML '92, 1992.

[109] R. Islam, R. Tian, L. Batten, and S. Versteeg, "Classification of Malware Based on String and Function Feature Selection," in *Proceedings of the 2010 2nd Cybercrime and Trustworthy Computing Workshop*, ser. CTC '10, 2010.

[110] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications.* CRC Press, 1999.

[111] Y. Ji, X. Zhang, and T. Wang, “Backdoor attacks against learning systems,” in *2017 IEEE Conference on Communications and Network Security*, ser. CNS ’17, 2017.

[112] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, “Binary Function Clustering Using Semantic Hashes,” in *Proceedings of the 2012 11th International Conference on Machine Learning and Applications - Volume 01*, ser. ICMLA ’12, 2012.

[113] G. H. John and P. Langley, “Estimating Continuous Distributions in Bayesian Classifiers,” in *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*, 1995.

[114] R. Johnson and S. Christie. JTAG 101 IEEE 1149.x and Software Debug. Accessed: 2018/03/28. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/jtag-101-ieee-1149x-paper.pdf

[115] Kaspersky Lab. (2015) Equation Group: Questions and Answers. Accessed: 2018/03/28. [Online]. Available: https://web.archive.org/web/20150217023145/https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf

[116] G. Keizer. (2010) Is Stuxnet the ‘best’ malware ever? Accessed: 2018/03/28. [Online]. Available: https://www.infoworld.com/article/2626009/malware/is-stuxnet-the--best--malware-ever-.html

[117] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice.* CRC Press, 2009.

[118] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, “Testing Intermediate Representations for Binary Analysis,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’17, 2017.

[119] R. Kohavi, “The Power of Decision Tables,” in *Proceedings of the 8th European Conference on Machine Learning*, 1995.

[120] J. Z. Kolter and M. A. Maloof, “Learning to Detect and Classify Malicious Executables in the Wild,” *Journal of Machine Learning Research*, vol. 6, 2006.

[121] J. Kornblum, “Identifying Almost Identical Files Using Context Triggered Piecewise Hashing,” *Digital Investigation*, vol. 3, 2006.

[122] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental Security Analysis of a Modern Automobile," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10, 2010.

[123] Kryptowire. (2016) How a Crypto 'Backdoor' Pitted the Tech World Against the NSA. Accessed: 2018/03/28. [Online]. Available: https://www.kryptowire.com/adups_security_analysis.html

[124] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, 1955.

[125] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast Location of Similar Code Fragments Using Semantic 'Juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, ser. PPREW '13, 2013.

[126] D. Lee. (2012) Flame: Massive cyber-attack discovered, researchers say. Accessed: 2018/03/28. [Online]. Available: http://www.bbc.com/news/technology-18238326

[127] K. Lu, S. Xiong, and D. Gao, "RopSteg: Program Steganography with Return Oriented Programming," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14, 2014.

[128] H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, and D. Gao, "Software Watermarking Using Return-Oriented Programming," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '15, 2015.

[129] O. Maimon and L. Rokach, *Data Mining and Knowledge Discovery Handbook*. Springer-Verlag New York, Inc., 2005.

[130] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary Hunting with Inter-procedural Control Flow," in *Proceedings of the 15th International Conference on Information Security and Cryptology*, ser. ICISC '12, 2012.

[131] D. Miyani, "BinPro: A Tool for Binary Backdoor Accountability in Code Audits¡Paste¿," Master's thesis, University of Toronto, 2016.

[132] B. Möller, T. Duong, and K. Kotowicz. (2014) This POODLE Bites: Exploiting The SSL 3.0 Fallback. Accessed: 2018/03/28. [Online]. Available: https://www.openssl.org/~bodo/ssl-poodle.pdf

[133] H. D. Moore. (2013) Ray Sharp CCTV DVR Password Retrieval & Remote Root. Accessed: 2018/03/28. [Online]. Available: https://community.rapid7.com/community/metasploit/blog/2013/01/28/ray-sharp-cctv-dvr-password-retrieval-remote-root

[134] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, N. Japkowicz, and Y. Elovici, "Unknown malcode detection and the imbalance problem," *Journal in Computer Virology*, vol. 5, no. 4, 2009.

[135] M. Muench, A. Francillon, and D. Balzarotti, "Avatar$^2$: A multi-target orchestration platform," in *Workshop on Binary Analysis Research*, ser. BAR '18, 2018.

[136] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification," in *2011 International Symposium on Visualization for Cyber Security*, ser. VizSec '11, 2011.

[137] B. H. Ng and A. Prakash, "Exposé: Discovering Potential Binary Code Re-use," in *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*, ser. COMPSAC '13, 2013.

[138] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis.* Springer-Verlag Berlin Heidelberg, 1999.

[139] J. Oakley and S. Bratus, "Exploiting the Hard-working DWARF: Trojan and Exploit Techniques with No Native Executable Code," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, ser. WOOT '11, 2011.

[140] A. Osborne, *An Introduction to Microcomputers Vol 1: Basic Concepts.* McGraw-Hill, 1980.

[141] F. Pagani, M. Dell'Amico, and D. Balzarotti, "Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis," in *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18, 2018.

[142] D. Papp, L. Buttyán, and Z. Ma, "Towards Semi-automated Detection of Trigger-based Behavior for Software Security Assurance," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17, 2017.

[143] Y. H. Park, D. S. Reeves, V. Mulukutla, and B. Sundaravel, "Fast malware classification by automated behavioral graph matching," in *Proceedings of the 6th Cyber Security and Information Intelligence Research Workshop*, ser. CSIIRW '10, 2010.

[144] R. Parvez, P. A. S. Ward, and V. Ganesh, "Combining Static Analysis and Targeted Symbolic Execution for Scalable Bug-finding in Application Binaries," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '16, 2016.

[145] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-Architecture Bug Search in Binary Executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15, 2015.

[146] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, “Leveraging Semantic Signatures for Bug Search in Binary Programs,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC ’14, 2014.

[147] R. Quinlan, *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, 1993.

[148] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets.* Cambridge University Press, 2011.

[149] G. Ramalingam, “The Undecidability of Aliasing,” in *ACM Transactions on Programming Languages and Systems*, ser. TOPLAS ’94, 1994.

[150] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware Evolutionary Fuzzing,” in *Proceedings of the 2017 Network and Distributed System Security Symposium*, ser. NDSS ’17, 2017.

[151] K. Rieck, P. Trinius, C. Willems, and T. Holz, “Automatic analysis of malware behavior using machine learning,” *Journal of Computer Security*, vol. 19, no. 4, 2011.

[152] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-Oriented Programming: Systems, Languages, and Applications,” *ACM Transactions on Information and System Security*, vol. 15, no. 1, 2012.

[153] N. S. (2012) Pin - A Dynamic Binary Instrumentation Tool. Accessed: 2018/03/28. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[154] J. Salwan and F. Saudel, “Triton : Framework d’exécution concolique et d’analyses en runtime,” in *Proceedings of the 2015 Symposium sur la Securite des Technologies de l’Information et des Communications*, ser. SSTIC ’15, 2015.

[155] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamzeh, “Malware detection based on mining API calls,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC ’10, 2010.

[156] R. Santamarta. (2013) Identify Backdoors in Firmware By Using Automatic String Analysis. Accessed: 2018/03/28. [Online]. Available: http://blog.ioactive.com/2013/05/identify-back-doors-in-firmware-by.html

[157] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, “Idea: Opcode-Sequence-Based Malware Detection,” in *Proceedings of the 2010 International Symposium on Engineering Secure Software and Systems*, ser. ESSoS ’10, 2010.

[158] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, “N-grams-based File Signatures for Malware Detection,” in *Proceedings of the 11th International Conference on Enterprise Information Systems*, ser. ICEIS ’09, 2009.

[159] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, “Data Mining Methods for Detection of New Malicious Executables,” in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, ser. SP ’01, 2001.

[160] F. Schuster and T. Holz, “Towards Reducing the Attack Surface of Software Backdoors,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, ser. CCS ’13, 2013.

[161] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications,” in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP ’15, 2015.

[162] B. Schwarz, S. Debray, and G. Andrews, “Disassembly of Executable Code Revisited,” in *Proceedings of the Ninth Working Conference on Reverse Engineering*, ser. WCRE ’02, 2002.

[163] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, “Detecting unknown malicious code by applying classification techniques on OpCode patterns,” *Security Informatics*, vol. 1, no. 1, 2012.

[164] R. Shapiro, S. Bratus, and S. W. Smith, ““Weird Machines” in ELF: A Spotlight on the Underappreciated Metadata,” in *Proceedings of the 7th USENIX Conference on Offensive Technologies*, ser. WOOT ’13, 2013.

[165] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing Functions in Binaries with Neural Networks,” in *Proceedings of the 24th USENIX Security Symposium*, ser. SEC ’15, 2015.

[166] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware,” in *Proceedings of the 2015 Network and Distributed System Security Symposium*, ser. NDSS ’15, 2015.

[167] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, ser. SP ’16, 2016.

[168] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “BitBlaze: A New Approach to Computer Security via Binary Analysis,” in *Proceedings of the 4th International Conference on Information Systems Security*, ser. ICISS ’08, 2008.

[169] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting Fuzzing Through Selective Symbolic Execution,” in *Proceedings of the 2016 Network and Distributed System Security Symposium*, ser. NDSS ’16, 2016.

[170] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, “Verifying information flow properties of firmware using symbolic execution,” in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE ’16, 2016.

[171] M. Sumner, E. Frank, and M. Hall, “Speeding up Logistic Model Tree Induction,” in *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2005.

[172] S. J. Tan, S. Bratus, and T. Goodspeed, “Interrupt-oriented Bugdoor Programming: A Minimalist Approach to Bugdooring Embedded Systems Firmware,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC ’14, 2014.

[173] X. Tang, Y. Liang, X. Ma, Y. Lin, and D. Gao, “On the effectiveness of code-reuse-based android application obfuscation,” in *Proceedings of the 19th International Conference on Information Security and Cryptology*, ser. ICISC ’16, 2016.

[174] S. L. Thomas, T. Chothia, and F. D. Garcia, “Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality,” in *Proceedings of the 22nd European Symposium on Research in Computer Security*, ser. ESORICS ’17, 2017.

[175] S. L. Thomas and A. Francillon, “Backdoors: Definition, Deniability & Detection,” in *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID ’18, 2018, to appear.

[176] S. L. Thomas, F. D. Garcia, and T. Chothia, “HumIDIFy: A Tool for Hidden Functionality Detection in Firmware,” in *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA ’17, 2017.

[177] R. Tian, L. M. Batten, M. R. Islam, and S. Versteeg, “An automated classification system based on the strings of trojan and virus families,” in *Proceedings of the 4th International Conference on Malicious and Unwanted Software*, ser. MALWARE ’09, 2009.

[178] R. Tian, L. M. Batten, and S. Versteeg, “Function length as a tool for malware classification,” in *Proceedings of the 3rd International Conference on Malicious and Unwanted Software*, ser. MALWARE ’08, 2008.

[179] A. Tillequin. amoco. Accessed: 2018/03/28. [Online]. Available: https://github.com/bdcht/amoco

[180] J. Vanegue, "The Weird Machines in Proof-Carrying Code," in *Proceedings of the 2014 IEEE Security and Privacy Workshops*, ser. SPW '14, 2014.

[181] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on iOS: When Benign Apps Become Evil," in *Proceedings of the 22th USENIX Conference on Security Symposium*, ser. SEC '13, 2013.

[182] D. Woodhouse. (2001) JFFS2: The Journalling Flash File System, version 2. Accessed: 2018/03/28. [Online]. Available: https://www.sourceware.org/jffs2/

[183] C. Wysopal and C. Eng, "Static Detection of Application Backdoors," *Black Hat USA*, 2007, accessed: 2018/03/28. [Online]. Available: http://blackhat.com/presentations/bh-dc-08/Wysopal-Eng/Whitepaper/bh-dc-08-wysopal-eng-WP.pdf

[184] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog Malicious Hardware," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, ser. SP '16, 2016.

[185] Y. Ye, T. Li, K. Huang, Q. Jiang, and Y. Chen, "Hierarchical associative classifier (HAC) for malware detection from the large and imbalanced gray list," *Journal of Intelligent Information Systems*, vol. 35, no. 1, 2010.

[186] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Proceedings of the 2014 Network and Distributed System Security Symposium*, ser. NDSS '14, 2014.

[187] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, "Implementation and Implications of a Stealth Hard-drive Backdoor," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13, 2013.

[188] K. Zetter. (2013) How a Crypto 'Backdoor' Pitted the Tech World Against the NSA. Accessed: 2018/03/28. [Online]. Available: https://www.wired.com/2013/09/nsa-backdoor/

[189] K. Zetter. (2015) Secret Code Found in Juniper's Firewalls Shows Risk of Government Backdoors. Accessed: 2018/03/28. [Online]. Available: http://www.wired.com/2015/12/juniper-networks-hidden-backdoors-show-the-risk-of-government-backdoors

[190] Y. Zhang and V. Paxson, "Detecting Backdoors," in *Proceedings of the 9th USENIX Conference on Security Symposium*, ser. SEC '00, 2000.

[191] X. Zhu and A. B. Goldberg, “Introduction to semi-supervised learning,” *Synthesis lectures on artificial intelligence and machine learning*, vol. 3, no. 1, 2009.