The Pennsylvania State University
The Graduate School

INTRUSION DETECTION IN IOT NETWORKS WITH KERNEL-LEVEL HARDWARE MONITORING

A Thesis in
Computer Science and Engineering
by
Adrien Cosson

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2020

Abstract

IoT systems have been broadly adopted, and we are now seeing increasing deployment in both home and commercial environments. However, with this broad distribution of new technology, there has been an introduction of new classes of attacks, specifically targeting IoT networks and devices. Due to the constrained natures of IoT devices, as well as the opacity of IoT framework, standard intrusion detection systems cannot be applied here. In this paper, we introduce *Sentinel*, a new framework aimed at facilitating the conception of novel detection system. By leveraging common features of IoT frameworks, we expose, collect and centralize low-level system information of each smart device in a network. We demonstrate that the data collected contains some strong signal, by designing a proof of concept intrusion detector that reaches a 95.7% accuracy. We also perform a power consumption analysis to prove that *Sentinel* is compatible with the power requirements of battery-operated devices, by increasing the power usage by less than 1%. We believe that this framework can be used to design highly performant, specialized IoT intrusion detection systems.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1900873 and Grant No. CNS-1564105, Amendment No. 003. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Chapter 1 | Introduction

Internet of Things (or IoT) is a new technology domain that has been steadily increasing in popularity in the past decade. It is today a multi-billion dollar industry, and has found many applications. The most commonly encountered use of IoT is in smart homes, where IoT devices (such as light bulbs, cameras, or door locks) are installed and connected to provide a way of automating daily tasks and overseeing the state of a home. Another main application of IoT is industrial environment monitoring, where wireless sensors are deployed to provide real-time feedback of the state of a production line or warehouse. A more recently emerging application domain is healthcare, where patient care and monitoring machines are connected to provide a centralized and more accessible view of their current condition.

Over the past decade, security issues have been frequently found in most devices and frameworks. Attacks specialized against IoT systems have been developed and deployed. The most notable one is the Mirai botnet. This malware, that was first detected in late 2016, infected more than half a million devices in a span of a few months, and used this network to launch a series of DDoS attacks. This included the largest one to date, with a throughput of 623 Mbps [1].

These insecurities are mostly due to economics; as IoT devices have to be relatively cheap to be competitive, manufacturers and application developers often forgo good security practices as a way to keep costs down. Furthermore, as the IoT market is fast-moving, with new products announced every month, a short time-to-market is often key for manufacturers, causing security concerns to be delegated to low-priority, or even post-launch items. Although intrusion detection for standard networks is a mature field, it is still developing for IoT networks. This stems from these devices having different constraints, like low computing power, meshed and ad-hoc structure, or limited battery capacity. These properties require the use of intrusion detection methods that do not incur an overhead in processing and power usage. Since traditional intrusion detection methods rely heavily on local pre-processing work, they are nor applicable to IoT systems [2].

We believe that these security issues are inherent to IoT. Indeed, IoT frameworks implementations are too opaque and high-level to provide a useful level of insight into the low-level state of the devices. These frameworks abstract away the specifics of the hardware, by presenting a simple software interface layered on top of an embedded platform, to which the user is not granted direct access to. We posit that this lack of visibility inside the devices is an key advantage for adversaries, allowing them to hide their actions among the high-level traffic. By piercing through this layer of opacity, and exploiting the information we obtain, we thwart this advantage. Further, it has been shown that low-level access can be a powerful way to detect anomalous behavior [3].

We hypothesize that, as IoT devices have a simple and unique purpose, low-level data obtained by accessing their underlying OS contains a strong signal, that can be used for malicious behavior detection. As such, we are not trying to replace existing security mechanisms and policies, but rather add a new layer, based on exploiting so far unused data. Indeed, most state-of-the-art IoT Intrusion Detection systems rely on protocol-specific network information [4–7].

In order to verify this hypothesis, we design a novel infrastructure around existing IoT ecosystems. A key insight is that providing this system-level data reporting can be achieved for a low performance cost, and in a scalable fashion. This novel system also leverages the presence of a centralized high-capacity, non power-constrained node in the IoT network (a "hub"), but is agnostic to the actual IoT framework used, making it compatible with most of the common existing ones [8–10]. The data collected can be processed either locally at the hub, or remotely to build a more centralized detector.

This work focuses on the collection and examination of a set of metrics from devices made to perform a single task. This simplicity of the nodes makes it possible to keep the number of parameters measured low, while still being representative of the state of the device. This approach could not be applied to more traditional network environments, where nodes are either multi-purpose workstations or dedicated servers, as these have higher complexity when compared to IoT devices. As such, our approach is fit for IoT networks, and only those.

In this paper, we propose a novel intrusion detection framework specialized for IoT environments, *Sentinel*. This framework exposes low-level system information to the user-space of every IoT device, periodically sampling this data, and centralizing it at the hub of the network. The low-level data is extracted by a Linux Kernel Module installed on the devices, that retrieves the latest values of the queried metrics from the kernel. A polling application, running on each IoT node, is responsible for collecting this information at a regular rate and forwarding it to the network hub. This hub runs a data collection application that listens, receives incoming data samples, and stores the data for potential intrusion detection systems to use.

We evaluate the efficacy of low-level system monitoring by using *Sentinel* to record data with synthetic experiments. These consist of cycling each device in the network through their possible states, while subjecting them to a set of common IoT attack side-effects. The dataset gathered is then used to train and evaluate multiple Intrusion Detection models (based on Decision Trees), in order to prove that the data provided by *Sentinel* contains valuable signal for malicious behavior detection. We verify this by looking at the detection rates of the Decision Trees, as well as their true positive and false positive rate. We find that the true positive rate for each attack is above 91%, with an average detection rate of 95.8%. We also prove that the additional power use from the components of our framework fit the requirements of battery-powered devices by observing the instantaneous electrical consumption. We measure this power usage for different values of data polling rate. We find that for a sampling period of 10 seconds or more, *Sentinel* causes an increase of power consumption of less than 1% when compared to a non-instrumented device, while retaining an detection rate of 93%.

In summary, this paper makes the following contributions:

- We introduce *Sentinel*, a real-time low-level system monitoring framework for IoT devices.
- We generate datasets by running simulated attacks against an IoT network, and record their effects.
- We use these datasets to confirm that *Sentinel* provides pertinent information for intrusion detection.
- We measure the impact *Sentinel* has on power consumption to validate that it can realistically be used in a production environment.

In performing this work, we prove that *Sentinel* is a useful addition to the IoT security space, by leveraging up until now unexploited data sources. Further, this data only contains truly valuable information due to the single-purpose nature of IoT devices, and would most likely not be useful for a general-purpose machine, such as a desktop workstation.

Chapter 2 | Background

2.1 IoT Frameworks

An IoT framework is a set of systems that connect and establish communication between multiple devices and a centralized location, while providing a unified user interface. All IoT frameworks will almost always posses the same core elements:

1. A hub, which is a device located at the center of the network, in charge of maintaining and controlling a list of connected nodes.

2. A device API, that allows smart things manufacturers to expose their functionalities in a way that can be used by the hub.

3. A user interface, usually in the form of a smartphone app or a web-app.This interface allows the user to see the state of their devices, control them, add new ones, and create rules to automate their behavior.

This architecture is presented in Figure 2.1.

In an IoT environment, it is common for the nodes to have limited computation power and energy capacity. As these devices are very specialized, they are designed to just fit their power requirements, often leaving very little room for additional software. This implies that any additional security feature that may want to be added has to be as minimal as possible, and offload some work to a sturdier machine, be it the hub or a dedicated device.

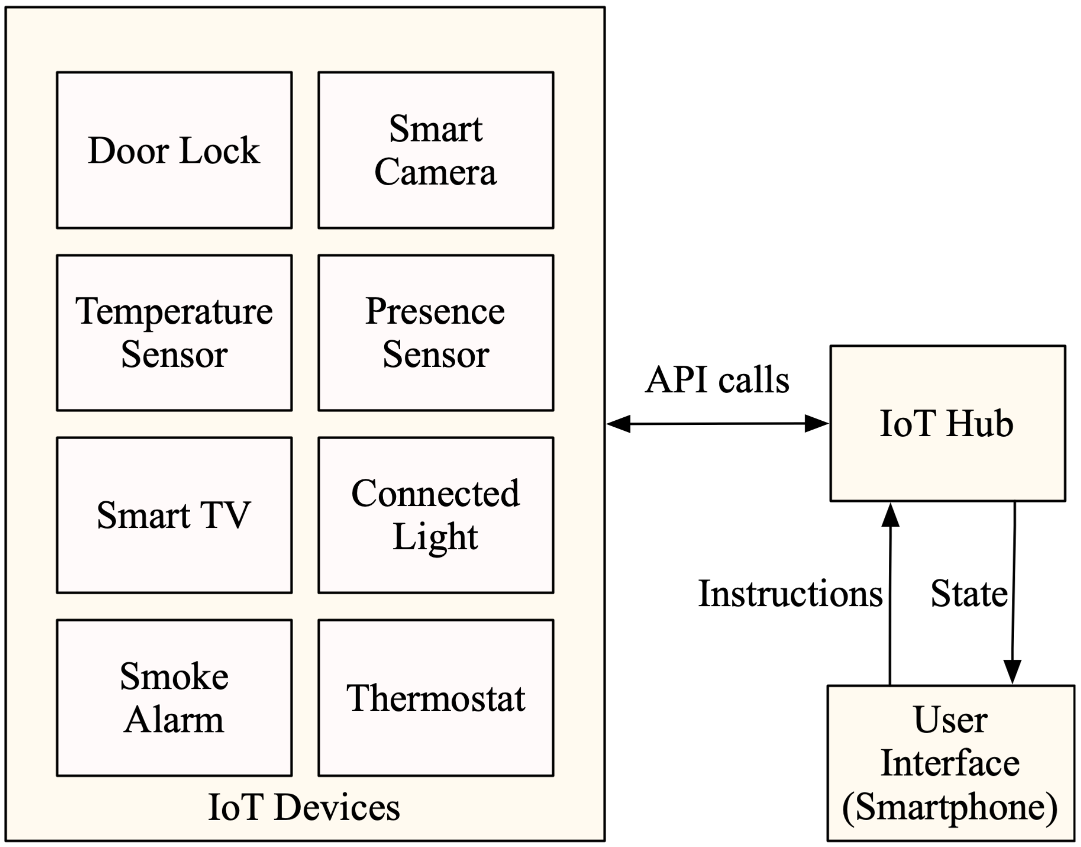

Figure 2.1. Architecture of an IoT framework

2.2 Intrusion Detection Systems

Intrusion detection is a domain of computer security dedicated to monitoring a system for any malicious behavior. Intrusion Detection Systems (IDS) exist to cover different classes of systems, and are generally separated in two architectures.Network-based IDS (NIDS) monitor the state of an entire network in search of malicious agents, by gathering network-level metrics and processing them at a central location. Host-based IDS (HIDS) run on a specific host and search for malware operating inside of it, through the use of system-level and process-level information [11].

There are three approaches an IDS designer can use to detect malicious behavior. Signature-based detection can be performed against threats that have been detected and identified in the past, by comparing the pattern of collected data to a list of known malicious signatures in search of a match. While very efficient, this type of detector is powerless against new attacks [12].

Anomaly-based detection takes a different path, by building an internal representation of the system, which is compared to an expected baseline state. This baseline has to be learned by the system through the observation of known benign behavior. Any discrepancy can then be detected and handled. However, this method is more prone to false positives, as it relies heavily on a statistical approach to the detection [13].

The third main Intrusion Detection logic is specification-based. Similarly to the anomaly-based detection, the system possesses a set of baseline and threshold values that are compared to the current situation. While the previous method infers these values from observation, they are manually defined by a human expert in this method. This allows the system to start acting immediately after being turned on, as it does not require any training. Further, the false positive rate is usually lower compared to anomaly-based detection. However, the need for human intervention makes this system poorly scalable, as any change made to the infrastructure will force the rules to be updated [13].

All three approaches rely on the same base principle: the collection of actionable data, from which decisions are made. However, due to their heterogeneous nature, IoT systems seldom present a standardized access method to this crucial information.

2.3 IoT network attacks

IoT attacks are usually sorted in three different categories: node-level, network-level, and application-level attacks [14].

2.3.1 Node-level

These attacks focus on targeting a single device. Due to their low-power nature, IoT devices are very vulnerable to DoS (Denial of Service) attacks. This class of attacks is characterized by an attacker rendering a device unresponsive. This is often achieved by flooding it with requests to saturate its CPU [15]. For battery-powered devices, this can also be done through a “battery draining” attack, where the attacker sends a constant flow of requests to the device. This prevents it from entering sleep mode which exhausts its battery at a much higher rate than normal, and causes the device to shut down once it is depleted [16, 17].

Physical attacks take a different path to node-disruption. In this instance, the attacker needs to have physical access to the device to compromise it, but the resulting attacks are much harder to detect. A common type of physical attack is RFID tampering, where the adversary leverages common vulnerabilities in the RFID protocol to disrupt the network [18].

2.3.2 Network-level

Network-level attacks correspond to attacks having an influence on the IoT network as a whole. The most often encountered attacks of this type are routing attacks, which uses a wide variety of methods to manipulate the network flow to the adversary's advantage. These attacks only work in meshed networks, where every device can be responsible for routing packets to their destination. The simplest routing attack is the black hole attack, where a compromised node will advertise itself as the optimal route to every other node, and drop every packets received, effectively stopping communications in the network. A grey hole attack operates in the same way, but only drops a fraction of received traffic [19].

Another class of network-level attacks, encountered both in traditional networks and in IoT network, are passive listening attacks. In this scenario, the attacker, having taken over a device, uses it to eavesdrop on the network traffic to gather insufficiently protected sensitive information. This data gathering can also be achieved by exploiting network side-channels, such as packet timings or channel bands used.

2.3.3 Application-level

Finally, application-level attacks are designed to disrupt or take down a specific application running on a node (so-called “edge computing” nodes). This can be achieved in many different ways, depending on the application running on the node. For instance, a node can be tricked into downloading a malicious file from the Internet, allowing an attacker to take it over [20]. Another way of targeting an application is by running a Man-in-the-Middle attack. This is done by spoofing a service the device needs to connect to, and using this connection to sniff sensitive information [21]. As these attacks are very dependent on the actual application running on the device, our work does not focus on them.

2.4 Mirai Botnet

Mirai is the latest significant IoT botnet in history, having infected hundreds of thousands of devices over a few months and using them to launch large-scale DDoS attacks. Mirai infects devices by using default or common login-password combinations. A Mirai bot works by scanning for vulnerable machines. Once a vulnerable machine is found, it is reported to the Mirai “C&C” server. The C&C server then sends this information to a different server that hosts an SQL database. Finally, it notifies a third server, the “loader”, which exploits the vulnerability, sends the payload and executes a new instance of the bot [1].

The separation of C&C and loader is done in order to prevent the botnet from being taken down if the C&C is blocked. Furthermore, the loader is not referenced by IP address but by domain name in the bot executable, allowing for a short downtime in case the C&C is targeted. The attacker only has to spawn a new machine and update the DNS records to point to it, restoring the connection to the entire botnet.

2.5 Related work

The idea of collecting low-level host data for intrusion detection purposes is not new. Garfinkel & Rosenblum [22] proposed an IDS architecture relying on this very idea, by running the host’s application in a virtual machine, and exposing low-level information about the application to a local IDS. Such an architecture provides isolation between the potentially compromised host application and the IDS, while still giving access to a fine-granularity level of detail. Forrest et al. [3] showed that observing sequences of privileged syscalls made by an application could be used to reliably detect certain classes of attacks. However, it seems that so far no similar approach has been done for IoT environments.

There have been many works focused on building IoT intrusion detection systems, most of them based on a network-level approach. SVELTE is an IDS focused on 6LoWPAN networks, developed by Raza et al. [4]. This system, aimed at detecting routing attacks, builds a map of the meshed network during a learning phase, then monitors network flow for any behavior not matching this mapping. INTI [7] is another 6LoWPAN IDS, that detects attacks by having the nodes first categorize themselves in clusters based on their proximity. Packet flow monitoring is then performed at every node, and used to calculate a level of trust for each one. When a node’s trust dips too low, it is considered compromised, and is eliminated from its cluster. Kalis, proposed by Midi et al. [6] is a more general-purpose IoT IDS. The core idea being: while most attacks can be detected reliably by an existing IDS, there is no IDS that is able to identify all attacks on its own. The system learns the specifics of the network (devices type, network layout) and uses this knowledge to select the IDS that make sense in this scenario. Moustafa et al. [23]

presented an NIDS for IoT based on statistical flow features. From a given network features dataset, a subset is picked based on the features with the lowest cross-correlation (i.e. the features the most independent from every other). These features are then passed on to a set of three different machine learning models, which each determine whether the traffic observed is malicious or benign. Their decisions are used in a weighted vote to arrive at a final decision.

Chapter 3 Sentinel

3.1 Threat Model

We assume that the attacker has access to the network, and is able to run arbitrary code on any compromised node. Further, the attacker can take over the IoT application itself to make it run arbitrary code. However, we assume that the attacker does not have privileged access, nor can they disrupt the kernel, as it would then be trivial to replace our kernel module with a malicious one that would only report fake information.

Regarding the hub, we assume that the attacker does not have any access to the component of *Sentinel* running on it, and cannot alter its behavior nor kill it. We make no assumption regarding the capability of the attacker to get access to the information exposed by *Sentinel*.

The attacker can have a wide array of goals. Possibilities include: using corrupted devices as part of a botnet for DDoS attacks, sniffing network traffic to collect sensitive information, or disrupting the network to make it non-functional.

3.2 Architecture

Sentinel is a novel framework designed to help detect node-level and network-level attacks on IoT networks. The purpose of this system is to serve as a data aggregation platform, and expose the collected information to an IDS specialized for a given environment. By providing an IoT framework-agnostic system, it relieves IDS designers from the burden of adapting their methods to the low-level specifics of the network. As IoT is a fundamentally constrained domain, both in computation power and power consumption, *Sentinel* aims to be as lightweight as possible on the network nodes, by offloading most of the heavy work to the hub, a centralized, higher power device.

Sentinel is built upon a Linux Kernel Module running on every IoT device in the monitored network (number 1 in Figure 3.1), and providing various low-level metrics to the userspace. Each of these devices then runs a data sampling application (2) that periodically collects theses metrics and sends them over to the IoT hub. In the hub runs a data collection component (3), in charge of receiving and storing this data, and exposing it through an API, to allow an IDS (4) to use this information.

While the idea of running some part of *Sentinel* on every device we want to secure is a reminder of a host-based Intrusion Detection System (HIDS), due to the constraints inherent to the IoT devices, we cannot afford to perform any detection work locally. Instead, we centralize all the data at the hub and run an IDS either there (or on another remote machine), making our architecture more closely related to a network-based IDS (NIDS).

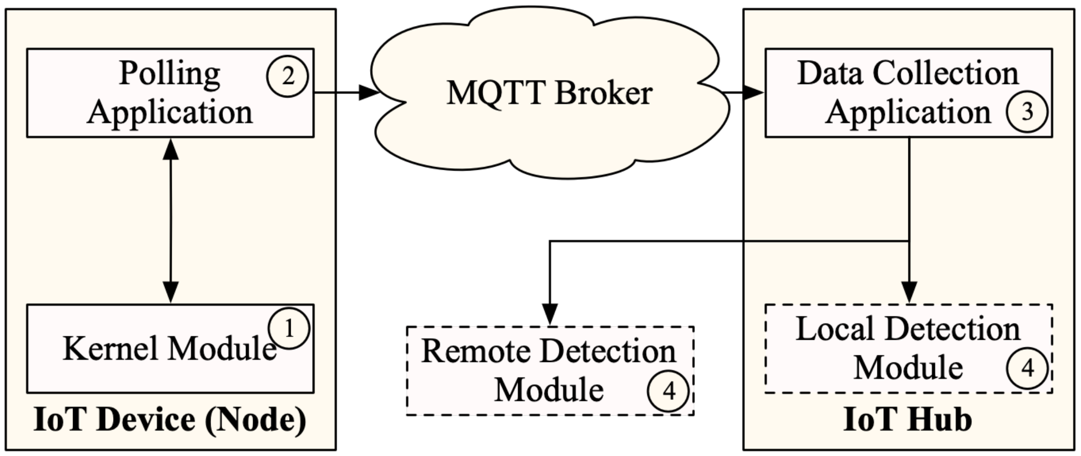

Figure 3.1. Structure of Sentinel

3.2.1 Kernel Module

The Sentinel Kernel Module (SKM) is in charge of exposing useful low-level metrics to userspace, in a machine-friendly and centralized location. This data can then be accessed by any process that needs it.

The SKM is a Linux Kernel Module, installed on every node that needs to be monitored. Once installed, it creates a set of entries in the `sysfs` filesystem, each one corresponding to a metric, with its contents mappable to a standard C type. Whenever a file is read, the SKM is informed, and fetches the wanted value directly from the relevant kernel data structure. `sysfs` is a RAM-based filesystem, introduced in the Linux

Kernel as a replacement for the legacy `profcs`. It provides a file-based view of the kernel data-structures by giving developers an easy interface to export `kobjects`. Each file in `sysfs` corresponds to a single item (e.g. the temperature of a device). These files can be either read-only (if it would not make sense to write to them), or can allow writing in order to configure the device they relate to [24].

We chose to use a Kernel Module for two main reasons. Such a module allows for lower performance overhead than a userspace application, as the transition between userspace and kernelspace does not provoke a context switch. Additionally, a kernel module needs less computing power to perform its task than a regular application, as all the data it needs already exists in the kernel memory and simply needs to be read. Meanwhile, a userspace application would have to parse the output of specific commands or kernel files.

We chose to use `sysfs` because this filesystem is considered to be the standard way to expose kernel information to userspace [25].

The implementation of the SKM allows any user on the device to access the exposed data. This might be considered too lax, as this information could be deemed sensitive. One way to remediate this issue would be to only allow certain users to access it, limiting the access to only the monitoring application. Another route would be to perform some cryptographic manipulation of the data inside the SKM, so that only the hub could interpret it after receiving it. This would provide end-to-end confidentiality of the data, at the expense of higher computation cost on the node.

The metrics exposed by the SKM are available in Table 3.1. In addition to the system-level parameters made available by the SKM, we also allow exposing information about an arbitrary process. The target process can be dynamically changed at any time. The values we collect and make available are presented in Table 3.2. We choose to expose these specific metrics for two main reasons. They are easy to find in the kernel, and do not require any complex computation to be obtained, allowing us to keep the performance hit of the SKM minimal. Further, these metrics have been proved to provide actionable information for intrusion detection, as seen in the related work.

Adding new metrics to be reported by the SKM is straightforward, as it only requires to add a new `sysfs` entry in the module source code, and write a function that fetches the corresponding data from a relevant kernel object.

Metric	**Unit**
Number of logical CPUs	N/A
Frequency of each CPU	kHz
Total, free, and available RAM	kB
Total, free, and available swap	kB
Number of running processes	N/A
1, 5, and 15 minutes loads	N/A

Table 3.1. System-level metrics exposed by Sentinel

Metric	**Unit**
Current physical memory used	kB
Current virtual memory used	kB
High-water physical memory	kB
High-water virtual memory	kB
Number of file descriptors open	N/A

Table 3.2. Process-level metrics exposed by Sentinel

3.2.2 Data Sampling

With the data exposed by the SKM, we can collect it from the node's userspace. The data sampling component periodically reads all the data available, and forwards it to the hub.

The communication between the nodes and the hub is done via a Message Queue Telemetry Transport (MQTT) layer, a publisher-subscriber protocol commonly found in home IoT networks [26]. In an MQTT system, clients can publish and subscribe to topics (e.g. `living_room/thermostat`). Any message published to a topic will be relayed to its subscribers. In our implementation, the entire MQTT traffic is secured with SSL certificates, both for the server and the clients. This ensures that the reported data is not sent to an attacker masquerading as the broker, and prevents an attacker from injecting malicious values.

As MQTT is almost ubiquitous in home IoT environments, having *Sentinel* piggyback on it helps reduce the need for extra software installation and maintenance. It also provides a scalable platform, allowing an arbitrary number of nodes to send their information to the hub.

In the current implementation, the device polling is done at a fixed rate. Setting this rate too high can cause an overabundance of data at the hub, and cause high CPU use on the nodes. On the other hand, setting it too low may cause an attack to be missed or detected later than it could have been. This is why our application also provides a way to *a fortiori* change the polling rate, in order to allow an IDS to implement some drill-down policies. For instance, the polling rate can be set to a low value in standard conditions, and increased whenever a suspicious behavior is detected.

3.2.3 Data Aggregation

Once the data from the nodes is sent to the MQTT broken, the hub needs to notify the broker it wants to receive it. As an IoT network needs to be flexible to allow for devices being frequently added and removed, the data collection component dynamically detects data coming from unknown nodes, and is able to handle nodes reconnecting.

Upon receiving a data record from a node over the MQTT connection, the hub unpacks it and inserts into a local PostgreSQL database instance. This database also provides an open interface for remote connections.

The use of a PostgreSQL database was dictated by the need for a data access method providing concurrent access, as well as remote access. PostgreSQL being an industry-tested framework also helps the stability of our system.

Having the database instance run locally is convenient as it provides faster access time. However, this causes the hub to be a centralized point of failure, as the loss of the device will cause all the stored data to be lost. For this reason, the data aggregation component is able to use a remote database instead, at the expense of performance.

3.2.4 Intrusion Detection

Once the data has been collected and transmitted over to the hub, it is stored and can be accessed either by the hub or by a remote machine, for real-time monitoring.

The data storage is done in a PostgreSQL instance. The data format used in this database is specified in a Python library based on SQLAlchemy. This library presents an API for easy data retrieval and manipulation, and new custom functions can be designed to fit a user's needs.

We choose to use a PostgreSQL database as it provides a reliable, production-tested data storage solution. It also supports native concurrent access, allowing a real-time data processing solution to be accessing it while *Sentinel* is still collecting more measurements. Finally, an SQL database also provides a well-known low-level API to allow any user to create their own data access functions without having to rely on the methods provided by *Sentinel*.

While it is reliable, PostgreSQL is not the fastest or lightest SQL solution that could be used. In particular, if no concurrent or remote access is needed, a simple system like SQLite could be used. However, this change is easy to implement, as no part of the system relies on any specific property of PostgreSQL.

This component allows *Sentinel* to be fully agnostic to the actual IDS system implemented, local or remote. However, in order to prove the usefulness of this framework, we provide a proof of concept for a simple Decision Tree-based IDS in the evaluation of our work. The goal of this proof of concept is to demonstrate that the metrics collected hold some intrinsic information that can be used to successfully detect and identify an attack, while remaining compatible with IoT constraints.

Chapter 4 | Testbed Environment

In this, we first present how we select and implement the attacks we use in our evaluation. We detail which IoT frameworks are used, and why they are chosen. We then list out which IoT devices are implemented, and how we orchestrate their behavior during data recording sessions.

4.1 Simulated Attacks

In order to evaluate how *Sentinel* can help detect IoT network attacks, we need to observe the data it collects while running an attack, or at least simulating the side effects of said attack. In order to determine which side effects should be implemented, we turn to Mirai.

From it, we can extract three main side effects of interest: the network scanning phase, the C&C connection, and the new target reporting. We then focus on these three behaviors, which also happen to be common traits of IoT network attacks, in our evaluation.

We decided to simulate the side effects of Mirai, rather than actually run it, because we believe that the minute details of the attack are not what *Sentinel* focuses on. Rather, we simply implemented the side effects of these behaviors, as would be seen on an infected device. Further, we also implement the effects of a black/grey hole attack, as it is a commonly encountered behavior in IoT network intrusions.

4.1.1 Network Scan / Pivoting

A pivoting action consists of scanning the newly reachable network with Nmap. In order to be as realistic as possible, we do the same here: the device running the attack ping scans a server continuously.

4.1.2 Exfiltration

For this behavior, the side effects are somewhat the inverse of the previous one: a large amount of outbound traffic, and no increase in inbound. We simulate this by sending large UDP packets to a server that simply discards them.

4.1.3 C&C Keep-alive

In order for the attacker to keep control of its infected devices, they need to periodically exchange a heartbeat message, to confirm that it is still reachable and compromised. This is simulated by periodically pinging a remote machine, that responds with an empty payload.

4.1.4 Black/Grey Hole Attack

The side effect of such an attack is a large amount of inbound network traffic and a small amount of outbound traffic. This is simulated by having the device connect to a server, and the server sending a large message ($\geq$1MB) in response. In addition, for a grey hole attack, a random amount of received messages will be sent out to simulate the partial packet drop created by the attack.

4.2 IoT Frameworks Considered

In order to assess the real-world usefulness of *Sentinel*, we install it on two different IoT networks, each of them running a different IoT framework, and with a similar device architecture. The two frameworks considered are Home Assistant and WebThings.

Home Assistant is an open-source framework, providing integration with most commercial IoT devices. A strong emphasis is put on user freedom, allowing them to create their own devices and guaranteeing a fully local processing. This framework's hub can be installed on any main OS, and provides extensive configurability. Connecting a device to the hub is done by creating an "integration", which defines what interactions are possible with the device, and how the hub can perform them. As creating an integration is complex task, a common way for enthusiasts to create their own devices is to use the pre-existing MQTT integration. This integration allows the user to define a new device (e.g. a new light) by simply listing its MQTT topic, as well as its capabilities. The only work to be done is then to write the client-side handler that reacts to the MQTT messages coming from the hub, and sends back acknowledgements.

WebThings is Mozilla's open-source implementation of the Web of Things (WoT), an initiative aiming to standardize IoT. As with Home Assistant, this framework focuses on giving users the tools they need to configure and control their networks as they want. The WebThings Gateway (WebThings name for a hub) can be run any Linux machine, and provides a local data-processing. Adding a device to the network simply is as simple as implementing a few API endpoints and running a web-server on the device. The Gateway then automatically detects and connects to the server. From here, the user only has to program the device to perform its work when a callback function is called.

Another very popular IoT framework is SmartThings. SmartThings is a closed-source framework owned by Samsung. It provides a way for consumers to connect their home IoT appliances, either through a commercially available hub, or through the SmartThings cloud. The user is able to write their own control logic to orchestrate the devices using the SmartThings web IDE. However, the only way to build a SmartThings network is to use a pre-approved commercial hub, on which it is not possible to install arbitrary kernel modules or software. As such, we decided against using it in our evaluation. However, as the commercial hubs run on Linux, we believe it would be possible for *Sentinel* to be integrated by their manufacturers.

4.3 IoT Network Layout

The network we simulate runs on a set of Raspberry Pi 4, each one representing a different IoT device. At the center of the network is the hub. Every other device runs an implementation of its role in the used IoT framework, as well as *Sentinel*. In order to help visualize the set of devices used in our testbed, Figure 4.1 illustrates an example floor plan of a studio using all of them. The hardware we use to simulate the devices are listed in the Table 4.1.

4.3.1 Home Assistant Implementation Details

All the IoT devices are implemented in Home Assistant with the default MQTT component. Each device is essentially a Python application that subscribes and publishes to the relevant MQTT topics. For instance, a door lock will interact with the following topics:

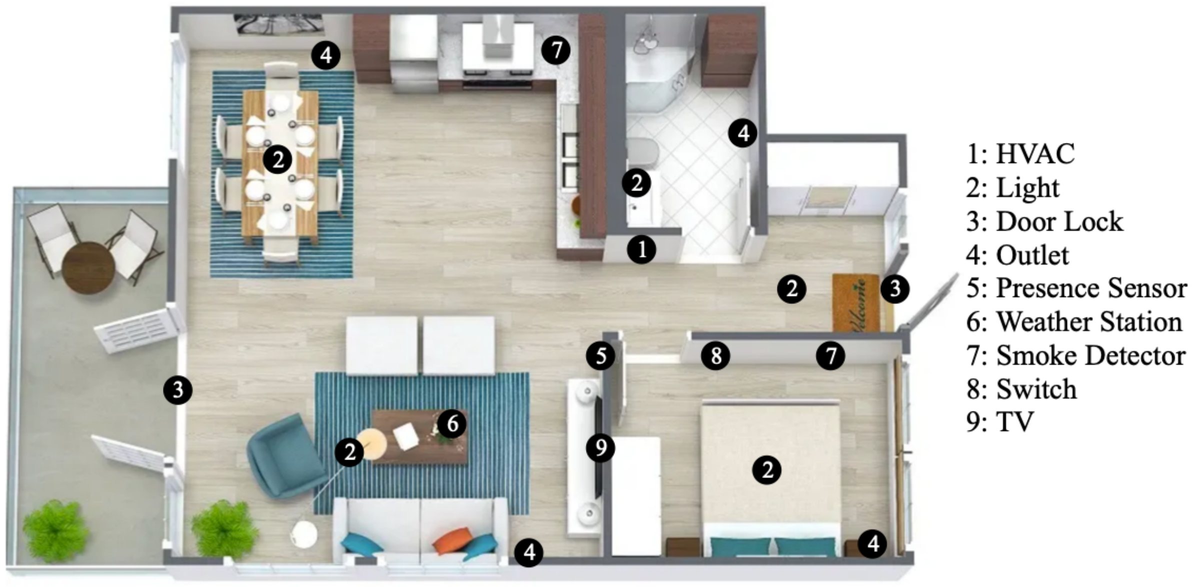

Figure 4.1. Floor plan of the experimental testbed

Device type	Hardware attached
Color light bulb	WS2812 LED strip
Smoke detector	Smoke sensor
Door lock	Servo-motor
Smart TV	N/A
Thermostat	BMP280 sensor and power relay
Weather station	BMP280 sensor
Presence detector	PIR motion detector
Physical switch	Double-throw switch
Outlet	Power relay

Table 4.1. Types of devices present in the network

- `home/mqtt_lock/available`: whether the device is connected and available
- `home/mqtt_lock/set`: listen for commands coming from the hub
- `home/mqtt_lock/state`: publish its current state, used as an acknowledgment for the hub

All the messages (both from the hub and the devices) are set to be retained by the broker, so if any party restarts, it is able to immediately assess the current state of the system and set its internal representation accordingly. For instance, if a device stops for any reason, when it restarts and connects to the broker it will be informed of the last command the hub sent it, and will be able to act accordingly.

4.3.2 WebThings Implementation Details

WebThings handles devices differently from Home Assistant. While the latter needs to have prior knowledge of the device type and already have a server-side handler (called “component”), the former only needs the nodes to create a simple web-server that exposes a predefined API. Once the hub has detected and connected to the node’s server, it can learn the properties the devices possesses (such as OnOffProperty, ColorProperty, etc.). For a device to be recognized as a specific type (e.g. a lamp), it needs to expose a specific set of properties [27].

For instance, a thermostat needs to implement the `TemperatureProperty` and `TargetTemperatureProperty` properties in order to have the “Thermostat” capability. Additionally, the `HeatingCoolingProperty` and `ThermostatModeProperty` properties will also be part of the interface provided by the “Thermostat” capability.

4.3.3 Collecting the Training Data

With the attacks defined, the nodes implemented, and the frameworks selected, we can start collecting the training dataset. The idea behind this is to generate and record the behavior of every single device for every possible configuration it can have. This gives us a clearly labelled set of data that can be used by an automated detection system. For each combination of attack, device state, and framework, we run each device for 20 minutes and record its metrics with *Sentinel*.

4.4 Simulating the IoT Network

In order to generate the training dataset, we build a home IoT network with standard devices, which we cycle through all the possible combinations of logical state and simulated attack. This is done by enumerating all these combinations in a text file, referred to as a “trace file”. This trace is then fed to a scheduling engine that parses it, and executes all the events founds inside.

As the purpose of this simulation is to evaluate how *Sentinel* is able to pickup behavioral discrepancies in running devices, we do not worry about running the devices in a realistic schedule. Instead we focus on generating clean and consistent data.

A complete overview of *Sentinel* instrumented for these experiments is available in Figure 4.2.

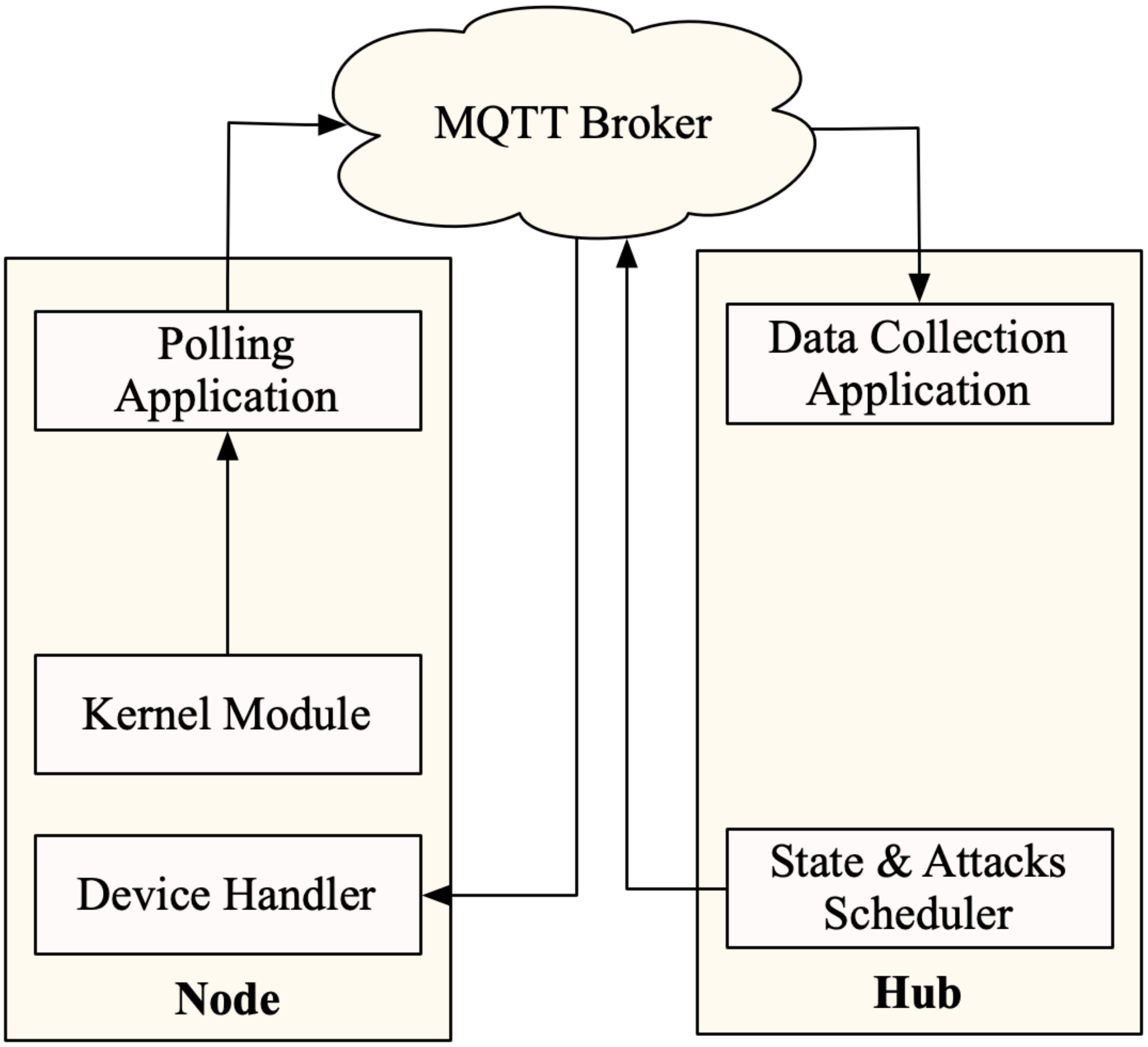

Figure 4.2. Sentinel instrumented for the experiments

4.5 Implementation

The devices used to represent the IoT network in our experiments are Raspberry Pi 4 model B, with 4GB of RAM. All the Raspberry Pi use a custom Raspbian Lite image, created with the `pi-gen` tool [28]. The image consists of a Raspbian Lite, to which we add WiFi credentials, customize username and password, enable SSH, and install all the tools that compose *Sentinel*. Each Python application has a `virtualenv` created where all the required packages are installed. The SKM is compiled, installed and set to load at boot time.

The nodes are connected through Wi-Fi, supported by a D-Link DIR-605L router running firmware v2.09.

Chapter 5 Evaluation of the Detection System

Our evaluation aims to prove that *Sentinel* is able to provide a novel and useful layer of security to IoT networks. In this section, we answer the following questions: (1) Can the power usage of *Sentinel* be low enough to be used in a energy-constrained device? (2) Can the data provided by *Sentinel* power an intrusion detection system? (3) How does the polling rate of *Sentinel* influence the accuracy of the detection? (4) How does the number of CPU cores available to an IoT node influence the detection accuracy?

5.1 Power consumption analysis

Another constraint for IoT devices is power use. Some devices are designed to be battery powered and be able to last for months or years on their supply. To inspect how *Sentinel* can impact the battery life expectancy of these devices, we perform a power consumption analysis. This is done by running the device handler on a Raspberry Pi with and without *Sentinel*, in different logical states, and with various reporting periods. The power draw is measured at the outlet with a watt meter in which only the Pi is plugged in. The results of the experiment are available in Figure 5.1.

For a reporting period of one second, *Sentinel* causes an increase of power use of about 10%, which will definitely impact the lifetime of a battery-powered device. However, as the polling frequency decreases, the power consumption overhead incurred decreases too. As such, for a ten seconds reporting delay, the overhead dips below 1%. This means it is possible to find a tradeoff between battery life and enhanced device security by varying the sampling rate.

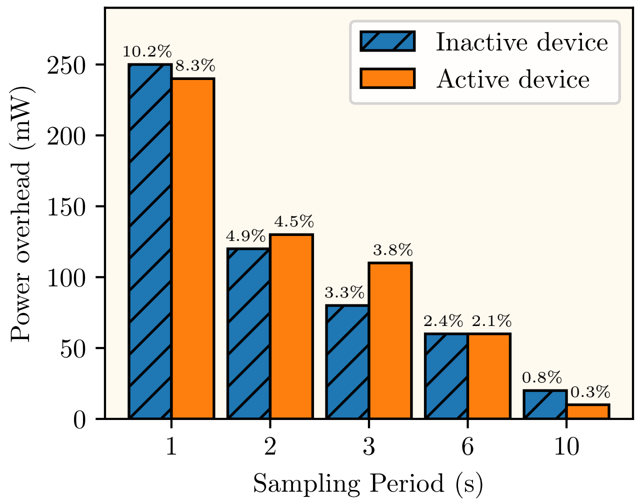

Figure 5.1. Power overhead caused by Sentinel for various polling periods, expressed as absolute and relative values

5.2 Training a Decision-Tree IDS

In this section, we use data collected by Sentinel to train models that are able to classify the current state of each node. We develop binary (detection whether an attack is happening) and multi-class classifiers (which particular attack is happening, if any).

For each node, we create a dataset where we prune some constant features that are recorded by *Sentinel* and won't be useful for our model (`nb_cpus`, `free_ram`, `total_ram`, `free_swap`, `total_swap`, `tracked_pid`). All the datapoints are then normalized between 0 and 1, based on the min and max value achieved for each metric. The only two exceptions are RAM and swap usage, which are expressed as a percentage of total use. The datasets created contain the samples recorded every second over the time window of the experiment and are labeled if there is an attack or not (binary classification), or which type of attack is under way with a "no attack" type (multi-class classification).

These datasets are then used to train a set of Decision Tree models. Decision Trees are a class of Machine Learning classifiers. They operate by traversing a binary tree built during a training phase. Starting at the root of the tree, each node corresponds to a comparison of a specific value of the datapoint. Based on the result of this comparison, one of the two branches of the node is taken, and the operation is repeated until it reaches a leaf node. This leaf node corresponds to the class the sample is predicted to belong to.

The concept of using Decision Trees for Intrusion Detection purposes is not new. Indeed, such models have been used in this manner with great success over the past two decades, brining with them faster reaction time and improved detection accuracy [29]. Further, Decision Trees are one of the most explainable machine learning systems, allowing us to understand which criteria was used to make each decision.

For our purposes, we trained the models using stratified k-fold validation. In this training method, the data is split into multiple subsets (called “folds”), while making sure that each class is evenly represented across each set. Each of the folds is then split randomly into an 80%/20% partition. We train the model on the first part, and evaluate it on the second. This method of training is useful in proving that the results obtained are not due to random chance and are consistent across multiple independent training passes.

As a classification system like a Decision Tree operates either in binary or in multi-class mode, we train models with both configurations, to see if they present any meaningful difference. The confusion matrices for the Home Assistant framework are presented in Table 5.1 for the binary classifier, and Table 5.3 for the multi-class system. The same results for the WebThings framework are presented respectively in Tables 5.2 and 5.4.

The multi-class systems clearly outperform the binary ones, as seen on their higher true positive rates. Further, the high false negative rates of the binary classifiers give us insight into the fact that bundling all the attacks under a single detection target does not perform optimally. Indeed, the actual low-level effects of these attacks are not all identical, making it harder for the classifier to find a unified method of detection. Of course, this detection work remains a proof of concept for the efficacy of *Sentinel*, and does not claim to be a cutting-edge IDS.

	Attack	**No attack**
Attack	87.77%	12.23%
No attack	4.29%	95.71%

Table 5.1. Confusion Matrix for Home Assistant Binary

	Attack	**No attack**
Attack	96.25%	3.75%
No attack	0.75%	99.25%

Table 5.2. Confusion Matrix for WebThings Binary

In order to determine the best value for the depth of the Decision Trees, we trained multiple models: one per possible depth value (between one and the total number of

	Attack 1	**Attack 2**	**Attack 3**	**Attack 4**	**Attack 5**	**No attack**
Attack 1	99.35%	0.13%	0.00%	0.00%	0.00%	0.52%
Attack 2	0.00%	91.31%	0.41%	0.00%	0.00%	8.28%
Attack 3	0.04%	0.43%	96.67%	0.04%	0.00%	2.83%
Attack 4	0.00%	0.00%	0.13%	99.11%	0.02%	0.74%
Attack 5	0.00%	0.00%	0.00%	0.00%	98.15%	1.85%
No attack	0.04%	1.36%	0.15%	0.06%	0.09%	98.31%

Table 5.3. Confusion Matrix for Home Assistant Multi-class

	Attack 1	**Attack 2**	**Attack 3**	**Attack 4**	**Attack 5**	**No attack**
Attack 1	98.76%	0.17%	0.02%	0.00%	0.00%	1.06%
Attack 2	0.167%	96.13	0.74%	0.20%	0.11%	2.65%
Attack 3	0.00%	0.00%	96.19%	0.35%	0.02%	3.33%
Attack 4	0.00%	0.17%	0.48%	96.56%	0.15%	2.65%
Attack 5	0.02%	0.00%	0.04%	0.07%	97.46%	2.41%
No attack	0.05%	0.26%	0.18%	0.17%	0.20%	99.15%

Table 5.4. Confusion Matrix for WebThings Multi-class

metrics), and one with no depth limit. The accuracy results are available in Figure 5.2. As accuracy converges after a depth of 14, we use this value for the rest of the experiments.

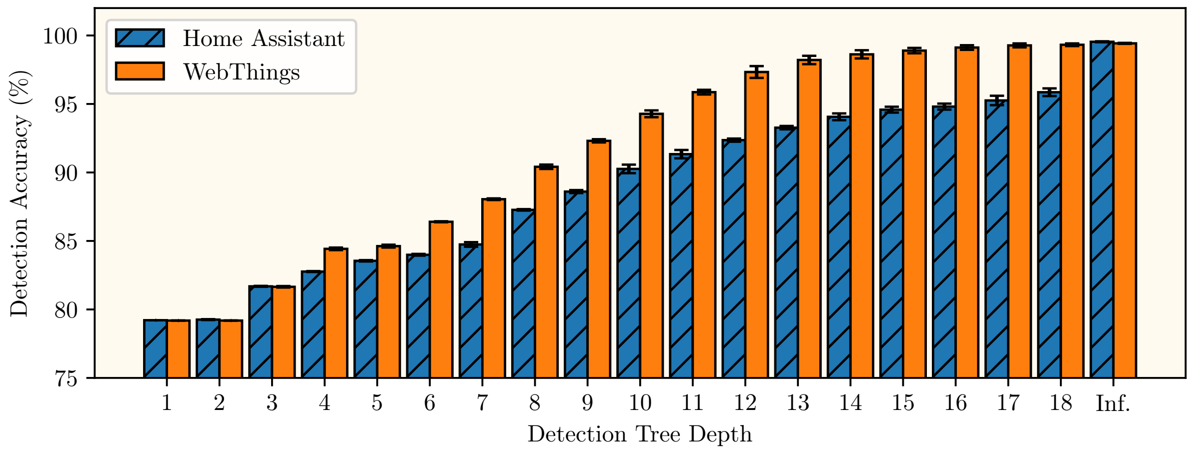

Figure 5.2. Detection accuracy for various Decision Tree depths

The models trained with the data collected previously present an accuracy of 99.8%. While further work would need to be put into testing the full detection capabilities of the data collected, we believe that this result demonstrates that *Sentinel* has some potential.

5.3 Sample Rate Influence

In order to characterize how the quality of the data collected by *Sentinel* is impacted by its sampling period, we run two data collection runs. The first one has a polling rate of one second, and each state is held for one minute. The second is done with a period of ten seconds, and a state duration of ten minutes. These configurations provide us with the same amount of measurements in both sets, guaranteeing that any difference found is not caused by variability of the training dataset sizes. The results of this experiment are presented in Table 5.5 and Figure 5.3.

	1 CPU Core	**4 CPU Cores**
Accuracy (95% confidence)	95.8% ± 1.4%	90.2% ± 1.9%

Table 5.5. Detection precision for a polling rate of 1s and 10s

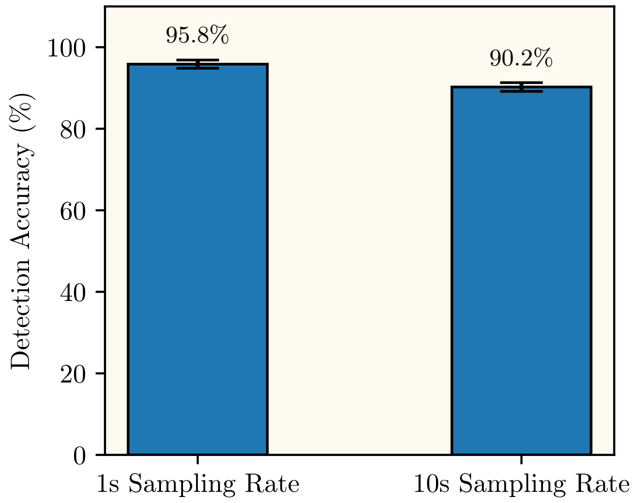

Figure 5.3. Detection Accuracy for a polling rate of 1s and 10s

When the polling rate is decreased to ten seconds, the detector accuracy decreases by 5.6 percentage points. While the longer sampling period comes with a potentially increased reaction time to an attack, *Sentinel* comes with a drill-down capacity, allowing an IDS to dynamically change the polling rate of individual devices. It is then possible to design a logic that temporarily decreases the sampling period if suspicious behavior is noticed.

5.4 Device performance

The Raspberry Pi 4, which was used to represent an IoT device, is significantly more powerful than the average IoT device, which are usually single-purpose, single-core devices. The Raspberry Pi 4 has a 1.5GHz four core CPU, which could have an effect on *Sentinel*. To evaluate how this impacts its performance, we perform another complete data gathering experiment similar to the previous, but we first disable three of the four cores of each node. The data is gathered with one second between each sample, and one minute per state. We then train a new Decision Tree on this dataset, and compare its precision to the four core dataset. The results of this experiment are available in Table 5.6 and Figure 5.4.

	1 CPU Core	**4 CPU Cores**
Accuracy (95% confidence)	93.0% ± 2.7%	95.8% ± 1.4%

Table 5.6. Detection precision for 1 and 4 core devices

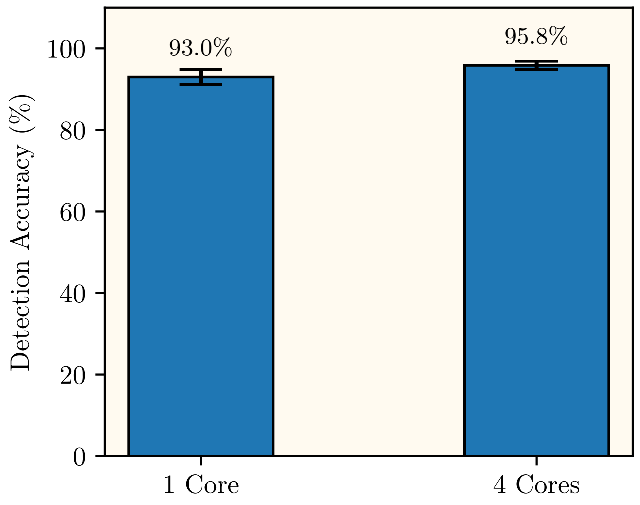

Figure 5.4. Detection Accuracy on a device with 1 and 4 CPU cores

With a core count reduced to one per device, the attack detection accuracy is decreased by 2.8 percentage points. This difference comes from the fact that, as the device now has to share a single core for all its processes, some metrics are not as insightful as with four cores. For instance, the `cpu_load` metric, that records the one-minute CPU load of the device, now has an increased value outside of an attack. When an attack occurs, its increase is consequently less noticeable.

However, it is to be noted that the detector performance was not degraded to useless a level by this change. We believe that this result shows that *Sentinel* could effectively run on a low core-count device.

Chapter 6 Discussion

As stated previously, *Sentinel* is not a fully-fledged Intrusion Detection System. Its purpose is to provide a lightweight framework-agnostic data-collection system. It should be used as a stepping stone to help build new IDSs specially designed for IoT environments, using the data made available to detect novel attacks. The detector presented in this paper are only a proof of concept to validate that the data collected has some inherent value for intrusion detection.

While the metrics collected by *Sentinel* proved to be useful in detecting malicious behavior, they are not optimized for every device. Adapting *Sentinel* for a new set of devices, or for a different IDS may require new features to be added. This can be done relatively easily by editing the source code of SKM. Further, as `sysfs` is passive (i.e. it the SKM only does some work when a file is read), there is no issue with adding more features than required, as the unused ones can simply be ignored by the data polling application.

As the iteration of *Sentinel* presented in this paper depends on every device running Linux and being able to communicate over MQTT, it does not faithfully represent the totality of the IoT devices available currently. However, we do not believe this to be an issue for multiple reasons:

- Any IoT device that does not run Linux will most likely run a very limited firmware, making it irrelevant to the attacks considered in this paper.
- *Watchtower* could be modified to work not only with MQTT, but also with other non WiFi-based IoT protocols, such as Zigbee or BLE.

The MQTT component of *Sentinel* relies on server-side and client-side certificates in order to provide adequate security properties. For the purposes of our experiments, the certificates were manually generated and distributed, for the sake of convenience.

However, in a real-world environment, where the IoT network would need more flexibility, the certificate distribution would have to happen through a centralized authority.

Our experimental MQTT setup allows any device with the proper credentials to publish and subscribe to any topic, potentially allowing an attacker to obtain sensitive information by corrupting a node. Most MQTT brokers provide an Access Control List feature, through which read/write access to topics can be restricted. However, this ACL is static and needs to be edited whenever a new device joins, as it will need its own set of topics to publish to. A potential alternative to this unwieldy method could be to add an end-to-end encryption layer on the data, by encrypting it in the SKM, and decrypting it at the hub. This would prevent any possibility of eavesdropping, but would require to design a key agreement method between the kernel module and the hub.

Fault detection is another active domain of IoT research, focusing more on safety, where intrusion detection focuses on security. However, both domains share features, as faults can have similar side effects as network intrusions. For instance, a device getting stuck in an infinite loop can resemble a sleep deprivation attack. We therefore believe that *Sentinel* could see some application in fault detection, as the metrics used by the system could be used as input of a fault detection system.

Our threat model assumes that the adversary does not have privileged access to the nodes, and as such cannot falsify the data reported by the SKM. However, it could be interesting to investigate if an attacker with root access forcing the node to misreport the system information could impact the detection abilities. In particular, when a device changes state, the attacker would have to adapt its falsification in order to remain covert. This is at first glance not trivial, as it requires the knowledge of how the device should behave in every scenario.

As stated previously, *Sentinel* does not claim to be an automated intrusion detection solution, but rather a data-gathering tool that can be leveraged by a dedicated detection system. In order to be improved, *Sentinel* could provide a larger variety of data, which would allow it to be tailored to fit a specific scenario, where some metrics are more relevant than others. However, in order to determine which metrics could benefit from being added to the monitoring capacities of *Sentinel*, some feedback from actual field experience would be required.

Chapter 7 Conclusion

Our work demonstrates that collecting low-level system information in IoT devices does provide valuable insight into the presence of an attacker. Through our evaluation, we prove that a simple detection model, with our generated datasets, is capable of assessing malicious behavior with a high degree of accuracy. Further, we show this data collection can be achieved for a low cost, making it compatible with the constraints inherent to IoT devices.

Future work on this topic will be focused on exploring the detection possibilities of additional IoT attacks, as well as expanding the set of metrics collected, and evaluating their individual values. Work will also be needed on implementing *Sentinel* on commercial IoT devices, to confirm our results obtained on open-source devices.

Bibliography

[1] Antonakakis, M., T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou "Understanding the Mirai Botnet," , p. 19.

[2] Benkhelifa, E., T. Welsh, and W. Hamouda (24) "A Critical Review of Practices and Challenges in Intrusion Detection Systems for IoT: Toward Universal and Resilient Systems," *IEEE Communications Surveys & Tutorials*, **20**(4), pp. 3496–3509.

[3] Forrest, S., S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff "A Sense of Self for Unix Processesey," , p. 9.

[4] Raza, S., L. Wallgren, and T. Voigt (2013) "SVELTE: Real-Time Intrusion Detection in the Internet of Things," *Ad Hoc Networks*, **11**(8), pp. 2661–2674.

[5] Surendar, M. and A. Umamakeswari (2016) "InDReS: An Intrusion Detection and Response System for Internet of Things with 6LoWPAN," in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, IEEE, Chennai, India, pp. 1903–1908.

[6] Midi, D., A. Rullo, A. Mudgerikar, and E. Bertino (2017) "Kalis — A System for Knowledge-Driven Adaptable Intrusion Detection for the Internet of Things," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, Atlanta, GA, USA, pp. 656–666.

[7] Cervantes, C., D. Poplade, M. Nogueira, and A. Santos (2015) "Detection of Sinkhole Attacks for Supporting Secure Routing on 6LoWPAN for Internet of Things," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, IEEE, Ottawa, ON, Canada, pp. 606–611.

[8] "Architecture — SmartThings Classic Developer Documentation," https://docs.smartthings.com/en/latest/architecture/index.html.

[9] "Architecture | Home Assistant Developer Documentation," https://developers.home-assistant.io/docs/architecture_index.

[10] "Mozilla IoT," https://iot.mozilla.org/about.

[11] Vacca, J. R. (2013) *Computer and Information Security Handbook*, Elsevier Science & Technology Books, San Diego.

[12] Liao, H.-J., C.-H. Richard Lin, Y.-C. Lin, and K.-Y. Tung (2013) "Intrusion Detection System: A Comprehensive Review," *Journal of Network and Computer Applications*, **36**(1), pp. 16–24.

[13] Mitchell, R. and I.-R. Chen (2014) "A Survey of Intrusion Detection Techniques for Cyber-Physical Systems," *ACM Computing Surveys*, **46**(4), pp. 1–29.

[14] Andrea, I., C. Chrysostomou, and G. Hadjichristofi (2015) "Internet of Things: Security Vulnerabilities and Challenges," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, pp. 180–187.

[15] Wood, A. and J. Stankovic (2002) "Denial of Service in Sensor Networks," *Computer*, **35**(10), pp. 54–62.

[16] Vasserman, E. Y. and N. Hopper (2013) "Vampire Attacks: Draining Life from Wireless Ad Hoc Sensor Networks," *IEEE Transactions on Mobile Computing*, **12**(2), pp. 318–332.

[17] Khouzani, M. H. R. and S. Sarkar (2011) "Maximum Damage Battery Depletion Attack in Mobile Sensor Networks," *IEEE Transactions on Automatic Control*, **56**(10), pp. 2358–2368.

[18] Mitrokotsa, A., M. R. Rieback, and A. S. Tanenbaum "Classification of RFID Attacks," , p. 14.

[19] Wallgren, L., S. Raza, and T. Voigt (2013) "Routing Attacks and Countermeasures in the RPL-Based Internet of Things," *International Journal of Distributed Sensor Networks*, **9**(8), p. 794326.

[20] Heer, T., O. Garcia-Morchon, R. Hummen, S. L. Keoh, S. S. Kumar, and K. Wehrle (2011) "Security Challenges in the IP-Based Internet of Things," *Wireless Personal Communications*, **61**(3), pp. 527–542.

[21] Stojmenovic, I., S. Wen, X. Huang, and H. Luan (2016) "An Overview of Fog Computing and Its Security Issues: AN OVERVIEW OF FOG COMPUTING AND ITS SECURITY ISSUES," *Concurrency and Computation: Practice and Experience*, **28**(10), pp. 2991–3005.

[22] Garfinkel, T. and M. Rosenblum "A Virtual Machine Introspection Based Architecture for Intrusion Detection," , p. 16.

[23] Moustafa, N., B. Turnbull, and K.-K. R. Choo (2019) "An Ensemble Intrusion Detection Technique Based on Proposed Statistical Flow Features for Protecting Network Traffic of Internet of Things," *IEEE Internet of Things Journal*, **6**(3), pp. 4815–4830.

[24] Mochel, P. and M. Murphy, "Sysfs - The Filesystem for Exproting Kernel Objects." https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.

[25] "Rules on How to Access Information in Sysfs — The Linux Kernel Documentation," https://www.kernel.org/doc/html/latest/admin-guide/sysfs-rules.html.

[26] "Mqtt/Mqtt.Github.Io," https://github.com/mqtt/mqtt.github.io.

[27] "WoT Capability Schemas - Mozilla IoT," https://iot.mozilla.org/schemas/.

[28] Cosson, A. (2020), "Fork of Pi-Gen Used to Generate the Sentinel Raspbian Images," .

[29] Kruegel, C. and T. Toth (2003) "Using Decision Trees to Improve Signature-Based Intrusion Detection," in *Recent Advances in Intrusion Detection* (G. Goos, J. Hartmanis, J. van Leeuwen, G. Vigna, C. Kruegel, and E. Jonsson, eds.), vol. 2820, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 173–191.