OASIS: An Intrusion Detection System Embedded in Bluetooth Low Energy Controllers

Romain Cayre
EURECOM, Apsys.Lab
Sophia-Antipolis, France
romain.cayre@eurecom.fr

Vincent Nicomette
Univ de Toulouse, INSA, LAAS
Toulouse, France
vincent.nicomette@insa-toulouse.fr

Guillaume Auriol
Univ de Toulouse, INSA, LAAS
Toulouse, France
guillaume.auriol@insa-toulouse.fr

Mohamed Kaâniche
CNRS, LAAS
Toulouse, France
mohamed.kaaniche@laas.fr

Aurélien Francillon
EURECOM
Sophia-Antipolis, France
aurelien.francillon@eurecom.fr

ABSTRACT

Bluetooth Low Energy has established itself as one of the central protocols of the Internet of Things. Its many features (mobility, low energy consumption) make it an attractive protocol for smart devices. However, numerous critical vulnerabilities affecting BLE have been made public in recent years, some of which are linked to the protocol's design itself. The impossibility of correcting these vulnerabilities without affecting the specification requires the development of effective intrusion detection systems, enabling the detection and prevention of these threats. Unfortunately, the protocol relies on peer-to-peer communications and introduces many complex and dynamic mechanisms (e.g., channel hopping), making monitoring complex, costly and limited. Existing intrusion detection approaches lack flexibility, are limited in scope and introduce high deployment costs.

In this paper, we explore a novel approach consisting in embedding an intrusion detection system directly within BLE controllers. This strategic position tackles these challenges by enabling a more advanced analysis and instrumentation of the protocol and opens the way to new defensive applications. We propose OASIS, a framework for injecting detection heuristics into controllers' firmwares in a generic way without affecting the normal operation of the protocol stack. It can be deployed in various contexts during the life cycle of a device, from the chip manufacturer to a software developer making use of proprietary components, or even in a full black box context by a security analyst to harden a commercial product. We describe its modular architecture and present its implementation within five of the most popular BLE chips from three different manufacturers, deployed in billions of devices and embedding heterogeneous protocol stacks. We present five modules for critical low-level protocol attack detection. We show that OASIS has a low impact on the controller performance (power, timing, memory) and evaluate its usage in a real-world setting.

ACM AsiaCCS 2024, 1-5 July 2024, Singapore
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**; **Embedded systems security**; **Intrusion detection systems**.

KEYWORDS

Intrusion Detection, Bluetooth, Controllers, Instrumentation

ACM Reference Format:
Romain Cayre, Vincent Nicomette, Guillaume Auriol, Mohamed Kaâniche, and Aurélien Francillon. 2024. OASIS: An Intrusion Detection System Embedded in Bluetooth Low Energy Controllers. In *Proceedings of ACM Conference (ACM AsiaCCS 2024).* ACM, Singapore, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The rapid expansion of Internet of Things has motivated the development of new wireless protocols, optimizing stack complexity or power consumption to fit the limited constraints of smart devices. Bluetooth Low Energy has established itself as one of the most popular wireless technologies over the years: its massive deployment in billions of smartphones, tablets and laptops, low complexity and versatility make it especially attractive for IoT devices. In this context, Bluetooth security has become a major concern. Multiple critical vulnerabilities [1; 3; 4; 10; 16] have been discovered recently, illustrating the growing interest of the community in this technology. While some of these vulnerabilities [3; 4; 16] are related to the stack implementation and can potentially be patched by the manufacturers, others [1; 6; 8; 10; 19] are related to the protocol design itself and cannot be easily fixed without either modifying the specification [5] or not adhering to the standard.

This situation highlights the need to develop defensive measures, such as Intrusion Detection Systems (IDS), to detect and prevent these threats. However, building such systems is difficult because the BLE protocol design and its concrete applications introduce many fundamental challenges. First, exhaustively analyzing the traffic from an external probe is unsuitable in many settings for which BLE was created (e.g., mobile use). Second, such monitoring is complex because BLE uses channel hopping algorithms during connections, requiring expensive and complex wideband monitoring. Moreover, the wireless nature of the protocol introduces many context-dependent factors impacting the completeness and representativeness of the monitored traffic. The protocol is also mainly used to establish direct point-to-point communications that are

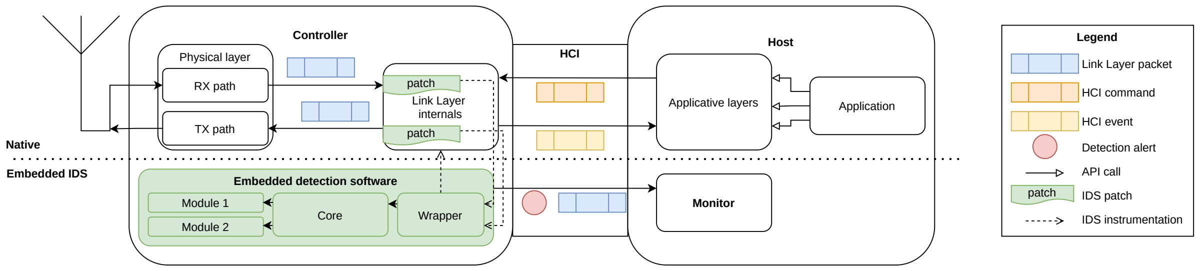


Figure 1: OASIS embedded IDS overview. Below the dashed line, OASIS modifications.

difficult to monitor from a central node. Finally, many BLE devices are intended for mobile use, resulting in a highly dynamic environment. These factors deeply impact the practicability of previous works exploring the domain of BLE Intrusion Detection. Several works [32; 36; 39] rely on existing BLE sniffers and inherit their limitations, focusing their work on a small subset of BLE communications (e.g., advertising mode). Other works [21; 23] explore various detection techniques on Bluetooth Low Energy without addressing these challenges by performing offline detection on datasets, making their deployment difficult in practice.

In this work, we explore an alternative system design, aiming at embedding our IDS directly into the BLE controllers of the devices themselves. It allows to monitor BLE traffic and detect attacks locally, at the lowest level that can be accessed by software, while providing useful contextual information and allowing a full instrumentation of the device that can be leveraged in a prevention strategy. While a typical BLE stack is composed of two main parts, the Host (which manages the application layers of the protocol) and the Controller (which is in charge of the lower layers), most of the low-level traffic is hidden from the Host by design. As a result, many attacks would be impossible to detect from the Host, motivating us to focus our work on the Controller instrumentation. This strategy allows an exhaustive view of the wireless traffic while giving access to a large set of relevant features, such as signal strength (RSSI) or checksum (CRC) validity. It is also strategically positioned to carry out some low-level defensive actions to prevent a detected attack.

This approach implies several technical challenges. First, the controllers implementations are generally proprietary and require reverse engineering to understand and instrument their internals. Second, controllers are difficult to instrument, as manufacturers generally do not provide an easy way to patch them to include defensive code. Third, a protocol stack implementation is time sensitive by design, resulting in an optimized code that can be difficult to modify and improve.

In this paper, we present OASIS, a user-friendly, modular, and easy-to-extend framework designed to tackle these challenges and largely automate both reverse engineering and instrumentation processes of BLE controllers, allowing the hardening of currently deployed embedded devices. It allows to embed various intrusion detection heuristics, suited for many BLE attacks, operating at any level of the protocol stack. It can also be used in multiple contexts and use cases, from an end user having only a binary firmware without any documentation to an embedded software developer using a closed-source stack in his application or even by a manufacturer trying to improve the security of its protocol stack. Our approach being based on embedded detection performed locally by the nodes, it does not suffer from many of the technical limitations of existing research works relying on sniffers. It can also easily complement other detection approaches or cooperate with other defensive components, implemented at a higher level in the device itself or taking advantage of a distributed strategy across cooperating devices. We demonstrate the relevance of our lightweight detection heuristics for detecting existing Bluetooth Low Energy attacks, this work being the first one to provide a practical detection of low-level attacks targeting the connected mode, such as InjectaBLE [10], BTLEJack [8] or KNOB [1].

More precisely, our contributions are threefold:

- We propose a novel approach to design a BLE IDS by embedding detection mechanisms directly into the BLE controllers.
- We provide a generic and modular framework, OASIS, dedicated to the integration of intrusion detection modules inside various BLE controllers' firmware. We illustrated its capabilities by implementing it on five popular chips from three major manufacturers, based on heterogeneous BLE stacks and deployed in billions of commercial products like smartphones and IoT devices.
- We show the relevance of OASIS with 5 of the most severe low-level structural attacks targeting BLE protocol. We designed 5 intrusion detection modules for these attacks and embedded them into various BLE controllers. We detected these attacks with very good false positive and negative rates and low-performance overhead. In particular, we are the first to propose a detection strategy for critical attacks such as InjectaBLE [10] or BTLEJack [8].

2 BLUETOOTH LOW ENERGY OVERVIEW

The BLE protocol uses *Gaussian Frequency Shift Keying* (GFSK) modulation with a data rate of 1MBps or 2MBps with 40 channels in the 2.4GHz ISM band.

In advertisement mode, the devices notify other nearby devices of their presence by broadcasting advertisement packets on the dedicated advertisement channels (37, 38, and 39). The other channels (data channels) are used in connected mode. In connected mode, communications are based on a Central/Peripheral topology, letting two connected devices exchange data. Each node is assigned a role according to its capabilities: a) a *Broadcaster (or Advertiser)* is only capable of transmitting advertisements; b) a *Scanner (or Observer)* is only capable of scanning advertisements; c) a *Peripheral* is capable of announcing its presence using advertisements and can accept incoming connections and d) a *Central* is capable of scanning advertisements and initiating a connection with a *Peripheral*.

In advertising mode, the advertising device jumps over the three advertising channels to broadcast its frames. The time between each *advertising event* (a complete cycle of hopping on the three channels) is the sum of *advInterval* and *advDelay*. The *advInterval* is an integer chosen by the device multiplied by 0.625ms. On the other hand, the *advDelay* is a random delay (between 0 and 10ms) automatically generated by the *Link Layer* for each advertising event. After every advertisement frame transmission, the device listens to the channel for possible *Scan Request* or *Connection Requests* from other devices.

When a *Connection Request* is accepted, the devices enter the *connected mode*. In *connected mode*, the devices use a channel hopping algorithm configured by several parameters included in the *Connection Request* and hop along the *data channels*. The device initiating the connection acts as a *Central* while the device accepting the incoming connection acts as a *Peripheral*. The devices communicate during time slots named *connection events*: first, the *Central* transmits a frame to the *Peripheral*, then the *Peripheral* waits for $150\mu s$ and transmits its frame to the *Central*. The duration of a *connection event* is defined by the *Hop Interval* parameter, included in the *Connection Request*. When a *connection event* is terminated, the devices jump to the next channel according to the selected channel hopping sequence.

Every *Link Layer* frame starts with a 1-byte preamble, followed by a 4-byte field (*Access Address*) used for synchronization. Each frame also includes a header, a payload, and a 3 bytes-long Cyclic Redundancy Check (CRC).

3 THE OASIS FRAMEWORK

This section describes the design of OASIS, a generic and modular framework for embedding intrusion detection mechanisms into controllers. We cover the threat model, detection requirements, design guidelines, global architecture, generated code structure, main component implementations, and a typical use case.

3.1 Threat model

This work considers an attacker who can perform active network attacks targeting BLE devices. He can use dedicated offensive hardware and attack any protocol stack layer, including the lowest ones (e.g., Physical and Link Layer). More specifically, we consider that the attacker can receive and transmit arbitrary signals and packets, impersonate a device, and sniff both new and existing BLE communications. We assume that the attacker does not have physical access to attacked devices and cannot perform physical attacks or alter the controller firmware behavior.

While OASIS could be used to detect some of those attacks, we do not consider the exploitation of vulnerabilities related to a specific controller implementation. As a consequence, we consider network attacks aiming at altering firmware memory, including the embedded IDS itself (e.g., remote exploitation of buffer overflow or format string), out of the scope of this paper. OASIS is designed to detect protocol-level attacks aiming to manipulate a BLE communication, either by disrupting it (e.g., Jamming-based attacks [8]), altering it (e.g., Man-in-the-Middle [6; 19], hijacking [8] and malicious injection [10] attacks) or downgrading its security [1].

3.2 Detection requirements

The embedded detection mechanisms in the BLE controllers require the instrumentation of: **a) Packet reception** (e.g., LL packets, RSSI, CRC validity), **b) Time management** (e.g., accurate timestamps, timed execution), **c) Connection and Device management** (e.g., connections metadata, BD addresses, controller state), **d) High-level operations** (e.g., scan mode trigger).

Implementing these mechanisms can be very heterogeneous depending on the stack used: to avoid developing multiple detection modules dependent on the stack, this motivates the development of a generic framework with wrappers allowing instrumentation of the stacks and exposing a consistent API.

3.3 Main guidelines

Many controller implementations are proprietary and not documented, making it often impossible to instrument the source code. Interacting with the BLE stack often requires patching the firmware binary and running OASIS code without disrupting the execution.

This motivated the development of a framework to generate the detection software, which must be able to run independently from the controller. This implies carefully choosing the hooked functions to avoid adding delays in time-sensitive components but also finding a way to inject our code and data into memory without impacting the controller execution. Our framework must also be **user-friendly**, i.e., allow a developer to easily implement a new detection module without requiring a deep understanding of the underlying controller architecture.

The controllers are also heterogeneous and cannot be instrumented without writing specific code for each of them. However, a detection module implements a logic independent of the underlying controller, and the corresponding code must be written only once. Consequently, every target-specific wrapper must expose a homogeneous API, facilitating the development of target-agnostic components. Therefore, one of the key principles that have guided our framework design is the **genericity**.

Some controllers only implement a subset of the BLE specification. For example, some IoT-oriented controllers only implement the *Peripheral* role, so only a subset of the generated code needs to be embedded. Given this situation and the strong constraints in terms of time and memory associated with the embedded approach, **modularity** is a fundamental design guideline of our framework. Similarly, extending the framework to add a new target or detection module should be straightforward.

3.4 Embedded detection software

The OASIS framework generates an embedded detection software ready to be loaded into the chip memory. This embedded software instruments the target controller by patching specific functions to extract relevant features. Then, these features are forwarded to the selected detection modules, which analyze them and potentially raise an alert if an attack is detected. The software interacts with the BLE stack but runs without interfering with its normal

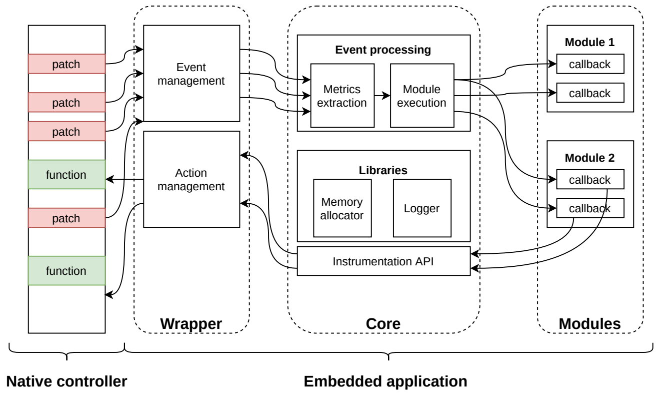


Figure 2: Embedded detection software overview.

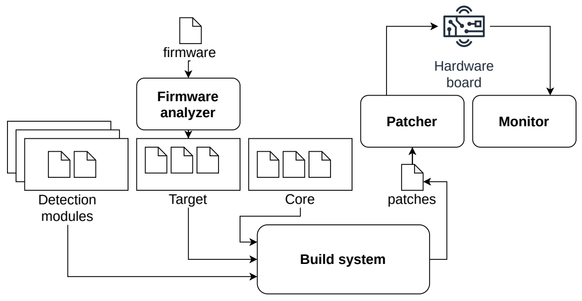


Figure 3: OASIS Framework architecture.

behavior. Therefore, it uses and manages its own memory space independently of the main firmware.

The detection software is composed of three main components, as illustrated by Figure 2: a *target-specific wrapper*, a *core*, and a *set of detection modules*. They are described in the following subsections.

3.4.1 Target-specific wrapper. The *wrapper* is the target-specific component for interacting with the controller. It is composed of two main systems: a) an *event management system* to react to specific events (e.g., packet reception, packet transmission, connection initiation) and to extract all available low-level features from the controller, and b) an *action management system*, to trigger specific actions in the controller (e.g., sending an event to the Host, entering a specific state).

The *event management system* comprises a set of wrapper functions corresponding to the monitored events. It instruments the controller by patching specific instructions of the BLE stack to redirect the execution flow to a trampoline function that saves the context and calls the corresponding wrapper function. Once the wrapper function has been executed, the trampoline function restores the context, executes the instruction altered by the patch, and redirects the execution flow to the next instruction in the stack. This mechanism allows calling the corresponding wrapper function when a specific event occurs. Then, the wrapper function extracts all available features and propagates them to the *event processing system* implemented in the core component.

The *action management system* comprises a set of functions to trigger a specific action in the instrumented controller. Depending on the instrumented stack, it can make a function call, mimic an HCI command transmitted by the Host, or modify a variable in the controller memory.

This component is the only one that depends on the target. Therefore, each implemented wrapper exposes a similar API, allowing the target-independent components to interact with the controller in a standardized and unified way.

3.4.2 Core. The core is the central component of the detection software. It is composed of an *event processing system*, a *set of libraries*, and an *instrumentation system*.

The *event processing system* handles the different events the detection software monitors. When the wrapper generates a specific event, the core collects the features extracted by the wrapper and possibly infers some complementary features from the extracted ones (e.g., the core can infer the *advInterval* used by an *Advertiser* or a *Peripheral* from the timestamps of the advertisements received from that device). Then, the *event processing system* propagates the event and a structure containing the collected features to the loaded detection modules by executing the corresponding callbacks.

The core also exposes an *instrumentation system*, that can be used by the detection modules to interact with the controller. This system propagates the function calls to the wrapper, allowing it to enter a specific state or trigger an action in a generic way. It also provides various *libraries* facilitating the modules' development. The core exposes a custom memory allocator, allowing to dynamically allocate and release of memory without interfering with the native memory management (the embedded detection software manages its own independent memory), a hashmap implementation, and a logging system, allowing to send detection alerts to the Host.

3.4.3 Detection modules. The detection modules implement the detection strategies: they are generally responsible for analyzing the features provided by the core component to detect attacks. They can declare a set of callbacks executed when a specific event occurs, for example, when a packet is received, or a connection is initiated. They also have access to features collected and inferred using a specific structure and can trigger various behaviors using the instrumentation API. We describe both supported low-level attacks and the corresponding detection algorithms in section 5.

Each module is independent and can be considered a small embedded detection software. With this design, the user can choose which modules to include in the embedded detection software and easily write new ones. The currently implemented detection modules require an average of 64 lines of standard C code, with 17 lines for the simplest heuristic (KNOB) and 108 lines for the more complex one (InjectaBLE). Depending on the requirements of the selected modules, a system of dependencies also allows one to compile and flash only a subset of the framework features. It is particularly relevant given the constraints in time and memory inherent to an embedded approach.

3.5 Architecture of the OASIS framework

The OASIS framework allows the generation of the aforementioned embedded detection software and injects it into the memory. The framework is composed of four main components, as shown in

Figure 3: the *Firmware analyzer*, the *Build system*, the *Patcher* and the *Monitor*. They are described in the following section.

3.5.1 Firmware analyzer. The framework relies on source code and configuration files describing a target to instrument the controller, including the wrapper source code, linker scripts, and configuration files. These files describe all the information the framework needs to patch the controller firmware, inject the detection software code into the memory, and interact with the controller.

Identifying the information needed to generate these files generally requires reverse engineering of the controller firmware, most proprietary and not documented. Since this process is tedious and error-prone when performed manually, the role of the *firmware analyzer* is to automate this reverse engineering task and the generation of the corresponding target-specific files.

The process is divided into two main steps. The first is dedicated to reverse engineering the provided firmware, while the second uses the collected information to generate the source and configuration files describing the target. The reverse-engineering step is mainly based on an automated static analysis of the firmware binary, which tries to identify the relevant functions, variables, and structures using regular expressions describing specific instruction patterns or values. It exploits the fact that different firmwares may share many similarities because of code reuse, which allows us to automate the analysis of several firmwares sharing the same controller architecture. This component currently supports Broadcom, Cypress, and Nordic Semiconductor proprietary stacks.

Once the firmware is analyzed, the tool generates the target's configuration and source code files to instrument it. It automatically disassembles the functions linked to a specific event to identify instructions to patch, allowing it to build the list of firmware instructions to patch. The wrapper source code, linker files, and configuration files are automatically generated from the previously extracted information.

3.5.2 Build system. Once generated, the target files are provided as input to the *build system*, with the target-agnostic software components (e.g., the core and the selected detection modules). The *build system* comprises a set of scripts for generating the final list of instruction patches and binary blobs that will be injected into the memory using standard tools such as the GNU GCC compiler and assembler.

The build system performs the following steps:

- **Detection modules compilation:** Each selected module is compiled without linking, allowing the corresponding binary blobs to be generated.
- **Modules callbacks generation:** For each selected module, the build system lists the callbacks needed by the module. Then, a glue C source code, including the module callbacks as function pointers for every event, is generated, allowing the core to redirect the execution flow to the correct module callback when the event occurs.
- **Trampoline functions generation:** For each patch required to instrument the target, the build system generates a trampoline function to save the context, restore it, and execute the removed instruction.
- **Compilation and linking:** The whole embedded software (including the core, the target wrapper, the detection modules, the glue code, and the trampoline functions) is compiled and linked. A dependency mechanism allows compiling only the required software components if the selected modules do not use some components.
- **Symbols extraction:** Each symbol contained in the compiled binary is extracted from the binary and stored in a temporary file containing the symbol name, address, and content.
- **Patches generation:** The final list of patches and binary blobs is generated by combining the symbols previously extracted from the binary and the patches that must be applied to the controller firmware to instrument it.

3.5.3 Patcher and monitor. Finally, once the list of patches has been generated, the framework can inject them into the memory using the *patcher* system. Depending on the type of controller used, a different back-end may be used to execute the patching process (e.g., InternalBlue [22], OpenOCD).

3.6 Framework usage

The framework can be easily used or extended thanks to the previously mentioned components. A typical workflow is composed of the following steps:

- **Generating the target-specific files (optional):** If the target-specific files have not been previously generated (the framework includes a set of pre-generated files for various targets), the users can dump the firmware and use the **firmware analyzer** to automatically perform the reverse engineering process and generate the corresponding target-specific files.
- **Selecting detection modules:** Users can easily select the modules they want to include in the final embedded detection software or write their own modules using standard C code. Other components do not require any modifications if the existing features are sufficient to perform the detection.
- **Building and patching the embedded detection software:** Users can then execute the **build system** to build the corresponding embedded detection software, then they can inject it into the memory using the **patcher**.
- **Monitoring the embedded detection software:** Users can debug the embedded detection software or monitor the generated logs and detection alerts using the **monitor**.

4 CONTROLLERS INSTRUMENTATION

We focused our work on three heterogeneous and widely used BLE stacks: the *Broadcom/Cypress stack*, embedded in a large number of Bluetooth chips from these manufacturers, the *SoftDevice* from Nordic Semiconductors, embedded in their BLE-enabled chips (e.g., nRF51 and nRF52 families), and the BLE stack included in the Zephyr open-source OS. This section briefly presents the internals and the instrumentation methodology we applied to the two proprietary stacks, illustrated in Figure 4, as they required a significant reverse engineering effort. We performed partial reverse engineering for each analyzed stack, targeting a representative set of firmwares implementing the stack. It allowed us to identify

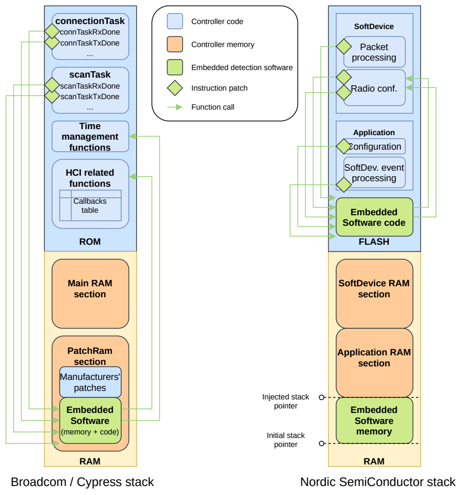


Figure 4: Embedded detection software integration in proprietary stacks.

the underlying software architecture, the implementation of the features listed in our detection requirements (presented in subsection 3.2), and the memory mapping.

4.1 Broadcom and Cypress BLE controllers

Bluetooth-enabled chips from Broadcom and Cypress use a proprietary stack based on a real-time OS named *ThreadX*. The chips involved, based on an ARM Cortex M3 processor, are common in the wild and can be embedded in various types of devices, such as smartphones (e.g., Nexus 5, Samsung Galaxy S20), computers (e.g., Raspberry Pi) or IoT devices (e.g., FitBit Charge). These chips are poorly documented, but several works[12; 13; 22] have partially documented their internals.

The BLE features are implemented as *tasks*, scheduled by the *Bluetooth Core Scheduler* and representing a specific state (e.g., *connection*, *scan*). A task is described by a set of functions linked to a specific event (initialization, packet reception, packet transmission, etc.) and listed in a specific callbacks table. We hooked the initialization and packet processing functions linked to each BLE task, allowing us to analyze the received and transmitted packets in real time while being able to detect the active GAP role. We also extracted the structures that store relevant features, such as connection parameters or Bluetooth device addresses, from some radio configuration functions.

A specific thread handles the high-level operations, especially the HCI management. The thread processes every HCI command and leads to the execution of a specific function, which is stored in a table of function pointers indexed by the command opcode. The HCI events are generated using an allocation function that allocates and initializes the event buffer while another function allows their transmission. We hooked both the processing of the command thread, allowing us to inject arbitrary commands to trigger high-level operations, and the HCI events functions used to build our logging system by passing detection alerts to the Host.

The firmware is stored in ROM, but the manufacturers have included a mechanism named *PatchRam*[2] for updating it: a specific memory area in RAM can be used to store a limited number of ROM changes. Manufacturer patches are written in a dedicated RAM area. A specific ROM instruction of the original firmware is patched using *PatchRam* to redirect the execution flow to the updated function in RAM. These mechanisms are triggered using vendor-specific HCI commands, allowing us to easily divert them to patch the existing firmware and inject our own code into memory. The embedded detection software code and data are stored in the manufacturer's patch section of RAM, while the *PatchRam* mechanism can be used to modify the firmware instructions in ROM to set up our hooks. *InternalBlue* tool makes this process much easier, so our framework uses it as a back end to patch and monitor these chips.

4.2 Nordic Semiconductors SoftDevice

Nordic Semiconductors designed a custom proprietary controller for its BLE-enabled chips (e.g., nRF51 and nRF52 families, based on ARM processors), named *SoftDevice*. These chips are commonly used in IoT devices, and multiple versions of the *SoftDevice* can be found in the wild.

The *SoftDevice* is provided by the manufacturer as a binary blob, which is loaded in the lowest parts of the flash. The user application is flashed in the upper part of the flash and communicates with the *SoftDevice* using a non-standard proprietary API based on supervisor calls. A typical application initializes the *SoftDevice*, configures it to enable the needed BLE features, then monitors the events transmitted by the controller by calling a specific function in an infinite loop. The *SoftDevice* manages the low-level operations: a single packet processing function is called by the radio interrupt when a packet is received or transmitted, which can identify the current GAP role and the current radio operation using a set of internal variables and structures. We also identified a set of configuration functions to store relevant features such as connection parameters in the internal structures. We mainly hooked packet processing and configuration functions in the *SoftDevice* component and extracted various features from the internal structures we identified. The function used by the application to collect the *SoftDevice* events has also been hooked, allowing us to generate the right supervisor call when we need to trigger a high-level action. Similarly, we hook the application's entry point to execute our initialization routine, allowing us to initialize the memory and configure a timer to facilitate time-management operations. The strategy to patch the firmware and inject our code into the memory is based on modifying the firmware binary. The firmware instructions to patch are altered in the binary itself and then the code and the memory of our detection software are appended at the end of the firmware. We also inject a decreased stack pointer initialization value in the interrupt vector, allowing us to set aside a specific zone of the RAM to avoid conflicts between the memory used by the SoftDevice, the application, and our detection software. This modified firmware is flashed into the chip's flash using OpenOCD, and our initialization hook copies the memory zone from the flash to the reserved RAM zone.

5 DETECTION OF LOW-LEVEL BLE ATTACKS

In this section, we describe five major low-level attacks related to the protocol design and discuss how these attacks can be detected using appropriate features at the BLE controller level to build relevant detection modules using OASIS framework. We purposely focus on attacks related to the protocol design, which cannot be easily fixed without changing the specification.

5.1 GATTacker and BTLEJuice: Man-in-the-Middle attacks

Attacks presentation. Two main strategies have been considered to perform a Man-in-the-Middle attack targeting a BLE connection. They are both based on a spoofing strategy targeting the advertisements transmitted by a *Peripheral* before initiating the connection, even if they adopt different approaches to perform this operation. *GATTacker*[19] advertises spoofed advertisement packets faster than the legitimate *Peripheral* to maximize the probability of receiving the *Connection Request* before the legitimate device. Once the *Central* is connected to the attacker fake *Peripheral*, the attacker initiates a connection using a second dongle with the legitimate *Peripheral* to establish the Man-in-the-Middle attack. *BTLEJuice*[6] establishes a connection with the target *Peripheral*, forcing him to stop transmitting its advertisements. Then, the attacker uses a second dongle to expose a spoofed *Peripheral*, waiting for a *Central* to initiate a connection. Let us emphasize that the spoofing strategies exploited by these Man-in-the-Middle attacks are also a mandatory preliminary step for more complex attacks, such as BLESA [37]. As a result, detecting such strategies provides a first line of defense for these attacks.

Detection strategies. Our strategy to detect *GATTacker* is based on the idea that a *Peripheral* transmitting advertisements must follow a specific channel hopping pattern, which depends on two parameters (the *advDelay* and the *advInterval*, as discussed in subsection 2). If an attacker is transmitting advertisements simultaneously, a node monitoring the advertising channels as a *Scanner* or a *Central* should receive both the legitimate and the spoofed advertisements and be able to detect that the received packets are not compliant with the protocol specification, indicating the presence of a malicious node. To detect this situation, we first estimate the *advInterval* for each device transmitting advertisements in the absence of attacks. This estimation is based on a sliding window that is filled with the duration between two consecutive advertisements from the same device received on the same channel. Once the window has been completely filled, the minimum value in the window is considered as our *advInterval* estimate (keeping the minimum value allows us to reduce the impact of the random *advDelay* parameter). Then we set a detection threshold to the *advInterval* value minus the maximum *advDelay* value, representing the worst legitimate case. Each time a new packet is received, a new estimate is calculated. An alert indicates the presence of a malicious node if the calculated value is lower than the detection threshold.

The *BTLEJuice* attack is more difficult to detect because a node monitoring the advertising channel has no guarantee to observe the *Connection Request* transmitted by the attacker. Therefore, we adopt another strategy, allowing the target *Peripheral* to detect its own spoofing by an attacker. When a connection is established, the *Peripheral* simultaneously maintains the connection and scans the advertisements. During this scan operation, the *Peripheral* checks if its own address is included in the advertisements packets and raises an alert if this situation is detected.

While these strategies provide effective detection, they have some limits that should be highlighted. The *GATTacker* detection must correctly estimate the legitimate *advInterval* before it can detect an attacker node. Consequently, the detection requires that the monitoring device has been able to fill its sliding window to estimate the interval before the attack begins. This learning phase could be removed in a controlled environment with known *advInterval* values. Likewise, the *BTLEJuice* detection requires that the target *Peripheral* can maintain a connection and scan the advertisements simultaneously: it may be problematic for specific controllers implementing only a subset of *BLE* roles. Random addresses can also be problematic. They could be associated with a device without storing each address by leveraging the Identity Resolving Key (IRK) knowledge if they are resolvable. Otherwise, a tradeoff between space and detection performance must be made.

5.2 BTLEJack attack

Attack presentation. Another attack that may have a significant impact on availability is named *BTLEJack*[8]. This attack, presented by D. Cauquil, is a jamming approach allowing to jam an established connection or to hijack the Central role under certain circumstances. The attacker first synchronizes with an established connection, then transmits a jamming signal when the *Peripheral* sends a reply to the *Central* at each connection event. The attack exploits a counter mechanism detecting link losses by incrementing the counter value for every missed or invalid packet. When this counter reaches a predefined threshold, the *Central* considers the connection as lost and exits, allowing the attacker to interrupt it or, in the worst case, to hijack the *Central* role if the *Peripheral* does not disconnect immediately after the *Central* disconnection.

Detection strategy. From a *Central* node perspective, detecting this attack can be performed easily: unlike a normal connection loss, the *Central* receives frames including an invalid CRC on multiple consecutive connection events during an attack, while none packet is received in a legitimate scenario. This situation has a very low probability of occurrence in a legitimate situation, as the channel hopping algorithm ensures the use of multiple channels distributed along the ISM band. The detection strategy involves raising an alert when the consecutively received frames with integrity corruption counter reach the value of the connection counter minus one.

5.3 KNOB attack

Attack presentation. The *KNOB* attack, presented by Antonioli et al. [1], allows an attacker performing a Man-in-the-Middle attack to inject a low entropy value during the pairing process. Indeed, the pairing process includes a protocol for entropy negotiation, allowing each involved device to indicate how many entropy bytes can be used during key generation. As a result, an attacker can perform an entropy downgrade attack by setting this number of bytes to 7 instead of 16 in the case of BLE. Consequently, the key can easily be brute-forced, compromising the security of future communications between the involved devices.

Detection strategy. This attack can be detected by both a *Central* or a *Peripheral* using a simple passive strategy. When a Pairing Request (i.e., the packet type used to negotiate the entropy value) is received, the entropy value field is extracted from the packet payload. An alert is raised if the value is less than 10 bytes of entropy. Even if the protocol technically allows such a lower value to be used legitimately, considering that a device should not be allowed to use an entropy value low enough to allow a brute force seems a reasonable assumption from a security perspective.

5.4 InjectaBLE attack

Attack presentation. The last attack we focus on is a new injection attack targeting BLE communications called *InjectaBLE* [10]. This attack abuses a feature used to compensate for potential clock drift between devices. When a *Peripheral* enters reception mode to receive a packet from the *Central* during a connection, it listens during a short window (named *window widening*) after and before the theoretical instant, allowing an attacker to exploit a race condition and inject a malicious packet before the legitimate *Central* node. This attack is critical, especially if the connection is not encrypted. It allows any role to be hijacked or a Man-in-the-Middle to be performed by injecting specific frames.

Detection strategy. This attack can be detected by monitoring the interval between two consecutive received packets by the targeted *Peripheral* itself. We can detect if a packet is injected by comparing the last interval to the legitimate connection interval. If the interval is less than the theoretical interval minus an empirically estimated threshold, we consider the frame as malicious and raise an alert.

6 EVALUATION OF OASIS CAPABILITY

We first conducted two sets of experiments to evaluate OASIS's detection capabilities. We tested each module independently in office conditions. Then, we evaluated our IDS detection capabilities on two off-the-shelf devices in real-life conditions. Finally, we evaluated the performance of the *firmware analyzer*.

6.1 Detection performance evaluation

We performed several experiments to evaluate the detection performance for each detection module on multiple devices connected to a central gateway monitoring detection logs while legitimate and malicious traffic was generated.

Table 1: Targets used for each experiment.

	Targets				
	Ra	Ne	Ga	D_1	D_2
GATTacker	✓	✓		✓	✓
BTLEJuice			✓	✓	✓
KNOB			✓	✓	✓
InjectaBLE	✓			✓	✓
BTLEJack		✓		✓	

6.1.1 Experimental setup. Our first set of experiments was conducted on five different targets: BCM4345C0 in a Raspberry Pi 3+ board; BCM4339 in a Nexus 5 smartphone; the Gablys, a smart keyfob with an nRF51822 controller; a CYW20735 in an IoT development kit, and an nRF51422 in the nRF51 Development kit. Each are embedding various SDK examples (e.g., *Scanner* and *Peripheral*), respectively called Ra, Ne, GA, D_1, D_2 in Table 1.

Most attacks used the *Mirage* offensive framework [11], which implements multiple offensive strategies. We extended it with a KNOB implementation. For Gattacker, BTLEJuice, and KNOB attacks, we performed 250 periods of attacks randomly alternated with 250 periods of legitimate traffic, targeting a connected light bulb or the evaluated boards themselves, depending on the attack. InjectaBLE and BTLEJack attacks require sniffing a connection, a non-trivial task [9; 26]. They can sometimes fail due to sniffer desynchronization. As a result, performing a fully automated experiment could lead to invalid results (e.g., an attack failure being considered a false negative), and we manually monitored the experiment. It allowed us to control the attack success but impacted the number of attacks that could be performed in a reasonable amount of time, leading to 100 attacks alternating with 100 periods of legitimate traffic in these specific cases.

6.1.2 Experiment results. For each experiment performed, we compute the number of true positives (TP) , false positives (FP), true negatives (TN) and false negatives (FN) the Recall and the Precision by the target. The results for each experiment are listed in Table 2. Multiple observations can be made from these results. First, we can emphasize that our detection strategies successfully detect attacks, as illustrated by the good Recall values we obtained (ranging from 0.94 to 1.0). Moreover, these experiments have been conducted in realistic conditions, using standard offensive tools. Similarly, the high Precision values, all between 0.93 and 1.0, show that our detection strategies generate only a very small number of false positives. In addition, four of our five experiments have a precision value equal to 1.0 for every tested target. The detection strategies that rely exclusively on passively monitoring advertisements (e.g., GATTacker) generate slightly more false positives: this can be explained by the fact that they have to compute some estimates that may be impacted by some environmental changes inherent to these intensively used channels. Finally, the results of a given experiment are globally homogeneous for each tested target. This shows that our detection modules are, as expected, independent of the underlying

Table 2: Experimental results.

Experiment	Target	TP	FP	TN	FN	Recall	Precision
GATTacker	Ra	250	0	250	0	1.0	1.0
	Ne	250	0	250	0	1.0	1.0
	D_1	250	0	250	0	1.0	1.0
	D_2	250	19	231	0	1.0	0.93
BTLEJuice	Ga	245	0	250	5	0.98	1.0
	D_1	239	0	250	11	0.96	1.0
	D_2	250	0	250	0	1.0	1.0
KNOB	Ga	247	0	250	3	0.99	1.0
	D_1	250	0	250	0	1.0	1.0
	D_2	249	0	250	1	0.99	1.0
InjectaBLE	Ra	99	0	100	1	0.99	1.0
	D_1	100	0	100	0	1.0	1.0
	D_2	94	0	100	6	0.94	1.0
BTLEJack	Ne	95	0	100	5	0.95	1.0
	D_1	98	0	100	2	0.98	1.0

wrapper implementations. Even if some of our strategies cannot be implemented on every target due to role requirements, these experiments demonstrate that these defensive detections can be implemented on various devices, including a smartphone, a Raspberry Pi, and a commercial connected object with minimal resources.

6.2 Real life experiment

Our second experiment evaluated our embedded IDS in conditions as close as possible to a real-life deployment. The main goal of this experiment was to analyze the behavior of our detection heuristics under various environments in a typical daily use case. More precisely, it allowed us to understand our environment-dependent heuristics better and identify the factors impacting the false positive rates. We experimented with two commercial products: the Nexus 5 smartphone and the Gablys smart keyfob. We included two detection modules on the smartphone (KNOB and GATTacker) and two detection modules on the keyfob (BTLEJuice and KNOB), covering both the connected and the advertising modes. We recorded the alerts on both devices simultaneously for twenty-four hours. We intensively used the BLE controller during this period by performing scan operations and connections. We moved with the devices running OASIS between different environments representing various radio profiles: an apartment, a crowded coffee shop, a street, and a store. We manually performed 29 attacks at random instants in the apartment and coffee shop environments. We only used legitimate traffic in street and store environments.

Our experimental results are detailed in Appendix D. We detected every attack we performed during this experiment without interfering with the legitimate use of devices. We observed that the GATTacker heuristic was too sensitive in some noisy environments. We suspect that packet corruption may impact the advertising interval estimation. Two factors can be optimized to decrease GATTacker sensitivity: the sliding window size and the time threshold before raising an alert when the estimation decreases.

6.3 Firmware Analyzer evaluation

We evaluated the *firmware analyzer* on 9 Broadcom / Cypress firmwares extracted from BLE controllers embedded in heterogeneous devices[1] and on 361 Nordic SemiConductors firmwares for nRF51422 and nRF51822 chips. The 361 firmwares were generated from the BLE examples included in six nRF5 SDK versions with nRF51 support (9.0.0 to 12.3.0) and are representative of various roles (e.g., Peripheral, Central or Both) and different SoftDevices (e.g., s110, s120, s130). Our fully automated approach managed to find 100% of functions and data in 307 of the 370 firmwares (stripped binaries). On average, we find 99.34% of functions and 97.68% of data, the worst case being 88.88% and 80%. These results show the feasibility of building a set of heuristics allowing the detection of relevant functions and data from monolithic firmware blobs, making our embedded approach practical for common proprietary BLE controllers. Regarding the generalization of this strategy, let us emphasize that 1) only a limited number of strategic functions and data are required to build the instrumentation code, 2) open-source implementations of BLE stacks are more and more common in the wild (e.g., Zephyr, NimBLE, BTStack, Blessed), considerably facilitating the controller instrumentation, 3) automatic reverse engineering of monolithic firmwares is an active research field, and this component could be improved by exploiting complementary research works[17; 34] to implement a more sophisticated approach.

[1]Nexus 5, iPhone 6, Raspberry Pi 3/3+/Zero W, Samsung Galaxy S8, Samsung Galaxy S10, MacBook Pro 2016, Development kits

7 EVALUATION OF OASIS IMPACT

Devices relying on BLE are usually low-power, run on batteries, and have low memory and CPU resources. Therefore, the impact of OASIS on power consumption and runtime performance is essential to evaluate. For this purpose, we performed high-resolution power measurements on development boards in typical use cases. As the detection modules are generally specific to a given role, we tested several combinations of detection modules, covering typical BLE use cases (Table 3). Then, we evaluate the impact on packet timings and the memory usage of OASIS.

7.1 Experiment 1: fine-grained power consumption analysis

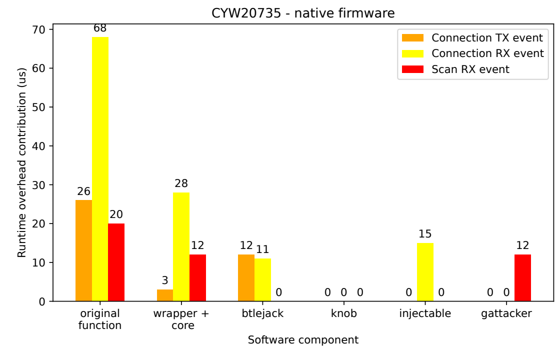
The first experiment evaluated each profile's power consumption with high precision. All tests were conducted on the nRF52-DK with current consumption monitored using Power Profiler Kit (PPK) from Nordic Semiconductor [30]. For this test, we used Zephyr's "HCI over UART" example, which supports all BLE roles, allowing us to evaluate all the profiles. For each profile, we collected 4-minute long traces under various configurations (with or without OASIS running one or a combination of supported modules). We observed a low but measurable increase in average power consumption when OASIS is deployed. The increase depends on the profile selected and the number of modules. Depending on the conditions, the power consumption increases from a minimum of 0.54% to a maximum of 1.11% as described in Table 4 (Details in Appendix A). We also observed that the power consumption increase was consistent with the modules' complexity and the number of modules loaded. We could also note the marginal cost of embedding multiple modules instead of the most costly one in profiles using connected mode (increase of respectively 0.07 % and 0.09 % for central and peripheral profiles), suggesting that the power consumption overhead in connected mode is mainly related to wrapper and core

Table 3: Micro-Benchmarking configurations for power consumption overhead of OASIS.

Profile	Supported modules	Benchmark action
Scanner (P_S)	GATTacker	running a scan
Peripheral (P_P)	InjectaBLE, KNOB, BTLEJuice	accepting connection
Central (P_C)	BTLEJack, KNOB	initiating connection
Multiple (P_M)	all	alternating scan & connections

Table 4: Power consumption of different OASIS profiles.

	Average power consumption (μW)		Percentage increase
	without module	with module	
InjectaBLE	1226	1232	0.54 %
BTLEJuice	1226	1234	0.67 %
KNOB	1212	1220	0.68 %
GATTacker	1223	1236	1.07 %
BTLEJack	1212	1225	1.11 %



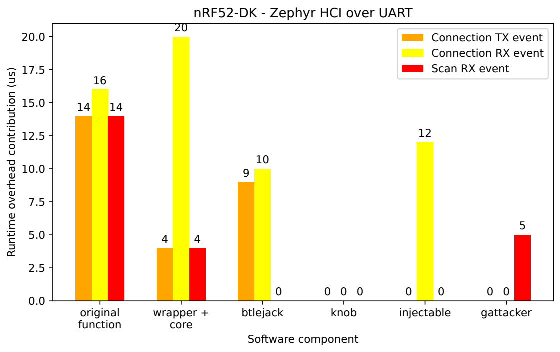


Figure 5: Contribution of components to the packet delay for each event.

processing rather than the modules. As a result, it motivates us to focus on potential optimization work on these specific components. We complemented this evaluation by a large-scale analysis on 100 Raspberry Pi, described in Appendix B.

7.2 Experiment 2: Execution time analysis

We performed experiments to evaluate the runtime overhead of OASIS under various conditions. We performed these analyses on development boards from two manufacturers (CYW20735 and nRF52-DK). On those devices, we instrumented several points using hooks (see Figure 6). This instrumentation has low impact and logs events with microseconds accuracy. For each profile (from Table 3), we ran two-minute benchmarks under various configurations (without the IDS, with the IDS, and with different combinations of modules) and collected the execution time. Figure 5 shows the average timing overhead contribution of OASIS core and wrapper as well as each detection module (average execution time), which we computed from the measurements.

For both devices, our results are consistent with the power measurements (computations dominate power consumption). These results confirm that the wrapper and core components are the main contributors to the overhead. We can see the very low overhead of the simpler modules, such as KNOB, which is below the microseconds accuracy. On the other hand, the more complex modules, with higher time overhead, perform many memory accesses. In the worst case (OASIS loaded with all modules, CYW20735 board), processing a packet and response overhead is $54\mu s$, and the packet response

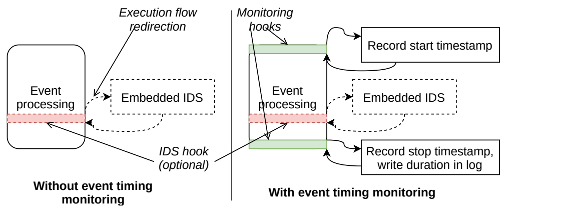

Execution flow redirection
Event processing
Embedded IDS
IDS hook (optional)
Without event timing monitoring
Monitoring hooks
Record start timestamp
Event processing
Embedded IDS
Record stop timestamp, write duration in log
With event timing monitoring

Figure 6: Event instrumentation (hooks) to measure execution time.

becomes $122\mu s$. This remains under the standard's $150\mu s$ response delay requirement.[2] This is consistent with the fact that we did not see any error or communication failure due to using OASIS. If more complex modules are integrated, parts of the processing can be deferred after the packet response, reducing the impact on time-critical code paths.

7.3 Experiment 3: Memory analysis

The memory used by OASIS can be split into three main sections: code, static data, and dynamic data (e.g., heap). Dynamic data is only used by modules relying on a hashmap (e.g., btlejack, gattacker and injectable), and is dependent on the environment. The end user can arbitrarily configure an upper limit (default is 2048 bytes). We analyzed the amount of memory used for static data and code by each OASIS component for five different targets: detailed results are presented in Table 5.

We observe an overall static memory consumption between 4291 (Nexus 5) and 6305 bytes (nRF51 peripheral example) when all modules are loaded. The difference can be explained by a) the wrapper complexity (nRF51 require more instrumentation code because of SoftDevice structure), b) the architecture in use (ARMv6-M is used by nRF51, introducing a supplementary cost in memory compared to ARMv7-M to handle some operations). We can observe that most of the static memory in use is related to the wrapper and core components. The static memory consumption for a module is included between 48 (best case, knob module) and 500 bytes (worst case, injectable module). It is consistent with their complexity.

These results show that our approach only requires a few kilobytes of memory to run with every module loaded. On a nRF51822 with 256KB of memory or a BCM4339 with 196KB of memory, the framework with all modules represents 3.2% of the memory (assuming a full heap with a size equal to 2048 bytes). Supplementary engineering efforts may significantly reduce memory usage, by introducing a fine grain dependencies management or exploiting more aggressive compiler optimizations.

8 RELATED WORK

8.1 State of the art

One of the main challenges to efficiently monitor BLE is related to channel hopping algorithms. In [26], Ryan highlighted the problems associated with these algorithms and developed some heuristics for inferring connection parameters. This algorithm was improved by Cauquil [7] to analyze channel maps. A similar implementation

[2] Bluetooth Core Specification [5] Rev. 5.3, Vol. 6, Part B, Section 4.1.1, p. 2737

Table 5: Memory consumption per target (code and static data)

Target \ Component		total (all)	wrapper	core	injectable	btlejack	btlejuice	gattacker	knob
nRF51 SoftDevice (peripheral)	code	5278	1266	2708	496	256	124	380	48
	data	1027	587	427	4	4	1	4	0
Raspberry Pi 3	code	3860	730	1902	432	236	124	384	52
	data	477	41	423	4	4	1	4	0
Nexus 5	code	3798	668	1902	432	236	124	384	52
	data	493	41	439	4	4	1	4	0
CYW20735	code	3904	774	1902	432	236	124	384	52
	data	484	41	430	4	4	1	4	0
nRF52 Zephyr (hci_uart)	code	3886	692	1958	432	236	124	392	52
	data	457	21	423	4	4	1	4	0

was adapted to the Ubertooth One by Sarkar et al. [27]. Cauquil developed an approach to synchronize a sniffer with the second PRNG-based channel hopping algorithm [9]. Finally, Qasim Khan presented *Sniffle* [25], a new sniffer implementation that increases the probability of successfully synchronizing the sniffer with a connection by tracking the device during advertising.

Unfortunately, these passive approaches suffer from several serious limitations that significantly impact the completeness and representativity of monitored communications. First, most of these sniffers can only monitor one connection at a time. Del Arroyo et al. [18] proposed an opportunistic algorithm based on a scheduler in the Ubertooth One to monitor multiple connections simultaneously, but the underlying outdated hardware limits this work and may miss some packets depending on the environment. Second, most existing implementations are unstable, partially because of the use of various heuristics unsuitable for some devices (e.g., frequent channel map updates). Consequently, comprehensive monitoring of the Link Layer traffic of BLE communications from an external probe remains an open challenge, especially in connected mode.

This situation significantly impacts the research on intrusion detection systems for BLE. Indeed, those are mostly based on sniffers; those are therefore limited to monitoring advertisements, limiting the detection to spoofing or DoS attacks targeting the advertising mode. Wu et al. presented BlueShield [36], an approach to detect spoofing attacks by profiling monitored devices using features inferred from the advertisement packets. Sung et al. [32] explored using Received Signal Strength Indicators (RSSI) to detect intruders. Finally, Yaseen et al. [39] presented *MARC*, a framework to detect Man-in-the-Middle attacks by exploiting four features inferred from advertisement packets, such as the advertising interval or RSSI.

Other research works also explored the analysis of traffic in connected mode. For example, Newaz et al. [23] combine an n-gram-based approach with various machine learning techniques to detect attacks by analyzing irregular traffic-flow patterns on Personal Medical Devices. Satam et al. propose a similar method for Bluetooth Classic networks in [28; 29]. Similarly, Lahmadi et al. [21] explores using Machine Learning techniques to identify Man-in-the-Middle attacks by building a model of legitimate behaviors based on features such as RSSI, channel numbers, or distance. While these works provide interesting results regarding traffic analysis, they performed offline detection on datasets and are difficult to deploy.

Some complementary research works propose defensive strategies that do not rely on intrusion detection or prevention. For example, Wu et al. propose LightBlue [38], an automatic approach to perform Bluetooth protocol stacks debloating to remove unused code exposing an avoidable attack surface or providing resources exploitable by an attacker (e.g., ROP Gadgets).

Several researchers approached monolithic firmware analysis. Gustafson et al. proposed Shimware [17], a methodology to retrofit monolithic firmwares and automate some aspects of this process. FirmXRay [34] is an automated Bluetooth vulnerability detection mechanism providing scalable firmware analysis. These works may complement our work on OASIS firmware analyzer. InternalBlue [22] is a framework to instrument and experiment with Broadcom Bluetooth Firmware. Cui et al. proposed software Symbiotes [14], which deploys a wide number of randomized hooks for including various detection mechanisms in embedded devices. Unfortunately, this approach is not applicable to Bluetooth controllers according to their constraints regarding resources and patching limit (e.g., patching mechanisms with a limited number of slots), motivating the development of a lightweight approach.

8.2 Comparison with previous work

Several papers [21; 23; 32; 36; 39] proposed BLE IDS in recent years. In Table 6, we conduct a qualitative comparison based on multiple factors: extensibility, implementation availability, and scope. We indicate if the detection can be performed with a given approach for each attack. We include the number of supported detection features, grouped by categories. However, establishing a fair comparison between OASIS and previous work is challenging, as threat models differ, and OASIS is the first work to explore an online embedded intrusion detection approach, aiming at protecting the device itself instead of a specific environment. Because of the lack of embedded IDS in previous work, we cannot compare the IDS memory/CPU usage on the device. We also note that previous work is often limited to spoofing attacks (e.g., BTLEJuice or GATTacker) and systematically relies on BLE sniffers, implying the deployment of static probes (known to be a significant challenge actively studied in literature [24; 33]) or the manual collection of traffic. Previous work also performs offline Intrusion detection on small datasets when the connected mode is considered. Moreover, most of these works can't be easily reproduced: only BlueShield [36] source code is available, but its complex deployment significantly complicates an experimental evaluation (details in Appendix C).

9 DISCUSSION

This paper focused on low-level attacks, which are difficult to detect and mitigate by design. However, our approach could be easily applied to many other active attacks targeting the BLE protocol. Indeed, implementing the detection at the lowest level accessible by software allows detecting low-level attacks but is also relevant to detect attacks targeting the upper layers or being linked to a specific implementation. We believe this embedded detection approach is relevant in the IoT context. The need to adapt IoT IDS for low resource consumption has already been highlighted by previous works, such as TWINKLE [31], where a two-mode adaptive security model, alternating between a low consumption mode and a vigilant mode with higher performance is proposed. Furthermore, OASIS can also easily cooperate with other defensive components running on the device itself (e.g., offloading costly modules on Host). While

Table 6: Comparison between OASIS and current state-of-the-art BLE detection approaches
✓: supported, ✎: can be implemented, ✗: not supported

		OASIS	BlueShield [36]	MARC [39]	HEKA [23]	I.S. IT [32]	MiTM ML [21]
Online Detection		✓	✓	✓	✗	✓	✗
Extensible		✓	✗	✗	✗	✗	✗
IDS Mobility		✓	✗	✗	✗	✗	✗
Scope		Generic	Stationary Networks	Medical	Medical	Beacon Tags	Generic
Detected Attacks	BTLEJuice	✓	✓	✗	✓	✗	✓
	GATTacker	✓	✓	✓	✗	✗	✓
	InjectaBLE	✓	✗	✗	✗	✗	✗
	BTLEJack	✓	✗	✗	✗	✗	✗
	KNOB	✓	✗	✗	✗	✗	✗
	Device DoS	✎	✗	✗	✓	✗	✗
	Replay	✎	✗	✗	✓	✗	✗
	False Data injection	✎	✗	✗	✓	✗	✗
	Physical Intrusion	✗	✗	✗	✗	✓	✗
Modes		Adv. / Conn.	Adv.	Adv.	Conn.	Adv.	Adv. / Conn.
Features collection		Embedded	Static Probe	Static Probe	Manual	Static Probe	Manual
Feat.	**Advertising**	4/4	4/4	3/4	0/4	0/4	0/4
	Connection	4/4	0/4	0/4	1/4	0/4	0/4
	Metadata	6/7	3/7	1/7	0/7	1/7	3/7
Implementation available		✓	✓	✗	✗	✗	✗

we demonstrated during our experiments that OASIS is lightweight and can even be implemented in small IoT devices with limited resources, a tradeoff between the number and cost of detection modules and the available resources is unavoidable on such systems.

We can highlight some challenges of this approach. First, implementing the detection on local nodes complicates the collection of alerts by a Security Operations Center (SOC). However, this can be solved by establishing a channel dedicated to alert reporting between a central node and the local nodes analyzing traffic. Such a channel could also exploit the decentralized nature of our embedded approach, allowing to coordinate complex detection algorithms.

Embedding defensive mechanisms in the device itself also raises the question of the integrity of the detection code and data. While we consider that attacks aiming at altering the controller itself (e.g., memory corruption attacks) are out of the scope of this paper, they may allow to alter or evade the detection logic. Well-known mitigations can be deployed to harden OASIS against memory corruption attacks (e.g. stack canaries, ASLR) [20]. A hybrid (Host instrumentation) or decentralized (cooperation between multiple Controllers) approach could also allow a remote attestation mechanism. Regarding OASIS evasion, the five detection modules are based on observing the necessary consequences of attacks. To our knowledge, no evasion technique should be possible under the assumption of traffic completeness and bug-free implementation.

Another challenge is related to the instrumentation of heterogeneous stacks, sometimes proprietary. While we showed in subsection 6.3, the feasibility of building heuristics to automatically identify relevant functions and data in common proprietary controllers architectures, integrating support for other proprietary architectures may require a reverse engineering effort. This effort must be put into perspective because 1) open-source wireless stack implementations are more and more common, 2) only a limited number of hooks are required, 3) the engineering efforts required by our approach must be compared to current state-of-the-art BLE IDS based on sniffers, which need to address significant challenges (e.g., probe placement, traffic completeness, technical limitations), whereas our solution can immediately protect devices.

Finally, our framework enables IDS deployment in an adverse context to provide an effective solution to IoT insecurity. It may not be systematically possible for a third party to deploy it on commercial devices, as a firmware update is required. Nonetheless, it can also be used while developing an IoT device to harden the BLE controller and protect it from attacks or even integrated by the chip manufacturers themselves.

10 CONCLUSION

In this paper, we presented OASIS, an embedded IDS for BLE. It is based on the instrumentation of the controller firmware, which gives visibility and control over the lowest layers. We demonstrated its relevance by conducting experiments under realistic conditions on various targets, including smartphones and IoT devices with limited resources. We presented detection modules to detect five critical low-level attacks, including ones targeting the connected mode, which were difficult to detect with previous approaches. We further performed micro benchmarks to measure power, memory and packet timing overhead, and real-world experiments. This demonstrates that OASIS is lightweight and can be used in various environments. We provide a modular, generic, and user-friendly framework for instrumenting BLE controllers, suitable for collecting low-level detection features and released as open-source[3].

This framework provides a simple way to instrument BLE controllers to security community. In addition, it could facilitate research in various areas (e.g., vulnerability research, intrusion detection). Future work aims to include new types of controllers, detection modules, and explore new prevention strategies. In addition, we plan to explore the feasibility of building a cooperative IDS using a set of decentralized nodes capable of cooperating and communicating over a secure wireless communication channel.

ACKNOWLEDGMENTS

This work has been partially supported by the French National Research Agency under the France 2030 label (Superviz ANR-22-PECY-0008). The views reflected herein do not necessarily reflect the opinion of the French government. We thank the reviewers for their insightful comments and their suggestions to improve the final version of this article.

[3]**Repository:** https://github.com/RCayre/oasis

REFERENCES

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1047–1061, 2019.

[2] ARM.com. Cortex-M3 Technical Reference Manual r2p0, 7.3 About the Flash Patch and Breakpoint Unit (FPB). ARM Documentation, February 2010. Available at: https://developer.arm.com/documentation/ddi0337/h?lang=en.

[3] Armis. Blueborne Technical White Paper. https://go.armis.com/hubfs/BlueBorneTechnicalWhitePaper.pdf, 2017.

[4] Armis. BleedingBit Technical White Paper. https://go.armis.com/hubfs/BLEEDINGBIT-TechnicalWhitePaper.pdf, 2018.

[5] Bluetooth SIG. *Bluetooth Core Specification*, 07 2021. Rev. 5.3.

[6] Damien Cauquil. Btlejuice: The Bluetooth Smart MITM framework. In *DEF CON*, volume 24, 2016. Available at https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20villages/DEF%20CON%2024%20Internet%20of%20Things%20Village%20-%20Damien%20Cauquil%20-%20Btlejuice%20The%20Bluetooth%20Smart%20Mitm%20Framework.mp4.

[7] Damien Cauquil. Sniffing BTLE with the Micro:Bit. *PoC or GTFO*, 17:13–20, 2017.

[8] Damien Cauquil. You'd better secure your BLE devices or we'll kick your butts ! In *DEF CON*, volume 26, 2018. Available at https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/DEFCON-26-Damien-Cauquil-Secure-Your-BLE-Devices-Updated.pdf.

[9] Damien Cauquil. Defeating Bluetooth Low Energy 5 PRNG for fun and jamming. In *DEF CON*, volume 27, 2019. Available at https://media.defcon.org/DEF%20CON%2027/DEF%20CON%2027%20presentations/DEFCON-27-Damien-Cauquil-Defeating-Bluetooth-Low-Energy-5-PRNG-for-fun-and-jamming.PDF.

[10] Romain Cayre, Florent Galtier, Guillaume Auriol, Vincent Nicomette, Mohamed Kaâniche, and Géraldine Marconato. InjectaBLE: Injecting malicious traffic into established Bluetooth Low Energy connections. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2021)*, Taipei (virtual), Taiwan, June 2021.

[11] Romain Cayre, Vincent Nicomette, Guillaume Auriol, Eric Alata, Mohamed Kaâniche, and Geraldine Marconato. Mirage: towards a Metasploit-like framework for IoT. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 261–270. IEEE, 2019.

[12] Jiska Classen and Matthias Hollick. Inside job: Diagnosing Bluetooth Lower Layers Using Off-the-Shelf Devices. *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, May 2019.

[13] Jiska Classen and Matthias Hollick. Extracting Physical-Layer BLE Advertisement Information from Broadcom and Cypress Chips. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 337–339, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Ang Cui and Salvatore J. Stolfo. Defending Legacy Embedded Systems with Software Symbiotes. In *The 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[15] Great Scott Gadgets. Ubertooth Retirement. Available at: https://greatscottgadgets.com/2022/12-22-ubertooth-retirement/.

[16] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. SweynTooth: Unleashing Mayhem over Bluetooth Low Energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.

[17] Eric Gustafson, Paul Grosen, Nilo Redini, Saagar Jha, Andrea Continella, Ruoyu Wang, Kevin Fu, Sara Rampazzi, Christopher Kruegel, and Giovanni Vigna. Shimware: Toward Practical Security Retrofitting for Monolithic Firmware Images. In *In Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, October 2023.

[18] Jose Gutierrez del Arroyo, Jason Bindewald, Scott Graham, and Mason Rice. Enabling Bluetooth Low Energy Auditing through Synchronized Tracking of Multiple Connections. *Int. J. Crit. Infrastruct. Prot.*, 18(C):58–70, sep 2017.

[19] Sławomir Jasek. Gattacking Bluetooth Smart Devices. In *BlackHat USA*, 2016. Available at http://gattack.io/whitepaper.pdf.

[20] Mahmood Jasim Khalsan and Michael Opoku Agyeman. An overview of prevention/mitigation against memory corruption attack. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, pages 1–6, 2018.

[21] Abdelkader Lahmadi, Alexis Duque, Nathan Heraief, and Julien Francq. MitM Attack Detection in BLE Networks using Reconstruction and Classification Machine Learning Techniques. In *MLCS 2020-2nd Workshop on Machine Learning for Cybersecurity*, 2020.

[22] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth Binary Patching and Experimentation Framework. *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, Jun 2019.

[23] AKM Iqtidar Newaz, Amit Kumar Sikder, Leonardo Babun, and A. Selcuk Uluagac. HEKA: A Novel Intrusion Detection System for Attacks to Personal Medical Devices. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2020.

[24] Steven Noel and Sushil Jajodia. Optimal IDS sensor placement and alert prioritization using attack graphs. *Journal of Network and Systems Management*, 16:259–275, 2008.

[25] Sultan Qasim Khan. Sniffle: A sniffer for Bluetooth 5 (LE). In *Hardwear.io*, 2019. Available at https://www.hardwear.io/netherlands-2019/speakers/sultan-qasim-khan.php.

[26] Mike Ryan. Bluetooth: With Low Energy Comes Low Security. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, Washington, D.C., August 2013. USENIX Association.

[27] Sopan Sarkar, Jianqing Liu, and Emil Jovanov. A Robust Algorithm for Sniffing BLE Long-Lived Connections in Real-Time. In *2019 IEEE Global Communications Conference, GLOBECOM 2019, Waikoloa, HI, USA, December 9-13, 2019*, pages 1–6. IEEE, 2019.

[28] Pratik Satam, Shalaka Satam, and Salim Hariri. Bluetooth Intrusion Detection System (BIDS). In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications, AICCSA 2018*, Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA. IEEE Computer Society, January 2019.

[29] Shalaka Satam, Pratik Satam, and Salim Hariri. Multi-level Bluetooth Intrusion Detection System. In *2020 IEEE/ACS 17th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–8, 2020.

[30] Nordic Semiconductor. Power Profiler Kit. Available at: https://www.nordicsemi.com/Products/Development-hardware/power-profiler-kit.

[31] Devkishen Sisodia, Samuel Mergendahl, Jun Yu Li, and Hasan Çam. Securing the Smart Home via a Two-Mode Security Framework. In *Security and Privacy in Communication Networks*, 2018.

[32] Yunsick Sung. Intelligent Security IT System for Detecting Intruders Based on Received Signal Strength Indicators. *Entropy*, 18(10):1–16, October 2016.

[33] Juan E. Tapiador and John A. Clark. The placement-configuration problem for Intrusion Detection nodes in Wireless Sensor Networks. *Computers & Electrical Engineering*, 39(7):2306–2317, 2013.

[34] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. *FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware*, page 167–180. Association for Computing Machinery, New York, NY, USA, 2020.

[35] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Mathias Payer, and Dongyan Xu. BlueShield GitHub repository. Available at: https://github.com/allenjlw/BlueShield/.

[36] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Mathias Payer, and Dongyan Xu. BlueShield: Detecting Spoofing Attacks in Bluetooth Low Energy Networks. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 397–411, San Sebastian, October 2020. USENIX Association.

[37] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave (Jing) Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. BLESA: Spoofing attacks against reconnections in bluetooth low energy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[38] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. LIGHTBLUE: Automatic Profile-Aware debloating of bluetooth stacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 339–356. USENIX Association, August 2021.

[39] Muhammad Yaseen, Waseem Iqbal, Imran Rashid, Haider Abbas, Mujahid Mohsin, Kashif Saleem, and Yawar Abbas Bangash. MARC: A Novel Framework for Detecting MITM Attacks in eHealthcare BLE Systems. *Journal of Medical Systems*, 43(11):324, 2019.

A POWER CONSUMPTION FOR HIGH PRECISION POWER MEASUREMENTS IMPACT

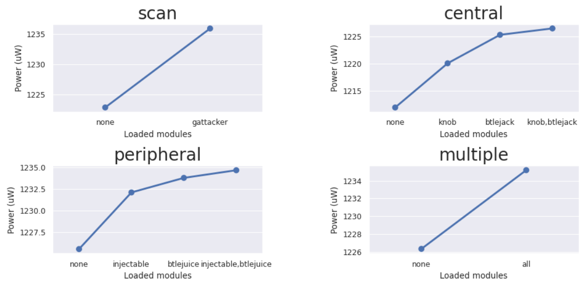


Figure 7: Average power consumption for each profile under different configurations.

B LARGE-SCALE POWER CONSUMPTION ANALYSIS

To evaluate the impact of OASIS on power consumption in a realistic network of devices, we measured the precise power consumption on a 100 Raspberry PI 3B+ network (Figure 10). OASIS was installed on Raspberry's onboard Bluetooth controller (Broadcom BCM4345C0). To reduce the impact of external factors, we perform 144 rounds of experiments of 10 minutes each, where devices perform random connections and communications. For each round, we randomly assign half of the devices to behave as Centrals (performing scanning and connections as Central) while the other half behave as peripherals (performing advertising and connections as Peripheral). We also alternate rounds with and without IDS. The results show that OASIS has a small but measurable effect (illustrated in Figures 8 and 9). A Welch's independent samples t-test showed that the 0.51 % difference was significant ($t(51485.37) = 53.85, p = 0.0 < 0.05$)), the 95% confidence interval is [1.18,1.27] while the cohen-d is 0.472299, indicating a moderate effect.

The power consumption impact is thus very limited, which shows that OASIS is a lightweight IDS that can be deployed with minimal impact on power consumption.

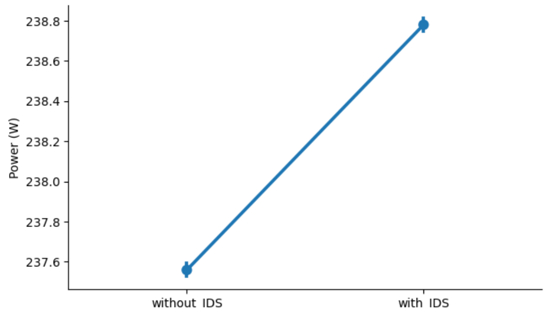


Figure 8: Mean power measurements with and without OASIS, with associated 95% confidence intervals.

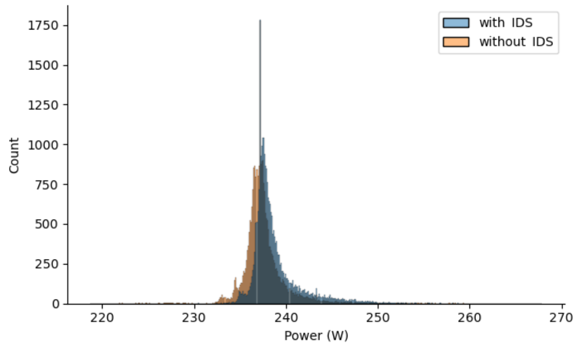


Figure 9: Power consumption distributions with and without OASIS.

The mean power consumption in the group running the IDS was $238.78W$ with a standard deviation of 2.71 %, whereas the mean power consumption in the group not running the IDS was $237.56W$ with a standard deviation of 2.45 %.

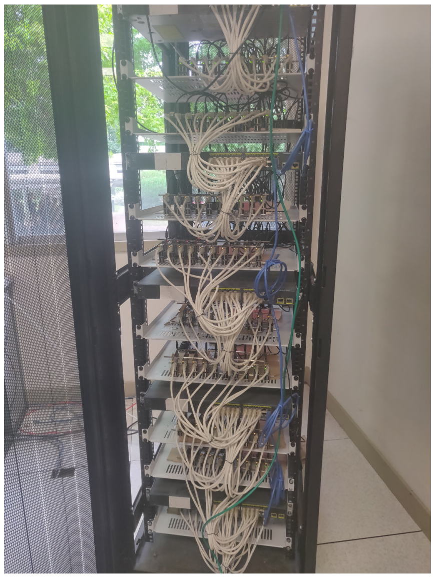


Figure 10: Experimental setup for large-scale performance evaluation, based on a bay of 100 Raspberry Pi.

C BLUESHIELD EXPERIMENT DETAILS

We reproduced the BlueShield experimental setup described in [36], using code provided in the associated GitHub repository [35]. Unfortunately, BlueShield relies on now deprecated software (e.g., Python 2.7) and hardware (e.g., the Ubertooth One has been recently discontinued by its manufacturer, Great Scott Gadgets [15]) running custom firmware. We also had to write minor fixes in C code to be able to compile both the custom firmware for Ubertooth One and the associated Host libraries and tools. Since the code of BlueShield relies on outdated Python 2.7 libraries, we had to run some software components in Docker containers to reproduce a functional software environment. We used two Raspberry Pi 4 and one Raspberry Pi 3B+, connected to three Ubertooth One running the custom BlueShield firmware, as collectors. We connected them to a Local Area Network (LAN) with a computer acting as monitor. An overview of collectors and one of the profiled devices can be found in Figure 11.

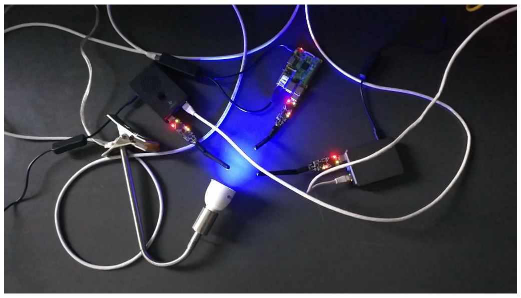


Figure 11: BlueShield collectors and profiled BLE Lightbulb

We encountered multiple deadlocks and exceptions during the profiling phase, which were linked to Ubertooth reception issues during the interval and RSSI estimation step. After several attempts with different probe locations and minor bugfixes, we successfully ran BlueShield profiling on two static BLE devices (a TI-based lightbulb and an nRF51 Development Board running *ble_app_blinky* example from nRF5 SDK 11.0). However, the monitoring phase did not raise any alert in our environment, despite the fact that we ran multiple successful spoofing attacks targeting the profiled devices from various BLE transceivers (including BLE dongles used during Blueshield evaluation, such as CYW20735 or CSR 4.0). We noticed a significant impact of the probe location on the exhaustivity and integrity of BLE traffic captured by Ubertooth One, which may cause packet losses impacting the profile accuracy or the detection.

D REAL-LIFE EXPERIMENT RESULTS

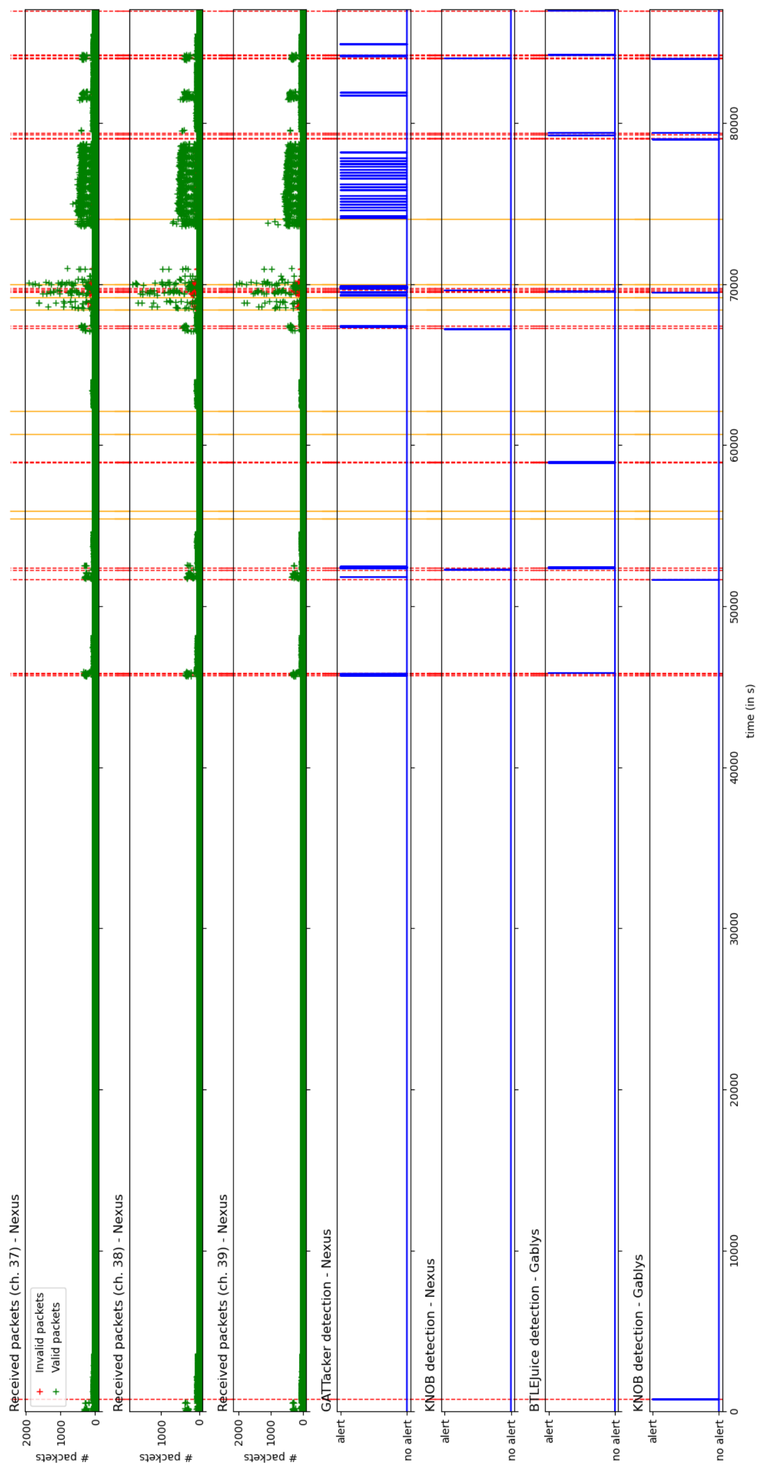


Figure 12: Results of real-life experiment. Red dotted lines indicate the occurrence of attacks, Orange lines indicate environment change.