FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware

Haohuang Wen
wen.423@osu.edu
The Ohio State University

Zhiqiang Lin
zlin@cse.ohio-state.edu
The Ohio State University

Yinqian Zhang
yinqian@cse.ohio-state.edu
The Ohio State University

ABSTRACT

Today, Bluetooth 4.0, also known as Bluetooth Low Energy (BLE), has been widely used in many IoT devices (*e.g.*, smart locks, smart sensors, and wearables). However, BLE devices could contain a number of vulnerabilities at the BLE link layer during broadcasting, pairing, and message transmission. To detect these vulnerabilities directly from the bare-metal firmware, we present FirmXRay, the first static binary analysis tool with a set of enabling techniques including a novel base address identification algorithm for robust firmware disassembling, precise data structure recognition, and configuration value resolution. As a proof-of-concept, we focus on the BLE firmware from two leading SoC vendors (*i.e.*, Nordic and Texas Instruments), and implement a prototype of FirmXRay atop Ghidra. We have evaluated FirmXRay with 793 unique firmware (corresponding to 538 unique devices) collected using a mobile app based approach, and our experiment results show that 98.1% of the devices have configured random static MAC addresses, 71.5% *Just Works* pairing, and 98.5% insecure key exchanges. With these vulnerabilities, we demonstrate identity tracking, spoofing, and eavesdropping attacks on real-world BLE devices.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; **Embedded systems security**; **Mobile and wireless security**.

KEYWORDS

Firmware analysis, Bluetooth Low Energy, Embedded system security, Vulnerability discovery

ACM Reference Format:
Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372297.3423344

1 INTRODUCTION

Over the past several years, we have witnessed a rapid growth of the Internet-of-Things (IoT), thanks to a variety of enabling technologies from sensors, micro-controllers, actuators, to mobile and cloud computing. Among the deployed IoTs, the BLE-enabled ones are ubiquitous and have been widely used in many applications (*e.g.*, health care, retail, asset tracking [19], and recently contact tracing [59]). The key reason for its success is its low technical barrier from both hardware and software. Today, there are many System on Chip (SoC) vendors such as Nordic [11] and Texas Instruments (TI) [15], which provide both hardware chips and software development kits (SDKs) for IoT developers. There are also numerous software platforms (*e.g.*, Android), frameworks (*e.g.*, Google Home), and clouds (*e.g.*, AWS) that enable application programmers to easily assemble hardware gadgets with software components. Therefore, such a low technical barrier has attracted a huge number of developers, and together they have produced billions of BLE-IoT devices.

However, a secure BLE device needs proper hardware capability (*e.g.*, I/O), and also correct configuration for its broadcasting, pairing, and message encryption. Otherwise, it could lead to a number of vulnerabilities at the BLE link layer. For instance, a BLE device can be vulnerable to identity tracking [27] and device fingerprinting [63] [20] if developers configure MAC addresses and universally unique identifiers (UUIDs) statically for broadcasting. Meanwhile, a BLE device can be vulnerable to active man-in-the-middle (MITM) attacks (*e.g.*, spoofing) if it is configured to only support *Just Works* pairing [41] [49]. In addition, passive MITM attacks (*e.g.*, eavesdropping) are also possible if it fails to enforce the Low Energy Secure Connections pairing [36] to secure the key exchange [44] [32].

While it is important for a BLE device to be secure against these attacks, it is in fact hard to do so for several reasons. First, the configurations are complicated. For instance, to use secure pairing methods (*e.g.*, *passkey entry* and *OOB* [16]) instead of *Just Works*, developers have to clearly specify the MITM protection requirement and also the device I/O capability in the pairing feature packets. Second, many security features also rely on capabilities provided by device hardware. For example, to configure *passkey entry* pairing, the device must have a keyboard or a touchable screen to let users manually enter a passkey to authenticate the pairing device. Third, some extra implementations are required. For example, to configure periodically randomized MAC addresses, developers also need to implement the exchange of Identity Resolving Key (IRK) [16].

Therefore, it is imperative to identify the aforementioned vulnerabilities in BLE devices. There could be multiple approaches to do so, such as packet analysis with real devices, or using companion mobile apps. However, these approaches are either not scalable or have only limited view. Fortunately, we notice that these vulnerabilities can be directly identified from the low-level configurations in the corresponding bare-metal firmware (*i.e.*, firmware without OS support, which is particularly popular for BLE due to its extremely low energy requirement). While there is a large body of research in firmware analysis for vulnerability discovery such as firmware

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3423344

emulation [56] [23], fuzzing [47] [25] [60], rehosting [26] [31], and static analysis [35] [33], nearly all of them focus on non-bare-metal firmware (*e.g.*, devices such as IoT routers and cameras [25] with Linux kernels). Additionally, none of them systematically investigates the vulnerabilities in bare-metal BLE-IoT devices.

To advance the state-of-the-art, we present FirmXRay, the first static analysis tool to detect BLE link layer vulnerabilities from configurations in the bare-metal firmware at scale. Specifically, we have developed three techniques in FirmXRay. The first is *Robust Firmware Disassembling*, which uses absolute pointers to model base address constraints and infers the base address to disassemble the firmware. The second is *Precise Data Structure Recognition*, which leverages the static SDK function signatures to identify the configurations from function parameters. The third is *Configuration Value Resolution* to extract the configuration generation path and resolve the configuration values. We have implemented FirmXRay atop Ghidra [8], and target the firmware built with the SDKs from Nordic or TI, the two leading global BLE SoC vendors [46].

To evaluate FirmXRay, we have to collect bare-metal firmware at scale, which is challenging since IoT vendors seldom release the device firmware publicly, and also there is no centralized platform to collect them. Interestingly, we notice that bare-metal firmware typically do not directly connect to the Internet through cellular network or Wi-Fi, and thus they must rely on relays (*e.g.*, mobile apps) to transfer update packets wirelessly. Therefore, we design a scalable mobile app based approach to collect the bare-metal firmware. With this approach, we successfully downloaded 793 unique firmware corresponding to 538 unique devices.

Among these devices, FirmXRay discovered that 71.5% of them adopt *Just Works* pairing that provides no protection against active MITM attacks such as active eavesdropping and spoofing. In addition, nearly all of them have configured random static MAC addresses and insecure key exchanges, which allows tracking and eavesdropping attacks that can leak user's personal identity and private data. Our results show that there is a wide spread of vulnerabilities across various bare-metal BLE-IoT devices. To show the security implications of the identified vulnerabilities, we demonstrate three types of concrete attacks on 5 real-world BLE devices.

Contributions. Our paper makes the following contributions:

- We design the first automated static analysis tool FirmXRay to detect BLE link layer vulnerabilities from the configurations of bare-metal firmware with a novel algorithm to recognize the base address, and then identify and resolve the configurations.
- We propose a mobile-app-based scalable approach to efficiently collect bare-metal firmware images from only mobile apps, resulting in 793 unique ones corresponding to 538 unique devices.
- We implement FirmXRay atop Ghidra, and evaluate it with 793 unique firmware, in which our tool discovered that 71.5% of the devices use *Just Works* pairing, and nearly all of them have configured random static MAC addresses and insecure key exchanges.

2 BACKGROUND

2.1 Bare-metal Firmware

Bare-metal firmware is ubiquitous among various IoT embedded devices such as smart sensors, smart toys, smart locks, and smart lights, because of its low energy consumption and also the trade off between price and performance. Since it directly runs on a logic hardware without any operating systems, fundamentally it is barely a binary blob that only contains the program code to manage the device functionality using an infinite loop and interacts with other software components through interrupts.

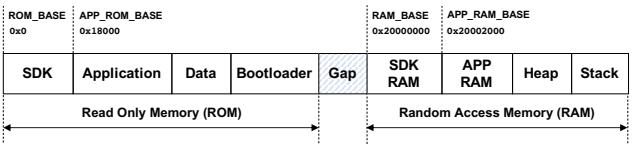


Figure 1: Memory layout of bare-metal IoT devices.

Nowadays, many manufactures, such as Nordic [11], TI [15], and Dialog [6], have developed various Micro Controller Units (MCUs), which are small and self-contained computers on micro chips to support the bare-metal firmware. Meanwhile, they often adopt low-energy technologies such as BLE, and low-end processors such as ARM Cortex-M0. Moreover, to facilitate the development of an embedded device, these manufactures also provide software development kits (SDKs) that have integrated a number of basic functionalities. Typical examples are SoftDevice [12] from Nordic and BLE-Stack [2] from TI, which enable developers to implement specific device logic such as BLE pairing and data exchange, atop the programming interfaces provided in the SDK.

Memory layout of bare-metal firmware. The memory layout of a typical bare-metal IoT device is presented in Figure 1 [10]. At a high level, the layout consists of two main regions: (*i*) read only memory (ROM) containing program code and persistent data, and (*ii*) random access memory (RAM) holding run-time variables. The ROM is located at the lower address space (*e.g.*, `0x0`) whereas the RAM is at higher address space (*e.g.*, `0x20000000`), and there is a gap between these two memory regions. On the ROM side, there are multiple isolated sections including the *SDK* provided code for the precompiled vendor-specific functions, *application* code for device logic, and *bootloader* for boot logic. On the RAM side, there are multiple RAM sections correspondingly for the *application* and *SDK* to store static variables, as well as the *stack* and *heap* to store local and dynamically allocated variables. For each section in the ROM and RAM, it starts from an absolute base address, such as `APP_ROM_BASE` for *application*, which can be customized before compilation.

Over-the-air upgrade of the firmware. While the *bootloader* and the *SDK* are preloaded into the device memory and seldom get changed overtime, there is a need for developers to upgrade the *application* with new patches (*e.g.*, when fixing vulnerabilities or bugs). Since the *application* code does not rely on OS and is isolated from other sections in the ROM, it is usually small in size (less than one megabyte according to our observation), which thus allows the upgrade procedure to directly replace the old *application* with a new one. Additionally, since bare-metal devices often do not have direct Internet access (*e.g.*, cellular network or Wi-Fi), they rely on other entities (*e.g.*, smartphones) to serve as intermediate relays to download the upgraded firmware from remote servers, and then transfer the firmware to the devices. Such an upgrade process is called *over-the-air (OTA) upgrade* because the transfer is through wireless network such as Bluetooth. After receiving the upgraded

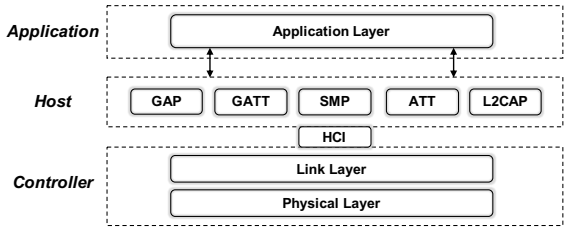


Figure 2: Bluetooth Low Energy protocol stack.

firmware, the device reboots and the bootloader replaces the old firmware with the latest one.

2.2 Bluetooth Low Energy

BLE protocol stack. The architecture of the BLE protocol stack [16] is shown in Figure 2. At a high level, it is divided into three components: application, host, and controller. At the bottom of the stack, the link layer directly interacts with the physical layer, and is responsible for basic functions including advertising, connection, and encryption. Meanwhile, the host communicates with the link layer through the Host Controller Interface (HCI) and defines secure device communication protocols such as Generic Attribute Profile (GAP). At the top of the stack, the application layer leverages the abstractions from the host to implement specific application logic.

BLE workflow. The general workflow of Bluetooth Low Energy is presented in Figure 3, which illustrates how a central device (*e.g.*, a smartphone) pairs with a peripheral device (*e.g.*, a BLE smart band), and exchanges data. At a high level, the workflow is broken down into eight steps across three main stages: (I) Broadcast and connection, (II) Pairing and bonding, and (III) Data Transmission. The details of each stage are described as follows.

(I) Broadcast and connection. In this stage, the smartphone recognizes the broadcasting smart band and establishes a connection with it. Initially, in order to indicate the willingness of connection, the smart band needs to broadcast data packets to all nearby devices, which include identifiable information such as Media Access Control (MAC) address and universally unique identifiers (UUIDs). A device that broadcasts information and waits for connection (❶) is regarded as a *peripheral*, while the one scans the advertised BLE packets (❷) from the peripherals and initiates the connection is called a *central*. After the central initiates the connection request (❸) to the peripheral, the connection is successfully established (❹).

(II) Pairing and bonding. The channel between the central and peripheral often needs to be encrypted, and thus the pairing process is for them to negotiate the cryptographic key. While broadcast and connection is a mandatory stage for all BLE communications, the pairing and bonding stage is optional. If none of the device requests for pairing, the transferred data will be in plain text. Specifically, the pairing process consists of the following three steps:

- **Pairing feature exchange (❺)**. At first, the two devices exchange their pairing features so that an appropriate pairing method (*e.g.*, *passkey entry*) can be negotiated. The exchanged features include their I/O capabilities, MITM requirement, BLE version, *etc.* If MITM protection is needed and certain I/O requirements (*e.g.*, having a keyboard or display) are satisfied, they will select a secure pairing method including *passkey entry*, *numeric comparison*, and *Out Of Band* (*OOB*). Otherwise, they have to use *Just Works*, which has the weakest security protection.
- **LTK/STK generation (❻)**. After the pairing method is decided, the two devices then negotiate the encryption key. This step performs differently according to specific BLE versions. When two devices are below BLE 4.2, they use BLE Legacy Pairing to generate a temporary short term key (STK) to encrypt the long term key (LTK), in which the STK is generated based on the selected pairing method (*e.g.*, requiring a user to manually enter a 6-digit passkey) [16]. If the two devices support at least BLE 4.2, the LE Secure Connection (LESC) pairing can be used. Based on the Elliptic-Curve Diffie–Hellman (ECDH) protocol, each of them generates a public-private key pair and only exchanges the public key, and then an LTK is directly calculated on both sides to encrypt the session. Note that the selected pairing method is used to authenticate the pairing process (*e.g.*, asking the user to enter a password). If bonding is specified, the negotiated key is stored in non-volatile memory for future communications.
- **Transport specific key distribution (❼)**. After the STK or LTK has been generated, the transport specific keys are distributed from one entity to the other. The distributed keys include the LTK (in Legacy pairing), Identity Resolution Key (IRK), Connection Signature Resolving Key (CSRK), and so on.

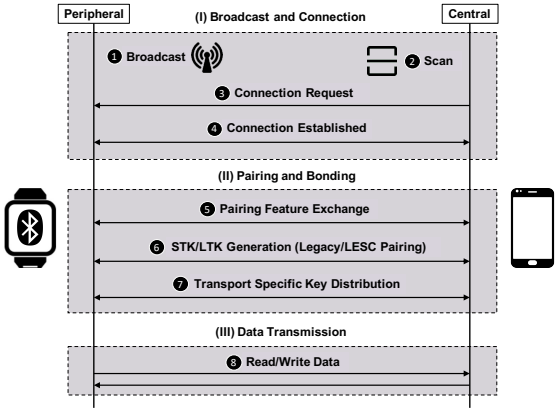


Figure 3: Bluetooth Low Energy workflow.

(III) Data Transmission. When the first two stages are completed, the central and the peripheral start to communicate with each other (❽). The communication is through reading or writing data on certain BLE attribute called characteristic. To be more specific, the BLE stack maintains a set of hierarchical attributes including services, characteristics, and descriptors [7], which are identified by UUIDs.

3 OVERVIEW

3.1 Threat Model, Scope, and Assumptions

Threat model. In this paper, we consider that nearby attackers can compromise the devices by leveraging the vulnerabilities at the BLE link layer. These attackers are capable of sniffing BLE packets during broadcast and data transmission, and also performing MITM attacks. These attacks can be launched by using a programmable

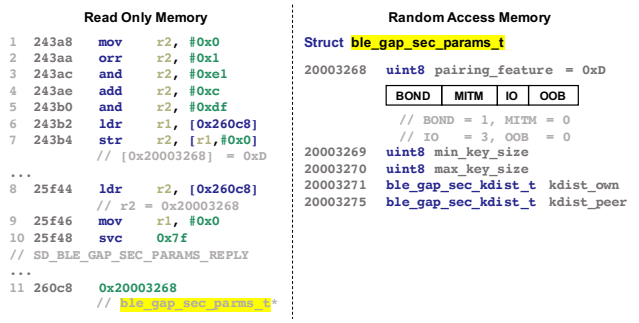


Figure 4: An example of a *Just Works* pairing vulnerability.

BLE development board such as a Nordic nRF52-DK [13] to build a Bluetooth sniffer and MITM proxy.

Scope. While there are many attacks against BLE (*e.g.*, [30, 34, 44, 48, 63]), we particularly focus on those caused by the vulnerabilities at the BLE link layer, which is responsible for broadcast, pairing, and encryption. To summarize, there are three types of such attacks: (*i*) identity tracking, (*ii*) active MITM, and (*iii*) passive MITM.

(*i*) **Identity tracking**. This attack enables an attacker to keep track of a victim's identity based on the advertised information such as MAC address from a BLE peripheral. While MAC address is mandatory in each BLE packet, which makes identity tracking possible [27], it can be configured to be a static address (*e.g.*, public IEEE address [40]), or a randomly generated address which keeps changing periodically (*e.g.*, every 15 minutes [40]). Therefore, in this case, how resilient the device against identity tracking depends on the device configuration.

(*ii*) **Active MITM**. An active MITM attack allows intercepting (*e.g.*, active eavesdropping) and modifying messages (*e.g.*, spoofing). In BLE, such a vulnerability can also be identified from configurations. Specifically, among the four types of pairing, *numeric comparison* and *passkey entry* are able to prevent active MITM attacks since they rely on a third-party entity (*e.g.*, a human being) to authenticate the connection with a dedicated I/O (*e.g.*, by manually entering a passkey on the screen). Meanwhile, *OOB* can mitigate this attack by narrowing down to an extremely short connection distance. However, if the firmware fails to properly configure with MITM protection, or lacks certain I/O capabilities (which will still be reflected in the configurations), it has to use *Just Works* pairing, which provides no protection against active MITM attacks.

(*iii*) **Passive MITM**. Passive MITM attack allows an attacker to read messages, such as passive eavesdropping. As in BLE, although the communication traffic is encrypted after pairing, the LTK can still be eavesdropped since the two devices have to first establish a temporary encryption key negotiated in plain-text when no public key cryptography is used. To mitigate this vulnerability, Bluetooth Special Interest Group (SIG) [3] has adopted the Elliptic-Curve Diffie–Hellman (ECDH) protocol for key exchange since BLE 4.2 [16], which is known as the Low Energy Secure Connection (LESC) pairing. However, such a protection also relies on user configuration, because both the central and peripheral must explicitly request for LESC pairing, and meanwhile have to invoke the ECDH key exchange [36].

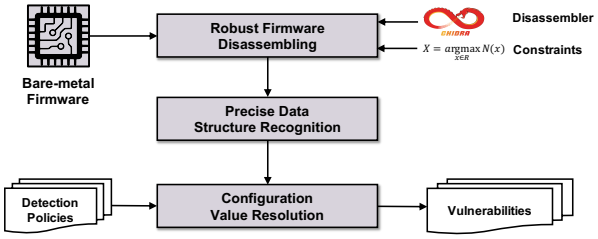


Figure 5: Overview of FirmXRay.

Assumptions. We focus on the bare-metal firmware developed based on the Nordic or TI SDKs, which are all ARM Cortex-M architecture. We also assume that they are not obfuscated and no address space layout randomization (ASLR) is deployed, and the firmware are distributed via the relay of mobile apps (to achieve the OTA upgrade).

3.2 Motivating Example

To clearly illustrate how the link layer vulnerabilities can be identified from configurations, and the corresponding technical challenges, we present a motivating example in Figure 4. The example comes from an IoT wristband firmware developed based on the Nordic SDK. In particular, starting from line 1, the firmware loads a value `0x0` into register `r2`. Going through a series of operations including logical or (`orr`), logical and (`and`), and arithmetic add (`add`) (line 2-5), the value of `r2` becomes `0xD`. Next, this value is stored into a specific location `0x20003268` (line 6-7), which refers to a static data structure `ble_gap_sec_params_t` in RAM. Afterwards, an address `0x20003268` is loaded into `r2` (line 8), which essentially makes `r2` a pointer pointing to that data structure. Finally, a supervisor call is invoked by an `svc` instruction [9] along with an svc number `0x7f` (line 10). After the `svc` is called, the SDK function `SD_BLE_GAP_SEC_PARAMS_REPLY` is invoked, taking `r0`, `r1`, and `r2` as parameters to reply the peer device with its pairing features.

The configuration `pairing_feature` is a `uint8` integer located at the starting address of the structure, where the pairing features are represented by different bits of the integer. More specifically, the first bit specifies whether bonding is performed, the second indicates whether MITM protection is necessary, and the third to the fifth bits represent the specific I/O capabilities, according to the SDK specification [12]. As a result, the value `0xD` can be interpreted as a pairing configuration: the device requires bonding, no MITM protection, and does not have I/O capabilities. Therefore, we can conclude that the wristband contains a vulnerability that uses *Just Works* to pair with a smartphone.

3.3 FirmXRay Overview

Based on the above motivating example, we can notice that in order to identify the vulnerabilities, we must first correctly disassemble the firmware to recognize the instructions and parameters, then identify the configuration data structures, and finally compute the configuration values. More specifically, we need:

- **Robust Firmware Disassembling.** While the firmware disassembling in ARM (RISC) is relatively easy than x86 (CISC), we still need to recognize the base address for the disassembling since the firmware code we acquire start from customized bases.

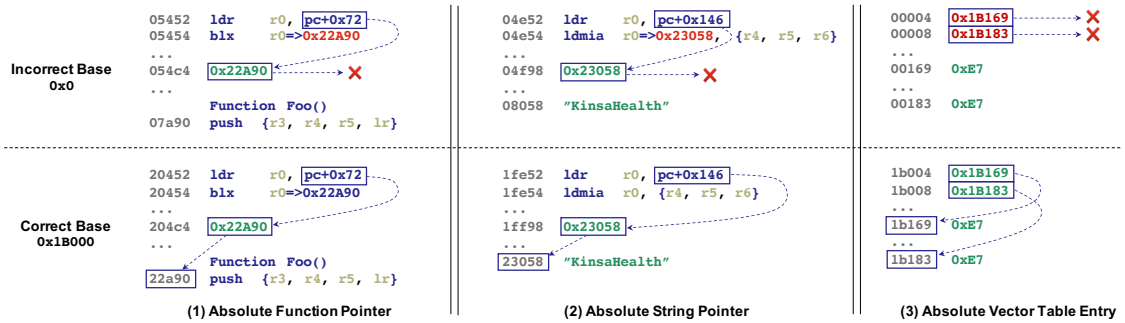


Figure 6: Effect of disassembling with different base addresses across absolute pointers.

- **Precise Data Structure Recognition.** After disassembling, FirmXRay has to identify the configurations from the disassembled code. However, as shown in Figure 4, configurations are often embedded in complicated data structures, and the names of variables and functions, and their types, etc., of bare-metal firmware are completely stripped.
- **Configuration Value Resolution.** As indicated in Figure 4, the configurations are not directly hardcoded in the program, but instead are generated through complicated computations such as logical, arithmetic, and bit-wise operations. Hence, it is necessary to design an algorithm to resolve the configuration values.

As such, we have designed three corresponding techniques, as shown in Figure 5. FirmXRay first takes a bare-metal firmware as input, and recognizes the base address using *Robust Firmware Disassembling* (§4.1). Next, based on the disassembled firmware, it identifies the configuration data structures using *Precise Data Structure Recognition* (§4.2). Finally, with the identified configurations, FirmXRay resolves the concrete configuration values using *Configuration Value Resolution* and identifies the vulnerabilities with the corresponding detection policies (§4.3).

4 DETAILED DESIGN

4.1 Robust Firmware Disassembling

Observations. When given a firmware image, FirmXRay first has to recognize the base address for robust firmware disassembling. To clearly illustrate the challenges, we present three simplified real-world examples in Figure 6. We can notice that if the firmware is correctly rebased, as shown in the bottom half of Figure 6(1), the corresponding instructions such as `blx` would successfully recognize the target function `Foo` through its absolute function pointer address at `0x22A90` pointed by a pointer at `0x204C4`; otherwise, this absolute address falls beyond the firmware address space as shown in the top half of Figure 6(1). Similarly, the absolute string pointers in Figure 6(2) and the vector table entries in Figure 6(3) would also point to wrong locations if their target addresses are not properly resolved.

As shown in the above three cases, if the firmware starts from an incorrect base, the *absolute pointers* (*e.g.*, the above pointers using absolute addresses) would be dereferenced at wrong locations, which causes incorrect disassembly. The root cause is that the addresses of their targets (*e.g.*, function entries, strings, interrupt numbers) shift along with the base address, while the absolute pointer values remain unchanged. For example, the address of the target function `Foo` shifts from `0x22A90` to `0x7A90` when the base changes from `0x1B000` to `0x0`, while the absolute pointer address remains `0x22A90` regardless of the base. Therefore, we must recognize the correct base that properly links these pointers to their right targets.

Although there exist a handful of efforts (*i.e.*, [61] [50]) in base address recognition, these approaches rely on a single type of clues (namely the function prologues), which can lead to incorrect results when there is insufficient number of such clues in the firmware, as shown in our experiment (detailed in §5.2.1). As a result, we propose a more systematic approach based on the observation that the absolute pointers must point to certain instructions or variables with respect to their types, and such *point-to relations of absolute pointers* can provide strong clues to infer the base address. For instance, as illustrated in Figure 6(1), a function pointer must point to a valid function entry. If the firmware starts from an incorrect base, this function pointer will point to a wrong location. Therefore, only the correct base address can link an absolute pointer (*e.g.*, a function pointer) with the intended target (*e.g.*, a function entry). Based on this observation, we can model the base address recognition as a point-to constraint solving problem of absolute pointers.

Our Approach. Consequently, we propose a two step approach to recognize the base address. In particular, the first step is to extract all absolute pointers from the firmware, and the second step is to associate constraints between the absolute pointers and their intended targets, and finally solve the constraints, from which to infer the base address. The details of these two steps are described as follows.

Step-I: Absolute pointer recognition. Without the knowledge of the base address, FirmXRay first loads the firmware with a `0x0` base address and disassembles the program instructions. The reason for why a zero base works is that the ARM instructions are always aligned with 2 or 4 bytes [9]. For disassembling, we apply a linear sweep algorithm [45] to exhaustively disassemble all possible instructions. To identify the absolute pointers, we can particularly focus on all of the load instructions (*i.e.*, `ldr` in ARM), since they must be loaded into registers before being dereferenced. However, not all the absolute pointers in the load instructions are useful as many of them point to the RAM locations to deference run-time values, which are not visible statically. Therefore, we must look for absolute pointers that reference the static code or data. Fortunately, as illustrated in Figure 6, there are three types of absolute pointers that fall into this category: (*i*) absolute function pointers, (*ii*) absolute string pointers, and (*iii*) vector table entries. These pointers can also be easily distinguished with the pointers that deference

run-time values which are located at higher address space as shown in Figure 1. In the following, we describe how we identify these three types of pointers.

(*i*) **Absolute function pointer**. After being loaded via `ldr`, an absolute function pointer will be dereferenced and go through the `blx` instruction for function invocation, as shown in Figure 6(1). As a result, FirmXRay identifies function pointers by checking whether they are eventually taken by a `blx` or `bx` instruction. We use P_F to denote the set of absolute function pointers.

(*ii*) **Absolute string pointer**. Unlike absolute function pointers that can be easily identified, it is actually hard to recognize absolute string pointers because there is no instruction that takes an explicit string as operand. We therefore have to rely on other clues to identify them. One clue is the SDK functions that take strings as parameters. By recognizing these functions, we are able to identify the ones that use absolute string pointers. We use P_S to denote the set of absolute string pointers.

(*iii*) **Vector table entry**. We also identified a special type of bare-metal unique absolute pointers, which reside in a vector table of interrupt handlers. The entries in the vector table point to the locations that store the specific interrupt numbers (*e.g.*, `0xE7`). Since this vector table is located at `APP_ROM_BASE`, FirmXRay scans the firmware from the base address to identify this vector table, which has a strong signature (*i.e.*, an array of absolute addresses). We use P_V to denote the set of vector table entries.

Additionally, FirmXRay also searches for necessary gadgets to build up the constraints, including function entries, strings, and interrupt numbers. The function entries are recognized through the function prologues which are usually the instructions to push register values onto the stack (*e.g.*, `PUSH`, `STMFD`). Meanwhile, FirmXRay recognizes the possible readable strings according to the printable ASCII values and their ending null bytes. Finally, it recognizes the interrupt numbers based on the manufacture-reserved values.

Step-II. Constrained Base address modeling and solving. Having identified all absolute pointers and their possible targets, we need to resolve the firmware base address based on the *point-to constraints of absolute pointers* identified in P_F, P_S, and P_V. It might appear that we can resolve the firmware base address by using just a single pointer in P_F, or P_S, or P_V. For example, as illustrated in the top half of Figure 6(1), by subtracting the absolute pointer value (`0x22A90`) with the address of its intended target (`0x7A90`), the base address is resolved as `0x1B000`. However, it is actually hard to resolve the base address by solely relying on just one (or a few) absolute pointer. Back to our example, if we link the absolute function pointer (*e.g.*, `0x22A90`) to another valid function entry (*e.g.*, `0x8A90`), we can resolve a different base address (*e.g.*, `0x1A000`) that satisfies the point-to constraint as well. Therefore, we must combine all the absolute pointers we identified to resolve the base address.

With these pointers, by looking at each individual one, we may obtain multiple candidate base addresses, but there must be one optimal base address that has the maximum number of matches of the identified point-to constraints. For instance, the base address `0x1B000` satisfies the four constraints illustrated in Figure 6. In general, assume there are N absolute pointers, there will be N constraints. Ideally, there exists one optimal base address that satisfies all N constraints. However, this cannot be always true, since many constraints cannot be resolved. For example, there exist a few function pointers that do not point to typical function prologues (*e.g.*, `push`) but instead point to code snippets that start from various instructions (*e.g.*, `ldr`). Therefore, the optimal base address should be the one that satisfies the most number of constraints. We thus define a target function

$$N(x) = N_F(x) + N_S(x) + N_V(x) \tag{1}$$

to measure how many constraints a base address x can satisfy, where $N_F(x)$, $N_S(x)$, $N_V(x)$ denote the number of satisfied constraints in P_F, P_S, and P_V, respectively. With this target function, we can traverse the address space R to find the optimal base address X with the maximum function value, which can be formulated as

$$X = \arg\max_{x \in R} N(x) \tag{2}$$

Intuitively, we can start from `ROM_BASE` and iterate through the ROM to try all possible bases. However, we find that the search space R can be optimized with a restricted boundary. Specifically, we use the absolute addresses to infer the upper bound, which is the smallest absolute pointer address. Therefore, we only need to search R in the following range

$$R = \{x \mid 0 < x < A_{min}\} \tag{3}$$

where A_{min} denotes the minimum absolute pointer address (*e.g.*, `0x1B169`) among all the identified absolute pointers (*e.g.*, `0x22A90`, `0x23058`, `0x1B169`, `0x1B183`). To search for X, we design a simple probe-and-test algorithm. Starting from the lower bound of R, FirmXRay iterates each candidate x in R and calculates $N(x)$. Note that we only need to probe those x with even values, since ARM instructions are aligned with 2 or 4 bytes. To this end, we define

$$d(x, p) = p - x \tag{4}$$

where $p \in P_F \cup P_S \cup P_V$ and $d(x, p)$ denotes the concrete target memory address pointed by p with the given base address x. Then, for each potential x and each absolute pointer p, FirmXRay performs the following three checks:

(i) If $p \in P_F$, FirmXRay checks if $d(x, p)$ is a valid function entry. If so, it increases $N_F(x)$ by 1.

(ii) If $p \in P_S$, FirmXRay checks if $d(x, p)$ is a valid string. If so, it increases $N_S(x)$ by 1.

(iii) If $p \in P_V$, FirmXRay examines whether $d(x, p)$ is an interrupt number. If so, $N_V(x)$ is added by 1.

After all of the candidate x have been probed and tested, FirmXRay selects the x with the maximum $N(x)$ value as the optimal base address, and this x satisfies the most number of constraints.

4.2 Precise Data Structure Recognition

Given the disassembled firmware code, FirmXRay needs to recognize the configuration data structures. While there are many techniques for reverse engineering data structures from stripped binaries, they cannot be easily applied to our problem. For instance, dynamic approaches such as Rewards [39] and Howard [51], are not suitable for bare-metal firmware because they require vendor-specific execution contexts such as hardware inputs for execution. While TIE [38] does not require to run the firmware, it still falls

short because it attempts to recover all data structures using type inference, while we only focus on those that must be taken as static SDK function parameters.

As a result, we develop our own customized static analysis. Our key insight is that no matter where these data structures are defined in the memory, they will finally be taken as parameters by the SDK functions, because the firmware needs to invoke these precompiled functions to configure the device hardware. For example, in Figure 4, the structure pointer is taken as a parameter (stored in a register `r2`) by function `SD_BLE_GAP_SEC_PARAMS_REPLY` to set up the pairing feature. As such, FirmXRay first identifies the SDK functions, and further recognizes the configurations through function parameters.

To identify the SDK functions, FirmXRay requires vendor-specific knowledge to establish signatures of the function invocation points, and these knowledge were gathered manually from the SDK specifications prior to the analysis. In particular, the Nordic and TI SDKs use special mechanisms to invoke these functions. Nordic uses supervisor calls (*i.e.*, `svc`) [14] where each function is associated with a corresponding `svc` number (*e.g.*, `0x7F` for `SD_BLE_GAP_SEC_PARAMS_REPLY` as shown in Figure 4). TI uses ICall [5] to invoke SDK functions, and each function is dispatched by the ICall interface with a specific command, which allows us to identify them precisely. Based on these knowledge, FirmXRay scans through the disassembled code to recognize these SDK functions, and identifies the configuration data structures from their parameters.

4.3 Configuration Value Resolution

Having identified the configuration data structures, we design the following three-step analysis to resolve the configuration values:

Step-I. Configuration path extraction. In this step, our goal is to extract the program path with the instructions involved in the configuration value generation, and we adopt a backward program slicing [54] algorithm. At a high level, FirmXRay starts from the SDK function invocation points identified in §4.2 and backward traverses the program control flow graph G to record all instructions (*e.g.*, the `orr`, `and`, and `add` instructions in Figure 4) that are necessary for computing the configuration value. At first, the algorithm takes the following inputs: a function invocation point A (*e.g.*, an `svc` instruction), the current function block B, a dependent variable (*e.g.*, registers and memory locations) set D, and the current configuration path s. It backward iterates the instructions in B starting from A. For each instruction i in B, if it modifies the value of any target variables in D, the algorithm adds all other variables in i to D, removes the target variables, and records the instruction i in s. In particular, to determine whether i should be involved as part of the program path, the algorithm focuses on three types of dependencies, including register to register, register to memory, and memory to register. Note that unlike x86 instructions, ARM does not have dependencies between memories. As such, FirmXRay focuses on two types of data dependencies:

- **Register to Register dependence**. It is quite common when the value of a register depends on another register. For instance, in instruction `add r1 r2 r3`, the value of `r1` depends on `r2` and `r3`. Therefore, if `r1` is in D, we append this instruction to s, and add `r2` and `r3` to D as the new dependent variables.

Policy	SDK Function Name	Reg. Index	Description
(i)	`SD_BLE_GAP_ADDR_SET`	0	Configure the MAC address
	`SD_BLE_GAP_APPEARANCE_SET`	0	Set device description
	`SD_BLE_GATTS_SERVICE_ADD`	0, 1	Add a BLE GATT service
	`SD_BLE_GATTS_CHARACTERISTIC_ADD`	2	Add a BLE GATT characteristic
	`SD_BLE_UUID_VS_ADD`	0	Specify the UUID base
	`GAP_ConfigDeviceAddr*`	0	Setup the address type
	`GATTServApp_RegisterService*`	0	Register BLE GATT service
(ii)	`SD_BLE_GAP_SEC_PARAMS_REPLY`	2	Reply peripheral pairing features
	`SD_BLE_GAP_AUTH`	1	Reply central pairing features
	`SD_BLE_GAP_AUTH_KEY_REPLY`	1, 2	Reply with an authentication key
	`SD_BLE_GATTS_CHARACTERISTIC_ADD`	2	Add a BLE GATT characteristic
	`GAPBondMgr_SetParameter*`	2	Setup pairing parameters
	`GATTServApp_RegisterService*`	0	Register BLE GATT service
(iii)	`SD_BLE_GAP_LESC_DHKEY_REPLY`	0	Reply with a DH key
	`GAPBondMgr_SetParameter*`	2	Setup pairing parameters

Table 1: Targeted SDK functions in our detection policies (Note: functions w/ * are for TI, and otherwise for Nordic).

- **Memory to Register (or vice versa) dependence**. This is when the register value depends on certain memory location. For instance, in `ldr r2 [0x260c8]`, `r2` loads the value by dereferencing memory location `0x260c8`. Therefore, if `r2` is in D, FirmXRay records this instruction to s, and also adds `0x260c8` to D.

Finally, when a fixed point is reached where the slicer state remains unchanged (*e.g.*, D becomes empty), the algorithm adds s to S and returns. Otherwise, the algorithm continues to jump to previous blocks by first setting up a new context (*i.e.*, creating new copies of D and s) for each path, and then recursively invoking itself in each of the previous block b in G. Ultimately, the algorithm produces S containing a set of configuration value generation paths that eventually generate the target values. Note that to prevent branch explosion, we have limited the length of the configuration path of our static analysis with a threshold approach as in other works (*e.g.*, [55] [42]).

Step-II. Configuration value generation. Based on the extracted configuration value generation paths, FirmXRay then statically executes each instruction in a forward order to generate the concrete values of our targets. Specifically, FirmXRay first creates an execution context for each path, including the registers (*e.g.*, `r1`-`r12`, `sp`) and memory (*e.g.*, RAM and stack). Next, it forward executes each instruction in order and modifies the context accordingly (*e.g.*, updating values in registers and memory) based on the instruction semantics defined in the official documentation [9]. However, the execution of a configuration path may have dependencies on other paths (*e.g.*, initialization of global variables), which indicates that we must execute them in a correct order. As a result, FirmXRay maintains a queue for all the configuration paths, and each executed path will be removed from the queue. When FirmXRay encounters a path that should be executed after others, it removes the path from the head of the queue and pushes it to the end. If the queue becomes empty, which means all the paths have been statically executed, FirmXRay retrieves the concrete values of our targets from the corresponding execution context and outputs the results.

Step-III. Vulnerability detection. When the configuration values are resolved, the final step is to identify the vulnerabilities from them. Since the configurations are identified through SDK function parameters, FirmXRay requires SDK-specific knowledge to recognize the configuration semantics. In particular, for each vendor, FirmXRay focuses on the SDK functions listed in the second column of Table 1, and their descriptions and parameters of our interest are also described in the table. Based on these SDK

functions, we further define three detection policies to detect the vulnerabilities that lead to the attacks mentioned in §3.1.

(i) **Identity tracking vulnerability detection.** At a high level, there are two types of identity that can be tracked: static MAC address [27] and static UUIDs [63] [20]. Therefore, we have two corresponding policies. The first is through static MAC address identification by checking the MAC address types through APIs such as `SD_BLE_GAP_ADDR_SET` for Nordic and `GAP_ConfigDeviceAddr` for TI. According to the BLE specification [16], the MAC address can be configured as three types: (1) public address that never changes, (2) random static address that may change only when reboot, and (3) private address that change periodically (*e.g.*, every 15 minutes). Although a random static address may alter after a device power cycle, it is still not resilient to tracking because (i) BLE devices seldom power off since they are supposed to run for a long time due to the low energy cost (*e.g.*, sensors), (ii) as revealed in previous research [27], many random static addresses remain unchanged even after reboot on some devices such as fitness trackers. As a result, if the firmware uses public address or random static address, it is vulnerable to identity tracking.

The second policy to detect identity tracking is to check whether there are static UUIDs specified in the firmware. Similarly, we target corresponding APIs that take static UUIDs as parameters to identify them, such as `SD_BLE_GATTS_SERVICE_ADD` for Nordic and `GATTServApp_RegisterService` for TI.

(ii) **Active MITM vulnerability detection.** As mentioned in §3.1, the insecure pairing method such as *Just Works* can lead to active MITM attacks. Fundamentally, the pairing method is negotiated when two devices exchange the pairing features. Therefore, we focus on APIs that specify the pairing feature, such as `SD_BLE_GAP_SEC_PARAMS_REPLY` and `GAPBondMgr_SetParameter`. If no MITM protection or no I/O capability is specified, the device has to use *Just Works* pairing. In addition, the correct implementation of the secure pairing method (*e.g.*, *passkey entry* and *OOB*) may require the invocation of other procedures, such as exchanging an authentication key with the `SD_BLE_GAP_AUTH_KEY_REPLY` API for Nordic. If the corresponding APIs are not invoked, the pairing will be downgraded to *Just Works*.

The MITM vulnerability can also be revealed in the security permissions of characteristics, which is the second layer of protection against active MITM attacks. Note that for each characteristic, there are three levels of security permissions: no protection, encrypted read/write, and authenticated read-/write [49]. For instance, a developer may specify authenticated read and write to ensure that the characteristic can be read and written only when the authentication is in place. Our detection focuses on APIs that specify these permissions, such as `SD_BLE_GATTS_CHARACTERISTIC_ADD` for Nordic and `GATTServApp_RegisterService` for TI.

(iii) **Passive MITM vulnerability detection.** Recall in §3.1, failed to enforce the LESC pairing can lead to passive MITM attacks. To detect this vulnerability, we check if the LESC configuration is enabled using API `GAPBondMgr_SetParameter` for TI, or whether the ECDH key exchange is invoked during pairing using API `SD_BLE_GAP_LESC_DHKEY_REPLY` for Nordic.

Category	Total		Statistics		Vulnerabilities					
					IT		AM		PM	
	# F	# D	Avg. Size (KB)	Median Time (m)	# F	# D	# F	# D	# F	# D
Nordic-based Firmware										
Wearable	204	138	98.2	42.8	204	138	171	112	203	137
Others	76	22	223.5	48.8	76	22	63	14	75	21
Sensor	67	51	80.9	9.5	67	51	51	37	66	50
Tag (Tracker)	58	41	84.2	164.3	58	41	45	29	57	40
Robot	41	21	117.7	37.2	41	21	35	18	25	20
Medical Devices	41	21	138.6	82.3	41	21	22	10	37	20
Bike Accessory	41	35	92.3	21.1	41	35	36	30	41	35
Car Accessory	25	21	75.6	250.3	21	17	20	17	25	21
Smart Light	21	19	81.2	179.1	20	18	16	14	21	19
Switch	20	11	72.8	111.8	20	11	11	8	20	11
Smart Home	20	18	63.0	10.3	20	18	10	10	20	18
Smart Eyeglasses	19	7	58.1	56.4	19	7	19	7	19	7
Thermometer	16	13	54.2	27.9	16	13	10	9	16	13
Smart Lock	15	9	67.0	1.6	15	9	8	5	14	8
Beacon	13	12	61.4	0.9	13	12	9	8	12	11
Firearm Accessory	11	5	87.7	150.1	11	5	7	4	11	5
Agricultural Equip.	10	10	142.8	29.6	5	5	9	9	10	10
Battery	9	9	34.3	1.9	9	9	7	7	9	9
Game Accessory	9	9	67.4	12.5	9	9	8	8	9	9
Keyboard	7	5	63.4	21.6	7	5	7	5	7	5
Mouse	6	6	58.2	20.9	6	6	5	5	6	6
Printer	6	2	24.1	3.5	6	2	6	2	6	2
Surf Board	6	6	71.9	655.9	6	6	2	2	6	6
Sports Accessory	4	4	88.9	136.7	4	4	3	3	4	4
Smart Toy	4	4	58.0	1.8	4	4	3	3	4	4
Smart Clothes	3	2	57.6	7.6	3	2	3	2	3	2
Sailing Accessory	3	2	73.5	256.1	3	2	2	1	3	2
Diving Accessory	3	1	19.6	2.2	3	1	3	1	3	1
Network Device	3	3	74.2	261.3	3	3	2	2	3	3
Camera	3	3	143.0	0.1	3	3	0	0	3	3
Alarm	2	2	41.7	23.7	2	2	2	2	2	2
Headphone	2	1	122.7	40.1	2	1	0	0	2	1
TI-based Firmware										
Sensor	19	19	132.9	0.2	19	19	0	0	19	19
Smart Lock	2	2	46.3	0.1	2	2	1	1	2	2
Smart Toy	2	2	47.8	0.1	2	2	0	0	2	2
Medical Devices	1	1	70.2	0.1	1	1	0	0	1	1
Others	1	1	76.7	0.2	1	1	0	0	1	1
Total	793	538	102.7	21.9	783	528	596	385	767	530

Table 2: Experiment results across firmware categories. (F: Firmware, D: Device, IT: Identity Tracking, AM: Active MITM, PM: Passive MITM)

5 EVALUATION

We have implemented a prototype of FirmXRay[1] based on Ghidra [8] with more than 5K lines of our own code. While there are a great number of MCU manufactures, our implementation particularly targets the bare-metal firmware developed based on the Nordic or TI SDK. In this section, we present our evaluation results. We first describe the experiment setup in §5.1. Then, we provide the detailed experiment results in §5.2, followed by the attack case studies in §5.3.

5.1 Experiment Setup

Bare-metal firmware collection. To evaluate FirmXRay, we first need to collect the bare-metal firmware. Intuitively, we can either crawl firmware through the manufacturer's websites or dump them from the actual device hardware. However, such approaches are not scalable for two reasons. First, developers seldom make the device firmware publicly available. Second, it will be costly to buy all these devices. Therefore, we must look for cost-effective and scalable approaches. To this end, as indicated in §2.1, bare-metal firmware are usually transferred from mobile apps to devices for over-the-air upgrade, and thus the apps should at least have the

[1]The source code is available at https://github.com/OSUSecLab/FirmXRay.

Pointer Type	# True Positive (%)	# False Positive (%)
Absolute Function Pointer	40 (95.2%)	2 (4.8%)
Absolute String Pointer	14 (33.3%)	28 (66.7%)
Vector Table Entry	40 (95.2%)	2 (4.8%)
Three Sets Combined	42 (100%)	0 (0)

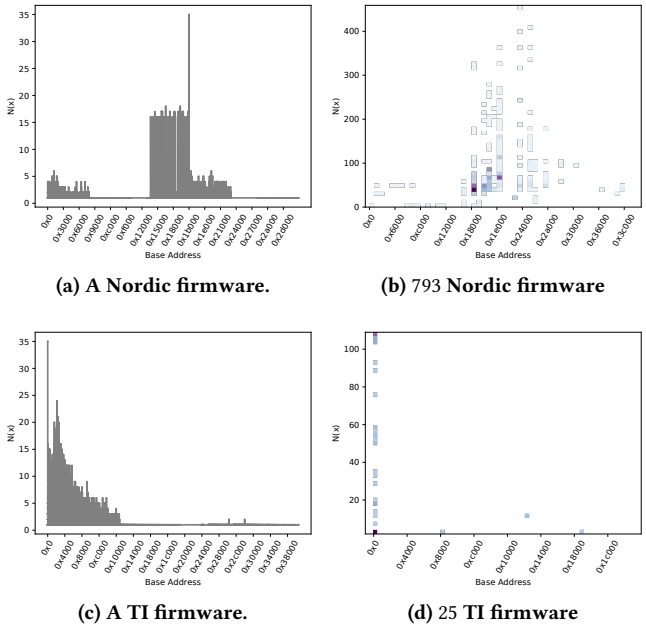
Table 3: Base address recognition w/ one set of pointers.

capability to download them, which consequently enables us to develop a mobile app based approach to collect firmware by reverse engineering its download logic. Also surprisingly, we noticed many apps actually did not implement such logic, but instead they even directly leverage the mobile app update mechanism from app stores to upgrade the firmware by embedding the firmware image inside the app package.

As a result, we developed a simple script to automatically unpack the mobile apps to extract the bare-metal firmware at scale. In particular, we first crawled about 2 million free apps from Google Play in February 2020 as our dataset. We further selected those using BLE by scanning relevant APIs such as `startScan`, and ultimately obtained 135, 486 apps in total. From these BLE apps, we directly unpacked the APKs and extracted the Nordic and TI firmware, since they have distinct signatures, which can be easily distinguished (*e.g.*, magic bytes in the firmware header). Ultimately, we successfully obtained 793 such unique bare-metal firmware (768 from Nordic and 25 from TI). Note that one app may contain multiple firmware because (1) one device may have different versions of firmware (*e.g.*, different versions of a medical device firmware extracted from the same ShockLink app), or (2) one app may have multiple device of the same category (*e.g.*, two types of thermometer from the Kinsa app). We further group different versions of the same device together, and find that the 793 firmware represent 538 unique devices.

Firmware categorization. To better understand the security implications across firmware, we would like to first categorize them. While it is challenging to directly infer their categories from the firmware code, we notice there are two sources that can help: (*i*) the parameters of the SDK function `SD_BLE_GAP_APPEARANCE_SET` from Nodic that specifies the device types (*e.g.*, sensor, keyboard, etc., and there are 50 such types), and (*ii*) the mobile app description associated with the firmware. Therefore, we use the following approach to infer the firmware categories: if `SD_BLE_GAP_APPEARANCE_SET` API is available in the firmware code, we directly obtain the device type; otherwise we manually infer the device category based on the app description. With this approach, we eventually identified 108 firmware categories (note that the extra category beyond what is defined by the API comes from our manual analysis of the app description). The categories with at least two devices are in the first column of Table 2, and we can notice that the top 5 most popular categories are (1) wearable (*e.g.*, smart band and smart watch), (2) sensor (*e.g.*, speed and humidity sensor), (3) tag (*e.g.*, device tracker), (4) robot (*e.g.*, robot dog), and (5) medical device (*e.g.*, blood pressure monitor). For the category (*e.g.*, drone, sim charger, smart luggage) that has only one device, we aggregate them in the Others category shown in the 2nd row of Table 2.

Experiment environment. Our analysis was performed on a Linux sever equipped with twelve Intel Core i7-8700 (3.20 GHz) CPUs and 32 GB RAM, running Ubuntu 18.04.2 LTS.



(a) A Nordic firmware. **(b)** 793 **Nordic firmware**

(c) A TI firmware. **(d)** 25 **TI firmware**

Figure 7: Distribution of target function value $N(x)$ across: (a)(c) candidate base addresses of a single firmware, and (b)(d) recognized base addresses among all firmware.

5.2 Experiment Results

Among the 538 unique devices (from 793 firmware), FirmXRay has identified 528 (98.1%) of them configured with random static MAC addresses, 385 (71.5%) *Just Works* pairing, and 530 (98.5%) insecure key exchange using Legacy pairing. The detailed statistics of the vulnerabilities across firmware categories are shown in the 6th - 11th columns of Table 2. In the following, we zoom in how FirmXRay reaches these results. In particular, we first describe the effectiveness of firmware base address recognition in §5.2.1, followed by the results of the three types of vulnerabilities identified in §5.2.2, §5.2.3, and §5.2.4, respectively.

5.2.1 Effectiveness of Base Address Recognition. To validate the effectiveness of our base addresses recognition, we need to first search for the ground truth of the base addresses. Interestingly, we notice that there are 42 firmware exposing their SDK versions (*e.g.*, S110, S130) in their file names, which enables identifying their base addresses according to the SDK specifications. Among these firmware, we found FirmXRay *correctly recovered all the base addresses without any false positives.* We further demonstrate the advantage of combining three sets of pointers, by using only one of the three sets to infer the base address. The detailed experimental results are presented in Table 3, which shows the number of firmware that are (in)correctly recovered with base addresses among the 42 ground truth samples. As shown, *using only a single set of pointers will all result in false positives ranging from* 4.8% *to* 66.7%. We further investigated these cases, and found that the failure is due to the lack of enough absolute pointers. For instance, the nrf52810_xxaa.bin firmware only contains 7 absolute function pointers, while there are usually tens of such pointers in other firmware. Therefore, we have to combine all three sets of pointers together to reduce false positives.

Firmware Name	Mobile App	Category	# Device
cogobeacon	com.aegismobility.guardian	Car Accessory	4
sd_bl	fr.solem.solemwf	Agricultural Equip.	2
LRFL_nRF52	fr.solem.solemwf	Agricultural Equip.	2
orb	one.shade.app	Smart Light	1
sd_bl	com.rainbird	Agricultural Equip.	1

Table 4: Firmware using private MAC address.

To further understand why our algorithm works, we plot the distribution of the target function values $N(x)$ across different candidate base addresses in Figure 7a and Figure 7c using two firmware samples from Nordic and TI, respectively. As shown, the target function value at the recovered base address (*e.g.*, $N(x)$=35) is the maximum one, and this *peak effect* exists in all the firmware we tested. Next, we present the distribution of the recovered base addresses and their $N(x)$ values across all the firmware in Figure 7b and Figure 7d for Nordic and TI firmware, respectively. It can be inferred that the base addresses range from `0x0` to `0x3C000`, showing that they are actually highly diversified across firmware, and the distribution of $N(x)$ is significantly different between the two vendors.

5.2.2 Identity Tracking Vulnerability Identification. According to the detection policies described in §4.3, there are two ways to identify identity tracking vulnerabilities: one is through detecting static MAC address, and the other is through detecting static UUIDs.

Static MAC address identification. Among these 538 devices, FirmXRay discovers that *nearly all of them* (98.1%) *have configured random static addresses* that do not change periodically. For the rest 10 (1.9%) devices, their MAC addresses are configured to be private that change periodically, which are shown in Table 4. We can notice that these devices belong to car accessories, agricultural equipment, and smart lights. Note that some of the devices have the same firmware name such as cogobeacon.

Static UUID identification. As shown in Table 5, FirmXRay has discovered 1,807 service UUIDs and 1,699 characteristic UUIDs, from 651 (82.1%) firmware (for the rest 17.9% firmware, they do not have static UUIDs as the corresponding APIs are not invoked). To understand why there are so many static UUIDs, we look into the firmware code to see how these UUIDs are defined, *e.g.*, by SIG standard or by each manufacture. Based on the parameters of the APIs in Table 1, we are able to identify the UUID types. For the service UUIDs, they can be divided into three types: primary type (standardized), secondary type (vendor-specific), and invalid type. Most of the UUIDs (98.5%) are primary type, but a very small portion of them (1.5%) are defined as invalid by the developers (which is likely caused by programmer's mistakes). As for the characteristic UUIDs, 64.8% of them are customized (vendor-specific), and 20.1% are standardized by Bluetooth SIG [1], while the rest 15.1% are unknown. These results imply that most of the firmware manufactures tend to use customized static UUIDs.

5.2.3 Active MITM Vulnerability Identification. To understand how pairing and bonding are performed among devices, we first present the results of the pairing mode, as shown from the 2nd row to 5th row of Table 6. For a particular device, it can be configured to be only (1) a peripheral (*e.g.*, smart band), (2) only a central (*e.g.*, robot dog), (3) either a peripheral or a central (*e.g.*, smart home switch), and (4) neither of them (*e.g.*, beacon). As presented in Table 6, it is

Item	#	%
# Service	1,807	100
Type		
# Primary service UUID	1,779	98.5
# Invalid service UUID	28	1.5
# Characteristic	1,699	100
Type		
# Standard characteristic UUID	341	20.1
# Customized characteristic UUID	1,101	64.8
# Unknown characteristic UUID	257	15.1
Read Permission		
# Characteristic w/ No Protection	1,631	96.0
# Characteristic w/ Encrypted Read	59	3.5
# Characteristic w/ Authenticated Read	9	0.5
Write Permission		
# Characteristic w/ No Protection	1,655	97.4
# Characteristic w/ Encrypted Write	37	2.2
# Characteristic w/ Authenticated Write	7	0.4

Table 5: Identified UUIDs and characteristic permissions.

quite common for the device to be of peripheral-only mode (47.4%) since many devices rely on mobile apps to discover and connect with them. Developers also tend to configure them to support both peripheral and central mode (34.5%), but rarely specify central-only mode (1.1%). Interestingly, we find there are 16.9% of devices that do not invoke any of these pairing functions, and the reasons may be two-folds. First, they are non-connectable beacons [24], which keep broadcasting information but never connect to the surrounding devices. Second, a number of devices directly skip the pairing procedure to favor user convenience. With respect to bonding (the 6th row), FirmXRay discovers that 48.9% of devices perform bonding to maintain the session key after pairing. For the rest 41.1%, they need to restart the pairing process with the peer device in future connections, which may expand the attack surface for eavesdropping of the encryption key during pairing.

Next, we present the details of how FirmXRay identifies vulnerabilities based on *Just Works* pairing and characteristic permissions.

Just Works pairing identification. As presented in the 8th row of Table 6, 59.1% of devices directly specify *Just Works* pairing since they do not explicitly declare MITM protection or I/O capabilities. Among those that do not use *Just Works* pairing, FirmXRay has identified two types of pairing methods: *passkey entry* and *OOB*. We find that none of the devices in our study supports numeric comparison. Interestingly, although there are 67 (12.5%) Nordic-based devices that support *passkey entry* or *OOB* pairing, their implementations are actually flawed, which means they will be eventually downgraded to *Just Works* pairing. These 67 devices can be broken down into 37 and 30 devices that have incorrectly implemented the *passkey entry* and *OOB* pairing, respectively, as shown in the 9th and 10th rows of Table 6. The reason is that their firmware fail to invoke the `SD_BLE_GAP_AUTH_KEY_REPLY` API to reply the peer device with an authenticated key, which is a mandatory step for correct implementations [12]. Therefore, by adding the 318 devices that directly specify *Just Works* pairing with those having flawed implementations, we have 385 (71.5%) *devices use Just Works pairing*. These devices essentially do not provide any protection against active MITM attacks at the BLE link layer.

In contrast, we find that only 30 devices have correctly implemented *passkey entry* or *OOB* pairing. We further investigate their categories, and it turns out they are smart keyboards, smart debit

Item	N	T	Total	%
# Total Device	513	25	538	100
Pairing Mode				
# Peripheral only	230	25	255	47.4
# Central only	6	0	6	1.1
# Peripheral and central	186	0	186	34.5
# No pairing	91	0	91	16.9
# Device w/ bonding	250	13	263	48.9
# Device w/ active MITM vulnerability	384	1	385	71.5
# Device w/ *Just Works* pairing only	317	1	318	59.1
# Device w/ flawed *passkey entry* implementation	37	0	37	6.9
# Device w/ flawed *OOB* implementation	30	0	30	5.6
# Device w/ secure pairing	6	24	30	3.8
# Device w/ correct *passkey entry* implementation	3	24	27	3.4
# Device w/ correct *OOB* implementation	3	0	3	0.4

Table 6: Pairing configurations of devices (N:Nordic, T:TI).

cards, wearable, and so on. In addition, it is more common for TI firmware to configure secure pairing methods such as *passkey entry*, as indicated in the last two rows of Table 6. Overall, our results reveal that most of the BLE devices tend to use only *Just Works* pairing, which is possibly due to the lack of hardware capability or the misconfiguration from the developers.

Characteristic permission analysis. In addition to the pairing configurations, the characteristic permission can also reveal the active MITM vulnerability. As shown in Table 5, we further analyze the security permissions of the 1,699 BLE characteristics, which come from the Nordic-based firmware (we did not identify them from any TI firmware because them do not invoke corresponding APIs to configure these permissions). The results are broken down into read and write permissions, respectively. To our surprise, we discover that *the vast majority (over 96%) of the characteristics specifies the lowest security level of permissions*, showing that they can be arbitrarily read or written by peer devices without any encryption and authentication. This further implies that they can be directly exploited once the firmware is compromised by MITM attacks. In contrast, only very few number of the characteristics require encryption or authentication before read (4.0%) and write (2.6%). These characteristics usually come from the security-sensitive devices such as smart locks, smart home switches, and medical devices.

5.2.4 Passive MITM Vulnerability Identification. The passive MITM vulnerability is determined by whether the firmware has enforced LESC pairing to secure the key exchange, as described in the detection policy in §4.3. Among the 538 devices, FirmXRay discovers that 98.5% of them fail to do so, as reported in the last row of Table 2. *As such, these devices can be vulnerable to passive MITM attacks if there is no application-layer encryption*, allowing any attackers to eavesdrop the encryption key and read sensitive device data. In contrast, only 8 (1.5%) devices have eliminated this vulnerability by enforcing the LESC pairing. The detailed descriptions of these devices are shown in Table 7. Among them, there are many firmware versions mapped to the same device such as DogBodyBoard and CPRmeter.

5.3 Attack Case Studies

To exploit the three types of vulnerabilities identified, we correspondingly design three types of attacks with real devices. Due to limited budget, we purchased 5 vulnerable devices as shown in Table 8. We have three criteria when choosing the devices: (1) the device needs to be in the top categories in Table 2, (2) the device should not be too costly (we therefore excluded medical device and robot), and (3) the device should simultaneously contain the three vulnerabilities identified (to maximize the coverage of the vulnerabilities). Based on the functionality of these devices, we design three types of attacks: user tracking, unauthorized control, and sensitive data eavesdropping. To launch these attacks, we built an attack device based on a Nordic NRF52-DK board [13].

Firmware Name	Mobile App	Category	# Version
DogBodyBoard	com.wowwee.chip	Robot	16
BW_Pro	com.ecomm.smart_panel	Tag	1
Smart_Handle	com.exitec.smartlock	Smart Lock	1
Sma05	com.smalife.watch	Wearable	1
CPRmeter	com.laerdal.cprmeter2	Medical Device	4
WiJumpLE	com.wesssrl.wijumple	Sensor	1
nRF Beacon	no.nordicsemi.android.nrfbeacon	Beacon	1
Hoot Bank	com.qvivr.hoot	Debit Card	1

Table 7: Firmware that enforce LESC pairing.

A1. User tracking. Vulnerable BLE devices carried along with users are desired targets for tracking attacks, and we have three such devices: Nuband, Chipolo, and XOSS. Although BLE devices stop broadcasting after they are connected, we demonstrated our attack still succeeded in the following two scenarios. The first scenario is when companion apps are closed in both Android and iOS, and then the device disconnects with the app and starts to broadcast its MAC address. The second scenario is in iOS when the companion app enters the background, it in fact terminates the connection due to the limited Bluetooth capability in background [4], and the device also starts broadcasting.

A2. Unauthorized control. For devices that use *Just Works* pairing, they can be vulnerable to active MITM attacks such as spoofing. We demonstrated this attack with a smart home button pusher, which is usually placed on electronic switches for remote control, such as lights, coffee machines, and even more safety-critical ones such as door locks. However, since the device has to use *Just Works* pairing, it does not provide any authentication to recognize unauthorized users, and thus we successfully sent spoofed commands to remotely control the lights in our test.

A3. Sensitive data eavesdropping. Among the 5 devices, many of them, *e.g.*, Nuband, Kinsa Smart, and XOSS, carry sensitive user data (*e.g.*, steps, temperature, and travel distance). We demonstrated that it is possible to perform eavesdropping attack to obtain the LTK to decrypt the BLE data. Specifically, we first listened to all messages during the pairing process, and applied an offline brute-force searching to find the appropriate TK to calculate the LTK, which took only a few seconds. Note that in *Just Works* pairing, the attack is much easier since the TK is a hardcoded 0-string [32].

6 DISCUSSION

False positives (FP) and false negatives (FN) of FirmXRay. In theory, FP can exist due to the incorrect disassembling, which can be caused by a lack of sufficient absolute pointers in firmware, resulting in less number of constraints to differentiate the optimal base address with others. However, as demonstrated in our evaluation, FirmXRay correctly recovered all the base addresses according to our ground truth evaluation. Similarly, the incorrect base address

Device Name	Category	Vulnerabilities			Attacks		
		IT	AM	PM	A1	A2	A3
Nuband Activ+	Wearable	✓	✓	✓	✓		✓
Kinsa Smart	Thermometer	✓	✓	✓			✓
Chipolo ONE	Tag	✓	✓	✓	✓		
SwitchBot Button Pusher	Smart Home	✓	✓	✓		✓	
XOSS Cycling Computer	Sensor	✓	✓	✓	✓		✓

Table 8: Vulnerable BLE devices and attack case studies.

may also cause FN, since incomplete disassembled code may be produced that does not cover the desired configurations. Other sources of FP or FN could come from the fundamental limitations of program analysis, such as branch explosion [55] [42].

On the exploitability of the vulnerabilities. While FirmXRay has identified three types of vulnerabilities from firmware, not all of them can be exploited. For instance, stationary devices such as smart home devices are not subject to identity tracking even though they use random static MAC addresses identified by FirmXRay. In addition, many devices may have additional layer of security to mitigate active and passive MITM attacks, such as authentication and encryption in application layer. However, through the case studies of the 5 devices, we have not witnessed such a case among them.

Disclosure of findings. In June 2020, we disclosed our vulnerability findings to all device vendors (in total 205) through emails. As of the time of this writing, 12 vendors have acknowledged our findings and taken our suggestions into account, such as Wattbike, INPEAK, SRM, WOOLF, goTenna, and Chipolo.

Root causes of the vulnerabilities. We believe there are two main root causes for the identified vulnerabilities. The first is the lack of hardware capabilities. For instance, a device without I/O capability is very likely to be configured as no I/O. The second is the misconfiguration by the developers of the firmware. For example, a device with sufficient I/O support is misconfigured as no I/O. There are multiple reasons for developers to misconfigure firmware. First, according to our engagement with vendors, many of them were actually aware of the security problems (*e.g.*, using *Just Works* pairing), but they still prefer simpler implementation to favor user experience. Second, there are also limitations of the BLE module on smartphones, which make some implementations (*e.g.*, randomized MAC address) challenging in practice. For instance, the iOS has limited the BLE capability for app developers [4].

Future work. First, as described in §5.1, we directly unpacked the mobile app APKs to extract the embedded firmware. Thus, there may also exist other firmware we cannot obtain, such as those downloaded from servers. Second, while we have demonstrated FirmXRay for Nordic and TI, FirmXRay can also be adapted to other SDKs and architectures. Third, FirmXRay faces a challenge of confirming the vulnerabilities due to the static analysis, and we also plan to enable emulation and dynamic analysis on bare-metal IoT firmware to confirm our results.

7 RELATED WORK

Firmware analysis. Over the past decade, firmware has been an attractive target for security analysis. With static analysis, FIE [28] detects memory-safety bugs in micro-controllers, FirmUSB [35] and ProXray [33] vet the embedded USB devices, FirmAlice [47] uncovers authentication bypassing vulnerabilities, and Karonte [42] detects insecure interactions between multiple embedded firmware binaries. With dynamic analysis, Avatar [56] forwards I/O assess from emulators to real embedded devices, FirmDyn [23] employs full system emulation for scalable and automatic analysis of Linux binaries, Firm-AFL [60] combines system-mode and user-mode emulations for high-throughput firmware fuzzing, P^{2}IM [31] automatically models the I/O behaviours of peripherals to achieve hardware-independent firmware testing; HALucinator [26] relies on replacing high-level hardware abstraction layer functions to achieve firmware re-hosting. Once firmware is able to be executed, a common technique to find vulnerabilities is fuzzing, such as IoTFuzzer [25] for IoT devices and PeriScope [52] for Linux kernel peripherals.

BLE security. There have been numerous efforts in BLE attacks and defenses, including the discovery of vulnerable pairing (*e.g.*, [43, 44, 57]) and BLE packets eavesdropping [44]. Recently, there were also identity tracking attacks that leverage the static MAC address [27], signal strength [29], and advertised information [37], and static UUIDs [63] [20]. To mitigate these privacy attacks, Fawaz *et al.* proposed BLE-Guardian [30], a channel-level protection to allow only authorized peripherals to connect with the protected device. Most recently, a handful of other research focus on other types of attacks, such as cross-app co-located attacks [48] and downgrade attacks [58].

Misconfiguration detection. Misconfiguration has been a problem in IoT platforms [17] and cloud services [62] [18]. In particular, as summarized by Alrawi *et al.*, many devices use insecure default configurations [17]. In addition, some devices have also been revealed to have misconfigurations, such as smart lock [53], smart speaker [22], and smart light [21], which have led to serious security concerns such as insecure access control. Compared with these works, FirmXRay represents a scalable and binary code only approach to uncover vulnerabilities reflected in the configurations from bare-metal firmware.

8 CONCLUSION

We have presented FirmXRay, the first automated static binary analysis tool to detect BLE link layer vulnerabilities from bare-metal firmware. It features a novel algorithm to systematically recognize firmware base address for robust disassembling, then precisely identifies configurations from SDK functions, and finally resolves configuration values to detect the vulnerabilities. The prototype of FirmXRay has been implemented atop Ghidra. To evaluate FirmXRay, we developed a mobile app based approach to collect bare-metal firmware at scale, which resulted in 793 unique ones corresponding to 538 devices. Among them, FirmXRay discovered that 71.5% of these devices only use *Just Works* pairing, and nearly all of them have configured random static MAC addresses and insecure key exchanges. We have demonstrated concrete attacks with 5 real-world BLE devices, which not only undermine user privacy but also safety.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their constructive feedback. This research was supported in part by NSF 1750809 and 1834215, DARPA N6600120C4020, and ONR N00014-17-1-2995.

REFERENCES

[1] 16 bit uuids for members. https://www.bluetooth.com/specifications/assigned-numbers/16-bit-uuids-for-members/. (Accessed on 09/20/2020).

[2] Ble-stack bluetooth low energy. https://www.ti.com/tool/BLE-STACK. (Accessed on 09/20/2020).

[3] Bluetooth sig, inc. https://www.bluetooth.com/. (Accessed on 09/20/2020).

[4] Core bluetooth background processing for ios apps. https://developer.apple.com/library/archive/documentation/NetworkingInternetWeb/Conceptual/CoreBluetooth_concepts/CoreBluetoothBackgroundProcessingForIOSApps/PerformingTasksWhileYourAppIsInTheBackground.html. (Accessed on 09/20/2020).

[5] Developing a bluetooth low energy application. http://software-dl.ti.com/simplelink/esd/simplelink_cc13x0_sdk/2.20.00.38/exports/docs/blestack/software-developers-guide/ble-stack-2.x/index.html. (Accessed on 09/20/2020).

[6] Dialog semiconductor. https://www.dialog-semiconductor.com. (Accessed on 09/20/2020).

[7] Gatt overview. https://developer.android.com/reference/android/bluetooth/BluetoothGatt. (Accessed on 09/20/2020).

[8] Ghidra. https://ghidra-sre.org/. (Accessed on 09/20/2020).

[9] Instruction sets - arm developers. https://developer.arm.com/architectures/instruction-sets. (Accessed on 09/20/2020).

[10] Memory layout. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v11.0.0%2Fbledfu_memory.html. (Accessed on 09/20/2020).

[11] Nordic semiconductor. https://www.nordicsemi.com. (Accessed on 09/20/2020).

[12] Nordic semiconductor infocenter. https://infocenter.nordicsemi.com/topic/struct_nrf52/struct/nrf52_softdevices.html?cp=4_5. (Accessed on 09/20/2020).

[13] nrf52 dk - development kit for bluetooth low energy and bluetooth mesh. https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52-DK. (Accessed on 09/20/2020).

[14] Supervisor call interface. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v14.1.0%2Flib_svc.html. (Accessed on 09/20/2020).

[15] Texas instruments. http://www.ti.com. (Accessed on 09/20/2020).

[16] Bluetooth specification version 4.2. https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=286439, 2014.

[17] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. Sok: Security evaluation of home-based iot deployments. In *40th IEEE Symposium on Security and Privacy (SP)*, pages 1362–1380, May 2019.

[18] Omar Alrawi, Chaoshun Zuo, Ruian Duan, Ranjita Kasturi, Zhiqiang Lin, and Brendan Saltaformaggio. The betrayal at cloud city: An empirical analysis of cloud-based mobile backends. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 551–566, Santa Clara, CA, August 2019.

[19] Igor Bisio, Andrea Sciarrone, and Sandro Zappatore. A new asset tracking architecture integrating rfid, bluetooth low energy tags and ad hoc smartphone applications. *Pervasive and Mobile Computing*, 31:79–93, 2016.

[20] Guillaume Celosia and Mathieu Cunche. Fingerprinting bluetooth-low-energy devices based on the generic attribute profile. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pages 24–31, 2019.

[21] Alex Chapman. Hacking into internet connected light bulbs. https://www.contextis.com/us/blog/hacking-into-internet-connected-light-bulbs, 2014.

[22] Alex Chapman. Alexa, are you listening? https://www.contextis.com/us/blog/hacking-into-internet-connected-light-bulbs, 2017.

[23] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *2016 Network and Distributed Systems Security Symposium (NDSS)*, volume 16, pages 1–16, 2016.

[24] Dongyao Chen, Kang G Shin, Yurong Jiang, and Kyu-Han Kim. Locating and tracking ble beacons with smartphones. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 263–275, 2017.

[25] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS 18)*, San Diego, CA, February 2018.

[26] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[27] Aveek K Das, Parth H Pathak, Chen-Nee Chuah, and Prasant Mohapatra. Uncovering privacy leakage in ble network traffic of wearable fitness trackers. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*, pages 99–104, 2016.

[28] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. {FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, 2013.

[29] Ramsey Faragher and Robert Harle. Location fingerprinting with bluetooth low energy beacons. *IEEE journal on Selected Areas in Communications*, 33(11):2418–2428, 2015.

[30] Kassem Fawaz, Kyu-Han Kim, and Kang G Shin. Protecting privacy of ble device users. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1205–1221, 2016.

[31] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[32] Daniel Filizzola, Sean Fraser, and Nikita Samsonau. Security analysis of bluetooth technology. 2018.

[33] Farhaan Fowze, Dave Jing Tian, Grant Hernandez, Kevin Butler, and Tuba Yavuz. Proxray: Protocol model learning and guided firmware analysis. *IEEE Transactions on Software Engineering*, 2019.

[34] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[35] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, pages 2245–2262, 2017.

[36] Kai Ren. Bluetooth pairing part 4: Le secure connections - numeric comparison. https://blog.bluetooth.com/bluetooth-pairing-part-4, 2017.

[37] Aleksandra Korolova and Vinod Sharma. Cross-app tracking via nearby bluetooth low energy devices. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 43–52, 2018.

[38] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *18th Network and Distributed Systems Security Symposium (NDSS)*, 2011.

[39] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, February 2010.

[40] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Foppe, Travis Mayberry, Erik Rye, Brandon Sipes, and Sam Teplov. Handoff all your privacy–a review of apple's bluetooth low energy continuity protocol. *Proceedings on Privacy Enhancing Technologies*, 2019(4):34–53, 2019.

[41] Sode Pallavi and V Anantha Narayanan. An overview of practical attacks on ble based iot devices and their security. In *2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS)*, pages 694–698, 2019.

[42] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *41st IEEE Symposium on Security and Privacy (SP)*, pages 431–448.

[43] Tomas Rosa. Bypassing passkey authentication in bluetooth low energy. *IACR Cryptology ePrint Archive*, 2013:309, 2013.

[44] Mike Ryan. Bluetooth: With low energy comes low security. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, 2013.

[45] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54. IEEE, 2002.

[46] Nordic Semiconductor. Quarterly presentation q4 2019. https://www.nordicsemi.com/-/media/Investor-Relations-and-QA/Quarterly-Presentations/2019/Q4_Quarterly_presentation_2019.pdf?la=en&hash=EE265776035B52B96B1BE14541877A97500CD05B, 2019.

[47] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Network and Distributed Systems Security Symposium (NDSS)*, 2015.

[48] Pallavi Sivakumaran and Jorge Blasco. A study of the feasibility of co-located app attacks against ble and a large-scale analysis of the current application-layer security landscape. In *28th USENIX Security Symposium*, pages 1–18, 2019.

[49] Pallavi Sivakumaran and Jorge Blasco Alis. A low energy profile: Analysing characteristic security on ble peripherals. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 152–154, 2018.

[50] Igor Skochinsky. Intro to embedded reverse engineering for pc reversers. In *REcon conference, Montreal, Canada*, 2010.

[51] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *18th Network and Distributed Systems Security Symposium (NDSS)*, 2011.

[52] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael

Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *26th Network and Distributed Systems Security Symposium (NDSS)*, pages 1–15. Internet Society, 2019.

[53] Blase Ur, Jaeyeon Jung, and Stuart Schechter. The current state of access control for smart devices in homes. In *Workshop on Home Usable Privacy and Security (HUPS)*, volume 29, pages 209–218. HUPS 2014, 2013.

[54] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[55] Yan Xiong, Cheng Su, Wenchao Huang, Fuyou Miao, Wansen Wang, and Hengyi Ouyang. Smartverif: Push the limit of automation capability of verifying security protocols by dynamic strategies. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 253–270, 2020.

[56] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *21st Network and Distributed Systems Security Symposium (NDSS)*, volume 14, pages 1–16, 2014.

[57] Wondimu K Zegeye. Exploiting bluetooth low energy pairing vulnerability in telemedicine. International Foundation for Telemetering, 2015.

[58] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking secure pairing of bluetooth low energy using downgrade attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 37–54, 2020.

[59] Qingchuan Zhao, Haohuang Wen, Zhiqiang Lin, Dong Xuan, and Ness Shroff. On the accuracy of measured proximity of bluetooth-based contact tracing apps. In *International Conference on Security and Privacy in Communication Networks*, 2020.

[60] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[61] Ruijin Zhu, Yu-an Tan, Quanxin Zhang, Fei Wu, Jun Zheng, and Yuan Xue. Determining image base of firmware files for arm devices. *IEICE TRANSACTIONS on Information and Systems*, 99(2):351–359, 2016.

[62] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *40th IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310, May 2019.

[63] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, pages 1469–1483, November 2019.