LIGHTBLUE: Automatic Profile-Aware Debloating of Bluetooth Stacks

Jianliang Wu[1*], Ruoyu Wu[1*], Daniele Antonioli[2], Mathias Payer[2], Nils Ole Tippenhauer[3], Dongyan Xu[1], Dave (Jing) Tian[1], Antonio Bianchi[1]

[1]Purdue University, [2] EPFL, [3] CISPA Helmholtz Center for Information Security

{wu1220, wu1377, dxu, daveti, antoniob}@purdue.edu, daniele.antonioli@epfl.ch, mathias.payer@nebelwelt.net, tippenhauer@cispa.de

Abstract

The Bluetooth standard is ubiquitously supported by computers, smartphones, and IoT devices. Due to its complexity, implementations require large codebases, which are prone to security vulnerabilities, such as the recently discovered BlueBorne and BadBluetooth attacks. While defined by the standard, most of the Bluetooth functionality, as defined by different Bluetooth profiles, is not required in the common usage scenarios.

Starting from this observation, we implement LIGHTBLUE, a framework performing automatic, profile-aware debloating of Bluetooth stacks, allowing users to automatically minimize their Bluetooth attack surface by removing unneeded Bluetooth features. LIGHTBLUE starts with a target Bluetooth application, detects the associated Bluetooth profiles, and applies a combination of control-flow and data-flow analysis to remove unused code within a Bluetooth host code. Furthermore, to debloat the Bluetooth firmware, LIGHTBLUE extracts the used Host Controller Interface (HCI) commands and patches the HCI dispatcher in the Bluetooth firmware automatically, so that the Bluetooth firmware avoids processing unneeded HCI commands.

We evaluate LIGHTBLUE on four different Bluetooth hosts and three different Bluetooth controllers. Our evaluation shows that LIGHTBLUE achieves between 32% and 50% code reduction in the Bluetooth host code and between 57% and 83% HCI command reduction in the Bluetooth firmware. This code reduction leads to the prevention of attacks responsible for at least 20 CVEs, such as BlueBorne and BadBluetooth, while introducing no performance overhead and without affecting the behavior of the debloated application.

1 Introduction

Bluetooth provides wireless, short-range, generic, and affordable communication capabilities for billions of devices [6]. Bluetooth is specified in an open standard that defines around forty (40) profiles, encompassing a large variety of applications and devices. Each profile corresponds to a use case, such as Advanced Audio Distribution Profile (A2DP) for audio, OBject EXchange (OBEX) for data-exchange, and Human Interface Device Profile (HID) for input-output peripherals.

The large number of diverse profiles is one of the key reasons why the Bluetooth standard is so complex. For example, the Bluetooth 5.2 core specification alone is 3255 pages long [8] and each Bluetooth profile is specified in a dedicated document. Consequently, Bluetooth stack implementations require large codebases. For instance, there are about 436,000 lines of code in the Android 11 Bluetooth host code, in addition to the closed source Bluetooth firmware counterpart.

Bluetooth is vulnerable to severe attacks both at the specification and implementation levels. Recent academic works have shown standard-compliant attacks against the Bluetooth specification affecting both Bluetooth Classic and Bluetooth Low Energy [2, 35, 42, 45]. Besides, 132 vulnerabilities, such as buffer overflow and authentication bypass, have been found since 2017 affecting the Bluetooth host code on different versions of Android [27, 38]. Attacks such as BlueBorne [5] and BadBluetooth [46] demonstrated that remote code execution and local privilege escalations are common with Bluetooth host code. In addition, recent research [26, 34] demonstrates frequent vulnerabilities in Bluetooth firmware.

Given the large codebases, attack surface reduction via debloating unneeded code is an effective way to secure Bluetooth stacks. While existing tools [23, 29, 30, 36] can debloat software codebases, none of them can be applied to Bluetooth stack implementations directly, due to the intrinsic structure of the Bluetooth standard and its implementations. First, Bluetooth stack implementations often operate as event-driven state machines with different callbacks per functionality. Current debloating approaches cannot handle callbacks or understand the used state machines. Second, a full-stack Bluetooth implementation includes both the Bluetooth host code running within the host machine and the Bluetooth firmware running within the Bluetooth chip. No existing tools can debloat

*The two authors contributed equally.

code on heterogeneous architectures at the same time. Lastly, Bluetooth firmware are typically closed source, defeating any source-based debloating tool.

To enable effective debloating of Bluetooth stacks, in this paper, we present LIGHTBLUE, a framework performing automatic profile-aware debloating of Bluetooth stacks, spanning from Bluetooth host code to Bluetooth firmware, and allowing users to automatically minimize their Bluetooth attack surface by removing Bluetooth features not required by current applications. To address the unique challenges in Bluetooth debloating, LIGHTBLUE starts with a target Bluetooth application and detects the Bluetooth profile used by the application. Then, it transforms the Bluetooth host code into a single-entry program, simulating the transitions of its state machine, and applying a combination of control-flow and data-flow analysis to detect and remove unneeded code within the host code. To remove the unused functionality within the firmware, LIGHTBLUE also identifies unused Host Controller Interface (HCI) commands and patches the HCI dispatcher within the firmware binary to ignore them.

We evaluate LIGHTBLUE on four different Bluetooth host code and three pieces of different Bluetooth firmware. Our results show that LIGHTBLUE achieves between 32% and 50% code reduction in the Bluetooth host code and between 57% and 83% HCI command reduction in the Bluetooth firmware. This code reduction leads to the prevention of attacks corresponding to, at least, 20 CVEs. We run the debloated host code and firmware with the target applications and do not observe abnormal system behaviors.

In summary, our main contributions are as follows:

- We develop a technique to debloat the Bluetooth host code by removing unneeded code, using a combination of profile-aware control-flow and data-flow analysis.
- We bridge the gap between Bluetooth host code debloating and Bluetooth firmware debloating. To achieve this goal, we extract the list of HCI commands needed by a specific profile, and we remove the unused HCI command handlers from the firmware.
- We design and implement LIGHTBLUE as a fully automated pipeline framework to output a debloated and usable Bluetooth stack implementation that can support a given target application without interfering with its intended functionality.
- We evaluate LIGHTBLUE on four different Bluetooth host code and three different pieces of Bluetooth firmware, and we demonstrate that LIGHTBLUE can achieve around 32%-50% host code reduction, around 57%-83% HCI command reduction within the firmware, and prevent attacks related to 20 CVEs.

Our code is available at https://github.com/purseclab/lightblue.

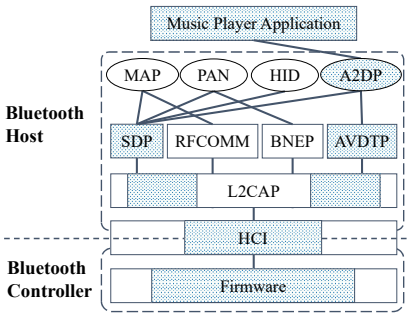

Figure 1: Architecture of the Bluetooth stack. In the figure we highlighted code components used by a hypothetical application acting as a music player.

2 Background

As shown in Figure 1, the Bluetooth stack is split into two parts: the *Bluetooth host* (host for short) layered upon the Host Controller Interface (HCI) and the *Bluetooth controller* (controller for short) locating beneath the HCI. The HCI is defined by the Bluetooth specification [8] as the protocol used for the communication between the host and the controller. Correspondingly, the Bluetooth stack implementation is split into two parts: the *Bluetooth host code* (host code for short) running upon or within an operating system (e.g., Android) and the *Bluetooth firmware* (firmware for short) running on a dedicated Bluetooth chip. The host code sits right below applications and includes several middle layers, and the firmware implements the link layer and interacts with the baseband and radio hardware.

2.1 Bluetooth Host

Profiles. Roughly speaking, a Bluetooth profile corresponds to a specific use case. Profiles define the standard way of using the different protocols and their features. For example, the Advanced Audio Distribution Profile (A2DP) [7] defines the protocols and procedures that implement the streaming of high-quality audio content, including Audio/Video Data Transport Protocol (AVDTP) and Service Discovery Protocol (SDP). Profiles may employ different physical transports. For example, the A2DP profile transmits audio data through Bluetooth Classic (i.e., Bluetooth Basic Rate/Enhanced Data Rate), and the Generic Attribute Profile (GATT) specifies the procedures of data transmission via Bluetooth Low Energy (BLE), which is mainly for power-constrained devices. As shown in Figure 1, a typical profile only uses parts of the stack across different layers.

Protocols. The Bluetooth specification defines a number of protocols acting as a middle layer between profiles and lower-level Bluetooth packets. A protocol is usually employed by a limited number of profiles. For example, the Bluetooth Net-

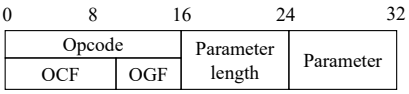

Figure 2: Format of an HCI command. Each HCI command has an opcode field composed of OCF and OGF and a parameter field that depends on the opcode.

work Encapsulation Protocol (BNEP) is only used by the Personal Area Network (PAN) profile. The Logical Link Control and Adaption Protocol (L2CAP) fragments, reassembles, and multiplexes packets generated by higher layer protocols and provides TCP-like services for Bluetooth.

2.2 Host Controller Interface (HCI)

The Host Controller Interface (HCI) connects the host and the controller by defining commands, events, and data packets communicating between the two parts. For example, the host can send an HCI command to the controller, which answers with an HCI event. As shown in Figure 2, an HCI command is composed of two parts: an opcode and a command parameter. The opcode differentiates HCI commands and has the Opcode Command Field (OCF) and the Opcode Group Field (OGF). The parameter field depends on the opcode.

2.3 Bluetooth Controller

The Bluetooth firmware runs on the controller and processes HCI commands from the Bluetooth host. In particular, when it receives an HCI command from the host code, it parses and dispatches the command to the corresponding command handler (based on the command's OGF and OCF), and returns an HCI event to the host code. A controller can include vendor-specific command handlers implementing ad-hoc functionalities, such as reading and writing the firmware's RAM at runtime.

The firmware support different radio links for different purposes. For example, L2CAP employs the Asynchronous Connection-Less (ACL) link for asynchronous data transfer, and the Synchronous Connection-Oriented (SCO) link to transmit synchronous data, such as audio. To set up a radio link, the host code needs to issue HCI commands to the firmware to establish a link with a remote device. For example, an SCO link can be started using the `HCI_Setup_Synchronous_Connection` command.

3 Threat Model and Motivation

Threat model. We target the host and the device controller supporting Bluetooth Classic and/or BLE. We only require source code access to the host code, while the firmware can be closed source and available only as a binary blob. We assume

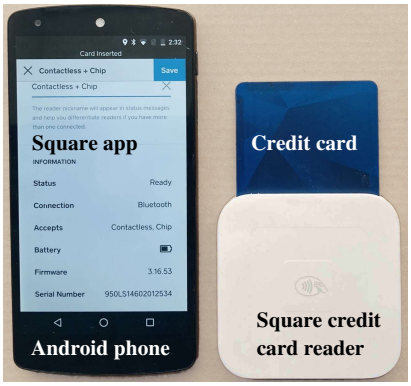

Figure 3: The Square app on Android phone with debloated Bluetooth stack communicates with the Square credit card reader.

that one major application dominates the Bluetooth usage within the device. While we trust the Bluetooth hardware, adversaries might try to exploit vulnerabilities within the host code and firmware to further compromise the whole system. In this scenario, LIGHTBLUE aims to reduce the attack surface exposed by the Bluetooth stack implementations.

Motivating example. A concrete usage scenario of LIGHTBLUE would be a Point-of-Sale app (e.g., Square [39]) running on a dedicated Android tablet or phone (shown in Figure 3). This app interacts with a dedicated Square credit card reader using the Bluetooth interface of the phone. This is a common usage scenario in shops and restaurants.

In the threat model we described, an attacker could exploit vulnerabilities in the host code and/or the firmware affecting the whole Bluetooth stack. However, in this specific usage scenario, the Android phone uses the Bluetooth interface exclusively to receive credit card data through the Square credit card reader. Technically, this feature only requires the usage of the GATT profile over BLE. Therefore, we can significantly reduce the attack surface of the Bluetooth stack by removing the code dealing with protocols and profiles that are not needed by the Point-of-Sale app (see Section 7.2 for more details).

LIGHTBLUE use cases. More broadly, by reducing the Bluetooth host and controller's attack surface, LIGHTBLUE serves a variety of potential users: (1) Original Equipment Manufacturers (OEMs) can use LIGHTBLUE to specialize their products (e.g., Point-of-Sale tablets), (2) enterprise users can use LIGHTBLUE to customize their devices (e.g., patient check-in tablets in hospitals), and (3) experienced end-users can use LIGHTBLUE to harden their Bluetooth stack (e.g., hardened Bluetooth stack for Android or LineageOS).

Protection scope. In general, LIGHTBLUE can protect both the host code and the firmware by reducing their attack surface. LIGHTBLUE secures the code in three different ways. First, it removes unneeded but potentially vulnerable func-

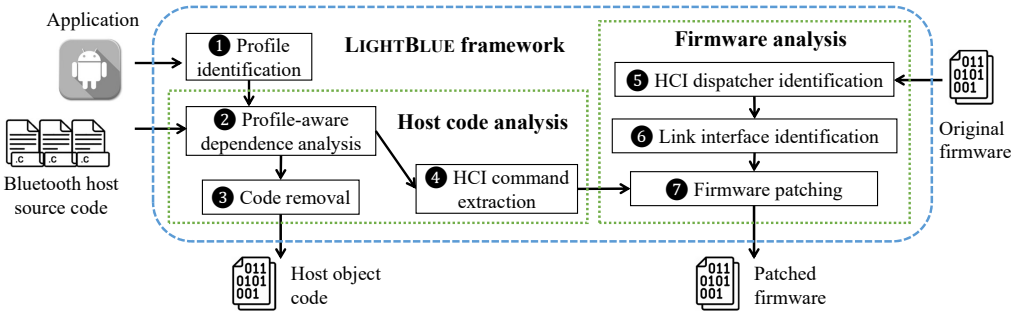

Figure 4: Workflow of LIGHTBLUE

tions (see Section 7.4 and Section 8.1). Second, it reduces the number of ROP code gadgets, hindering the exploitability of a bug (see Section 7.3.1). Third, it prevents Bluetooth attacks exploiting protocol malleability, such as BadBluetooth (see Section 8.2). These attacks allow malicious access and exploitation of normally unused code (which LIGHTBLUE can remove).

4 Debloating Challenges and Solutions

Full-stack Bluetooth debloating imposes unique challenges compared to "single program" software debloating. We enumerate the three major challenges and provide a summary of how LIGHTBLUE addresses them.

Profile state transition and profile coupling. The design of Bluetooth host code is different from a single-entry program. A profile is implemented as an event-driven state machine in which different callbacks can be called at different times. Furthermore, the executions of different profiles are sometimes coupled together. For example, in the host code of Android 6.0.1, a broker function receives and dispatches all the received events. Therefore, the code of different profiles cannot be partitioned just by operating at the function granularity. Some approaches [25, 36] suggest solving this issue by using a combination of data-flow and control-flow analysis. However, this hybrid method only works on programs having a single entry point and receiving inputs through a limited number of interfaces (e.g., a program receiving inputs through command-line arguments and standard input). Therefore, these approaches are not suitable for a multi-entry, callback-driven software, such as the Bluetooth host code.
Approach: LIGHTBLUE uses a profile-aware analysis (see Section 5.2) to decouple the profile-specific code chunks. This technique transforms the multi-entry host code into a single-entry program. After this transformation, LIGHTBLUE can use a data-flow-based approach (inspired by TRIMMER [36]) to separate code chunks used by different profiles.

Semantic gap between the host code and the firmware. The host code does not directly invoke the firmware code since these two codebases run on separate CPUs. While a profile-based analysis provides a way to debloat the host code, we need to find a way to extend the debloating from the host code to the firmware, achieving full-stack debloating.
Approach: LIGHTBLUE exploits the fact that the Bluetooth specification defines an HCI layer to bridge the host and the controller. In particular, LIGHTBLUE extracts the HCI commands needed by a specific profile of interest and maps them to the corresponding HCI command handlers in the firmware (Section 5.3).

Diversity and accessibility of the firmware. The firmware of a Bluetooth controller is usually proprietary and closed source. On top of that, controllers from different vendors have different software stacks (e.g., different real-time operating systems and Bluetooth controller implementations) and might even run on different architectures (e.g., ARM and MIPS). The absence of source code, together with the heterogeneity of the firmware and the architecture of the controller, prevents the application of existing and generic debloating techniques to this specific domain.
Approach: LIGHTBLUE exploits the fact that the firmware, regardless of its specific implementation, needs to dispatch the received HCI commands to the corresponding handler functions if HCI is supported. Therefore, LIGHTBLUE focuses on identifying the HCI dispatcher function using a two-step approach, as we will explain in Section 5.4.

5 System Design

The workflow of the LIGHTBLUE framework is illustrated in Figure 4. As input, LIGHTBLUE takes an application and a Bluetooth stack implementation (host code and firmware).

Internally, LIGHTBLUE is composed of three parts: (i) *Profile identification*, (ii) *Host code analysis*, and (iii) *Firmware analysis*. To remove unneeded host code, LIGHTBLUE first identifies the profile that is used by the application (step ❶ in Figure 4) [1]. Then, LIGHTBLUE analyzes the source code of the host, performing a profile-aware dependence analysis.

[1] For simplicity, in this description, we assume that LIGHTBLUE is used to keep a single profile and remove the others. In Section 7.3.1, we will show how LIGHTBLUE can also be used with multiple profiles.

This analysis generates a profile-specific dependency graph (step ❷). LIGHTBLUE then removes the code outside this graph (step ❸) and generates a debloated host code.

Additionally, LIGHTBLUE generates a list of HCI commands that are used by the target profile (step ❹). This list is used to remove the unneeded functionalities in the firmware. Specifically, LIGHTBLUE takes the original firmware and analyzes it to find the code to handle HCI commands (step ❺). Since the host code interacts with the firmware via HCI commands to set up radio links, LIGHTBLUE can identify the interfaces for setting up different types of links based on the identified code that handles the HCI commands (step ❻). For instance, the host code issues the HCI command `HCI_Setup_Synchronous_Connection` to the firmware to establish an SCO link with another device.

Therefore, LIGHTBLUE can take the code that handles this HCI command (identified in step ❺) as one of the interfaces of the SCO link. After this step, LIGHTBLUE produces a patched version of the firmware (step ❼) by debloating the unneeded functionalities. Specifically, LIGHTBLUE debloats the code handling unneeded HCI commands, based on a list of HCI commands extracted during step ❹. LIGHTBLUE also removes the unneeded link interfaces based on the profile's specification, which indicates the needed links. Finally, the compiled host code and the patched firmware are linked and flashed on the Bluetooth device.

5.1 Profile Identification

Profile identification is the first step of the LIGHTBLUE pipeline. The goal of this step is to understand the profile needed by a specific high-level application (e.g., an Android app). Our key observation is that the host code provides fixed interfaces to the application so that the application can use the functionalities provided by the profile. For example, in Android, the app can call the `getProfileProxy()` API to get the interfaces of a specific profile. On Linux, the application can access the profile provided by the Bluetooth stack by accessing the relevant D-Bus [14] services. LIGHTBLUE identifies the profile by scanning for these interfaces in the application's code.

5.2 Host Code Analysis

A profile is exposed as a series of APIs (i.e., functions) to the application. For instance, the A2DP profile exposes 8 interfaces in Android 6.0.1 running on the Nexus 5 phone. LIGHTBLUE builds the profile-specific dependency graph starting from these exposed functions and generating a call graph that encompasses all the functions potentially reachable in the host code. At this stage, the call graph is built using a conservative approach. In particular, for each function, LIGHTBLUE scans each instruction belonging to it (Line 4 in Algorithm 1) and if a function is called or referenced, it is added to the call graph.

Obviously, this approach leads to major over approximations, especially since the host code contains "dispatching" functions, which are used by many profiles. In other words, the functions implementing the functionality of different profiles are coupled together, as described in Section 4. Listing 1 shows an example of one of these dispatching functions. In this example, the value of the variable `service_id` represents the profile that is currently being executed, and based on this value, different functions are called. Our initial analysis includes in the call graph all the functions potentially called by the function shown in the example. For this reason, it overapproximates the number of functions that are reachable when a specific profile is executed.

A traditional way [25, 36] to solve this issue is to perform a data-flow analysis from the entry point of the program. This data-flow analysis can detect that, if a specific profile is executed, some branches cannot be taken, and therefore, some function calls cannot happen. For instance, in the example, the data-flow analysis could understand that, when the Hands-Free Profile (HFP) is executed, the value of the `service_id` variable must be `BTA_HFP_SERVICE_ID`, and therefore, the `btif_in_execute_service_request` function cannot call the `btif_av_execute_service` function nor the `btif_av_sink_execute_service` function.

However, existing techniques cannot be directly applied since they assume that a program has a single-entry point. To solve this problem, LIGHTBLUE adds a dummy function invoking the different interface functions exposed by a specific profile, taking into consideration the ordering in which these interfaces are called by applications using the specific profile (Line 5 in Algorithm 1). For instance, the application needs to first initialize (`init()`) the profile, then connect to the remote device (`connect()`), disconnect and close the connection (`close()`) at last. Then, LIGHTBLUE takes the constant values within the profile interface functions as the source, and propagates them across the host code, using an approach similar to what is proposed in TRIMMER [36] (Line 6 in Algorithm 1).

Finally, LIGHTBLUE scans each function for conditional jumps and checks if the conditional value is known. If it is, LIGHTBLUE removes the basic blocks that are only reachable from the unsatisfiable branch of the conditional instruction. In turn, every edge in the call graph originated from a function call located in one of these removed basic blocks is removed as well, leading to a smaller, and more accurate, profile-aware call graph (Line 7 to Line 10 in Algorithm 1). Using the generated dependency graph, which is profile-aware, LIGHTBLUE removes all code that is not in the profile dependency graph.

```
bt_status_t btif_in_execute_service_request(
    tBTA_SERVICE_ID service_id, BOOLEAN b_enable){
  switch (service_id){
    case BTA_HFP_SERVICE_ID:
      btif_hf_execute_service(b_enable); break;
    case BTA_A2DP_SOURCE_SERVICE_ID:
      btif_av_execute_service(b_enable); break;
    case BTA_A2DP_SINK_SERVICE_ID:
      btif_av_sink_execute_service(b_enable); break;
    ....}
}
```

Listing 1: A code snippet from `btif_dm` in Android

Algorithm 1 Profile-aware analysis algorithm

```
1:  procedure ProfileAwareAnalysis(source, profile)
2:      for each v ∈ Variables do
3:          C[v] ← ∅
4:      D ← callgraphBuilding(source, profile)
5:      P ← stackTransformation(profile)
6:      C ← constantPropagation(P)
7:      for each F ∈ D do
8:          for each conditionalBranch ∈ F do
9:              if C[condition] ≠ ∅ then
10:                 pruneFunction(F, C[condition])
11:     return D
```

5.3 HCI Command Extraction

LightBlue leverages the well-defined HCI send/receive interfaces to extract the HCI commands from the dependency graph. LightBlue performs data-flow analysis from all the functions in the profile dependency graph to the HCI interfaces. Then, it obtains the used HCI commands, by recovering the first two bytes used to generate the HCI command packets. These two bytes contain the OGF and OCF fields of the packet, and they determine the invoked HCI command. For instance, if the first two bytes are `0x0405`, LightBlue recovers OGF and OCF by extracting the upper 6 bits and lower 10 bits. Based on the value of OGF and OCF (`0x1` and `0x5`), the HCI command is a `Create Connection` command.

5.4 Firmware Analysis and Patching

Given the firmware's binary, LightBlue first identifies the HCI command dispatcher, which dispatches the HCI command to different handlers. Once the dispatcher is located, LightBlue can further identify different HCI command handlers. From the identified HCI command handlers, LightBlue also recognizes the interfaces setting up different radio links (e.g., the SCO link). To know which ones are needed, LightBlue relies on the Bluetooth core specification [8] and the profiles' specifications.

LightBlue debloats the unneeded HCI commands by redirecting the handling of those commands to the error command handler and replacing the unneeded HCI command handler with dummy code. LightBlue employs the same approach to disable the interfaces of unneeded links. At last,

```
bt_status_t bthci_cmd_dispatcher(PTR* hci_cmd_pkt){
  opcode = *(hci_cmd_pkt + 9);
  OGF = opcode >> 10;
  OCF = opcode & 0x3ff;
  handler = error_cmd_handler;
  switch(OGF) {
    case 0x01: switch(OCF) {
        case 0x01: handler = handle_inquiry; break;
        ...}
    ...
    default: // handling error HCI command
      handler = error_cmd_handler; break;}
  handler(hci_cmd_pkt);
}
```

Listing 2: HCI command dispatcher example

LightBlue writes the patched binary back to the chip by a vendor-provided patching mechanism.

5.4.1 HCI Command Dispatcher Identification

Upon receiving different types of HCI commands, the firmware needs to parse their headers and handle them according to their opcode (Figure 2). Our key observation is that the dispatcher needs to perform bitwise operations to extract the `OGF` and `OCF` values from the opcode during parsing. This is due to the fact that, based on the `OGF/OCF`, the dispatcher either calls the corresponding handler directly or passes the handler to another function to execute. Listing 2 shows a simplified HCI command dispatcher example.

Dispatcher candidate scanning. The bitwise operation pattern that we use is the `OGF/OCF` extraction pattern from the opcode in the HCI command. Specifically, `OGF` is the upper 6 bits, and `OCF` is the lower 10 bits of the opcode. The extraction pattern consists of dividing a variable into two parts, one of which is the upper 6 bits and the other is the lower 10 bits.

To identify code exhibiting this pattern, for each function in the firmware, LightBlue marks every undefined reference as symbolic and performs symbolic execution. During the symbolic execution, if the aforementioned bitwise operation pattern is detected, the function is included in the candidate list. In addition, the source of the opcode is also identified (i.e., `hci_cmd_pkt+9` in the example code). The source of the opcode is further used to enable the dynamic under-constrained symbolic execution [31] and the binary patching described in the following sections.

Dispatcher candidate verification. LightBlue utilizes the semantic of different HCI commands defined in the specification to verify each candidate dispatcher and filter out false dispatchers. The specification mandates the `HCI_Read_BD_ADDR` and `HCI_Read_Local_Version_Information` commands to provide the Bluetooth MAC address and the device's manufacturer name. Exploiting this semantic information, we start under-constrained symbolic execution of each dispatcher candidate by setting the value of the opcode to `HCI_Read_BD_ADDR` first and

HCI_Read_Local_Version_Information later. If the Bluetooth MAC address and the manufacturer name are accessed during the two executions, we flag the analyzed candidate as the HCI dispatcher function, otherwise we discard it.

Algorithm 2 shows how LIGHTBLUE identifies the dispatcher through pattern scanning and candidate verification. We highlight that the implemented algorithm does not depend on any specific firmware implementation, and as we will show, it reliably works on a large variety of different implementations.

Algorithm 2 Dispatcher identification algorithm

```
1:  procedure IDENTIFYDISPATCHER(FW: Firmware)
2:      func_list ← IdentifyFunctions(FW)
3:      predef_val ← btDefinedValues
4:      candidate ← ∅
5:      dispatcher ← ∅
6:      for each func ∈ func_list do
7:          if PatternDetected(func) then
8:              op_src ← IdentifySource(func)
9:              candidate.add((func, op_src))
10:     for each (func, op_src) ∈ candidate do
11:         op_src ← informationalHCICmds
12:         acc_val ← SymbExec(func, op_src)
13:         if predef_val ∈ acc_val then
14:             dispatcher.add(function)
15:     return dispatcher
```

5.4.2 HCI Command Handler Identification

Once the dispatcher is identified, LIGHTBLUE symbolically executes the dispatcher multiple times, by concretizing the source of command's opcode with the value of all the possible opcodes corresponding to the different HCI commands.

For each execution with a different concrete opcode, LIGHTBLUE records all the visited functions, creating a set of functions corresponding to each considered opcode. Then, in each generated function set, LIGHTBLUE identifies the specific HCI command handler function by removing from it all the functions that are present in any other function set. The unique function in each generated function set is identified as the specific HCI command handler.

5.4.3 Link Interface Identification

Once the HCI command handlers are identified, LIGHTBLUE leverages the semantics of these HCI command handlers obtained from the Bluetooth specification to further identify the interfaces of different links. For example, we know from the specification that the opcode of 0x0428 refers to the HCI_Setup_Synchronous_Connection HCI command that starts an SCO link with another device. Therefore, we can create the mapping between the HCI command handler handling the HCI command whose opcode is 0x0428 and the link it uses. In this example, LIGHTBLUE marks this HCI command handler as one of the interfaces of the SCO link.

We manually analyze every HCI command in the specification to create the mapping beforehand for LIGHTBLUE to identify all the interfaces of different links. Table 8 shows the links and corresponding interfaces. We also manually create the mapping between the profile and the links that the profile depends on based on the profile's specification. We note that the mapping creation is a one-time effort, and LIGHTBLUE reuses the mapping when debloating profiles on all platforms.

5.4.4 Firmware Patching

Once the HCI command dispatcher and handlers are identified, LIGHTBLUE performs firmware patching to debloat the firmware. LIGHTBLUE inserts a snippet of binary code at the beginning of the HCI command dispatcher function, which modifies the unneeded HCI command's opcode to an invalid opcode based on the observation that the dispatcher function first checks whether the opcode is valid or not. Thus, the unneeded HCI command will be handled by the error handler instead of the original one. We employ the same approach to debloat the interfaces of the unneeded links.

This approach has two main advantages. First, the debloated firmware still emits an HCI event (e.g., HCI_Command_Complete) to the host to remain specification-compliant, preventing the firmware from crashing when it receives debloated HCI commands from the host. Second, this approach is applicable to the platforms that have limited patching capabilities (e.g., the popular BCM4339 chip and similar Broadcom chips) since the firmware needs to be patched at only one place. In fact, for these devices the firmware is stored in a non-reprogrammable memory, and we need to use vendor-specific mechanisms to patch the firmware (e.g., the *patchram* [26] mechanism of Broadcom chips).

Conversely, if the controller allows unlimited patching (e.g., the firmware is held in reprogrammable flash memory), we modify it more extensively. In these cases, LIGHTBLUE also replaces the functions of each unneeded HCI command handler and link interface with dummy code (e.g., bx lr for ARM).

6 Implementation

We implement the host code analysis as an LLVM pass on top of LLVM 9.0, using about 2.3 KLOC. The firmware analysis and patching are implemented with Python based on angr [37], using about 1.2 KLOC.

Profile identification implementation. LIGHTBLUE identifies the needed profile by scanning for the APIs of interest and performing static analysis. For example, LIGHTBLUE scans for the getProfileProxy() API used by Android apps and traces back the third argument which indicates the needed profile. We highlight that we demonstrate the feasibility of profile identification for Android apps in Section 7.1. We leave the

Table 1: Host code and Bluetooth chips on our evaluation platforms. The Bluetooth firmware on Plt. 4 is not available. AC: ARM Cortex

#	Device	Host OS	Host Stack	BT Chip	Processor
Plt. 1	Nexus 5	Android 6.0.1	BlueDroid	BCM4339	AC M3
Plt. 2	Raspberry Pi 3	Raspbian 9	BlueZ 5.52	BCM43430A1	AC M3
Plt. 3	Dell Laptop	Ubuntu 18.04	BlueKitchen	CYW20735B1	AC M4
Plt. 4	Google Pixel 3	Android 9.0.0	Fluoride	Kryo 385	AC A75

profile identification for additional types of applications as future work.

Host code analysis implementation. LIGHTBLUE compiles and links the host source code using Clang and generates the LLVM Intermediate Representation (IR). Then, LIGHTBLUE runs the LLVM pass to generate the profile dependency graph and removes unneeded code. After the code removal, LIGHTBLUE generates the object file and the linker links the object file to generate the binary file.

Firmware analysis implementation. After dumping the firmware with a vendor-specific method, LIGHTBLUE recovers the functions and builds the firmware's call graph using angr. Then, LIGHTBLUE automatically identifies the HCI command dispatcher and handlers using angr's symbolic execution (as discussed in Section 5.4.1 and 5.4.2). During symbolic execution, to verify the dispatcher candidates we set the maximum function call depth to 0 to avoid state explosion.

Then LIGHTBLUE automatically verifies each dispatcher candidate from the candidate list. If no dispatcher is found, we increase the maximum function call depth by 1 and do symbolic execution again until we find a dispatcher. After that, the maximum function call depth, which successfully confirms the dispatcher, is used as the maximum function call depth in identifying the HCI command handler. For collecting the accessed values during symbolic execution, we also consider the dispatcher candidate's return value as a function pointer which accesses the defined values.

Once the HCI command handlers are identified, LIGHTBLUE also identifies the interfaces of different links based on the mapping between the handlers and their semantics (see Table 8), as discussed in Section 5.4.3. Then LIGHTBLUE identifies the needed link of the profile according to the specification and marks the interfaces of other links as unneeded. Finally, LIGHTBLUE patches the firmware using the vendor-provided approaches (e.g., *patchram*), as discussed in Section 5.4.4.

7 Evaluation

We evaluate LIGHTBLUE on several popular platforms across different Bluetooth hosts and controllers. Specifically, we test LIGHTBLUE on 3 full Bluetooth stacks used by different devices: a Nexus 5 phone, a Raspberry Pi 3, and a Dell Latitude laptop, as shown in Table 1. The Nexus 5 (Platform 1 in the table, Plt. 1 for short) has BlueDroid [15] as the host code and the Broadcom BCM4339 [9] Bluetooth chip. The Raspberry Pi 3 (Plt. 2), uses BlueZ 5.52 for the host and the BCM43430A1 chip [33]. The Dell laptop (Plt. 3) employs BlueKitchen [20] as the host code and the CYW20735B1 [12] chip. We choose these three platforms due to their availability of both the host code (source code) and the firmware binary. All three pieces of firmware that we analyze are proprietary and closed source, and they run on different chips. Additionally, we also evaluate LIGHTBLUE on a Google Pixel 3 (Plt. 4) running Fluoride [16] as the host code to show that host code debloating can work separately.

Among the steps shown in our pipeline (Figure 4), all steps are automated on Plt. 1. As we mentioned in Section 3, in most cases, the user of LIGHTBLUE is aware of the profiles to keep, and therefore, the profile identification step (step ❶) is not implemented for Plt. 2 and Plt. 3. All the other steps on Plt. 2 and Plt. 3 (steps ❷ to ❼) are automated. On Plt. 4, for which we only perform host code debloating (due to the unavailability of its firmware), all the steps needed (steps ❶ - ❹) are automated. We note that the user could manually specify the profiles that the user wants to keep on all platforms. In fact, we envision different usage scenarios in which LIGHTBLUE potential users are aware of the needed profiles, as we discussed in Section 3.

We first demonstrate LIGHTBLUE's real-world practicality by investigating the usage of Bluetooth profiles of Android apps. Then, we evaluate LIGHTBLUE along three different aspects, the correctness of the debloated stack (i.e., its ability to work correctly when supporting app code using a single profile), the attack surface reduction on both the host code and the firmware, and the number of prevented CVEs. To show the generality of the HCI command handler identification, besides the three pieces of firmware shown in Table 1, we also evaluate the HCI command handler identification on the firmware of Zephyr [10].

7.1 Profile Identification of Android Apps

We evaluate the profile identification on Android by analyzing 10,650 popular apps automatically crawled from AndroidZoo [1] during January and February 2020. Among our dataset, 935 apps require Android's Bluetooth permission (thus they can access Bluetooth functionality), out of which 432 apps are detected as using Bluetooth profiles. We find that more than 90% of the 432 apps only use 1 or 2 profiles. Besides, as shown in Table 6 in Appendix A, some profiles are rarely used, such as SAP and SPP. Only three profiles (i.e., A2DP, HFP, and GATT) are frequently used by the apps. Therefore, most of the profiles are not needed and can be debloated under a particular Bluetooth use scenario.

During the identification, we find that there are apps using reflection together with string operations (e.g., appending) to load Bluetooth-related classes and profiles, which LIGHTBLUE cannot identify. We highlight that, LIGHTBLUE

Table 2: Applications that are used to test debloated Bluetooth stacks on different platforms. For all tested end devices, we did not observe unexpected results (i.e., issues due to debloating). N/A = untested as we did not have required devices.

Platform	Application	Profile	No Crash	No Issue
Plt. 1	Spotify	A2DP (AVRCP)	✓	✓
	Phone (Built-in)	HFP	✓	✓
	Bluetooth Tethering Manager	PAN	✓	✓
	Bluetooth (Built-in)[1]	HID	✓	✓
	Samsung Health	HDP	✓	N/A
	nRF Connect for Mobile	GATT	✓	✓
Plt. 2	blueman	A2DP (AVRCP)	✓	✓
		PAN	✓	✓
		HID	✓	✓
		HDP	✓	N/A
		SAP	✓	N/A
Plt. 3	a2dp_sink_demo	A2DP (AVRCP)	✓	✓
	hfp_hf_demo	HFP	✓	✓
	panu_demo	PAN	✓	✓
	hid_keyboard_demo	HID	✓	✓
	hsp_hs_demo	HSP	✓	✓
	spp_streamer	SPP	✓	N/A
	pbap_client_demo	PBAP	✓	N/A
	gatt_browser	GATT	✓	✓

1: If the built-in Bluetooth app supported profiles (e.g., HID profile) are removed from the Bluetooth stack, the built-in Bluetooth app does not crash and cannot set up connections for the removed profiles.

follows the developers' guidelines to identify the profile, while reflection is not the recommended approach to use Bluetooth profiles by Google [21]. Besides, the presence of obfuscation may fail our profile identification.

7.2 Correctness of Debloating

To test the correctness of the debloated stack, we run different apps that use distinct profiles on each platform to check whether the app and the Bluetooth stack crash or whether the app can communicate through the needed profile correctly. Besides, we also test if, when one profile is debloated, the debloated platform can still use that profile. Through our experiment, we find that A2DP and AVRCP profiles are tightly coupled, and the AVRCP profile is not functioning without A2DP on all platforms. Because AVRCP is used to control the audio playback, there would be nothing to control without A2DP that transmits the audio playback. Therefore, we consider A2DP and AVRCP as one profile in our evaluation. Table 7 in Appendix A shows the profiles and their corresponding functionalities to give intuition to the user about the functionality removed when debloating a profile.

For Plt. 1, we pick 6 popular apps that use different Bluetooth profiles, run the apps with the debloated host code and firmware, and test whether the apps can run and communicate with another Bluetooth device without any issues. The same apps are used to test the debloated host code on Plt. 4. We use the `blueman` Bluetooth manager with our debloated host code and firmware on Plt. 2 to connect to different Bluetooth devices via different profiles and to test whether the debloated stack works correctly. At last, we compile and run the demo app code using different profiles in BlueKitchen on Plt. 3 with the debloated development board as the Bluetooth chip to test whether the app can run and connect to other Bluetooth devices. The tested apps are shown in Table 2.

We use these apps to connect to different Bluetooth devices including a laptop, a headset, a keyboard, and a BLE device to test the A2DP, AVRCP, PAN, HSP, HFP, HID, MAP, and GATT profiles. For other profiles (i.e., HDP, SAP, SPP, and PBAP) which we do not have physical devices to run, we test whether the app and the stack crash or not.

Throughout our experiment, all apps can be executed on all platforms without any crashes, and the apps can communicate with other Bluetooth devices through the debloated Bluetooth stack without any observed issues.

We test LIGHTBLUE with the Square Android app [39] running on a Nexus 5 and the Square Reader [40]. LIGHTBLUE first analyzes the Square app and identifies the profile (i.e., the GATT profile) needed by the app. Then LIGHTBLUE debloats the host code and the firmware used by the Nexus 5 phone. At last, we run the app to connect to the Square reader, and we verify that the app is still functioning correctly after debloating, as shown in Figure 3.

To test whether the debloated profiles are actually removed and no longer available, we use `sdptool` [17] to get all profiles on the test platforms and check whether the debloated profiles can still be accessed. As expected, all the debloated profiles are no longer available on the test platforms.

7.3 Attack Surface Reduction

In this section, we evaluate LIGHTBLUE by presenting the attack surface reduction rate when debloating different profiles using LIGHTBLUE. We show the code reduction rate of both the host code and the firmware.

7.3.1 Bluetooth Host Code

We evaluate our host code analysis on four different platforms as shown in Table 1. We first show the attack surface reduction when keeping one profile, then we evaluate LIGHTBLUE when keeping multiple profiles. We evaluate the host code analysis using three different metrics: reduced number of instructions, reduced number of functions, and reduced number of ROP gadgets. The reduction of ROP gadgets is not a perfect metric for evaluating the attack surface reduction However, it is widely used [23, 29, 30, 36] and we follow prior literatures to use this metric.

Keeping one profile. In this evaluation, LIGHTBLUE keeps one needed profile and removes the others. Our baseline (100%) is the original host code enabling all profiles, and

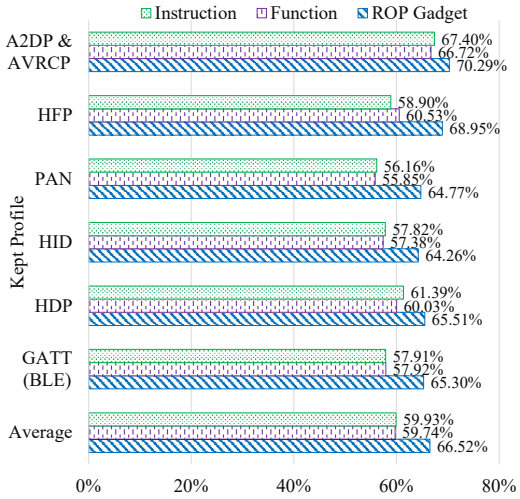

Figure 5: Comparison between debloated host code (keeping different profiles) and the Baseline on Plt. 1 (BlueDroid).

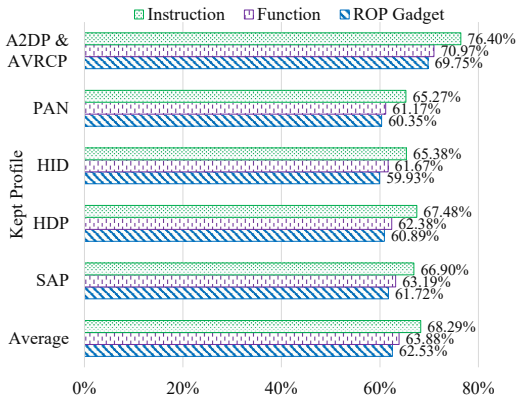

Figure 6: Comparison between debloated host code (keeping different profiles) and the Baseline on Plt. 2 (BlueZ).

we evaluate keeping, one-by-one all the supported profiles. We compile the binaries with the same optimization level.

The reduced attack surface of the host code on the four tested platforms is shown in Figure 5, 6, 7, and 8 respectively. As we can see from Figure 5, the average reduced instructions, functions, and ROP gadgets of BlueDroid on Plt. 1 are 40.07%, 40.26%, and 33.48%. The reduced attack surface is also different by keeping different profiles. Figure 6 shows the reduced attack surface of BlueZ by keeping different profiles. The average reduced instructions, functions, and ROP gadgets are 31.71%, 36.12%, and 37.47%. The average reduced instructions of BlueKitchen on Plt. 3 is 49.12%, while the reduced functions and ROP gadgets are 50.03% and 52.13% as shown in Figure 7. As shown in Figure 8, the average reduced instructions, functions, and ROP gadgets of Fluoride on Plt. 4 are 33.68%, 41.53%, and 30.07%.

Overall, BlueKitchen has the highest host code reduction rate among all the test platforms. One reason is that BlueKitchen is designed for low-end IoT devices which only have 1 profile. Therefore, the profiles are not coupled together as the host code on other platforms. Another reason is that BlueKitchen has the largest number of supported profiles, which enlarges the code baseline, and therefore, more code is removed when only 1 profile is kept.

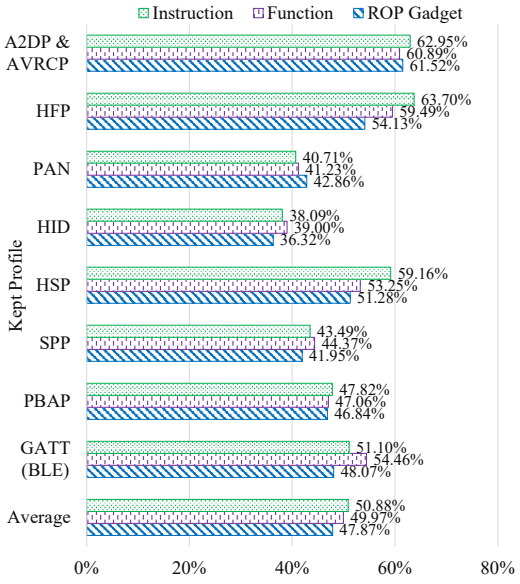

Figure 7: Comparison between debloated host code (keeping different profiles) and the Baseline on Plt. 3 (BlueKitchen).

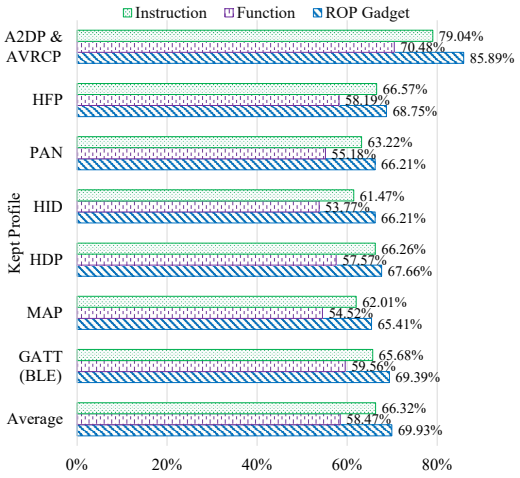

Figure 8: Comparison between debloated host code (keeping different profiles) and the Baseline on Plt. 4 (Fluoride).

Keeping multiple profiles. LIGHTBLUE also supports keeping multiple profiles. It is challenging to evaluate all possible profile combinations on the tested platforms. Therefore, we select the top 4 of the most popular profile combinations found during our profile identification step on Android (Section 5.1). In addition, we test one additional common use case (using headset and keyboard, requiring the A2DP and HID combi-

Table 3: Reduced attack surface percentage of host code when keeping multiple profiles on Plt. 1 and Plt. 4.

Profile Combination	Used by # of Apps	Reduced Attack Surface					
		Plt. 1			Plt. 4		
		Instruction #	Function #	ROP Gadget #	Instruction #	Function #	ROP Gadget #
A2DP & HFP	71	27.05%	26.33%	23.31%	15.82%	24.97%	9.80%
GATT & HFP	22	36.49%	34.62%	27.82%	29.13%	35.80%	26.50%
A2DP & GATT & HFP	18	22.57%	22.01%	18.56%	11.52%	18.96%	5.12%
A2DP & GATT	13	28.15%	28.96%	24.94%	16.59%	23.38%	9.07%
A2DP & HID	-	28.25%	29.49%	26.57%	20.81%	29.17%	13.47%

nation of profiles) to measure the reduced attack surface on Plt. 1 and Plt. 4. Table 3 shows the reduced attack surface results in these scenarios. We can see from the table that the reduced attack surface drops slightly compared with keeping only one profile. For example, compared with keeping only the A2DP profile, the percentage of removed instructions drops slightly from 32.60% to 27.05% when keeping A2DP and HFP simultaneously on Plt. 1.

7.3.2 Bluetooth Firmware

Among the three types of HCI packets, HCI commands are the only input to the firmware from the host code. Besides, the interfaces are also the code that handles HCI commands. Therefore, we use the number of disabled HCI command handlers in the firmware as a metric to evaluate the reduced attack surface of the firmware.

Through our experiment, we find that different profiles are using the same set of HCI commands based on the host code analysis. This happens because all the profiles use L2CAP as the transport protocol, which uses the same HCI commands to establish different connections. In addition, since all the tested platforms support Bluetooth Classic and BLE, the initializations of Bluetooth Classic and BLE stacks are tightly coupled. Therefore, even though only the A2DP profile is needed, the initialization of BLE is also executed. During this initialization, HCI commands of both Bluetooth Classic and BLE HCI are needed.

The analyzed firmware usually supports far more HCI commands than needed, which enables a significant amount of debloating, as we will show later in this section. In addition, the link interface identification and debloating enables additional firmware debloating. Therefore, LIGHTBLUE can further debloat HCI command handlers that handle these overestimated yet unneeded HCI commands in the firmware.

Table 4 shows the results of debloating in terms of removed HCI commands, on the three tested platforms. We manually check each platform to get the number of HCI commands that are processed in the original firmware and host code. We group the profiles in the table based on the links they need since the profiles that use the same link share the same interfaces (HCI commands).

We notice that removing HCI command handlers in the firmware also prevents the firmware from sending corresponding HCI events to the host code, and therefore, prevents those events from being processed by the host code. To better quantify this aspect, in the table, we specified how many debloated commands and corresponding events were processed by the host code before debloating takes place ("# of Cmds Processed by Host and Removed by Debloating" row). We noticed that a significant fraction of the removed HCI commands were not processed by the host code even before debloating. One reason for this aspect is that many of the debloated commands in the firmware are "vendor-specific." Commands of this class are used for firmware update, debugging, or adding new features, and they are typically not processed by the host code. The table lists the exact number of vendor-specific HCI commands we encountered in the debloated firmware samples.

The removal of some HCI command handler in the firmware may not affect (negatively or positively) the security of the host code. However, we note that LIGHTBLUE implements the HCI command handler removal functionality primarily to improve the security of the controller (while the security of the host is improved by debloating the host code, as evaluated in Section 7.3.1). In fact, debloating these unneeded HCI command handlers reduces the attack surface of the firmware, regardless of whether they are used in the host code or not. Besides, debloating the unneeded link interfaces, which are also HCI command handlers, can prevent the attacker from triggering certain vulnerabilities in the firmware, including CVE-2019-13916 (see Section 7.4).

In all the tested platforms and usage scenarios LIGHTBLUE removes more than half of the processed HCI commands. It is noteworthy that in BlueKitchen we can debloat a higher percentage of HCI commands. This happens because BlueKitchen is designed for low-end IoT devices that usually use one profile, and, therefore, the HCI commands for different profiles are not tightly coupled.

7.4 Preventing Bluetooth CVEs

At the time of writing, there are 15 CVEs about the Plt. 1 host code, out of which 11 can be removed via LIGHTBLUE. There are 23 CVEs about the Plt. 4 host code, out of which 8 can be removed through LIGHTBLUE. No CVE exists on Plt. 2 and Plt. 3 host code. There are 3 reported CVEs in the firmware of our tested platforms, out of which 1 can be prevented. Table 5 lists the CVEs that can be prevented by LIGHTBLUE. These CVEs are related to different profiles and functions

Table 4: Number of debloated HCI commands on different platforms.

Platform	**Plt. 1**			**Plt. 2**			**Plt. 3**		
Host Code	BlueDroid			BlueZ			BlueKitchen		
Bluetooth Chip	BCM4339			BCM43430A1			CYW20735B1		
# of Cmds Processed by Firmware	310			299			423		
out of which vendor-specific	*135*			*93*			*174*		
# of Cmds Processed by Host Code	138			144			131		
Kept Profile	HFP	GATT	Others[1]	HFP	GATT	Others	HFP	GATT	Others
Needed Link(s)	ACL & SCO	LE ACL[2] & ADVB[3]	ACL	ACL & SCO	LE ACL & ADVB	ACL	ACL & SCO	LE ACL & ADVB	ACL
# of Cmds Processed by Firmware and Removed by Debloating	192	196	195	171	172	174	352	354	354
out of which vendor-specific	*125*	*125*	*125*	*88*	*88*	*88*	*171*	*171*	*171*
# of Cmds Processed by Host Code and Removed by Debloating	20	24	23	16	17	19	60	62	62
# of Cmds Removed by Debloating	192 (64.2%)	196 (65.6%)	195 (65.2%)	171 (57.2%)	172 (57.5%)	174 (58.2%)	352 (83.2%)	354 (83.7%)	354 (83.7%)

1. Other profiles supported on the platform. 2. Low Energy Asynchronous Connection. 3. LE Advertising Broadcast link.

Table 5: Prevented CVE vulnerabilities and related profiles. N/A: not related to any profile. HO: the vulnerability is in the host code. FM: the vulnerability is in the firmware.

Vul. Loc.	**Related Profile**	**Platform**	**# of Vul. Functions**	**CVE Number**
HO	**A2DP & (AVRCP)**	Plt. 1	4	CVE-2018-9542*
				CVE-2018-9450*
				CVE-2017-13266*
				CVE-2018-9453
		Plt. 4	7	CVE-2019-2227*
				CVE-2018-9588*
				CVE-2018-9507*
				CVE-2018-9506*
				CVE-2019-2049
	PAN	Plt. 1	7	CVE-2017-0783*
				CVE-2017-0782*
				CVE-2017-0781*
				CVE-2018-9436*
				CVE-2018-9356*
				CVE-2018-9357
				CVE-2017-13269
	MAP	Plt. 4	1	CVE-2018-9505*
	HSP/HFP	Plt. 4	5	CVE-2018-9583*
	N/A	Plt. 4	1	CVE-2019-2226
FM	**GATT**	Plt. 3	1	CVE-2019-13916*

*: CVEs that can be triggered by over-the-air attacks.

which can be removed or prevented by debloating the corresponding profiles. Besides, LIGHTBLUE can also remove the code that is not needed by *any* profile when debloating a profile. Therefore, LIGHTBLUE can remove some vulnerabilities (e.g., CVE-2019-2226), regardless of the considered profile. Since there are different CVEs in different profiles, the vulnerabilities that can be removed by keeping different profiles are also different. We highlight that LIGHTBLUE can also potentially remove undiscovered vulnerabilities by debloating unneeded profiles.

It is noteworthy that LIGHTBLUE can protect both the host code and the firmware from over-the-air attacks (the CVEs with $*$ in Table 5). LIGHTBLUE protects the host code from over-the-air attacks by removing the relevant vulnerable code. For example, Section 8.1 shows how all the PAN-profile-related BlueBorne [5] vulnerabilities that can be triggered by over-the-air attackers can be removed by debloating the PAN profile completely. LIGHTBLUE protects the firmware from over-the-air attacks by disabling unneeded link interfaces. For instance, debloating the GATT profile would also disable the creation of BLE ACL and BLE Advertising Broadcast (ADVB) links since GATT is the only profile that uses these links. Therefore, LIGHTBLUE prevents the vulnerabilities inside the firmware triggered by malicious packet via these links, such as CVE-2019-13916. Section 9 discusses how to defend against more over-the-air attacks.

7.5 Accuracy of HCI Command Handler Identification in Firmware

To show the generality of LIGHTBLUE, we evaluate the accuracy of the HCI command handler identification on four different firmware. Besides the three pieces of firmware tested in Table 4, we also evaluate our automatic HCI handler identification accuracy on the open-source Zephyr firmware.

We obtain the ground truth about the Plt. 1 and Plt. 2 firmware by reverse-engineering them, helped by the InternalBlue [26] tool. Since the CYW20735B1 chip (used on Plt. 3) is a development board, for which debugging symbols are available, we get its ground truth using the symbols. The ground truth of Zephyr is obtained from its source code.

LIGHTBLUE can automatically detect all the HCI command handlers with 100% accuracy on our tested platforms. Yet, it may fail to identify the HCI command handlers on other platforms if the firmware employs complicated logic in the HCI command dispatcher function that leads to state explosion for angr, or if angr fails to decompile the firmware.

8 Case Study

In this section, we show how LIGHTBLUE can prevent real-world vulnerabilities, such as BlueBorne [5] and BadBluetooth [46], by presenting two case studies in detail.

8.1 Removal of BlueBorne CVEs

BlueBorne [5] is an airborne attack vector discovered by Armis in 2017. Based on the found vulnerabilities, the attacker can exploit vulnerable Bluetooth devices remotely. BlueBorne comprises several CVEs including CVE-2017-0781, CVE-2017-0782, and CVE-2017-0783, which affect devices running Android version 4.4.4 to 8.0. All these three CVEs are related to the PAN profile or the BNEP protocol which the PAN profile is built upon.

LIGHTBLUE can remove all these vulnerabilities by debloating the PAN profile. Specifically, the BNEP protocol would be removed since PAN is the only profile that needs BNEP. Therefore, when the adversary sends the malicious packets to trigger the remote code execution vulnerability, the connection cannot be established because of the unavailability of BNEP. The man-in-the-middle (MITM) attack will also fail because of the debloating of the PAN profile. Considering that the PAN profile is rarely used (in our dataset, it is used by only two apps, out of more than 10K), LIGHTBLUE can effectively defend against these vulnerabilities in common usage scenarios.

8.2 Defence Against BadBluetooth Attacks

BadBluetooth [46] introduces a new type of attack on Android smartphones from a malicious Bluetooth device. This attack is based on the weakness of the design that the Bluetooth profile authentication process is coarse-grained: the device still trusts the paired device (including all profiles) even if the paired device changes its profiles after pairing. Therefore, a malicious Bluetooth device can first use a user-expected profile to pair with a smartphone, and then switch to other profiles silently, to launch the attack on the smartphone. For example, a malicious Bluetooth speaker can pair with the smartphone using the A2DP profile at first, and then switch to the HID profile to inject input events to the smartphone.

LIGHTBLUE can naturally defend against this kind of attack. Taking the malicious Bluetooth speaker as an example, LIGHTBLUE can identify from the application that only the A2DP profile is needed and debloat the other profiles. After the smartphone pairs with the malicious speaker, the speaker changes profile to HID and try to inject malicious input events to the smartphone. However, since the HID profile is debloated and no longer supported by the smartphone, the injection fails, and the attack cannot be launched. At the same time, in the debloated smartphone, the audio transmission can still be functioning since the smartphone and the speaker can still communicate through the A2DP profile.

Though LIGHTBLUE cannot completely mitigate BadBluetooth attacks when enabling multiple profiles, it can still prevent some BadBluetooth attacks based on the removed profiles. Suppose that LIGHTBLUE is used to keep the A2DP and HFP profiles (the most popular combination of profiles). In this case, the BadBluetooth attack can be used to switch between these two profiles but cannot activate other Bluetooth functionality requiring another profile. For example, in the mentioned scenario, the attacker could use BadBluetooth to inject malicious voice commands to the voice assistant on the smartphone using the HFP profile but cannot inject malicious keystrokes to the smartphone, since this attack requires using the HID profile debloated by LIGHTBLUE. Similarly, LIGHTBLUE would prevent launching MITM attacks since the corresponding PAN profile is debloated by LIGHTBLUE.

9 Discussion and Limitation

Extending over-the-air protection. As discussed in Section 7.4, LIGHTBLUE can protect both the host and the controller from over-the-air attacks by removing unneeded code and debloating unneeded links. However, LIGHTBLUE cannot prevent all over-the-air attacks affecting the controller due to its top-down debloating approach (from the profile side of the host code to the firmware). Out of three known vulnerabilities affecting the controller of the considered devices (CVE-2019-13916, CVE-2019-18614, and CVE-2019-11516) [34], LIGHTBLUE can automatically prevent one (CVE-2019-13916, as explained in Section 7.4).

To further explore this issue, we studied the other two vulnerabilities and manually patched them. Specifically, we reverse-engineered the firmware of Plt. 1 and Plt. 3 to identify the radio-level interface. Based on the identified interface, we implemented patches for CVE-2019-18614 (on Plt. 3) and CVE-2019-11516 (on Plt. 1), both of which can be triggered over-the-air. For Plt. 3, we leveraged debug symbols to identify the radio-level interface for patching CVE-2019-18614. For Plt. 1, since there are no debug symbols, we used LIGHTBLUE to help us manually identifying the radio-level interface. In particular, LIGHTBLUE automatically identified the HCI command handler enabling the controller to receive data that can trigger the vulnerability. Following functions called by this command handler, we further reverse-engineered the firmware, located the targeted interface, and inserted additional length checks to patch CVE-2019-11516.

As shown, LIGHTBLUE can prevent some vulnerabilities affecting the controller automatically, and help an analyst to patch others manually. However, to fully handle these vulnerabilities, a complementary bottom-up approach (from the radio side of the firmware to the host code), which needs to identify the radio-level interfaces (i.e., the code dispatching packets coming from the radio interface), is needed.

Our study of the firmware revealed that, while LIGHTBLUE can automatically identify the HCI handler code in the firmware, automatically identifying the radio-level packet dispatcher code poses additional challenges. In fact, different from the HCI handler case, there is no well-defined specification on how this code is supposed to behave, and therefore,

this code can be implemented in different ways by different vendors. Consequently, implementing an automated radio-level dispatcher identification procedure remains an open challenge. In addition, even if the radio-level dispatcher is detected, an additional issue is to identify a set of properties of the incoming radio packets that can be used to effectively classify them and discard those triggering vulnerable code.

Configuration vs. debloating. Among our tested platforms, BlueDroid [15], Fluoride [16], and BlueZ [11] support disabling specific functionalities via configuration files during compilation. However, this configuration approach is not general, since it heavily depends on the stack implementation. For instance, BlueKitchen [20] does not support configuration at compilation time, and therefore, cannot be debloated via the configuration approach. Besides, some functionalities, such as the HFP profile, cannot be debloated even when the configuration approach is possible. Additionally, disabling a functionality does not remove all the related code, limiting the effectiveness of this approach. Finally, the configuration file approach is only applicable when source code is available, and therefore, it cannot be used to debloat the firmware. LIGHTBLUE, on the other hand, provides a general approach that can debloat unneeded functionality in both the host code and the firmware.

Blocking code path vs. debloating. Blocking the code path (e.g., introducing a new access control mechanism or dynamically enabling/disabling profiles) can also defend attacks like BadBluetooth [46]. However, the code-path blocking approaches cannot reduce the attack surface of the executable in memory or maybe even increase the attack surface due to the newly introduced mechanisms. Therefore, code-reuse attacks (e.g., ROP) can still exploit the executable gadgets of blocked profiles and newly introduced mechanisms.

Besides, blocking the code path cannot prevent attacks that jump to an existing function to perform their malicious operations (e.g., control flow hijacking attacks). Therefore, we implemented LIGHTBLUE so that it not only blocks the code path (i.e., disabling interfaces of unneeded profiles) but also reduces the attack surface of the executable by removing the unneeded code from the host code and the firmware (when a suitable firmware patching method is available).

Extending LIGHTBLUE to other protocol stacks. Theoretically, LIGHTBLUE is applicable to other stacks if they have a similar structure to the Bluetooth stack. The host code analysis can be applied to debloat such stacks if they have multiple functionalities exposed to the upper application layer via predefined interfaces (as discussed in Section 5.2).

For example, we envision extending LIGHTBLUE to support devices using the Near-Field Communication (NFC) protocol. The NFC protocol implements multiple functionalities, such as card emulation and peer-to-peer communication, but normally not all of them are used. In this case, the host code analysis could be applied to the NFC host code debloating the unused functionality (e.g., card emulation). The firmware analysis can also be applied to other stacks if the interface between the host code and the firmware is well defined. For instance, the firmware analysis can be applied to NFC firmware based on the well-defined NFC Controller Interface (NCI) [28] between the host and the controller. Similarly, LIGHTBLUE could be potentially applicable to Wi-Fi and 2/3/4/5G on Android, exploiting the separation between the host code and the firmware, i.e., Wi-Fi Hardware Abstraction Layer and Radio Interface Layer.

Extending profile identification. As mentioned in Section 6, LIGHTBLUE identifies the needed profile of Android apps by static analysis. The analysis fails when the application's Bluetooth profile usage cannot be determined statically, or obfuscation techniques are present. Moreover, LIGHTBLUE currently does not support profile identification other than on Android apps. We note that, however, profile identification implemented on Android is primarily to show that Bluetooth debloating is feasible because most Android apps only use a limited set of Bluetooth functionality. We plan to support automated profile identification of other types of apps in the future. We highlight that the user can always input the profiles instead of automatically identifying when the user is aware of the needed profiles.

Debloated stack testing coverage. We test the correctness of the debloated Bluetooth stack by checking whether the kept profile still works after debloating, as discussed in Section 7.2. We did not exhaustively test (e.g., fuzzing) the whole debloated Bluetooth stack for a long time. As future work, we could implement automated testing (e.g., fuzzing) of the debloated code, but fuzzing the entire Bluetooth stack is out of scope for this paper.

Usability. LIGHTBLUE is not mainly for general users (e.g., consumers) in its current implementation, since it requires actions such as rooting a phone to install the modified host code and firmware. In addition, usability of LIGHTBLUE might be limited in dynamic usage scenarios, in which the user frequently changes the needed functionality, since it requires reloading the host code frequently. Furthermore, the debloating of the firmware might not be possible if the Bluetooth controller vendors prevent firmware modifications (e.g., employing firmware integrity verification mechanisms). In summary, LIGHTBLUE is ideally suited for devices that serve a specific purpose and require a specific subset of the Bluetooth functionality, such as the use cases discussed in Section 3 or Bluetooth-enabled IoT devices (e.g., IoT devices using BlueKitchen).

10 Related Work

Program debloating. The following works discuss the debloating of binaries. Qian et al. [29] introduced a debloating framework for deployed binaries based on dynamic tracing.

Heo et al. [23] built a framework to debloat program based on reinforcement learning. Redini et al. [32] presented a debloating tool based on a new abstract domain. Debloating with source code was discussed by Quach et al. [30], who built a framework to remove the unneeded code, operating both at compile and load time. However, that framework needs a specific loader to load the binary. The works closest to ours are TRIMMER [36] and [25]. Both works only debloat single-entry programs and require knowing the inputs received by the program. However, the Bluetooth host code has multiple entries and does not take input directly. In addition, the listed prior works cannot debloat program code across different layers (the host code and the firmware).

Firmware analysis. We now discuss recent works that focus on firmware analysis both dynamically and statically. For dynamic analysis, Feng et al. [19] built a test framework to execute and fuzz the firmware by abstracting the diverse peripherals and handling I/O operations on the fly. Avatar [47] introduced a framework combining emulation and real device to successfully execute the firmware by forwarding peripheral accesses to the real device. Mantz et al. [26] build a patching and testing framework specifically for Broadcom Bluetooth chip. Examples for static analysis approaches are FIE [13] and FirmUSB [24], which applied symbolic execution to firmware analysis on MSP430 family firmware and USB firmware. FirmXRay [43] analyzed the configurations of the firmware on bare-metal BLE devices to detect link layer vulnerabilities. All these works focus on finding vulnerabilities, but none of them aims at reducing the attack surface of the firmware.

Bluetooth stack security. Antonioli et al. [2, 4] discovered a vulnerability in Bluetooth key length negotiation so that the key length can be one byte, therefore, encrypted data can be easily decrypted. Sivakumaran et al. [38] and Naveed et al. [27] revealed the "mis-bonding" problem between the application on the smartphone and the Bluetooth device leading to unauthorized access of the Bluetooth device. Xu et al. [46] showed how to attack a smartphone by replacing the user-expected profile with another one on after pairing. BIAS [3] can bypass the authentication and impersonate a paired benign Bluetooth device. BLESA [45] allows the attacker to inject malicious data into a smartphone when it reconnects to a previously paired BLE device. Tschirschnitz et al. [42] revealed the vulnerability during pairing, which allows the attacker to launch MITM attacks by confusing the user with two pairing methods. Ruge et al. [34] and Heinze et al. [22] proposed frameworks to fuzz the Bluetooth stack implementations. BLE-guardian [18] built a framework to protect the privacy of users of BLE devices by jamming the advertising channel. BlueShield [44] proposed a monitoring framework to detect spoofed BLE advertising messages. LBM [41] protects the Bluetooth host code by building a firewall in Linux kernel. All these works made Bluetooth safer by either discovering new vulnerabilities or providing different defense mechanisms, but none of them achieved the same goal by reducing the attack surface.

11 Conclusion

In this paper, we presented LIGHTBLUE, a novel framework for automatic Bluetooth stack debloating. LIGHTBLUE transforms the multi-entry, callback-driven, host code into a single-entry program. Then, a profile-aware, data-flow-based analysis is used to decouple profile-specific code chunks and identify chunks to be debloated. This analysis also yields unneeded HCI commands and link interfaces that are used for firmware debloating. At last, LIGHTBLUE debloats the firmware by removing the unused command handlers and link interfaces via firmware patching. In our evaluation with 4 different pieces of host code and 3 pieces of firmware, we demonstrated that LIGHTBLUE successfully removed 26 vulnerable functions, mitigating attacks from 20 CVEs.

Acknowledgments

We thank the reviewers for their valuable comments and suggestions. This project was supported in part by ONR under grants N00014-18-1-2674 and N00014-17-1-2513, NSF under grant CNS-1801601, and the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2016.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. BIAS: Bluetooth Impersonation AttackS. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[4] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy. *ACM Transactions on Privacy and Security*, 23(3), 2020.

[5] Armis. BlueBorne Technical White Paper. https://go.armis.com/blueborne-technical-paper. Accessed: January 29, 2020.

[6] Bluetooth Special Interest Group. 2019 Bluetooth Market Update. https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update/, 2019. Accessed: August 1, 2019.

[7] Bluetooth Special Interest Group. Advanced Audio Distribution v1.3.2. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=457083, 2019. Accessed: January 17, 2020.

[8] Bluetooth Special Interest Group. Bluetooth Core Specifications 5.2, 2019.

[9] Broadcom. BCM4339 data sheet. https://www.mouser.com/datasheet/2/100/Radio%20with%20Integrated%20Bluetooth%204.1%20and%20FM%20Receive-961626.pdf. Accessed: January 18, 2020.

[10] Zephyr Project Community. Zephyr Project. https://www.zephyrproject.org/. Accessed: February 3, 2020.

[11] BlueZ contributers. BlueZ. http://www.bluez.org/, 2019. Accessed: August 1, 2019.

[12] Cypress. CYW920735Q60EVB-01 Evaluation Kit. https://www.cypress.com/documentation/development-kitsboards/cyw920735q60evb-01-evaluation-kit. Accessed: August 1, 2019.

[13] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2013.

[14] D-bus. https://www.freedesktop.org/wiki/Software/dbus/. Accessed: January 18, 2020.

[15] Android Developers. Bluedride Bluetooth stack. https://android.googlesource.com/platform/external/bluetooth/bluedroid/, 2015. Accessed: August 1, 2019.

[16] Android Developers. Fluoride Bluetooth stack. https://android.googlesource.com/platform/system/bt/+/181144a50114c824cfe3cdfd695c11a074673a5e/README.md, 2019. Accessed: August 1, 2019.

[17] die.net. sdptool(1) - Linux man page. https://linux.die.net/man/1/sdptool. Accessed: August 1, 2020.

[18] Kassem Fawaz, Kyu-Han Kim, and Kang G. Shin. Protecting Privacy of BLE Device Users. In *Proceeings of the USENIX Security Symposium (USENIX Security)*, 2016.

[19] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[20] BlueKitchen GmbH. BlueKitchen BTSTACK. https://bluekitchen-gmbh.com/. Accessed: February 3, 2020.

[21] Google. BluetoothProfile. https://developer.android.com/reference/android/bluetooth/BluetoothProfile. Accessed: February 11, 2020.

[22] Dennis Heinze, Jiska Classen, and Matthias Hollick. ToothPicker: Apple Picking in the iOS Bluetooth Stack. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[23] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[24] Grant Hernandez, Farhaan Fowze, Dave Jing Tian, Tuba Yavuz, and Kevin RB Butler. FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[25] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-Driven Software Debloating. In *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2019.

[26] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth Binary Patching and Experimentation Framework. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2019.

[27] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[28] NFC Forum. New NFC Controller Interface Specification Makes It Easier to Deliver a Broad Range of NFC Devices and Solutions. https://nfc-forum.org/

new-nfc-controller-interface-specification-makes-it-easier-to-deliver-a-broad-range-of-nfc-devices-and-solutions/, 2012. Accessed: August 1, 2020.

[29] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.

[30] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2018.

[31] David A Ramos and Dawson Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2015.

[32] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2019.

[33] RPi-Distro. Bluetooth firmware. https://github.com/RPi-Distro/bluez-firmware. Accessed: August 1, 2020.

[34] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2020.

[35] Mike Ryan. Bluetooth: With Low Energy Comes Low Security. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2013.

[36] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[38] 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 *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2019.

[39] Inc Square. Square Point of Sale - POS. https://play.google.com/store/apps/details?id=com.squareup&hl=en_US, 2020. Accessed: August 1, 2020.

[40] Inc Square. Square Reader for contactless and chip. https://squareup.com/shop/hardware/us/en/products/chip-credit-card-reader-with-nfc, 2020. Accessed: August 1, 2020.

[41] Dave Jing Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Peter C. Johnson, and Kevin R. B. Butler. LBM: A Security Framework for Peripherals within the Linux Kernel. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[42] Maximilian von Tschirschnitz, Ludwig Peuckert, Fabian Franzen, and Jens Grossklags. Method confusion attack on bluetooth pairing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[43] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities from Bare-Metal Firmware. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[44] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Mathias Payer, and Dongyan Xu. BlueShield: Detecting Spoofing Attacks in Bluetooth Low Energy (BLE) Networks. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[45] 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 *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[46] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[47] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

A Appendix

Prevalence of each profile in tested Android apps.

Table 6 shows the number of Android apps in our dataset that use a specific profile.

Table 6: The prevalence of each profile in tested Android apps.

Profile	Used by # of Apps
A2DP (AVRCP)	182
HFP	192
PAN	2
HID	2
HDP	22
GATT	223
SAP	0
SPP	0
PBAP	2

Bluetooth profiles and corresponding functionalities.

Table 7 describes the functionality enabled by the different Bluetooth profiles.

Links and their interfaces.

Table 8 shows the mapping between links and their interfaces, as described by the Bluetooth specification.

Table 7: Profiles and corresponding functionalities.

Profile	Functionality
A2DP (AVRCP)	Advanced Audio Distribution Profile (A2DP) defines how one device streams audio to another one via Bluetooth. Audio/Video Remote Control Profile (AVRCP) provides functionality for one device to control the audio and video playing on another device through Bluetooth.
HFP	Hands-Free Profile (HFP) allows the car or headset to communicate with the mobile phone via Bluetooth so that the car or headset can make/answer phone calls or stream audio from the phone.
PAN	Personal Area Networking (PAN) Profile describes how to set up an ad-hoc network between different devices via Bluetooth and how to use it to access a remote network through a network access point.
HID	Human Interface Device (HID) Profile defines the procedures to be used by Bluetooth HID hosts (e.g., smartphones and laptops) to get input from and send output to Bluetooth HID devices (e.g., keyboards and mice).
HDP	Health Device Profile (HDP) allows communication between Bluetooth healthcare data source devices (e.g., blood pressure monitors and glucose meters) and data sink devices (e.g., smartphones).
GATT	Generic Attribute (GATT) Profile is designed to be used by an application or another profile so that a client can communicate with a server. Most BLE devices use this profile to communicate with smartphones.
SAP	SIM Access Profile (SAP) allows devices such as car phones with built-in GSM transceivers to connect to a SIM card in a Bluetooth enabled phone.
SPP	Serial Port Profile (SPP) allows Bluetooth enabled devices to emulate serial cable transmission via Bluetooth.
PBAP	Phone Book Access Profile (PBAP) provides the functionality to exchange phone books between Bluetooth enabled devices.

Table 8: Mapping of links and their interfaces.

Link Type	Link Interfaces
Asynchronous Connection Oriented (ACL), BT Classic	HCI_Create_Connection, HCI_Disconnect, HCI_Create_Connection_Cancel, HCI_Accept_Connection_Request, HCI_Reject_Connection_Request
Synchronous Connection Oriented (SCO) & Extended Synchronous Connection Oriented (eSCO), BT Classic	HCI_Setup_Synchronous_Connection, HCI_Accept_Synchronous_Connection_Request, HCI_Reject_Synchronous_Connection_Request, HCI_Enhanced_Setup_Synchronous_Connection, HCI_Enhanced_Accept_Synchronous_Connection_Request
LE Asynchronous Connection (LE ACL), BLE	HCI_LE_Create_Connection, HCI_LE_Create_Connection_Cancel, HCI_LE_Extended_Create_Connection
LE Advertising Broadcast (ADVB), BLE	HCI_LE_Set_Advertising_Enable, HCI_LE_Set_Scan_Enable, HCI_LE_Set_Extended_Advertising_Enable, HCI_LE_Set_Extended_Scan_Enable
LE Periodic Advertising Broadcast (PADVB), BLE	HCI_LE_Set_Periodic_Advertising_Enable, HCI_LE_Periodic_Advertising_Create_Sync, HCI_LE_Periodic_Advertising_Create_Sync_Cancel, HCI_LE_Periodic_Advertising_Terminate_Sync
Connected Isochronous Stream (CIS), BLE	HCI_LE_Create_CIS, HCI_LE_Accept_CIS_Request, HCI_LE_Reject_CIS_Request
Broadcast Isochronous Stream (BIS), BLE	HCI_LE_BIG_Create_Sync, HCI_LE_BIG_Terminate_Sync