argXtract: Deriving IoT Security Configurations via Automated Static Analysis of Stripped ARM Cortex-M Binaries

Pallavi Sivakumaran
Royal Holloway, University of London
Egham, United Kingdom
pallavi.sivakumaran.2012@rhul.ac.uk

Jorge Blasco
Royal Holloway, University of London
Egham, United Kingdom
jorge.blascoalis@rhul.ac.uk

ABSTRACT

Recent high-profile attacks on the Internet of Things (IoT) have brought to the forefront the vulnerabilities in "smart" devices, and have revealed poor device configuration to be the root cause in many cases. This has resulted in IoT technologies and devices being subjected to numerous security analyses. For the most part, *automated* analyses have been confined to IoT hub or gateway devices, which tend to feature traditional operating systems such as Linux or VxWorks. However, most IoT *peripherals*, by their very nature of being resource-constrained, lacking traditional operating systems, implementing a wide variety of communication technologies, and (increasingly) featuring the ARM Cortex-M architecture, have only been the subject of smaller-scale analyses, typically confined to a certain class or brand of device. We bridge this gap with `argXtract`, a framework for performing automated static analysis of stripped Cortex-M binaries, to enable bulk extraction of security-relevant configuration data. Through a case study of 200+ Bluetooth Low Energy binaries targeting Nordic Semiconductor chipsets, as well as smaller studies against STMicroelectronics BlueNRG binaries and Nordic ANT binaries, `argXtract` has discovered widespread security and privacy issues in IoT, including minimal or no protection for data, weakened pairing mechanisms, and potential for device and user tracking.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**.

KEYWORDS

IoT, Firmware Analysis, Stripped Binaries, ARM, Cortex-M, Bluetooth Low Energy, ANT, Nordic, STMicroelectronics

ACM Reference Format:
Pallavi Sivakumaran and Jorge Blasco. 2021. `argXtract`: Deriving IoT Security Configurations via Automated Static Analysis of Stripped ARM Cortex-M Binaries. In *ACSAC '21: Annual Computer Security Applications Conference, Dec 06–Dec 10, 2021, Texas, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/xxxxxxx.xxxxxxx

ACSAC '21, Dec 06–Dec 10, 2021, Texas, USA

ACM ISBN 978-1-4503-XXXX-X/21/12…$15.00
https://doi.org/10.1145/xxxxxxx.xxxxxxx

1 INTRODUCTION

The Internet of Things (IoT) is growing at a rapid pace, with an estimated 22 billion connected IoT devices in use around the world at the end of 2018, and projected to grow to 50 billion by 2030 [75]. These devices are increasingly handling users' personal data and health information, and performing security-related functions. It is therefore imperative to fully understand the security and privacy implications of IoT deployments.

Recent years have shown that there is legitimate cause for concern, as numerous flaws have been uncovered in IoT devices, some of which have been exploited at a large-scale (e.g., Mirai [22], Brickerbot [64]). Severe vulnerabilities have also been discovered in cardiac devices [45], baby heart monitors [80] and webcams [74]. The root cause on many occasions was poor device configuration, e.g., default passwords [5, 51] or poor protection for data [53, 91].

The configuration of an IoT device can therefore be a vital source of information regarding possible vulnerabilities, and the device firmware is often the most definitive source of information regarding its configuration. However, analysing IoT firmware is notoriously difficult, particularly in the case of *peripheral* firmware files, which are often only available as stripped binaries, i.e., files that do not contain headers, section information or debugging symbols. Additionally, many IoT peripheral devices now utilise ARM Cortex-M processors [9]. These feature inline data and instruction sets that are not yet fully supported by current disassemblers, which greatly complicates analysis [41].

In this work, we present `argXtract`, a framework for bulk extraction of security-relevant configuration information from stripped ARM Cortex-M binaries via a partial-knowledge automated static analysis. `argXtract` overcomes various challenges associated with analysing stripped Cortex-M binaries and provides a generic framework for extracting arguments to a *Call Of Interest* (COI); this can be an ARM supervisor call or a standard function call, identified via `svc` instructions and function pattern matching, respectively.

We also present three case studies, selected based on the availability of real-world Cortex-M binaries, where we use `argXtract` to analyse the security configurations from: (i) binaries implementing Bluetooth Low Energy (BLE) and targeting Nordic Semiconductor chipsets, where configurations are made via supervisor calls, (ii) binaries implementing BLE and targeting STMicroelectronics BlueNRG chipsets, where configurations are made via function calls, and (iii) binaries implementing the ANT [33] technology and targeting Nordic chipsets, with configurations via supervisor calls.

The results reveal widespread security issues in IoT peripherals, including minimal or no protection for data, inconsistent permissions, artificially weakened pairing mechanisms, and the potential for tracking devices (and possibly also users).

We summarise our main contributions as follows:

- We develop `argXtract`, a framework for performing automated static analysis of stripped ARM Cortex-M binaries, to enable extraction of security-related configuration information (§3). We evaluate `argXtract` in terms of the accuracy of function boundary estimation and pattern matching, and correctness of extracted results (§4).
- We use `argXtract` to extract security configurations from stripped Nordic BLE binaries (§5), STMicroelectronics' BlueNRG binaries and Nordic ANT binaries (§6). The results reveal widespread lack of protection for data, inconsistent data access controls and serious privacy vulnerabilities.
- We make `argXtract` available as open source code for the benefit of the research community at https://github.com/projectbtle/argXtract. The repository includes configuration files for case studies presented in this paper.

2 MOTIVATION

Configuration issues have been the root cause for several recent attacks on IoT devices. For example, Mirai exploited the use of default credentials to infect IoT devices [5], while some baby monitors, vaping products and e-skateboards did not implement basic protection for their data [52, 53, 91]. This left the devices vulnerable to false data injection and the potential for physical harm to the user. The configuration of an IoT device can therefore reveal exploitable vulnerabilities and is a vector of interest in security analyses.

There are several possible sources for this configuration information, including the devices themselves, the firmware they run, or any application or website they interface with. We analyse the merits and shortcomings of each of these potential sources in Appendix A. Arguably, firmware is the richest standalone source of information regarding a device's security configuration. With devices such as IoT hubs and gateways (e.g., mobile phones, routers), which often run some version of the Linux OS, familiar filesystem structures and commands may be identifiable within firmware, which can contribute towards the analysis. Even so, the analysis will generally not be straightforward. Analysis is much more complex for IoT *peripherals*, which may run a custom OS, or have no OS at all. This has resulted in far fewer analyses of IoT peripheral binaries. We bridge this gap by focusing on peripheral firmware analysis.

IoT peripherals may implement one or more communication technologies, such as BLE [16], Zigbee [93], ANT [33] or Thread [81]. Many of these technologies have fully-fledged stacks, which must be implemented within peripheral firmware. To ease development, many IoT chipset vendors implement the technology stacks themselves and provide APIs through which developers can configure the stacks within their applications [57, 78, 79]. Developers may also use library functions to perform other configurations.

We present an example configuration function in Figure 1, where a fixed passkey is defined for the BLE pairing process using an API call `sd_ble_opt_set`. Fixed passkeys reduce the security of BLE pairing and are therefore a vulnerability of interest. To identify fixed passkeys from firmware, we need to determine whether `sd_ble_opt_set` (which we term a Call Of Interest, or COI) is called with a fixed passkey as its argument. To do this, we need to pinpoint the location of the COI within the firmware binary, and then analyse the arguments that are passed to it. Figure 2 depicts the assembly instructions corresponding to this section of code, obtained by disassembling the firmware binary. From the instructions, we are able to identify that the function call occurs at address `0x1ead0`, that the passkey bytes occur at address `0x21f14` (as `323134333635` [little-endian], i.e., "123456"), and that they are referenced by their absolute location at address `0x1ed9c`. The ability to correctly deduce these pieces of information depends on a set of **conditions**:

C1 Knowledge of function location and callers' addresses (i.e., knowing that the code for `sd_ble_opt_set` is at address `0x1e748` and that it is called at address `0x1ead0`).

C2 Knowledge of locations of inline data (i.e., knowing that the bytes at addresses `0x1ed9c` and `0x21f0c` should be interpreted as data rather than as code).

C3 Firmware being loaded at the correct offset/application code base (such that the absolute address `0x21f14` results in bytes being loaded from the correct location).

This information is present within headers and symbol tables. However, due to storage considerations, IoT peripherals tend to ship firmware with this information removed, i.e., as *stripped binaries*. Figure 3 depicts the disassembly of the binary file with ELF

```
uint8_t passkey[] = "123456";
ble_opt_t ble_opt;
ble_opt.gap_opt.passkey.p_passkey = &passkey[0];
err_code = sd_ble_opt_set(BLE_GAP_OPT_PASSKEY, &ble_opt);
```

Figure 1: Source C code.

```
1eaba: 4ab8     ldr  r2, [pc, #736] ; (1ed9c)
1eabc: ab06     add  r3, sp, #24
1eabe: 6811     ldr  r1, [r2, #0]
1eac0: 2022     movs r0, #34 ; 0x22
1eac2: 9106     str  r1, [sp, #24]
1eac4: 8891     ldrh r1, [r2, #4]
1eac6: 8099     strh r1, [r3, #4]
1eac8: 7992     ldrb r2, [r2, #6]
1eaca: a908     add  r1, sp, #32
1eacc: 719a     strb r2, [r3, #6]
1eace: 9308     str  r3, [sp, #32]
1ead0: f7fffe3a bl   1e748 <sd_ble_opt_set>
1ed9c: 00021f14 .word   0x00021f14
21f0c: 2528 2000 0001 0700 3231 3433 3635 0000
```

Figure 2: Disassembly of unstripped binary.

```
0x3aba: b84a     ldr  r2, [pc, #0x2e0]
0x3abc: 06ab     add  r3, sp, #0x18
0x3abe: 1168     ldr  r1, [r2]
0x3ac0: 2220     movs r0, #0x22
0x3ac2: 0691     str  r1, [sp, #0x18]
0x3ac4: 9188     ldrh r1, [r2, #4]
0x3ac6: 9980     strh r1, [r3, #4]
0x3ac8: 9279     ldrb r2, [r2, #6]
0x3aca: 08a9     add  r1, sp, #0x20
0x3acc: 9a71     strb r2, [r3, #6]
0x3ace: 0893     str  r3, [sp, #0x20]
0x3ad0: fff73afe bl   #0x3748
0x3d9c: 141f     subs r4, r2, #4
0x3d9e: 0200     movs r2, r0
0x6f14: 3132     adds r2, #0x31
0x6f16: 3334     adds r4, #0x33
0x6f18: 3536     adds r6, #0x35
```

Figure 3: Disassembly of stripped binary.

headers and debugging symbols stripped out. The disassembly of the stripped binary does not contain information about function names (failing Condition C1), thereby making it difficult to deduce functionality. Also, data segments have been incorrectly interpreted by the disassembler as code (failing Condition C2),[1] which leads to incorrect results when performing value tracing and precludes the use of emulation frameworks (e.g., QEMU [13], unicorn [63]). Further, the code has been loaded at the incorrect offset (failing Condition C3), which means absolute addressing will fail.

Contributing to this problem is the fact that many resource-constrained IoT devices feature ARM processors [50] with Thumb or Thumb-2 instruction sets (instead of the ARM instruction set) for greater code density [35]. In fact, the ARM Cortex-M architectures, which are very popular in embedded systems [9], support *only* the Thumb and Thumb-2 instruction sets. These instruction sets are not yet fully supported by many disassemblers. Out of the current state-of-the-art reverse-engineering tools, `IDA` (free) [38] does not currently support ARM, while `Debin` [37] and `BAP` [18] do not fully support the Thumb instruction set. Testing free reverse-engineering tools that *do* support Thumb analysis against a simple stripped Cortex-M IoT binary, we found that `radare2` [2] failed to identify almost 40% of the functions within the binary (analysing using `aaa` and `aaaa`), `Ghidra` [55] failed to identify 30% of the functions, while `angr` [83] was unable to produce a valid Control Flow Graph - a step prior to analysis. Our observation regarding the robustness of `angr` and `radare2` for Thumb mode analysis is supported by [41], which also noted that `Ghidra` too has better support for the ARM instruction set than for Thumb. Further, because IoT peripheral binaries typically do not include the technology stack or ROM data, *dynamic* analysis approaches are unsuitable. This reveals a gap in the automated IoT security analysis landscape and prompted the development of `argXtract`.

3 ARGXTRACT

We design `argXtract` to take as input the disassembly of a Cortex-M binary,[2] perform several levels of processing, and finally extract and output arguments to security-relevant Calls Of Interest. The processing is divided into the following stages: **Application code base identification**, for correct absolute addressing (§3.1); **Data identification**, such that data is not incorrectly interpreted as code (§3.2); **Function block identification**, to enable call execution path generation and function pattern matching (§3.3); **COI identification** (function call or ARM supervisor call), to determine trace termination points (§3.4); **Tracing and argument processing**, to extract and process the arguments to a COI (§3.5).

3.1 Application Code Base Identification

As described in §2, an incorrect address offset will lead to the failure of absolute addressing. `argXtract` combines *known* address information with *obtained* addresses to compute the application code base. The addresses of core interrupt handlers are known, as they are present at specific offsets within the Vector Table (VT), which is located at `0x00000000` within the stripped binary [10]. Corresponding interrupt handler *code* within the stripped binary is identified by exploiting the fact that at least one interrupt handler is usually the *default handler*, i.e., an endless loop or self-targeting branch. `argXtract` examines the disassembly for self-targeting branches and compares their addresses against VT addresses. If the final 3 hex digits of a self-targeting branch and a VT address match, the offset is computed as $offset = vtAddress - selfTargetingBranchAddress$. The binary disassembly is reloaded at the correct offset, to satisfy Condition C3.

```
000272b4 <Reset_Handler>:
 272b4: 4906         ldr r1, [pc, #24] ;(272d0)
 272b6: 4a07         ldr r2, [pc, #28] ;(272d4)
 272b8: 4b07         ldr r3, [pc, #28] ;(272d8)
 272ba: 1a9b         subs r3, r3, r2
 272bc: dd03         ble.n 272c6
 272be: 3b04         subs r3, #4
 272c0: 58c8         ldr r0, [r1, r3]
 272c2: 50d0         str r0, [r2, r3]
 272c4: dcfb         bgt.n 272be
 272c6: f002 f8c9    bl 2945c <SystemInit>
 272ca: f7ff ffb9    bl 27240 <_mainCRTStartup>
 272ce: 0000        .short 0x0000
 272d0: 00042c54    .word 0x00042c54
 272d4: 20002b28    .word 0x20002b28
 272d8: 200031d0    .word 0x200031d0
```

Figure 4: Identification of `.data` using Reset Handler.

3.2 Data Identification

Stripped Cortex-M binaries do not contain section information. Their disassembly therefore produces a block of instructions with a `.text` (i.e., code) segment and often a `.data` segment, with no demarcation between the two and the `.data` segment misinterpreted as code. The `.text` segment also has inline data, often misinterpreted as code and resulting in value tracing errors.

The data identification component of `argXtract` uses information from the Reset Handler, whose address is read from the VT, to identify the location and correct starting address of the `.data` segment. It also identifies inline data using four primary sources: (i) PC-relative memory-loads (e.g., `ldr`, `ldrh`), (ii) direct write-to-PC operations (iii) table branches (`tbb`, `tbh`), and (iv) compact switch table helpers such as __ARM_common_switch8 and __gnu_thumb1 variants. These operations aid in satisfying Condition C2 (see §2). We describe the data identification mechanism for each of these sources in further detail below.

3.2.1 Identification of `.data`. The Reset Handler often contains the final address of the `.text` segment as well as the start and end addresses for the `.data` segment. This is present in the form of consecutive memory-loads, where the first memory-load reads in the address from which the `.data` segment starts and subsequent memory-loads read the (actual) start and end addresses for the `.data` segment. An example has been shown in Figure 4.

`argXtract` analyses instructions within the Reset Handler to determine whether they match the required structure. If they do, then the addresses starting after the final address of the `.text` segment and ending at the end of the file are marked as data, i.e., as the `.data` segment. The addresses within the newly-identified `.data` segment are also modified according to the information extracted

[1] Note that inline data is far more common in ARM than in x86/x64 [41], and the misinterpretation of such data as code is a problem that is encountered even with state-of-the-art disassemblers [32].

[2] The input disassembly is obtained via any existing disassembler and will very likely feature the issues described in §2.

```
1a83c: 2c17    cmp    r4, #23
1a83e: d8fc    bhi.n  1a83a
1a840: 4ac7    ldr    r2, [pc, #796]
1a842: 00a3    lsls   r3, r4, #2
1a844: 58d3    ldr    r3, [r2, r3]
1a846: 469f    mov    pc, r3
```

Figure 5: Example write-to-PC operation.

```
2894a: 2e08        cmp     r6, #8
2894c: d219        bcs.n   28982
2894e: e8df f006   tbb     [pc, r6]
28952: 1b181804    ; data
28956: 172f2f22    ; data
2895a: 8901        ldrh    r1, [r0, #8]
```

Figure 6: Example table branch structure.

from the Reset Handler. In the example in Figure 4, the memory-loads at addresses `0x272b4` and `0x272b6` denote that the addresses from `0x42c54` onward need to be reinterpreted as data, and need to be re-addressed with addresses starting from `0x20002b28`.

3.2.2 Identification of inline data via PC-relative memory-loads. A memory-load that loads data from an address within the firmware file will specify the source address relative to either the Program Counter (PC) or a register. Register-relative loads may require significant tracing in some cases. However, PC-relative loads are straightforward to analyse. `argXtract` performs a linear scan for PC-relative memory-loads, calculates the address from which data is loaded and marks it as data, re-processing residual bytes as instructions where required.

3.2.3 Identification of inline data via write-to-PC operations. Direct write-to-PC operations are sometimes used to accomplish code branches. Figure 5 depicts an example. This operation loads a branch address from an address within the firmware and writes the branch address to the PC. The address from which the branch address is loaded (i.e., the `ldr` source at `0x1a844`, obtained in this example by adding the contents of `r2` and `r3`) must be interpreted as data, but is misinterpreted as code within the disassembly of stripped binaries.

When a write-to-PC is encountered (at `0x1a846` in Figure 5), the preceding integer comparison (`0x1a83c`) is identified and the range of comparison values is determined using subsequent conditional branches (`0x1a83e`). Instructions following the branch and until the non-PC-relative memory-load (`0x1a844`) are executed for all possible comparison values, to produce a range of addresses from which the branch addresses are loaded. This range is marked as data. The branch addresses are obtained by executing until the PC-write instruction, and are used for function identification (§3.3).

3.2.4 Identification of inline data via table branches. Table branch instructions (`tbb`, `tbh`) were introduced in the ARMv7-M architecture to handle complex branching conditions. Figure 6 depicts a sample table branch instruction (at address `0x2894e`). The instruction is immediately followed by a branch table (`0x28952` and `0x28956`). This table should be interpreted as data, but is misinterpreted by disassemblers as code in the absence of debugging symbols.

In the case of table branch instructions, an index value is used to index into the branch table. An integer comparison is performed

```
  18228:  bl  182b0 <functionB>
000182b0 <functionB>:
  182b0:  push  {r4, lr}
  182b2:  cmp  r2, #32
  182b4:  blt.n  182c0  ;skips pop at 182be
  182b6:  mov  r0, r1
  182b8:  subs  r2, #32
  182ba:  lsrs  r0, r2
  182bc:  movs  r1, #0
  182be:  pop  {r4, pc}
  182c0:  mov  r3, r1
  182c2:  lsrs  r3, r2
  182c4:  lsrs  r0, r2
  182c6:  movs  r4, #32
  182c8:  subs  r2, r4, r2
  182ca:  lsls  r1, r2
  182cc:  orrs  r0, r1
  182ce:  mov  r1, r3
  182d0:  pop  {r4, pc}
  182d2:  nop
000182d4 <functionC>:   ;not called within code
  182d4:  ldrb  r2, [r0, #0]
```

Figure 7: Example assembly for function boundary identification.

against the register containing this index value prior to the table branch instruction. This provides an indication as to the size of the branch table. In Figure 6, the comparison (`0x2894c`) and conditional branch (`0x2894a`) indicate that the branch table has 8 entries. Because the table branch instruction in our example is `tbb`, the table will consist of single-byte offsets (if the instruction had been `tbh`, the table would contain halfword offsets). `argXtract` processes this information and marks the 8 bytes from the PC onward as data.

3.2.5 Identification of inline data via compact switch helpers. Prior to the introduction of table branch instructions, "helper" functions were utilised to handle switch-case constructs. The GCC compiler produces __GNU_THUMB1 variants, while Keil produces __ARM_COMMON_SWITCH8. These helper functions have identifiable function prologues, and calls to the functions are followed by an index table, similar to table branch instructions.

`argXtract` determines the locations of the helper functions and applies function-specific processing to determine the size of the index table. It also determines the addresses of resultant branches, to be used by the function boundary identification module.

3.3 Function Boundary Identification

Function boundary identification is used within `argXtract` to enable function pattern matching and call execution path determination. The challenges involved in function boundary identification have been widely studied. These include indirect function calls, absence of specific function prologues, indeterminate location of start instructions, absence of a clear exit point and presence of multiple exit points [90]. The presence of inline data within Cortex-M disassembly, which may be misinterpreted as code, can further complicate function boundary estimation [32].

`argXtract`'s function boundary identification is performed in two stages. First, an initial set of high-certainty candidates for function start addresses is generated by obtaining the addresses of all interrupt handler functions from the Vector Table (i.e., each interrupt handler is considered a separate function). Targets of

```
0x18228:  bl  0x182b0
...
0x182b0:  push  {r4, lr}
0x182b2:  cmp  r2, #0x20
0x182b4:  blt  #0x182c0
0x182b6:  mov  r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop  {r4, pc}
0x182c0:  mov  r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov  r1, r3
0x182d0:  pop  {r4, pc}
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
```

(a) Capstone disassembly (starting point).

```
0x18228:  bl 0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp  r2, #0x20
0x182b4:  blt  #0x182c0
0x182b6:  mov  r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop  {r4, pc}
0x182c0:  mov  r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov  r1, r3
0x182d0:  pop  {r4, pc}
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
```

(b) Identify function blocks using b, bl.

```
0x18228:  bl  0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp  r2, #0x20
0x182b4:  blt  #0x182c0
0x182b6:  mov  r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop r4, pc ;end?
0x182c0:  mov  r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov  r1, r3
0x182d0:  pop {r4, pc} ;end?
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
```

(c) Mark potential exit points.

```
0x18228:  bl  0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp  r2, #0x20
0x182b4:  blt  #0x182c0
0x182b6:  mov  r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop r4, pc ;end?
0x182c0:  mov  r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov  r1, r3
0x182d0:  pop {r4, pc} ;end?
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
```

(d) Identify instructions that skip exits.

```
0x18228:  bl  0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp  r2, #0x20
0x182b4:  blt  #0x182c0
0x182b6:  mov  r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop  {r4, pc}
0x182c0:  mov  r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov  r1, r3
0x182d0:  pop {r4, pc} ;end?
0x182d2:  nop
0x182d4:  ldrb  r2, [r0]
```

(e) Remove skipped exit instructions.

```
0x18228:  bl  0x182b0
...
0x182b0:  push r4, lr ;start
0x182b2:  cmp  r2, #0x20
0x182b4:  blt  #0x182c0
0x182b6:  mov  r0, r1
0x182b8:  subs  r2, #0x20
0x182ba:  lsrs  r0, r2
0x182bc:  movs  r1, #0
0x182be:  pop  {r4, pc}
0x182c0:  mov  r3, r1
0x182c2:  lsrs  r3, r2
0x182c4:  lsrs  r0, r2
0x182c6:  movs  r4, #0x20
0x182c8:  subs  r2, r4, r2
0x182ca:  lsls  r1, r2
0x182cc:  orrs  r0, r1
0x182ce:  mov  r1, r3
0x182d0:  pop  {r4, pc}
0x182d2:  nop
0x182d4:  ldrb r2, [r0] ;start
```

(f) Identify suitable start for next function.

Figure 8: Process used by argXtract for identifying function start addresses.

branch-and-link (bl) and branch (b) instructions are added to this set, subject to satisfying requirements regarding function prologues. Next, a function boundary estimation algorithm is executed against each block of instructions between two addresses in the sorted set. The algorithm operates on the basic principle that, while a function may have multiple exit instructions due to conditional executions, it must have mechanisms for bypassing all but one of the exit points. This could be via conditional branch instructions or a switch/branch table (as identified in §3.2). argXtract determines all potential exit points (e.g., data, pop, bx lr, unconditional branches to lower addresses or outside the current block) within the block of instructions that is being analysed and marks the exit point that *cannot be bypassed* as the ultimate function exit. The next valid instruction is determined to be the beginning of the next function. This procedure is performed iteratively to obtain the final list of function blocks.

We further illustrate this algorithm using the code example in Figure 7 as reference. This reference code contains two functions, denoted as functionB and functionC. Of these, functionB is called via a bl instruction at 0x18228, while functionC is called indirectly via a blx call (which means the starting address of functionC cannot be identified without some level of register tracing). Figure 8a depicts the equivalent assembly code obtained using the Capstone [62] disassembler against the *stripped* version of the binary. To estimate function boundaries for this disassembly, we first identify high-confidence function starts, including targets of bl instructions. This will result in 0x182b0 being identified as a function start (Figure 8b). This corresponds to functionB. We next apply our function boundary estimation algorithm to the block of assembly instructions beginning at 0x182b0 as follows:

(i) Mark out possible exit points, such as pop, load-to-PC and bx lr instructions, as well as data. As shown in Figure 8c, there are two potential exit points, at addresses 0x182be and 0x182d0.

(ii) Look for branches that skip the exit points. Figure 8d shows that the branch condition at 0x182b4 skips the exit point at 0x182be. This exit point is therefore considered to be part of the existing function (i.e., the one beginning at 0x182b0).

(iii) Remove exit points that are skipped. We are left with one other potential exit point at `0x182d0`. It is considered the final exit point of the function (see Figure 8e).

(iv) Identify the next valid instruction as the start of the next function. Initially, we consider the instruction at `0x182d2` to be the start of the next function. However, this address contains a `nop` instruction. Therefore, we skip it and mark `0x182d4` as the start of the next function, as shown in Figure 8f. This corresponds to the start address of `functionC`.

This process is then repeated from the start of the new function block (i.e., from `0x182d4`), until the end of the block is reached.

Function block annotation: `argXtract` maintains a data object containing information on cross-references to and from a function block, as well as the function's *call depth*. The call depth indicates the maximum number of functions that get called iteratively by a function. It is used for function pattern matching and tracing.

3.4 COI Identification

A COI in our framework could be a standard function call or could be translated to an ARM supervisor call (`svc`). `argXtract` identifies calls to the `svc`s or functions of interest using the techniques described in §3.4.1 and §3.4.2, respectively. In both cases, the addresses of the calling instructions are stored, to be used in the tracing step.[3] This then satisfies Condition C1. After this step, all pre-conditions for analysing a stripped ARM binary (see §2) will be satisfied.

3.4.1 Supervisor call identification. In `svc` analysis mode, an input object containing the `svc` numbers of interest (obtained from vendor SDKs) is provided to `argXtract`. A linear scan is performed over the disassembly to obtain the addresses of the relevant `svc` instructions.

3.4.2 Pattern matching. Identifying function calls is far more complex than identifying supervisor calls, as functions cannot be immediately identified within assembly. We exploit the fact that configuration API functions (such as those provided by vendors for performing configurations to IoT stacks) accept inputs in a specific order, which are passed within registers in a specific sequence for Cortex-M. In addition, most functions generate artefacts that are detectable within memory and/or registers, i.e., as output or intermediate values. For each function of interest, we define a "function pattern file", which is a collection of test sets containing register and memory inputs, and the corresponding outputs (which could be actual output values in registers or memory, or intermediate values at detectable locations). In the case of functions that store identifiable values at binary-specific locations that *cannot* be predetermined, we propose wildcard addresses, where expected values are specified at some predetermined *offset* from the wildcard address.

A pattern file is passed to each of the functions that have been identified for the binary under test (see §3.3). The function instructions are executed with the input register and memory values specified in the pattern file. Output register and memory contents are compared against the expected values. If a single function matches the given pattern, then this is taken to be the function of interest. In the case of nested function calls, the function with the lowest call depth that satisfies the given pattern is taken to be the function of interest. Polymorphism will be detected if the processing of the inputs differs between the functions such that the artefacts/outputs are different.[4] Note however that if two functions produce the same outputs for any given inputs, then function pattern matching will fail.

[3]Direct calls are identified. However, calls via `blx` are not.

```
{'sd_ble_opt_set':{
  'memory':{..., 20007f60:'31', 20007f61:'32', 20007f62
       :'33', 20007f63:'34', 20007f64:'35', 20007f65
       :'36', 20007f66:'00', 20007f68:'60', 20007f69:'7f
       ', 20007f6a:'00', 20007f6b:'20', ...},
  'registers':{...'r0':'00000022', 'r1':'20007f68', ..}}}
```

(a) Register/Memory Contents.

```
{"args": {
 "0": {...},
 "1": {"in_out": "in",
   "ptr_val": "pointer",
   "data": {
     "p_opt": {
       "ptr_val": "pointer",
       "length_bits": 48,
       "type": "hex" }}}}}
```

(b) Argument Definition Object.

```
"output": {
  "sd_ble_opt_set": [
  {
    "opt_id": 34,
    "p_opt":
         "313233343536"
  }
 ]
}
```

(c) Partial Output File.

Figure 9: Argument Processing

3.5 Tracing and Argument Processing

Once COIs have been identified (as described in §3.4), backward inter-procedural tracing is used to determine all call execution paths. Forward-tracing along the paths then leads to the COI(s). The arguments to a COI are contained within registers `r0-r3` (or on the call stack) [7, 8, 85]. Some registers contain the argument of interest, while others may hold pointers to data in memory. Therefore, when a COI is reached, the contents of both the register object and the memory map are returned to an argument analysis component for processing.

The type and format of data that are used as arguments to COIs are obtained from vendor SDKs and provided to `argXtract` in the form of *Argument Definition Objects*. These are JSON templates that describe the expected structure of bits for each input argument using predefined keywords.[5] For example, Figure 9b depicts the Argument Definition Object for the `sd_ble_opt_set` COI discussed in §2 (Figure 1). A corresponding trace output may look like that depicted in Figure 9a. Taking the second argument as an example, we note that it is defined as a pointer to a pointer to a 6-byte (48-bit) array. This argument is contained within register `r1`, which according to the trace output in Figure 9a contains a value of `20007f68`. As the Argument Definition Object indicates that this is a pointer, we refer to the contents of memory. The memory object in Figure 9a shows that the address `0x20007f68` contains the value `20007f60`. This (also being a pointer) is interpreted as a memory address, `0x20007f60`. This address contains the hex value `0x313233343536`, which corresponds to the ASCII string "123456", i.e., the value specified as the fixed passkey in our example in Figure 1. This results in the output depicted in Figure 9c.

[4]Note that input structures within function pattern files are provided in byte format. Therefore, differences in input *type* do not impact the analysis.

[5]We adopt this template-based approach for greater flexibility, such that supporting additional COIs only requires including new Argument Definition Objects, rather than needing to add extra COI-specific code.

4 EVALUATION

We implement argXtract using Python. We select Capstone as the disassembler, as it underpins ARM disassembly for a number of existing reverse-engineering and analysis tools, including radare, angr and binwalk. In this section, we evaluate our implementation in terms of the accuracy of function block identification and pattern matching, and the correctness of extracted configurations.

4.1 Test Set and Ground Truth

There is no existing ground truth for ARM Cortex-M, i.e., binaries with known function locations and configurations. We therefore generate a test set of 28 stripped binaries for testing and verification purposes. The binaries target chipsets from NXP, STMicroelectronics, Nordic Semiconductor and Texas Instruments, for multiple IoT technologies including Zigbee, ANT, BLE, Thread and 802.15.4. The binaries are compiled using GCC, IAR, Keil and Clang, depending on the chipset vendor. We provide a detailed description of the test binaries in our code repository. For ground truth, we obtain the configuration for each binary by disassembling its *unstripped* version using the GNU ARM embedded toolchain.

Table 1: True Positive Rates (TPR) and Effective False Positive Rates (EFPR) for function block identification against test binaries. EFPR is computed by discounting misidentifications that do not impact the trace.

Bin File		argXtract		radare2[1]		ghidra[2]		Bin File		argXtract		radare2[1]		ghidra[2]	
ID[†]	#Fns[‡]	TPR	EFPR	TPR	EFPR	TPR	EFPR	ID[†]	#Fns[‡]	TPR	EFPR	TPR	EFPR	TPR	EFPR
0a1	324	100	0.29	95.68	2.19	87.96	0	0d2	841	0	100	69.32	7.4	69.68	2.85
1d7	951	93.27	0.97	74.24	3	73.08	0.82	3b1	204	99.02	0	88.24	0	82.35	0
443	598	100	0	83.78	3.05	83.95	1.95	4d7	1563	95.71	1.17	78.57	3.28	77.8	0.15
589	1486	97.51	0.68	83.24	1.59	84.79	0	5d3	398	99.50	0.73	94.97	0	93.47	0
646	166	98.80	0	80.72	0.73	77.71	0.76	67e	2138	99.16	0.05	82.69	0.54	83.4	0.22
681	1961	97.86	0.56	94.19	0.7	87.51	0.12	6ac	265	98.11	0.37	73.96	0.5	72.08	0
70b	115	95.65	0	67.83	0	73.04	0	7e8	1529	97.58	0.66	81.62	1.57	84.96	0
928	520	95.38	0.93	90.19	0	71.15	0	938	2764	99.57	0.74	85.53	0.59	83.90	0.09
989	762	95.80	9.7*	69.27	8.62	63.10	9.39	ade	1951	99.33	0.89	89.54	0.46	87.69	0.12
bad	839	92.25	0.79	69.85	5.43	68.53	0	be7	2035	99.71	0.39	5.11	87.43	4.72	88
cb5	92	94.57	1.11	61.96	0	67.39	0	cc8	1582	94.82	0.71	82.68	1.91	83	0
dd9	801	96.63	6.9*	95.93	6.36	88.15	6.66	e2a	495	95.15	0.39	89.7	0	69.29	0
e2d	698	96.42	0.35	94.99	0	86.25	0	f2b	1926	99.79	0.65	81.15	1.01	79.85	0.06
f37	1585	95.21	1.16	78.23	3.18	78.36	0	fe9	1007	99.40	0.1	61.27	0.80	56.21	0.70

[†]ID = First three characters of SHA256 of binary. [‡]#Fns = Number of functions.
[1]radare2 was executed using aaa analysis mode, [2]Ghidra was executed using the *ARM Aggressive Instruction Finder* option. Both were provided with the application code base manually.

4.2 Accuracy of Function Identification

We evaluate the accuracy of argXtract's function identification (§3.3) by identifying function start addresses for the 28 stripped binaries within our test set and comparing them against the actual functions from the unstripped versions. For comparison, we also do the same using radare2 and ghidra. Table 1 presents the results.

The table shows that for all but five binaries, more than 95% of functions are correctly identified by argXtract. The results are more variable for radare2 and ghidra, but in general the TPRs obtained by these two tools are lower (often significantly lower) than those obtained by argXtract. Manual analysis of a sample of functions (across the test set) that were correctly identified by argXtract but not by radare2 or ghidra showed that many such functions occurred after inline data or less traditional function exit points. The techniques employed by argXtract for inline data identification and function boundary identification enable it to handle such instances and identify a greater proportion of function start addresses correctly. There was a single exception (binary with ID=0d2), where argXtract resulted in a TPR of 0% while radare2 and ghidra identified approximately 70% of functions. This was a binary where the .text segment was split into two sections, each with a different offset. argXtract was unable to compute the offsets in this case, which meant that further analysis was not possible. Additionally, manual analysis of the remaining four cases where argXtract produced a TPR < 95% showed that the unidentified functions were of unusual structure, e.g., functions accessed via direct conditional branches, or containing only a bx lr instruction. These are likely to be fragments of other functions or shared functions. We observe that for the vast majority of such cases, radare2 and ghidra also failed to identify the functions.

Examining false positives (regardless of the analysis tool), we found that in many cases misidentified functions were either where unannotated data had been identified as the start of function blocks, or where a logical function start *can* be assumed, e.g., blocks of alternating ldr instructions and data bytes causing each ldr to be considered as the start of a new function. In the former case, these particular "functions" will never be called during the tracing phase. In the latter, the functions *are* directly addressed as if they are individual functions. Therefore, such FPs will not affect the trace. We thus consider an "Effective FPR" to denote the false positives *excluding* such instances. The EFPRs obtained by argXtract are fairly similar to those obtained by ghidra (within 1-2% of each other). radare2 was more likely to result in a higher EFPR; manual analysis showed that this was often due to radare2 incorrectly considering push instructions to be the start of a function. Overall, argXtract resulted in EFPRs of < 1.5% for all but two binaries (marked with * in the table). These were both compiled by IAR, which is the only compiler we have observed that uses bl instructions to branch and link *within* a function. This accounts for the higher EFPR for these two binaries. While this will not impact the actual branching functionality, it will influence the call depth calculation, which in turn *could* impact tracing. We observe that radare2 and ghidra also resulted in high EFPRs for these two binaries.

4.3 Function Pattern Matching

We verified the functionality of the pattern matching module against the ot::KeyManager::SetKeyRotation OpenThread function, the mbedtls_ssl_conf_ciphersuites mbedtls library function, and the CryptoKeyPlaintext_initKey function from Texas Instruments' SimpleLink Platform. When testing for these functions, we generated stripped binaries using different vendor SDKs (where relevant), as well as different projects and compilers (Keil, IAR, Clang), to account for vendor/compiler-introduced variations. argXtract was able to identify the correct function in each case. To further check the accuracy of argXtract's function pattern matching, we manually verified it against the HAL_Write_ConfigData and aci_gap_init functions within a real-world STMicroelectronics binary by comparing their functionality against the functions within an unstripped reference binary.

4.4 Correctness of Results

For correctness checks, we perform tests using generated binaries with known configurations, as well as verification using a real-world binary and associated device. We use a subset of ten binaries from our test set, targeting Nordic and STMicroelectronics chipsets, compiled using GCC, Keil and IAR, and implementing ANT and BLE. For the ANT binaries, we define different channel settings and encryption keys. For Nordic BLE binaries, we define 3 BLE services - Heart Rate, Device Information and a custom service - with very specific configurations. Obtained configurations must be an exact match for the output to be taken as correct. For STMicroelectronics BLE binaries, we define different advertising addresses and privacy configurations.[6] In our experiments, all of the conditions were satisfied for all test binaries within our control set, i.e., the configurations were extracted exactly as expected. We additionally purchased a device, the Goji Go Activity Tracker, whose firmware we had extracted from its companion mobile application. The tracker had two SIG services, as well as the Nordic firmware update service and a developer-defined service. Comparing the results obtained using `argXtract` with those we obtained from manual analysis (via a combination of device interaction using the nRF Connect app [56] and profiling using the `ATT-Profiler` tool [71]), we found that `argXtract` accurately extracted the configuration of the device.

5 CASE STUDY: BLE SECURITY AND PRIVACY (NORDIC)

BLE is a predominant communications technology within the IoT, installed on billions of endpoint devices [14]. In this section we present a case study for the identification of BLE configuration vulnerabilities in binaries that target Nordic chipsets. The Nordic BLE stack accepts configuration requests via supervisor calls.

Building the firmware dataset: BLE peripherals typically interface with a mobile application, many of which enable a firmware upgrade and/or factory reset procedure. The firmware used for this purpose is either included within the mobile application itself or is downloaded from a server. The firmware for Nordic chipsets is identifiable due to its specific structure and included files.

We programmatically extract Nordic BLE binaries from a dataset of 35,000+ BLE-enabled mobile apps, obtained from Androzoo [1] and Google Play. We describe here the results obtained by executing `argXtract` against 243 unique[7] binaries. To additionally check for the possibility of cloned firmware (which can result in the same output for slightly different binaries), we use `ssdeep` [44], with a threshold of 70% to account for the fact that a lot of the Nordic baseline code will be the same across files. Seven clusters are present within our dataset, with an average of 3 files per cluster. We account for these, where relevant, when presenting our results.

Execution environment: We executed `argXtract` on a VM running Ubuntu 18.04.3 LTS with 64GB RAM and 10 processor cores, with 8 parallel processes, taking RAM usage into consideration.

Section outline: The remainder of this section describes our findings. We first review the (lack of) protection applied to BLE data across the binaries for the link and application layers (§5.1). We then analyse instances of weakened pairing due to the use of fixed passkeys (§5.2). Finally, we examine privacy concerns identified for our dataset due to the use of static addresses (§5.3) and device/manufacturer names (§5.4). Each subsection provides an overview of the relevant aspect of the BLE technology, describes the extracted data, and discusses the results and the security or privacy implications. The supervisor calls that are targeted are provided in Table 2.

5.1 Security of BLE Data

BLE data is stored in discrete structures known as attributes. *Characteristics* are a type of attribute that hold the data of interest, e.g., heart rate readings. Multiple characteristics are grouped into a *service*, which is also a type of attribute. A third type of attribute, *descriptors*, describes a characteristic value. Restricting access to attributes is facilitated via attribute *permissions*, which are a combination of the following: *Access permissions* control whether the attribute can be read and/or written; *Authentication/encryption permissions* indicate whether the link between the two devices must be authenticated/encrypted before the attribute can be accessed; *Authorisation permissions* require developer-specific checks and can be used to implement end-to-end (i.e., application layer) security.

When link layer protection is required (i.e., via authentication permissions), three security *modes* can be applied. We discuss only Mode 1 in this study as we have not observed Modes 2 and 3 in real-world devices. The Bluetooth specification defines four levels of protection for Mode 1: *Level 1* - No security; *Level 2* - Unauthenticated pairing with encryption, i.e., encryption with no requirement for Man-in-the-Middle (MitM) protection. This can be achieved using the *Just Works* pairing model, which uses an all-zero key as an input to the key derivation algorithm and requires no user interaction; *Level 3* - Authenticated pairing with encryption, i.e., encryption with MitM protection. This requires either the *Passkey Entry* or *Numeric Comparison* pairing models, both of which require user interaction; *Level 4* - Authenticated LE Secure Connections pairing with encryption using a 128-bit strength key.

Services are freely readable but not writable. Characteristics and specific descriptors can have authentication and authorisation permissions. Characteristics also have certain *properties*, to determine how their data can be accessed. For example, a characteristic value can be *read*. It can also be obtained via *notifications* or *indications*, by writing to a descriptor called the Client Characteristic Configuration Descriptor (CCCD), whereby the BLE peripheral informs the connected device of changes in the characteristic value. While the outcome is somewhat similar (i.e., the connected device obtains the characteristic value), the security requirements for the two mechanisms are different. A read request requires that the connected device satisfy the read permissions for the characteristic value itself, while subscribing to notifications requires that the connected device satisfy the *write* permissions of the characteristic's CCCD.

Extracting characteristic security configurations: We executed `argXtract` against our dataset with a maximum execution time of 1.5 hours per trace. This returned 199 valid output files.[8]

[6] Configurations were dependent on the options made available by the vendor.
[7] Determined by the SHA256 computed over the file bytes.
[8] We perform stringent validity checks and consider a characteristic data structure to be valid only if applicable characteristic permissions (based on characteristic properties) have correct values, as described by the BLE specification. Further, we reject any characteristic that cannot be uniquely tied to a service, even if the characteristic structure is otherwise complete and correct. See Appendix B for more details.

Table 2: Calls Of Interest used in case studies.

Case Study	Security Metric	Calls Of Interest
Nordic BLE	Security of BLE Data (§5.1)	Characteristic security is defined when the characteristic is added using `sd_ble_gatts_characteristic_add`. A characteristic is tied to a service via a `service_handle` field output by `sd_ble_gatts_service_add`.
	Use of Fixed Passkeys (§5.2)	Fixed passkeys are set via the `sd_ble_opt_set` svc with an `opt_id` (first argument) of 34.
	User Tracking due to Fixed Addresses (§5.3)	Address type can be changed via `sd_ble_gap_address_set` (older stacks); `sd_ble_gap_addr_set` and `sd_ble_gap_privacy_set` (newer versions). Default is random static address set at time of manufacture and unchanging during device lifetime.
	Manufacturer/Device Names and Privacy (§5.4)	Device name is set using `sd_ble_gap_device_name_set`. Manufacturer Name String is included within the Device Information Service (obtained in §5.1).
BlueNRG	BLE Address Privacy (§6.1.1)	Public addresses are configured using `aci_hal_write_config_data` [77], which internally calls `HAL_Write_ConfigData` with same arguments. Validation is performed via the extracted 2nd argument, which is a length field (value must be 6 for addresses). Privacy is configured via `aci_gap_init`. The function performs several tasks, most of which require runtime information. We exploit the fact that the function adds the BLE GAP service to the database when generating our test sets.
	BLE Pairing Security (§6.1.2)	Two function calls to enable BLE security: (i) `aci_gap_set_io_capability`, to set the device's input-output capability, and (ii) `aci_gap_set_authentication_requirement`, to set the pairing requirements (such as bonding, MitM protection, etc) [77]. Authorisation permissions are set using the `aci_gap_set_authorization_requirement` function.
Nordic ANT	ANT Channel Security (§6.2.1)	ANT is enabled using the `sd_ant_enable` supervisor call. Within this call, the number of required channels and the number (out of those created) that should be encrypted are specified.

Analysis times were < 2 minutes for half the binaries, 2-10 minutes for 25% of the binaries, and more than 10 minutes for the remaining binaries. The 199 binaries contained 6 of the previously mentioned clusters (identified using `ssdeep`). Examination of the corresponding output files showed that files within each cluster had the same service configurations. We therefore consider only 188 unique outputs.

5.1.1 Insufficient protection for BLE data. In this section, we discuss the protection applied to BLE data for the binaries in our dataset. Because BLE characteristics can either be defined by the Bluetooth Special Interest Group (SIG), with SIG-specified security configurations, or defined by the device developer with developer-specified security, we analyse the two instances separately.

(i) Protection for SIG-defined BLE data `argXtract` extracted SIG-defined characteristics from 103 binaries. We found that all devices follow SIG specifications regarding security configurations. While most SIG-defined characteristics have no security requirements, the results revealed an interesting observation for the SIG-defined characteristics that *do* have security requirements. In many such cases, the SIG specifies a choice of protection levels, normally Mode 1 Level 2 *or* Mode 1 Level 3. These can be achieved using *Just Works* or *Passkey Entry* pairing, respectively. While both *Just Works* and *Passkey Entry* are known to be vulnerable to passive eavesdropping attacks [15], *Passkey Entry* should be the choice for greater security, as it provides MitM protection. However, our results show that device developers have invariably opted for the *lower* security level, i.e., Mode 1 Level 2. We believe this may be due to lack of a user interface on the devices precluding the use of *Passkey Entry*. Note that, even if the BLE device does have a user interface, as long as the data only specifies a security requirement of Mode 1 Level 2, an attacker can often manipulate the pairing process by specifying no input-output capabilities, downgrade the pairing model to *Just Works*, and thereafter access the data.

(ii) Protection for developer-defined BLE data `argXtract` extracted at least one developer-defined characteristic from 170 binaries. Table 3 summarises the link layer and application layer protection applied to the developer-defined characteristics, broken down into readable and writable characteristics. From the table, we conclude that protection for reads is virtually non-existent at the link layer, with only five firmware binaries specifying Mode 1 Level 2 authentication requirements. Authorisation requirements are also not prevalent among readable characteristics, with only seven binaries specifying protection at higher layers. Writable characteristics fare similarly to readable characteristics in terms of link layer protection, with only four binaries specifying Mode 1 Level 2 authentication requirements. App-layer protection is slightly better for writable characteristics, but a significant proportion of binaries apply no protection at all to their writable characteristics (apart from those provided by Nordic itself, for firmware upgrades).

Security implications: The security of BLE data is strongly associated with authentication and authorisation permissions. *Having freely accessible BLE characteristics means any user in the vicinity of the BLE peripheral will be able to read and write the data*, subject to the characteristic being readable/writable. For that matter, even if the characteristic is protected by *Just Works* pairing, an attacker in the vicinity can pair with the device and access its data. We verified this with a fitness tracker, from which we were able to access characteristic values without pairing. Further, even if strong link layer protection is present, *absence of application layer protection makes the data vulnerable to access by unauthorised apps* [70]. We verified this with a custom Android app and emulated BLE device.

» *Implications for readable data within the dataset*: Among the binaries that had no protection for readable characteristics, we found (through the device name extracted in §5.4) numerous fitness trackers and healthcare devices, all of which potentially store detailed information regarding a user's activity or health. No protection or only *Just Works* protection means this personal and sensitive data is vulnerable to unauthorised access, via local *and* remote attacks.

» *Implications for writable data within the dataset*: Within the binaries that had writable characteristics, we found one that contained the SIG-defined Human Interface Device (HID) service. This only had Mode 1 Level 2 link layer protection applied to its characteristics. Again, this security requirement can be satisfied by *Just Works* pairing, which means that an attacker could transmit unsolicited messages to the HID device, and also read and modify the keyboard characters that are transmitted between the HID device and host via a MitM attack. This has been demonstrated in [43]. We have informed the developer of this vulnerability.

Table 3: Protection applied to developer-defined data.

Description	Reads†	Writes
Binaries with characteristics that have appropriate property	167	169
Binaries with Mode 1 Level 2 link layer protection	5	4
Binaries with Mode 1 Level 3 link layer protection	0	0
Binaries with application layer security	7	69*

All entries refer to developer-defined characteristics. †Including notifications/indications. *24 excluding Nordic DFU control point, from Nordic DFU library.

The vast majority of devices applied no protection to *any* of their writable characteristics. Among these were smart switches, medical respiratory devices and ECG monitors. Writing random bytes to characteristics on such devices could cause the devices to function improperly or cease to function entirely. If the behaviour of the device corresponding to the written values is known to an attacker, then they can write carefully chosen values in order to modify the expected behaviour, possibly with harmful consequences.

5.1.2 Different permissions for read vs. notify. As mentioned in §5.1, the value held within a characteristic can be accessed in different ways, either using a direct read request or via notifications/indications, and even though the mechanisms of access differ, the ultimate outcome is similar. Our results indicated that one binary within our dataset contained characteristics that had both `read` and `notify` properties, but with different security properties set for the two types of access. Mode 1 Level 2 security was required to be able to *read* the characteristics' values, while the values could be freely accessed via *notifications* (their CCCDs were writable without the need for any pairing or higher-layer protection).

Security implications: Different security levels for different value acquisition methods implies that the data can always be accessed using the less secure mechanism. In addition, there may be a false sense of security, as the protection will be assumed to be higher than it actually is. This shows that *developers may unintentionally leave "gaps" in security*, particularly when incorporating different functionalities. We have informed the developer about this issue.

5.2 Use of Fixed Passkeys

As mentioned in §5.1, *Passkey Entry* is a BLE pairing model which provides MitM protection by requiring that a user manually key in a passkey that is displayed on the BLE peripheral. However, some developers program a fixed passkey into the peripheral. This might be because many BLE peripherals don't have input/output capabilities (i.e., keypad or display), but could also originate from bad practices when programming devices that *do* have these capabilities.

Identifying fixed passkeys: `argXtract` identified a smartwatch binary within the dataset that called `sd_ble_opt_set` with an `opt_id` of 34, setting a fixed passkey of `0x303030303030`, i.e. "000000".

Security implications: Fixed passkeys undermine the security of the *Passkey Entry* model, particularly if the same passkey is used for all devices of a certain brand. In such a scenario, an attacker would only need to know the passkey for one device in order to be able to covertly connect to any device of the same brand. This effectively *removes the MitM protection afforded by the Passkey Entry model*. With the binary we identified, a fixed passkey of "000000" equates *Passkey Entry* to the *Just Works* model.

Table 4: Address types used in BLE peripherals.

Address Type	#Bins	Address Type	#Bins	Address Type	#Bins
Public	29	Private nonresolvable	1	Unknown	4
Random static	208	Private resolvable	1		

5.3 User Tracking due to Fixed Addresses

BLE peripherals periodically transmit messages on advertising channels in order to enable incoming connections. These messages contain the peripheral's hardware address and could be used to track the device. To overcome this, the BLE specification defines *resolvable private addresses*, which enable a device to change its advertising address while still allowing for reconnections from bonded peers.

There are in fact four types of addresses that can be used with BLE: Public, Random Static, Private Resolvable, and Private Non-resolvable. Public addresses do not change during the lifetime of a device; Random Static addresses do not change during a single power cycle and may not change for the lifetime of the device; Private Resolvable addresses change periodically in such a way as to enable reconnections by a bonded peer; Private Non-resolvable addresses change periodically, but do not allow for reconnections.

Extracting advertising address type: `argXtract` identified that 35 out of the 243 firmware files included one of the `svc` numbers for performing address type selection/setting, which meant that the remaining 208 files used the default setting of a random static address. Table 4 depicts a breakdown of the address types used within the BLE peripherals in our dataset. Out of the 243 binaries in our dataset, only a single binary used resolvable private addresses. The device name (obtained in §5.4) revealed that the binary was for a personal protection device. One binary within the dataset used non-resolvable addresses, which means it will not be vulnerable to tracking but will also not be able to form bonds with other devices. Its device name did not reveal its functionality. We found that overall, the results indicated that at least 95% of the BLE binaries use static (random or public) addresses.

Privacy implications: Because BLE peripherals tend to advertise constantly when not in a connection, *the use of public or random static addresses in advertising messages opens the BLE device, and by extension (depending on the device) its owner, to tracking*. In crowded locations such as shopping centres, repeated visits by a user can be covertly tracked simply by monitoring BLE advertisements and logging the device addresses. This has been previously demonstrated in [27]. It has also been shown to be feasible to set up a botnet to track users across a range of locations [40]. These attacks are particularly relevant in the case of devices such as wearables, which are generally always on the user's person. We found that *all of the wearable binaries within our dataset used public or fixed random static addresses*.

5.4 Manufacturer/Device Names and Privacy

BLE advertising messages usually contain the peripheral's name, which is often used by users to identify a device from (potentially) a number of other BLE devices that are also advertising in the vicinity. Peripherals may also include a Manufacturer Name String, which is normally obtained by sending a scan request. These advertising messages require no authentication in order to be read.

Extracting device and manufacturer names: argXtract extracted non-null values for device/manufacturer name from 156 binaries. An analysis of the names revealed that our dataset contained firmware from a variety of BLE devices, including fitness trackers, beacons, electric switch controls, parking aids, security devices, personal protection devices, medical equipment and behavioural monitoring devices.

Privacy implications: Device names can reveal a lot about the nature of the device. This is particularly concerning when the device is related to a user's health, or is of an otherwise private nature. Because no active connections are required to read advertising data, an attacker would simply need to monitor the BLE advertising channels and perhaps send a scan request for additional information. By continuously scanning BLE advertisements, extracting the device and manufacturer name, and combining this information with the Received Signal Strength Indicator (RSSI), along with user observation, an attacker may be able to determine which devices belong to which users in the vicinity. This could defeat private addresses (§5.3), as an attacker might instead be able to use the device name, along with other advertising data, to track the device [12, 19, 31]. Further, if a particular device has known issues (such as those identified in this work), then the attacker can take advantage of the device name to identify exploitable devices.

6 APPLICABILITY STUDIES

In this section, we apply argXtract to two smaller datasets, representing non-BLE technologies and non-svc-based stacks. In §6.1, we present a case study for the identification of BLE configuration vulnerabilities in firmware that targets STMicroelectronics chipsets, specifically the BlueNRG processor. Configurations for BlueNRG are performed via function calls. We therefore employ the function pattern matching module of argXtract (described in §3.4.2). In §6.2, we present a case study for the identification of vulnerabilities within binaries targeting Nordic Semiconductor chipsets and implementing the ANT technology stack. All relevant function calls and svcs are provided in Table 2.

6.1 Case Study: BLE Security and Privacy via Function Pattern Matching (BlueNRG)

We manually analysed 500 real-world .bin files extracted from APKs and found that two were STMicroelectronics BlueNRG binaries. argXtract identified that both had an application code base of 0x10051000, which corresponds to BlueNRG-1 v2.1+ [76].

6.1.1 BLE Address Privacy. In this section, we describe tests to identify the use of private addresses within BlueNRG binaries.

Extracting address configurations: argXtract revealed that one of the real-world binaries contained a public address derived from BlueNRG code samples. This, along with the binary's name, led us to conclude that the binary was for demonstration purposes. The second binary was a BLE-enabled cyclist safety aid. It *did not have privacy enabled*.

Privacy implications: A cyclist safety aid is likely to be about the user's person whenever they are cycling. A fixed address emanating from the device at all times enables the user to be tracked over time, as discussed in §5.3.

6.1.2 BLE Pairing Security. With BlueNRG binaries, if a BLE characteristic has authentication requirements, then specific configurations must be performed to enable pairing. We exploit two pairing-related functions in our tests. We additionally check for authorisation requirements.

Extracting pairing configurations: Focusing on the cyclist safety aid, argXtract found that the binary had no calls either to aci_gap_set_io_capability or to aci_gap_set_authorisation_requirement. This means that *BLE security was not enabled*.

Security implications: A lack of security in a cyclist safety aid means that an attacker could connect to the device and send commands to it without the need for any authentication. This could have serious consequences for the cyclist's safety. We have informed the developer regarding the identified issues.

6.2 Case study: ANT Security (Nordic)

To acquire Nordic ANT binaries, we follow the same procedure as for Nordic BLE (see §5), but focus on a different set of svc numbers. We obtained 9 ANT binaries from APKs.

6.2.1 ANT Channel security. ANT communications are channel-based, with a channel connecting two or more nodes together. Some ANT devices can have multiple channels. To secure the channels at the network layer, ANT supports 8-byte network keys and 128-bit AES encryption [34].

Extracting channel security configurations: argXtract extracted channel configuration parameters from 9 real-world ANT binaries, corresponding to 7 indoor exercise bikes, an analytical bike light (i.e., a bike light with additional sensors), and a heart rate monitor. Three binaries defined a single ANT channel, four defined 2 channels and two defined 4 channels. *None of the binaries specified encryption for any of their ANT channels.*

Security implications: As with the findings discussed in §5.1 for BLE, in ANT too data will be vulnerable to unauthorised access if channel security is not enabled. One of the binaries that was tested was a heart rate monitor, which means that a user's heart rate measurements (i.e., health indicators) are vulnerable.

7 LIMITATIONS AND FUTURE WORK

In this section, we discuss some limitations of argXtract, which could provide potential for future work.

Edge cases: argXtract is able to analyse most Cortex-M binaries. However, as seen in one example in §4, there are edge cases where the .text segment is split into subsections, with different address offsets for each subsection, where argXtract is unable to obtain individual code bases and accurate function estimates. This improvement is left as future work.

Function identification: With function boundary identification, argXtract assumes that the instructions belonging to a function are laid out in a contiguous range. If a function is split up into disjoint blocks of instructions, then argXtract may identify each such block as a separate function.

COI and callsite identification: As mentioned in §3.4, the function pattern matching performed by argXtract uses manually-defined test sets, when function outputs or artefacts are distinguishable. If two functions produce the same output for the same input

and one is not nested within the other, then a single function cannot be matched. Further, the function pattern matching process can take several hours when a binary contains a large number of functions. An ideal alternative would be *automated* function pattern matching, *without* executing function code. The most popular method to achieve this at present is via machine learning techniques. However, this requires a sufficiently large annotated training set for each function of interest, which is not yet available for the types of vendor-specific configuration functions that are of interest here. With callsite identification, direct calls are identified. However, calls via `blx` are not. (Note that `blx` will be identified and handled during tracing, but not for COI identification.)

8 RELATED WORK

In this section, we discuss previous works related to firmware analysis and IoT security. While it may seem like most aspects of firmware analysis have already been covered, most existing works focus on Linux-based systems [60]. We reiterate that the analysis of stripped binaries targeting non-traditional operating systems and the ARM Thumb instruction set, which is increasingly favoured by IoT peripherals, has still not been explored sufficiently.

Analysis of stripped binaries: The analysis of stripped binaries, particularly function block identification, has been the subject of widespread study. Control flow analysis has been used in [4, 36, 59, 61, 65] to determine functions in PE, ELF, COFF and XCOFF binaries, and a QEMU+LLVM approach for function boundary identification was presented in [29]. These approaches may not be suited to ARM IoT analysis due to errors introduced by inline data and compiler-introduced constructs such as Thumb switch-case conditions. Machine learning (ML) has also been proposed for identifying function entry points [11, 66, 68], but this approach requires a sufficiently large labelled training set, which is currently not available for IoT peripheral binaries. A semantics-based approach was used in Jima [3] for ELF x86/x86-64, which employs techniques for computing jump tables that are similar to those used in `argXtract` for computing table branch addresses. To the best of our knowledge, we are the first to employ the techniques we have described in this work for identifying the application code base,[9] the `.data` segment, as well as several sources of inline data. The inline data identification employed by `argXtract` also improves the performance of function identification (as we have shown in §4) and subsequent tracing.

Function matching and labelling: One approach for function pattern matching is to compute statistical similarities between instruction sequences of functions [46, 54], but this may suffer poor performance due to compiler-introduced variations and optimisations [17]. Dynamic similarity testing via function execution was employed in [30]. While this is in some ways similar to our approach, `argXtract` looks for functional *equivalence* based on *known* function behaviour, while [30] considers function *similarity* based on *random* executions. Most current approaches favour ML techniques [49, 58, 68, 89] but, as mentioned previously, this requires large training sets.

[9] Similar to our approach, Wen et al. [87] also use vector table entries as one input to compute the application code base, but without considering default handlers as we do.

Security analysis and patching of IoT firmware: Large-scale security analyses of embedded firmware files, predominantly Linux and VxWorks-based, were presented in [24, 25]. FIE [28], built from the KLEE symbolic execution engine, identifies vulnerabilities in embedded MSP430 firmware. Firmalice [69] detects authentication bypass vulnerabilities within the firmware of Linux and VxWorks-based binaries. FirmFuzz [73] specifically targets IoT firmware and is intended for security analysis. It uses QEMU and targets unstripped Linux-based binaries. These works analyse binaries that target at least pared-down versions of fully-fledged operating systems. They would not be suitable for analysing stripped firmware of embedded devices that do *not* have a proper OS. InternalBlue [48] enables testing and patching of Broadcom Bluetooth firmware, while LightBlue [88] analyses and performs debloating of unneeded Bluetooth profiles and HCI commands within firmware to reduce the potential attack surface. The randomness of RNGs used in Bluetooth chipsets was measured via firmware analysis in [82].

BLE configuration security analysis: On the BLE front, previous works have explored the security and privacy configurations of BLE peripherals by analysing devices [6, 27, 72, 84], and mobile applications [70, 94]. However, device analysis is expensive and may not directly provide indications about higher-layer security, while mobile applications do not provide insights about low-level pairing mechanisms. Multi-faceted analysis of BLE fitness trackers, in terms of configuration and behaviour, was performed in [23, 26, 39].

Independently to us, Wen et al. [87] developed a tool named `FirmXRay` that identifies BLE link layer configuration vulnerabilities by targeting supervisor calls on Nordic and ICalls on Texas Instruments BLE binaries. To compare `argXtract` and `FirmXRay`, we executed them against a random subset of 300+ binaries from the `FirmXRay` dataset. We found that a direct comparison was not possible due to insufficient information within `FirmXRay`'s output data structures (further details provided in Appendix C). In general, while `FirmXRay` is geared towards BLE vulnerabilities, our work is capable of handling generic analysis of any technology that targets ARM Cortex-M binaries. Further, `FirmXRay` only handles supervisor calls and ICalls, whereas `argXtract` performs function pattern matching to identify *any* function (provided the requisite artefacts can be identified within memory/registers). The template-based approach used in our framework also enables easy addition of new test functions. Within the BLE analysis, Wen et al. [87] have confined the discussion to link layer vulnerabilities, while we discuss application layer issues as well.

9 CONCLUSION

In this work, we present `argXtract`, a framework for performing partial-knowledge automated analyses of stripped IoT binaries, to extract security-relevant configuration information from ARM Cortex-M firmware. `argXtract` overcomes the challenges inherent to the analysis of stripped Cortex-M binaries and enables bulk processing of IoT peripheral firmware files. We use `argXtract` to extract configurations from three datasets: Nordic Bluetooth Low Energy (BLE) binaries, STMicroelectronics BlueNRG binaries, and Nordic ANT binaries. Our results reveal widespread lack of protection for data, inconsistent data access controls and serious privacy vulnerabilities.

REFERENCES

[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 468–471.

[2] Sergi Alvarez. 2021. radare2. https://github.com/radareorg/radare2.

[3] Jim Alves-Foss and Jia Song. 2019. Function boundary detection in stripped binaries. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 84–96.

[4] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 177–189.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the Mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*. 1093–1110.

[6] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. 2020. Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy. *ACM Trans. Priv. Secur.* 23, 3, Article 14 (June 2020).

[7] ARM. 2012. Supervisor calls. Available: https://developer.arm.com/documentation/dui0471/g/handling-processor-exceptions/supervisor-calls [Accessed: 28 July 2020].

[8] ARM. 2016. Calling SVCs from an application. Available: https://developer.arm.com/documentation/dui0471/m/handling-processor-exceptions/calling-svcs-from-an-application [Accessed: 28 July 2020].

[9] Arm. 2020. Record shipments of Arm-based chips in previous quarter. Available: https://www.arm.com/company/news/2020/02/record-shipments-of-arm-based-chips-in-previous-quarter [Accessed: 28 June 2020].

[10] ARM. 2021. Vector table. Available: https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/exception-model/vector-table [Accessed: 03 July 2020].

[11] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*. 845–860.

[12] Johannes K Becker, David Li, and David Starobinski. 2019. Tracking anonymized Bluetooth devices. *Proceedings on Privacy Enhancing Technologies* (2019), 50–65.

[13] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[14] Bluetooth Special Interest Group. 2019. 2019 Bluetooth Market Update. Available: https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update [Accessed 01-Feb-2021].

[15] Bluetooth Special Interest Group. 2019. Bluetooth Core Specification v5.2.

[16] Bluetooth Special Interest Group. 2019. Intro to Bluetooth Low Energy. Available: https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-low-energy/ [Accessed: 27 July 2020].

[17] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. 1–10.

[18] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.

[19] Guillaume Celosia and Mathieu Cunche. 2019. Saving private addresses: an analysis of privacy issues in the bluetooth-low-energy advertising mechanism. In *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. 444–453.

[20] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*, Vol. 16. 1–16.

[21] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.

[22] Richard Chirgwin. 2016. Finns chilling as DDoS knocks out building control system. Available: https://www.theregister.com/2016/11/09/finns_chilling_as_ddos_knocks_out_building_control_system. [Accessed: 11 June 2020].

[23] Jiska Classen, Daniel Wegemer, Paul Patras, Tom Spink, and Matthias Hollick. 2018. Anatomy of a vulnerable fitness tracking system: Dissecting the Fitbit cloud, app, and firmware. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 1 (2018), 1–24.

[24] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*. 95–110.

[25] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 437–448.

[26] Britt Cyr, Webb Horn, Daniela Miao, and Michael Specter. 2014. Security Analysis of Wearable Fitness Devices (Fitbit). *Massachusetts Institute of Technology* (2014).

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

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

[29] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. [n. d.]. rev. ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*.

[30] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 303–317.

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

[32] Jan Friebertshäuser, Florian Kosterhon, Jiska Classen, and Matthias Hollick. 2020. Polypyus–The Firmware Historian.

[33] Garmin Canada Inc. 2020. What is ANT+. Available: https://www.thisisant.com/consumer/ant-101/what-is-ant [Accessed: 27 July 2020].

[34] Garmin Canada Inc. 2020. What kind of security does ANT provide? Available: https://www.thisisant.com/developer/resources/tech-faq/what-kind-of-security-does-ant-provide-1 [Accessed: 07 Dec 2020].

[35] Liam Goudge and Simon Segars. 1996. Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications. In *COMPCON'96. Technologies for the Information Superhighway Digest of Papers*. IEEE, 176–181.

[36] Laune C Harris and Barton P Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News* 33, 5 (2005), 63–68.

[37] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. [n. d.]. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[38] Hex-Rays. 2021. IDA pro disassembler. Available: https://www.hex-rays.com/products/ida/support/download_freeware/. [Accessed: 31 Jan 2021].

[39] Andrew Hilts, Christopher Parsons, and Jeffrey Knockel. 2016. Every Step You Fake: A Comparative Analysis of Fitness Tracker Privacy and Security. (2016).

[40] Taher Issoufaly and Pierre Ugo Tournoux. 2017. BLEB: Bluetooth Low Energy Botnet for large scale individual tracking. In *2017 1st International Conference on Next Generation Computing Applications (NextComp)*. IEEE, 115–120.

[41] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. 2020. An Empirical Study on ARM Disassembly Tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 401–414. https://doi.org/10.1145/3395363.3397377

[42] Anastasis Keliris and Michail Maniatakos. 2018. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. *arXiv preprint arXiv:1812.03478* (2018).

[43] Gerhard Klostermeier and Matthias Deeg. 2018. Case Study: Security of Modern Bluetooth Keyboards. (2018). Available: https://www.syss.de/fileadmin/dokumente/Publikationen/2018/Security_of_Modern_Bluetooth_Keyboards.pdf [Accessed: 30 Nov 2020].

[44] Jesse Kornblum, Helmut Grohne, and Tsukasa OI. 2021. ssdeep - Fuzzy hashing program. Available: https://ssdeep-project.github.io/ssdeep/index.html [Accessed 16-Mar-2021].

[45] Selena Larson. 2017. FDA confirms that St. Jude's cardiac devices can be hacked. Available: https://money.cnn.com/2017/01/09/technology/fda-st-jude-cardiac-hack. [Accessed: 11 June 2020].

[46] Yeo Reum Lee, BooJoong Kang, and Eul Gyu Im. 2013. Function matching-based binary-level software similarity calculation. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*. 322–327.

[47] Franco Loi, Arunan Sivanathan, Hassan Habibi Gharakheili, Adam Radford, and Vijay Sivaraman. 2017. Systematically evaluating security and privacy for consumer IoT devices. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*. 1–6.

[48] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. 2019. InternalBlue-Bluetooth binary patching and experimentation framework. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. 79–90.

[49] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 309–329.

[50] Sanjay M Mishra. 2015. *Wearable Android: Android Wear and Google Fit app development*. John Wiley & Sons.

[51] Mitre. 2015. CVE-2015-2880. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2880 [Accessed: 14 July 2020].

[52] Mitre. 2018. CVE-2018-10825. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10825 [Accessed: 14 July 2020].

[53] Mitre. 2019. CVE-2019-16518. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16518 [Accessed: 14 July 2020].

[54] Ginger Myles and Christian Collberg. 2005. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*. 314–318.

[55] National Security Agency. 2020. Ghidra. https://github.com/NationalSecurityAgency/ghidra.

[56] Nordic Semiconductor. 2020. nRF Connect for Mobile. https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile.

[57] Nordic Semiconductor ASA. 2020. SoftDevices. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fug_gsg_ses%2FUG%2Fgsg%2Fsoftdevices.html [Accessed: 03 July 2020].

[58] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. 2020. Probabilistic Naming of Functions in Stripped Binaries. In *Annual Computer Security Applications Conference*. 373–385.

[59] Manish Prasad and Tzi-cker Chiueh. 2003. A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks.. In *USENIX Annual Technical Conference, General Track*. 211–224.

[60] Abdullah Qasem, Paria Shirani, Mourad Debbabi, Lingyu Wang, Bernard Lebel, and Basile L Agba. 2021. Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–42.

[61] Rui Qiao and R Sekar. [n. d.]. Function interface analysis: A principled approach for function recognition in COTS binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[62] Nguyen Anh Quynh. 2020. Capstone: The Ultimate Disassembler. https://www.capstone-engine.org.

[63] Nguyen Anh Quynh. 2020. Unicorn The Ultimate CPU emulator. Available: https://www.unicorn-engine.org [Accessed:25 Oct 2020].

[64] Radware. 2006. 'BrickerBot' Results In PDoS Attack. Available: https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-permanent-denial-of-service/. [Accessed: 11 June 2020].

[65] Giridhar Ravipati, Andrew R Bernat, Nate Rosenblum, Barton P Miller, and Jeffrey K Hollingsworth. 2007. Toward the deconstruction of Dyninst. *Univ. of Wisconsin, technical report* (2007), 32.

[66] Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code.. In *AAAI*. 798–804.

[67] Vinay Sachidananda, Suhas Bhairav, and Yuval Elovici. 2019. Spill the Beans: Extrospection of Internet of Things by Exploiting Denial of Service. *EAI Endorsed Transactions on Security and Safety* 6, 20 (2019).

[68] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*. 611–626.

[69] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.. In *NDSS*.

[70] Pallavi Sivakumaran and Jorge Blasco. 2019. 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 (USENIX Security 19)*. 1–18.

[71] Pallavi Sivakumaran and Jorge Blasco Alis. 2017. ATT Profiler. https://github.com/projectbtle/att-profiler.

[72] Pallavi Sivakumaran and Jorge Blasco Alis. 2018. A Low Energy Profile: Analysing Characteristic Security on BLE Peripherals. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 152–154.

[73] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. 2019. FirmFuzz: automated IoT firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*. 15–21.

[74] Mark Stanislav and Tod Beardsley. 2015. Hacking IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities. Available: https://www.rapid7.com/globalassets/external/docs/Hacking-IoT-A-Case-Study-on-Baby-Monitor-Exposures-and-Vulnerabilities.pdf. [Accessed: 11 June 2020].

[75] Statista Research Department. 2019. IoT connected devices worldwide 2030. Available: https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/. [Accessed: 29 June 2020].

[76] STMicroelectronics. 2018. AN4869: The BlueNRG-1, BlueNRG-2 BLE OTA (over-the-air) firmware upgrade.

[77] STMicroelectronics. 2019. PM0257: BlueNRG-1, BlueNRG-2 BLE stack v2.x programming guidelines.

[78] Texas Instruments. 2020. Bluetooth Low Energy software stack. Available: https://www.ti.com/tool/BLE-STACK [Accessed: 02 July 2020].

[79] Texas Instruments. 2020. A fully compliant Zigbee 3.x solution: Z-Stack. Available: https://www.ti.com/tool/Z-STACK [Accessed: 02 July 2020].

[80] Iain Thomson. 2016. Wi-Fi baby heart monitor may have the worst IoT security of 2016. Available: https://www.theregister.com/2016/10/13/possibly_worst_iot_security_failure_yet. [Accessed: 11 June 2020].

[81] Thread Group. 2019. What is Thread. Available: https://www.threadgroup.org/what-Is-thread [Accessed: 27 July 2020].

[82] Jörn Tillmanns, Jiska Classen, Felix Rohrbach, and Matthias Hollick. 2020. Firmware Insider: Bluetooth Randomness is Mostly Random. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[83] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9.

[84] Jiliang Wang, Feng Hu, Ye Zhou, Yunhao Liu, Hanyi Zhang, and Zhe Liu. 2020. BlueDoor: breaking the secure information flow via BLE vulnerability. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 286–298.

[85] KC Wang. 2017. Embedded real-time operating systems. In *Embedded and Real-Time Operating Systems*. Springer, 401–475.

[86] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. [n. d.]. Looking from the mirror: evaluating IoT device security through mobile companion apps. In *28th USENIX Security Symposium (USENIX Security 19)*.

[87] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware. (2020).

[88] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. 2021. LIGHTBLUE: Automatic Profile-Aware Debloating of Bluetooth Stacks. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[89] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 363–376.

[90] Xiaokang Yin, Shengli Liu, Long Liu, and Da Xiao. 2018. Function recognition in stripped binary of embedded devices. *IEEE Access* 6 (2018), 75682–75694.

[91] Kim Zetter. 2015. Hackers Can Seize Control of Electric Skateboards and Toss Riders. Available: https://www.wired.com/2015/08/hackers-can-seize-control-of-electric-skateboards-and-toss-riders-boosted-revo/ [Accessed: 27 July 2020].

[92] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. 2019. Discovering and Understanding the Security Hazards in the Interactions between IoT Devices, Mobile Apps, and Clouds on Smart Home Platforms. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1133–1150.

[93] Zigbee Alliance. 2019. What is Zigbee? Available: https://Zigbeealliance.org/solution/Zigbee/ [Accessed: 27 July 2020].

[94] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2019. Automatic fingerprinting of vulnerable BLE IoT devices with static uuids from mobile apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1469–1483.

A ANALYSIS OF POTENTIAL INFORMATION SOURCES FOR IOT VULNERABILITIES

There are several possible sources for obtaining information regarding the configurations of IoT devices. These include the devices themselves, the firmware they run, or any application or website they interface with. We analyse the merits and shortcomings of each of these potential sources below:

Devices: Several security and privacy analyses have been conducted against IoT devices [27, 47, 67, 72]. Interfacing with physical devices can reveal behavioural characteristics, particularly those where user interaction is required. Combining hardware device tests with an analysis of communication interfaces can yield even more details. However, large-scale analyses can be difficult to automate as well as being prohibitively expensive due to the need for purchasing devices. Further, a variety of communication protocols may be used, particularly with IoT peripherals, which could require specialist hardware or software for each traffic analysis.

Mobile applications: IoT peripherals tend to interface with a companion mobile app. Such apps are often available as large repositories, are reasonably easy to analyse, and can provide indications of higher-layer processing. As such, they have been used in security analyses to identify vulnerabilities in the associated devices [20, 21, 70, 86, 94]. However, one app may interface with multiple devices, making it difficult to separate out relevant information for a single device. Also, low-level protocol details may occur at the mobile OS level, transparent to the app, or the app may act as a conduit between a device and a server without processing the data itself. In such cases, it may not be possible to get a complete picture of the IoT device's security configuration via the app.

Web interface: If an IoT device communicates with an external server, the exchanged messages may reveal information regarding its configuration, particularly if it receives configuration commands from the server. However, performing tests against these servers may have legal implications. Also, automated tests may not always be feasible on a large scale in the absence of physical devices because the server may require authenticated requests from the device [92].

Firmware: The firmware on an IoT device tends to apply to a single type of device and generally reflects its configuration and functionality exactly. This has led firmware binaries to be the information source of choice for a number of security analyses [24, 25, 28, 42, 69]. However, firmware binaries are not always easy to obtain, as developers do not always make them publicly available. More importantly, firmware analysis is, due to its own nature, far more complex than, e.g., mobile app analysis.

B VALIDITY CHECKS

In order to ensure that our analysis is performed on valid data, we perform stringent validity checks on the output obtained from `argXtract`. This is particularly done in the case of complex output structures such as those for `sd_ble_gatts_characteristic_add`, which has several levels of nested fields.

`argXtract` stores service handles for every service that is added via the `sd_ble_gatts_service_add` call. For each characteristic extracted from `sd_ble_gatts_characteristic_add`, we attempt to match it to a service handle. If a characteristic cannot be uniquely matched with a service in this way, it is not considered further (even if its permission structure is fully valid as described below).

A BLE characteristic has certain properties (e.g., `read`, `write`, `notify`) indicating how it may be accessed. It will also have corresponding permissions. We test for the validity of such permissions according to the properties. That is, if a characteristic has the `read` property, we ensure that its `read_perm` (i.e., read permissions) has a valid security mode and level, as described in the BLE specification. If a characteristic has the `notify` property, we ensure that the write permissions *for its CCCD* are valid. If a property isn't set for a characteristic, then an invalid permission structure can be ignored. We obtained invalid results for a single binary within our dataset.

We also perform random manual checks on known characteristics, i.e., whose properties and permissions are known (e.g., SIG-defined or Nordic DFU). In this manner, we endeavour to produce the most accurate analysis results.


```
"sd_ble_gatts_characteristic_add": [
  {
    "service_handle": "4b8b",
    "p_char_md": {
      "char_props": {
        "ignore": 0,
        "auth_signed_wr": 0,
        "indicate": 0,
        "notify": 1,
        "write": 0,
        "write_wo_resp": 0,
        "read": 0,
        "broadcast": 0
      },
      "char_ext_props": {
        "wr_aux": 0,
        "reliable_wr": 0
      },
      "p_char_user_desc": 0,
      "char_user_desc_max_size": 0,
      "char_user_desc_size": 0,
      "p_char_pf": {
        "format": 0,
        "exponent": 0,
        "unit": 0,
        "name_space": 0,
        "desc": 0
      },
      "p_user_desc_md": {
        "read_perm": {
        "security_level": 0,
        "security_mode": 0
        },
        "write_perm": {
        "security_level": 0,
        "security_mode": 0
        },
        "ignore": 0,
        "wr_auth": 0,
        "rd_auth": 0,
        "vloc": 0,
        "vlen": 0
      },
      "p_cccd_md": {
        "read_perm": {
        "security_level": 1,
        "security_mode": 1
        },
        "write_perm": {
        "security_level": 1,
        "security_mode": 1
        },
        "ignore": 0,
        "wr_auth": 0,
        "rd_auth": 0,
        "vloc": 1,
        "vlen": 0
      },
      "p_sccd_md": {
        "read_perm": {
        "security_level": 0,
        "security_mode": 0
        },
        "write_perm": {
        "security_level": 0,
        "security_mode": 0
        },
        "ignore": 0,
        "wr_auth": 0,
        "rd_auth": 0,
        "vloc": 0,
        "vlen": 0
      }
    },
    "p_attr_char_value": {
      "p_uuid": {
        "uuid": "2a5b",
        "type": 1
      },
      "p_attr_md": {
        "read_perm": {
        "security_level": 0,
        "security_mode": 0
        },
        "write_perm": {
        "security_level": 0,
        "security_mode": 0
        },
        "ignore": 0,
        "wr_auth": 0,
        "rd_auth": 0,
        "vloc": 1,
        "vlen": 1
      },
      "init_len": 1,
      "init_offs": 0,
      "max_len": 20,
      "p_value": "00"
    },
    "value_handle": "2d5f",
    "user_desc_handle": "2567",
    "cccd_handle": "ac66",
    "sccd_handle": "3559"
    }
]
```


Figure 10: argXtract output.

```
"SD_BLE_GATTS_CHARACTERISTIC_ADD": [
  {
    "Solved": true,
    "Values": {
      "r2": 537034148,
      "readperm": 240,
      "writePerm": 240,
      "type": 1,
      "uuid": 10843
    }
  }
]
```


Figure 11: FirmXRay output.

C COMPARISON: ARGXTRACT, FIRMXRAY

We executed `argXtract` and `FirmXRay` against a random subset of 302 binaries from the `FirmXRay` dataset, focusing on the `sd_ble_gatts_service_add` and `sd_ble_gatts_characteristic_add` supervisor calls, as those are commonly available in both tools.

`argXtract` returned non-empty outputs for 161 binaries (1.5hr execution time). As described in Appendix B, we perform stringent validity checks on characteristic structures that are output by `argXtract`- particularly in terms of correct permission values. These permissions are checked depending on the characteristic properties. `argXtract` produced erroneous outputs for 16 binaries, of which 14 were found to be different versions of the same binary.

`FirmXRay` returned 282 non-empty outputs. However, a significant number (154) were found to contain invalid values for permissions. Despite this, we cannot immediately take the output to be incorrect because, as we have mentioned previously, this depends on the characteristic properties. Unfortunately, we are unable to perform the same type of validation as we do for `argXtract` (by examining the property set), as the characteristic property set is not available within `FirmXRay`'s output. That is, the characteristic properties for `sd_ble_gatts_characteristic_add` are obtained by parsing a structure pointed to by register `r1`, which also includes the CCCD permissions, while the read and write permissions for the characteristic *value* are obtained by parsing a structure pointed to by register `r2`. The complete structure is available within `argXtract`'s output, while only the fields `r2`, `readperm`, `writePerm`, `type` and `uuid` are present within the output obtained by `FirmXRay`.

To illustrate this issue, we provide sample output structures (for a single characteristic, due to space considerations) obtained by `argXtract` and `FirmXRay` for (the same characteristic within) the same input binary file in Figures 10 and 11, respectively. The values output by `FirmXRay` for `readperm` and `writePerm` are invalid in this example. However, the characteristic properties (within the `char_props` construct in `argXtract`'s output, as depicted in Figure 10) indicate that the characteristic only has the `notify` property, not `read` or `write`. This means the invalid values can be disregarded. However, information about characteristic properties is absent from `FirmXRay`'s output, making it infeasible to filter out invalid values. For this reason, we are unable to perform a meaningful comparison of the results of the two tools.