Polypyus – The Firmware Historian

Jan Friebertshäuser, Florian Kosterhon, Jiska Classen, Matthias Hollick
Secure Mobile Networking Lab, TU Darmstadt
{jfriebertshaeuser,flokosterhon,jclassen,mhollick}@seemoo.de

***Abstract*—Embedded systems, IoT devices, and systems on a chip such as wireless network cards often run raw firmware binaries. Raw binaries miss metadata such as the target architecture and an entry point. Thus, their analysis is challenging. Nonetheless, chip firmware analysis is vital to the security of modern devices. We find that state-of-the-art disassemblers fail to identify function starts and signatures in raw binaries. In our case, these issues originate from the dense, variable-length ARM *Thumb2* instruction set. Binary differs such as *BinDiff* and *Diaphora* perform poor on raw ARM binaries, since they depend on correctly identified functions. Moreover, binary patchers like *NexMon* require function signatures to pass arguments.**

As a solution for fast diffing and function identification, we design and implement *Polypyus*. This firmware historian learns from binaries with known functions, generalizes this knowledge, and applies it to raw binaries. *Polypyus* is independent from architecture and disassembler. However, the results can be imported as disassembler entry points, thereby improving function identification and follow-up results by other binary differs. Additionally, we partially reconstruct function signatures and custom types from *Eclipse* PDOM files. Each *Eclipse* project contains a PDOM file, which caches selected project information for compiler optimization. We showcase the capabilities of *Polypyus* on a set of 20 firmware binaries.

I. Introduction

Security of modern devices does not only depend on the operating system but also the hardware and firmware they run on. Systems on a chip run their own embedded Real-Time Operating System (RTOS) that is an additional target for attackers. Due to resource constraints, both the chip and the RTOS miss modern security mechanisms, and, thus, become an entry point into the operating system [2], [3], [5], [24], [29], [30], [34]. The firmware running on those chips is difficult to extract. Once extracted, analysis of raw binaries comes with many challenges. Even though most parts of raw binaries are not obfuscated to preserve performance, they often miss strings and other helpful indicators for reverse-engineering. Moreover, they lack memory region and function start annotations, in contrast to Executable and Linking Format (ELF) or Portable Executable (PE) binaries. Note that even on ELF and PE binaries, modern recursive disassemblers fail to identify $25\,\%$ of function starts on average if they are stripped [32, p. 9].

Especially within wireless baseband research, it is typical to encounter raw firmware binaries that have similarities, as they originate from the same family of wireless chips. In this paper's case study, we use a set of Bluetooth binaries that cover a history of approximately one decade of firmware development. For a subset of these, partially leaked or manually reverse-engineered symbols are available. However, to enable binary patching, security analysis, and further experimentation with Bluetooth and Wi-Fi firmware on popular devices, these symbols need to be ported to all firmware versions [30], [35].

Binary diffing compares binaries without source code. Currently, the state-of-the-art tools for this are *BinDiff* [42] and *Diaphora* [28]. They identify similar functions in a binary even if names are stripped. To this end, they can compare disassembler and decompiler output and consider call graph statistics. While *BinDiff* and *Diaphora* work well on most binaries, this is not the case for raw firmware. Both depend on functions being identified correctly. However, disassemblers like *IDA Pro* and *Ghidra*, which support these binary diffing tools, fail at function identification on the Advanced RISC Machine (ARM) *Thumb2* instruction set. Yet, fixing disassembler issues with the ARM *Thumb2* instruction set is rather complex and comes with different properties per compiler setting [27]. Surprisingly, due to avoiding false positives, the most recent version of *IDA Pro* fails to identify significantly more functions in firmware binaries than older versions.

Binaries within a firmware family are very similar. They are typically compiled for the same architecture—*ARM7*, *Cortex M3* and *M4* in our case—as well as with the same compiler options. Thus, matching functions look almost the same in binary format, except from relative branches and memory references. *Polypyus* takes advantage of these binary similarities to identify matches within seconds that state-of-the-art binary diffing tools cannot find. Our contributions are as follows:

- A benchmark of recent ARM *Thumb2* disassemblers and decompilers, including how their results impede *BinDiff* and *Diaphora* performance.
- Design and implementation of *Polypyus*, a binary-only firmware historian that works disassembler-independent.
- Evaluation of *Polypyus* on a decade of *Broadcom* and *Cypress* Bluetooth firmware history.
- Reconstruction of function signatures and types from *Eclipse* Persistent Document Object Model (PDOM) files for various *Cypress* wireless chips.

The *Polypyus* binary differ is available on *GitHub* [37]. While the evaluation in this paper focuses on Bluetooth firmware, we also successfully used the binary differ on *Broadcom* Wi-Fi firmware. Since it does not require any disassembler, it is not limited to ARM. However, its parameters might need to be adjusted for optimal results.

This paper is structured as follows. Sec. II evaluates the

Workshop on Binary Analysis Research (BAR) 2021
21 February 2021
ISBN 1-891562-69-X
https://dx.doi.org/10.14722/bar.2021.23xxx
www.ndss-symposium.org

function identification performance of various disassemblers and shows how failures lead to low-quality diffing results. Thus, we propose the *Polypyus* binary-only differ and describe its implementation in Sec. III. We evaluate binary-only diffing on 20 different Bluetooth firmware versions in Sec. IV. Moreover, we reconstruct function signatures and type information from 19 different PDOMs of *Cypress* Bluetooth, Wi-Fi and Zigbee firmware in Sec. V. We discuss related work in Sec. VI and conclude our findings in Sec. VII.

II. Function Start Identification Issues

This section shows that various disassemblers make severe mistakes on raw ARM *Thumb2* binaries and how these lead to subsequent errors within binary differs. Sec. II-A explains why already identifying instructions is a complex problem for disassemblers. This is confirmed by the benchmark results of *IDA Pro*, *Ghidra*, *Binary Ninja*, and *radare2* in Sec. II-B [26], [31], [41], [33], and Sec. II-C cherry picks common disassembler mistakes. Sec. II-D discusses the impact of these mistakes on binary differs. Our results agree with another recent study that compares disassemblers on stripped PE and ELF files [32].

A. Disassembling Instructions

Most functions start with pushing registers or loading a global variable. Thus, a disassembler can linearly inspect a binary, search for `push` instructions, and end functions on `pop` instructions. However, a function might have various branches that end at different `pop` instructions. Moreover, there are exceptions like branching to the Link Register (LR) or contents of a function pointer table. Thus, instead of aggressively scanning for `push` instructions, some disassemblers implement a recursive approach meaning that they only consider an initial function and follow its subsequent calls.

The underlying problem of identifying functions is the high instruction density. Considering 2 B random input and then running this with the *Capstone* disassembler [7], configured for ARM *Thumb2* little endian, random input is a valid 2 B instruction with 89 % probability or the start of a valid 4 B instruction with 9 % probability. Any mistake in identifying an instruction's start, which could be at a 2 B or a 4 B offset, leads to misinterpreted instructions.

B. Disassembler Benchmark

In practice, we observed many disassembler mistakes during manual firmware analysis. We design and run a disassembler benchmark with the most popular tools to quantify this observation in Sec. II-B1. The disassembler performance highly depends on their setup, thus, we also explain the initial configuration in detail. The results in Sec. II-B2 show significant differences in disassemblers.

```
0x000: 00 04 20 00        dcd bootcheck
0x004: bd 03 00 00        dcd __reset+1
0x008: 6d 01 00 00        dcd __tx_NMIHandler+1
0x00c: a1 01 00 00        dcd HardFaultInt+1
 ...
                      __tx_NMIHandler
0x16c: 00 bf               nop
0x16e: 00 bf               nop
0x170: 22 e0               b   WDogInt
```

Listing 1: *CYW20735* ROM starting with interrupt vectors.

1) Benchmark Considerations: A benchmark requires verifiable results—yet, known symbols for raw, stripped firmware binaries are rare. Hence, we only benchmark one specific example throughout this paper. However, we observed similar behavior on disassemblers within ARM-based Wi-Fi and fitness tracker firmware.

Partial symbols for the benchmarked Bluetooth firmware leaked as part of *WICED Studio* `patch.elf` files [13]. They include function names, function starts, and global variable names for the *BCM20703*, *CYW20719*, *CYW20735*, and *CYW20819* chips listed in Tab. I. Note that these symbols lack function ends. The start of a new function is not necessarily the end of the previous function, since compilers put variables after the end of functions. Such variables do not always belong to that specific function but sometimes are part of a library. Thus, we can only benchmark function starts.

It is required to know the ROM and RAM region of a Bluetooth firmware to dump it. Within the Bluetooth firmware the ROM starts at `0x0` and the RAM at `0x200000`.[1] ROM can be read and executed, while RAM can also be written. There is no execution prevention. The ROM of each firmware starts as shown in Listing 1. The `reset` vector is always located at offset `0x4`. It starts the firmware by initializing hardware components and booting the underlying *ThreadX* operating system. Even though this knowledge about a firmware binary has to be reverse-engineered in the first place, it is substantial for disassembler performance. Thus, we provide the disassemblers with the ROM and RAM position as well as the `reset` vector, if possible.

In the following, we use the *CYW20735* firmware for a benchmark. It has 10 791 functions. 10 584 of these functions are located in ROM. However, 207 functions are located in in RAM, instrumented by the so-called *Patchram* [30]. The ROM is changed with up to 256 breakpoints that point to RAM. Patched functions in RAM might be different from the leaked symbols, which can lead to a few mistakes. Thus, we ignore functions located in the RAM regions for our statistics.

IDA Pro: *IDA Pro 6.8* linearly sweeps over the whole firmware to identify instructions when possible and additionally approaches recursive descent. By default, *IDA Pro 7.x* only disassembles functions if a proper starting point is provided and continues with recursive descent.

For comparability, we disable the automatic analysis in *IDA Pro 6.8*. Then, for all *IDA* versions, we perform the following setup. We select ARM little endian as architecture and select *ARMv7-M* in the options. After binary import with default settings, we create ROM and RAM program sections and mark the first address in ROM as *Thumb* via `Alt+g`. For comparability, we mark the `reset` vector as offset and create a new function at its location as first step of the analysis. While *IDA Pro 7.x* starts analysis once provided with the `reset` vector, we manually re-enable automatic analysis after this step in *IDA Pro 6.8*.

IDA Pro 7.x can be forced to perform a linear analysis to create instructions where possible similar to the *IDA Pro 6.8* default mode of operation. The area for this analysis can be

[1] Note that on firmware prior to 2012, which is based on the *ARM7TDMI-S* core, the RAM starts at `0x80000`.

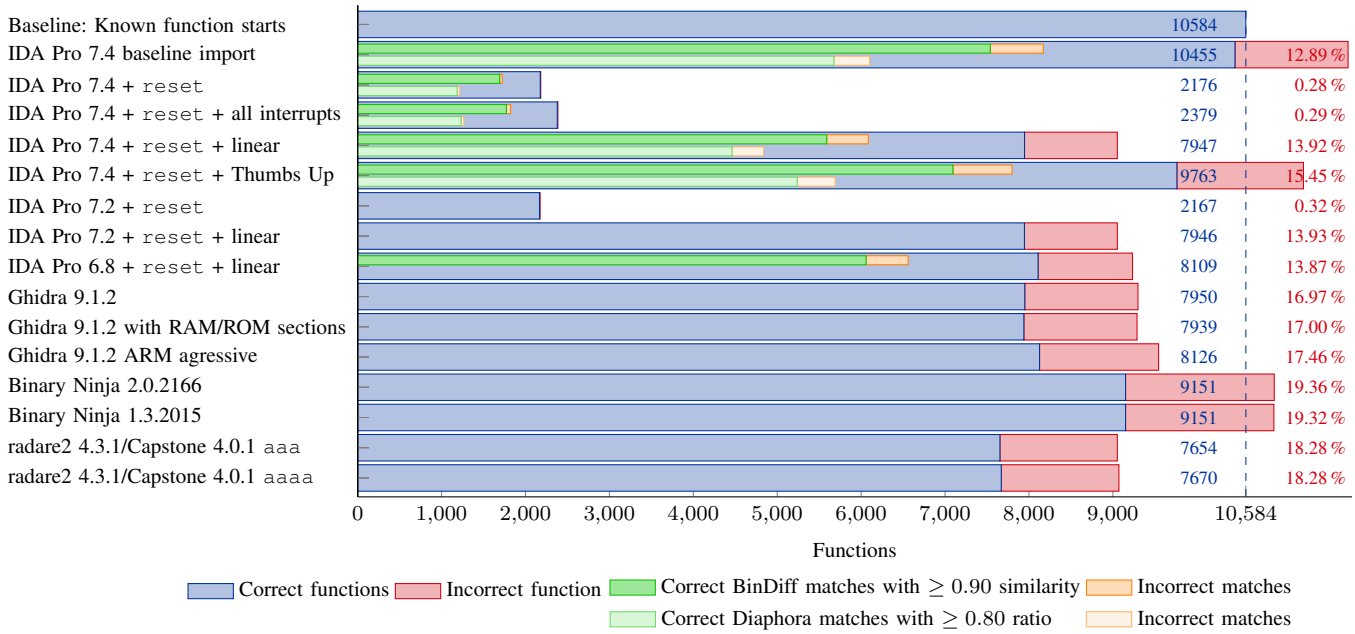

Fig. 1: Disassembler performance for function start identification on the *CYW20735* firmware.

defined with the following command, which we use within the ROM section:

```
idc.plan_and_wait(start, end)
```

Another tool to recognize function starts in ARM binaries is the *Thumbs Up* plugin for *IDA Pro* [27]. Initially, it performs a linear analysis on ROM sections utilizing the same command as above, and then makes heuristics on the already identified functions to apply them to the remaining ROM.

Ghidra: We import the binary as ARM *v7* little endian, with default compiler and analysis options. Then, we run the same analysis again with the experimental aggressive instruction finder. For comparison, we also create a fresh import and initially run the aggressive instruction finder, which leads to the same result as running these steps separately.

Binary Ninja: We set analysis and triage to `full`. Moreover, we configure the base address and the entry point. *Binary Ninja* automatically detects the correct architecture options.

radare2: For *radare2* we use the *Capstone* disassembler by setting the architecture to `arm` and 16 bit length. First, we run the basic auto analysis with `aaa`, and then we run the extended analysis using `aaaa`.

2) Benchmark Results: Fig. 1 shows an overview of the disassembler performance. *Binary Ninja* identified the most functions, while the recursive approach in *IDA Pro 7.x* has the lowest false positive rate.

IDA Pro: On *IDA Pro 7.x*, just starting at the `reset` vector is insufficient to find subsequent functions via branches. This only slightly increases when considering the full interrupt vector table. Too many function calls in the *CYW20735* firmware are implemented as offsets in function tables and other constructions the disassemblers fail to identify. Yet, this approach resulted in only 7 false positives.

The forced linear sweep identifies 7947 functions but also recognizes many false positives. Surprisingly, the automatic linear sweep without any options in *IDA Pro 6.8* provides better results with 8109 functions.

For the disassembler quality, there is almost no difference between *IDA Pro 7.2* and *IDA Pro 7.4*. In contrast, the *Thumbs Up* [27] scripts significantly improve the *IDA Pro 7.4* disassembler but do not solve all issues.

The subsequent analysis based on correctly provided functions still leads to false positives in *IDA Pro 7.4*. This even leads to some functions that cannot be marked as such as they are already located within other functions. Note that automatic analysis was still enabled during function import.

Ghidra: *Ghidra 9.1.2* identifies 7950 correct functions. The experimental aggressive instruction finder identifies 176 additional correct functions. At the same time, it also identifies 70 false positives.

Adding memory segments and marking the ROM as non-writable also does not significantly change the results, even though it improves readability of the decompiler output. In our setup, the number of false positives is identical, while the correct functions decrease by 11 when adding sections. This is the opposite of the expected impact.

Binary Ninja: Both *Binary Ninja* versions perform very similar. With 9151 correctly identified functions, they find the most true positives. In the newer version, the correct functions stay the same but the false positives increase by 4, which might just be some non-determinism within the disassembler.

radare2: *radare2*'s `aaa` analysis identifies 7654 correct functions. The *radare2* extended analysis adds 16 correct functions but 3 false positives. Overall, running the extended analysis is not much improvement but takes a lot of time. Despite taking very long, *radare2* identifies the fewest functions.

C. Common Disassembler Mistakes

Many disassembler mistakes are easy to spot as a human, yet hard to solve programmatically. In the following, we provide an intuition on these mistakes. Even though it is not the purpose of this paper to improve the disassembler per se, *Polypyus* is able to provide the disassembler with some hints.

a) Invalid Function Length: A very common mistake is ending functions too early, e.g., on the first return instruction when there are multiple function returns. Also the opposite is possible, e.g., *IDA Pro 7.2* considers the `reset` vector function on the *BCM4335C0* to be 14 736 B, which is a major part of this firmware. Also *Ghidra 9.1.2* has issues with function lengths, e.g., it often identifies a function start with a `push` instruction correctly but ends before a return instruction.

b) Correct Instructions but no Function: Even when the *IDA Pro* disassembler does not make any mistakes with the instruction offsets, it often does not recognize function starts within the instructions. This happens if there are no references to the function [27], however, firmware often has indirect function offset tables, meaning that no correctly identified reference does not necessarily indicate dead code.

c) Data Marked as Code: The symbol import of all correct function starts still produces false positives in *IDA Pro*. Often, data is misinterpreted as code. Even the reset vector table is interpreted as function at offset `0x0`.

D. Impact on Binary Diffing

Disassembler issues lead to subsequent binary diffing problems. Sophisticated statistics based on call graphs fail if a few functions within such a graph are missing. Moreover, binary diffing is also applied to false positives. Yet, *BinDiff* and *Diaphora* both require function starts, and thus, some false positives do less harm than not identifying most functions. Our observations on binary diffing performance agree with a previous comparison [40], however, that test case was a more narrow patch analysis and used a symbolized binary.

BinDiff: We show the impact of function recognition on binary diffing by running *BinDiff 6* on *IDA Pro 7.4* with the *CYW20735* and *CYW20819* firmware for the following settings: *(1)* all function starts but no names imported from the known symbols, *(2)* `reset` vector with forced linear analysis in the ROM section, and *(3)* recursive only function detection via `reset` vector. *BinDiff* provides a similarity score and we consider all results with a score ≥ 0.90. Functions below this threshold might still be correct, however, they contain also many false positives. In practice, such false positives impede further reverse-engineering.

Setting *(1)* returns 7539 correct matches, setting *(2)* decreases to 5590 matches, and *(3)* further decreases to 1692 matches. In contrast, setting *(1)* has a false positive ratio of 0.084, setting *(2)* is almost the same with a ratio of 0.089, but setting *(3)* lowers this to 0.018. These numbers show that *BinDiff* results depend on the function starts identified by the disassembler. However, false positives within function start identification significantly increase false positives.

Moreover, we run *BinDiff 4.2* on *IDA Pro 6.8*. As *IDA Pro 6.8* sightly outperforms *IDA Pro 7.x* during database initialization of both firmware versions, *CYW20735* and *CYW20819*, this effect duplicates within the *BinDiff 4.2* results, and identifies 6059 functions correctly in a setup comparable to *(2)*, at a similar false positive ratio of 0.083 Thus, surprisingly, the outdated setup works better.

Thumbs Up looks very promising when considering disassembler results. However, the false positives it identifies impede binary diffing as well.

Diaphora: We take the same statistics as with *BinDiff*. We run *Diaphora* including slow heuristics for all settings and consider the best matches as well as all partial matches with a ratio ≥ 0.80. In contrast to *BinDiff*, *Diaphora* considers decompiler output. As a first baseline, we run *Diaphora* on the database with `reset` and linear sweep. Without decompiler and slow heuristics, it finds 3320 matches. The decompiler increases this to 4409 matches without slow heuristics and 4838 matches including slow heuristics.

Using these thresholds, *Diaphora* provides a similar false positive rate as *BinDiff* but with fewer matches. However, as it is an open-source project, it would be possible to add more heuristics in the future.

radiff2: When running *radiff2* on the binaries, we encounter two issues. First, raw binaries are not supported [39]. Then, after adding all analysis options to the `-G` parameter, *radiff2* only identifies 42 matches within a few seconds and prints *"Exceeded anal threshold while diffing fcn.cafe and fcn.babe"* multiple times.

Ghidra Versioning: The *Ghidra* versioning works best if provided with correct function signatures. It sometimes identifies similar functions correctly without them but performs significantly worse than *BinDiff*, even on non-raw binaries. Moreover, enabling all analysis options typically leads to crashes, preventing us from making a fair comparison.

III. Raw Binary Diffing Approach

As shown in the previous section, faults within the disassembler lead to subsequent issues with binary diffing. Yet, firmware compiled with similar options is often byte-identical except from relative references. Based on this observation, we build the *Python*-based tool *Polypyus*, following the Greek historian Polybius. *Polypyus* learns byte similarities from a firmware history and applies these to new firmware binaries.

A. Polypyus Workflow and Tool Integration

The history can be collected from firmware with reverse-engineered or leaked symbols. *Polypyus* supports the `.symdefs` format generated by common ARM compilers and a proprietary `patch.elf` format provided by *WICED Studio*. Moreover, it can read and write function definitions as `.csv`.

Function import and export scripts allow integration into existing disassemblers, however, *Polypyus* does not depend on them. It runs standalone. When importing *Polypyus* results into a projects, this does not only add matched functions but indirectly improves disassembler quality, because the disassembler is provided with correct function starts and disassembles subsequent functions. After importing *Polypyus* matches to the disassembler, it is still possible to run binary diffing tools. They find more matches than *Polypyus* and profits from improved assembly quality in general, as previously shown in Fig. 1.

```
                mm_freeACLBuffer
0x17b4c: 00 28              cmp  r0, #0
0x17b4e: 02 d0              beq  locret_17b56
0x17b50: 08 38              subs r0, #8
0x17b52: f2 f7 43 b8        b.w  dynamic_memory_Release
                    locret_17b56
0x17b56: 70 47              bx   lr
```

Listing 2: `mm_freeACLBuffer` within *CYW20735*.

```
                mm_freeACLBuffer
0x0d0dc: 00 28              cmp  r0, #0
0x0d0de: 02 d0              beq  locret_d0e6
0x0d0e0: 08 38              subs r0, #8
0x0d0e2: f4 f7 f7 bf        b.w  dynamic_memory_Release
                    locret_d0e6
0x0d0e6: 70 47              bx   lr
```

Listing 3: `mm_freeACLBuffer` within *CYW20819*.

B. Binary Matchers

Listing 2 and 3 show the same function within the memory manager for two different firmware versions. Except from a relative branch instruction they are byte-identical. *Polypyus* combines these functions as follows:

```
mm_freeACLBuffer: 00 28 02 d0 08 38 ** f7 ** ** 70 47
```

We call this representation a matcher. The more input files provided to *Polypyus*, the more matchers are created.

During the learning phase, *Polypyus* applies fuzzyness to equally long functions with the same name. Bytes that differ, as in branch instructions, are masked. Even non-fuzzy functions can be byte-identical, such as mathematical operations.

Creating a new matcher requires a function's length. If the length is not determined by the input format, it is required to determine the function length. Functions do not necessarily end where the next function starts, as the compiler tends to put variables at the end of functions, which are not always part of the same function. Thus, we recommend using an external disassembler and importing the length determined by the disassembler into *Polypyus*, instead of simply assuming function starts being equal to function ends.

A matcher with a high fuzziness comes at the risk of false positive matches. The most common example for this is a function that only contains a 2 B or 4 B branch instruction to another function. This would create the following matcher:

```
jump_function: ** ** ** **
```

However, there can be more complex and still very fuzzy matchers. For example, the function `bb_setBdAddress`, which sets the Bluetooth device MAC address, reads identical with symbols, as shown in Listing 4. Due to the relative addresses, it generalizes to the following matcher:

```
bb_setBdAddress: 10 b5 ** f0 ** ** ** 49 ** 48 ff f7 b2 ff
    ** 49 ** 48 ff f7 e9 ff bd e8 10 40 ** ** ** **
```

As the branch to `bb_computeAC` happens to be at the same offset within the baseband library, it still creates four matching bytes despite being a location-dependent operation.

```
bb_setBdAddress
    push  {r4, lr}
    bl    bcs_pmuWaitForBtClock
    ldr   r1, =bb_localAccessCode
    ldr   r0, =rm_deviceBDAddr
    bl    bb_computeAC
    ldr   r1, =bb_localAccessCode
    ldr   r0, =rm_deviceBDAddr
    bl    bb_progMasterPiconetInfo
    pop.w {r4, lr}
    b.w   bcs_pmuReleaseBtClock
```

Listing 4: `bb_setBdAddress` on *CYW20735* & *CYW20819*.

C. Minimum Length and Cost Function

Polypyus prevents false positives by considering a minimum function length and a fuzziness cost function that is relative to the function length. While the parameters presented in this paper are tuned to work well with the Bluetooth firmware set, binaries of different architectures might benefit from adapting these parameters.

Minimum Length: A minimum function length prevents short fuzzy matchers causing false positives. In the Bluetooth firmware set, thresholds between 10 B and 24 B are reasonable. For all functions in our symbolicated firmware subset, we consider a prefix of length k. Then, we search for other functions with the same prefix, as depicted in Fig. 2.

Each function contains at least one 2 B instruction. Thus, at $k = 2$, each function at least matches itself, meaning that the amount of total matches is the same as the total number of functions. With increasing k, the total matches slightly decrease, as some functions are smaller than the prefix length and, thus, are not considered as equal. To determine a threshold for the minimum length, we consider the mean and the maximum matches depending on k. The mean represents the average number of functions per prefix, and the maximum the highest number of functions with the same prefix.

The numbers in Fig. 2 indicate that there exist a few short functions that can be found several times in all firmware versions. Hence, *Polypyus* favors longer functions in case there are multiple matching matchers. This solves the issue caused by one matcher being the prefix of another matcher.

Furthermore, we want to limit the problem that a small function matches all function prologues of unknown functions. The maximum number of functions per prefix drops significantly before the prefix length of $k = 10$ and again at $k = 24$, after which it stays almost constant. This drop shows that additional bytes in the prefix help to distinguish

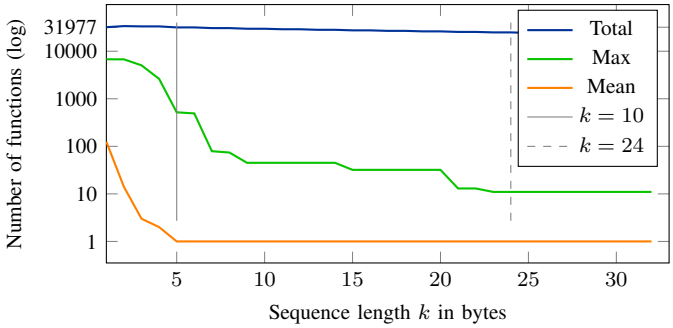

Fig. 2: Relation between the prefix length to the number of functions in the symbol data with the same prefix.

between functions. Based on this observation, the *Polypyus* default configuration for the minimum function length is 24 B.

Cost Function: Without a cost function, *Polypyus* would simply create fully fuzzy matchers for each byte length and ignore the bytecode. Intuitively, a cost-based approach restricts the creation of matchers. More fuzziness means higher costs. Dense clusters of fuzziness are more expensive than sparsely distributed fuzzy bytes. A function's allowance is based on its length and configurable thresholds. We define the following requirements for a matcher's fuzziness cost function:

(1) Fuzziness should not be allowed in short functions.
(2) The maximal fuzziness should only be allowed if it is sparsely distributed through the function.
(3) The cost attributed to a long succession of fuzziness should be higher than the sum of multiple shorter succession of the same total length.
(4) The cost function should be adjusted to the high ARM *Thumb2* code density.

Polypyus applies the cost of a matcher as follows:

$$\text{matcher-cost}(m) = \sum_{(k,d)\in F_m} \text{sequence-cost}(k,d) \tag{A}$$

$$\text{matcher-cost}(m) \leq (|m| - \mu) \cdot \phi \tag{B}$$

$$\text{sequence-cost}(k,d) = \frac{k}{p(\lceil k/2 \rceil)} \cdot (1 + \text{proximity-penalty}(d)) \tag{C}$$

$$\text{proximity-penalty}(d) = \begin{cases} 0 & , d = 0 \\ 0.9^d & , \text{ otherwise} \end{cases} \tag{D}$$

where m is the matcher, μ is the minimal function size defined above, d and ϕ are the minimal and the maximal relative distance to the next fuzzy byte in the matcher, and $p(k)$ is the likelihood of a uniformly sampled half-word sequence to be code without assumptions on the first half-word in ARM *Thumb2*. Fig. 3 presents the costs of individual fuzzy sequences for combinations of the byte distance d and the sequence length k.

We validate weather the cost function satisfies the requirements *(1)–(4)*.

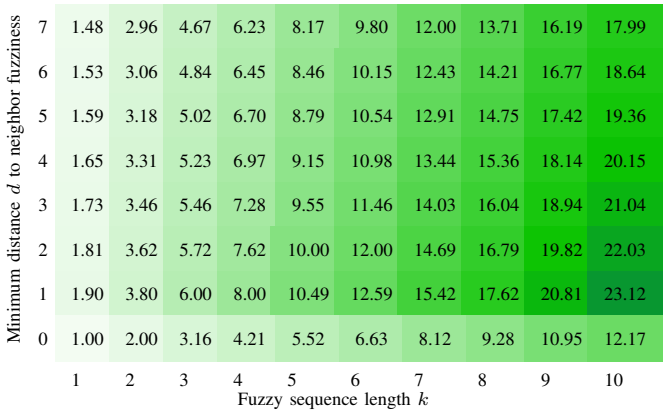

Fig. 3: The cost of a fuzzy-marked sequence of bytes based on the sequence length and its minimal distance to another fuzzy-marked sequence in the same function.

(1) Disallow Short Functions: The cost function does not allow fuzziness in short functions. According to Equation A, a fuzzy sequence of length k always costs at least k. Equation B limits the allowed maximal matcher cost to a term that is negative for functions shorter than the minimal function length. Matchers that are smaller than the minimal function length plus $1/\phi$ involve no costs. Then, costs grow linearly with a gradient of ϕ, which disallows short functions. Considering a fixed matcher size, the equation underrates the maximal fuzziness to be smaller than ϕ by a relative factor that is proportional to the function size, and descends with a growing matcher size.

(2) Require Sparse Fuzziness: Equation D defines a penalty factor that exponentially falls with an increasing minimal distance to the next fuzzy byte sequence and converges to zero. Meanwhile, the term $\frac{k}{p(\lceil k/2 \rceil)}$ in Equation C exponentially increases with the sequence length k. This way, short clusters of fuzzy bytes and long successions of fuzzy bytes are costly.

(3) Penalty for Long Successions: The function $p(k)$ that represents the probability of a byte to be valid code can be approximated by 0.977^k (see Sec. II-A). We ensure that the penalty for clustering decreases slower than the cost for long fuzzy sequences rises. If this is the case and if the third requirement given above is not violated by any of the initial values, then no combination of the byte distance d or sequence length k can violate the requirement. Therefore, the proximity penalty is an exponentially decreasing function with 0.9^k that falls slower than $\frac{k}{0.977^k}$ grows, with respect to the approximation error. As shown in Fig. 3, the initial values do not violate this requirement.

(4) ARM Thumb2 Code Density: The cost of a fuzzy sequence is proportional to the likelihood that it will not match just code. This is a reasonable metric for our scenario.

The Equations A–D describe the requirements to create a common function class in *Polypyus*. If a function class is not meeting the requirements, we test, with a greedy algorithm, whether subsets of the functions meet them, in which case these subsets form a common function class. All functions that are not part of common function class form standalone, non-fuzzy matchers to find exact matches of these functions.

D. Function Prologue Matchers

Function-based matchers substitute the need for binary diffing with other tools. However, as described in the typical workflow in Sec. III-A, *Polypyus* is meant to support existing disassemblers and state-of-the-art binary diffing tools. Thus, also identifying functions without assigning their correct name helps in the overall process.

In addition to the normal matchers, *Polypyus* has the option to also learn function prologues. This feature must be explicitly enabled. Within the Bluetooth firmware history, many functions start with common instructions, such as pushing and initializing the same registers. *Polypyus* learns the most common prologues from the history and applies these to the target binary to create unnamed functions. The default setting in the user interface is 8 B prologue length, and *Polypyus* only considers frequent prologues.

TABLE I: Bluetooth firmware history for disassembler and *Polypyus* benchmarks.

Chip	Device	Build Date	Symbols	Known Functions	ROM Size	ARM Core	Full Matches	+ Starts
BCM2046A2	iMac Late 2009	2007	—	—	0x31c00	ARM7TDMI-S (?)	0	10
BCM2070B0	MacBook 2011, Thinkpad T420	Jul 9 2008	—	—	0x57800	ARM7TDMI-S (?)	0	9
BCM20702A1	Asus USB Dongle	Feb (?) 2010	—	—	0x5fc00	ARM7TDMI-S	0	33
BCM4335C0	Google Nexus 5	Dec 11 2012	—	—	0x8f000	Cortex M3	266	2392
BCM4345B0	iPhone 6	Jul 15 2013	—	—	0xb3000	Cortex M3	371	2989
BCM20703A1	MacBook Pro early 2015	Dec 23 2013	—	—	0xc7000	Cortex M3	708	4350
BCM43430A1	Raspberry Pi 3/Zero W	Jun 2 2014	—	—	0x8f400	Cortex M3	209	1859
BCM4345C0	Raspberry Pi 3+/4	Aug 19 2014	—	—	0xc1c00	Cortex M3	307	2546
BCM4358A3	Samsung Galaxy S6, Nexus 6P	Oct 23 2014	—	—	0x8f000	Cortex M3	275	1978
BCM4345C1	iPhone SE	Jan 27 2015	—	—	0xb7000	Cortex M3	295	2477
BCM4364B0	MacBook/iMac 2017–2019	Aug 21 2015	—	—	0xd4000	Cortex M3	340	2804
BCM4355C0	iPhone 7	Sep 14 2015	—	—	0x90000	Cortex M3	231	1838
BCM20703A2	MacBook/iMac 2016–2017	Oct 22 2015	✓	8603	0xc7000	Cortex M3	101	1583
BCM4347B0	Samsung Galaxy S8	Jun 3 2016	—	—	0xf4800	Cortex M4	414	2720
BCM4347B1	iPhone 8/X/XR	Oct 11 2016	—	—	0xfc000	Cortex M3	695	3599
CYW20719B1	Evaluation board	Jan 17 2017	✓	15 036	0x1d2000	Cortex M4	1519	6214
CYW20735B1	Evaluation board	Jan 18 2018	✓	10 791	0x14f000	Cortex M4	2241	7129
CYW20819A1	Evaluation board	May 22 2018	✓	10 276	0xf6800	Cortex M4	1543	5772
BCM4375B1	Samsung Galaxy S10/S20	Apr 13 2018	—	Canaries	0x130000	Cortex M4 (?)	14	362
BCM4378B1	iPhone 11/SE2	Oct 25 2018	Strings	Canaries	0x132800	Cortex M4 (?)	15	380

Polypyus is provided with inputs from four symbolicated firmwares. Matching results are for function identification and starts with *Polypyus* defaults: 24 B minimum length for function identification and optional 8 B prologues for function starts.

E. Fast Matching Algorithm

When provided with the four symbolized firmware binaries and one target binary, *Polypyus* creates matchers and finds matches based on those within a few seconds on a recent consumer-grade laptop, e.g., 8.5 s to create matchers from three binaries in a history and 9.5 s to find matches in a target binary. This performance is achieved by several optimizations:

- A matcher prefix tree deduplicates the matchers.
- Match finding is a depth first search in the prefix tree.
- Bins in the tree reduce the number of visited matcher fragments.
- Search is restricted to partitions of the binary that potentially contain code.

IV. Raw Binary Diffing Evaluation

In the following, we analyze the quality of matches on a history collected from 20 chips with build dates ranging over a decade. An overview of these is shown in Tab. I.

A. Known Firmware Changes and History

Over this decade, the underlying ARM core changed. The oldest available datasheet of the *BCM20702A1* chip from 2010 states that it uses an *ARM7TDMI-S*-based microprocessor with integrated RAM and ROM [11]. The slightly newer *BCM4335C0* firmware from 2012 is already based on a *Cortex M3* [10]. In 2016, the core changed again to a *Cortex M4* [12]. Even though datasheets are not available for all chips, the *Cortex M4* introduces additional instructions, which can be determined with a disassembler. Not all *Cortex M* versions introduce new instructions, meaning that the newest chip series could also be based on a different core.

TABLE II: Firmware similarity by function names.

Firmware A	Firmware B	Unique Symbols	Common Symbols
BCM20703A2	CYW20719B1	12 161	5655
BCM20703A2	CYW20735B1	8402	5496
BCM20703A2	CYW20819A1	8215	5332
CYW20719B1	CYW20735B1	4790	10 435
CYW20719B1	CYW20819A1	6115	9515
CYW20735B1	CYW20819A1	1927	9570

In 2016, *Broadcom* sold their wireless IoT division to *Cypress* [9]. However, *Broadcom* kept customers like *Apple* and *Samsung*. Since then, firmware was developed independently.

In the *Samsung Galaxy S10* firmware, stack canaries were introduced. These change function prologues and harm performance on matchers created without stack canaries.

The *iPhone 11* firmware introduced a few debug strings with function names. These could be annotated and used to train *Polypyus*. However, some of the function names indicate that *Broadcom* did a lot of refactoring and firmware improvements on this latest firmware. Moreover, the additional debug print statements change the binary structure.

Overall, refactoring is a significant factor within the Bluetooth firmware history. As shown in Tab. II, when considering common function names, the *BCM20703A2* firmware from 2015 significantly differs from those of the *Cypress* evaluation boards. However, the evaluation board firmware from 2017 and 2018 is very similar, except from some additional libraries depending on the board version that appear as unique symbols.

Additionally, firmware versions have different feature sets. For example, the oldest evaluation board within the Bluetooth firmware set has the most functions. Moreover, chips in laptops tend to have larger ROMs than those in smartphones.

B. Polypyus Binary Diffing Results

We run *Polypyus* on the full firmware set listed in Tab. I. For the history, all firmware versions with symbols are considered. If the target is a firmware with symbols, we exclude the target itself from the history. Matches found by the function matchers configured to the 24 B default are listed separately to the matches found with 8 B prologues.

a) History Observations: First of all, the matches in Tab. I are very platform-dependent. *Polypyus* does not find any matches in the older *ARM7TDMI-S* core, as this is using a different binary format. Moreover, for the *BCM20703* firmware, which is still a *Cortex M3*, the evaluation boards that run on the *Cortex M4* do not provide high-quality input. However, the *BCM20703* symbols help identifying functions in other *Cortex M3* based firmware.

Even though we do not have datasheets for the newest *BCM4375B1* and *BCM4378B1* firmware, we assume that they are still based on a *Cortex M4* due to their instruction set. The stack canaries are compiled into most functions, and this impedes *Polypyus* performance. However, if *Polypyus* would be trained with a new classifier that specifically searches for stack canaries, these could as well have a positive impact on function start identification.

The test set of the three evaluation boards performs the best. The *CYW20735* evaluation board, which was built after the *CYW20719* but before the *CYW20819*, profits a lot from the known symbols of the other firmware.

b) Match Quality: For the firmware with symbols, we evaluate the match quality as shown in Fig. 4. The quality of the identified functions is very high. The prologues have more false positives, which can be improved by increasing the prologue size—however, this would also reduce the total number of prologue matches.

Approximately ⅓ of the incorrect functions in Fig. 4 is actually correct when we analyze these by hand. The symbol names do not match because these functions were obviously renamed. Renaming includes obvious typos, such as `md4` to `md5`, or consistency of names within libraries. However, we did not remove them from the experiments, as we do not know this for every function.

The false positives that remain can be further eliminated by increasing the minimum function length to a value like 30 B. However, we found that also matches at a minimum function length of 8 B are still more accurate than *BinDiff* on a flawed *IDA Pro* database. In the *CYW20735* firmware, the default of 24 B identifies 2184 correct functions at a false positive rate of 2.88 %. A minimum function length of 30 B reduces this to 1804 functions at a false positive rate of 2.66 %. However, a 8 B minimum function length increases the correct functions to 3521 with a false positive rate only of 3.95 %. This shows how effective the *Polypyus* cost function is.

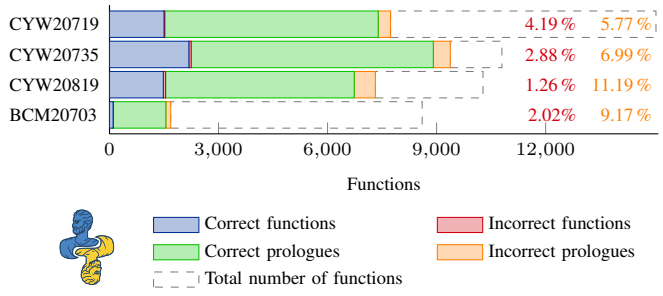

Fig. 4: *Polypyus* matches within the symbolized firmware, incorrect matches due to function name changes not removed.

C. Disassembler Improvement

Since *Polypyus* is meant to integrate into existing workflows with other disassemblers, we evaluate its impact when working with *IDA Pro 7.4*. We evaluate all firmware versions with symbols. First, we start with the `reset` vector, then we add function names and prologues identified by *Polypyus* (market with + *Polypyus*), and finally, we run a linear sweep over the ROM (marked with + *linear sweep*). For comparison, we run the same experiment without *Polypyus* and just mark the `reset` vector and run a linear sweep.

The results shown in Fig. 5 indicate that *Polypyus* indeed helps with identifying functions in all firmware versions. With one exception, it also reduces the false positive ratio. This might either be the case because *Polypyus* identified more false function prologues in the *CYW20819* firmware compared to the others or because of *IDA Pro 7.4* dynamics after the import that identify false functions, as it was already the case for the fully correct function import in Fig. 1.

Another interesting observation that can be seen in Fig. 4 is that function detection works better on the *BCM20703* firmware with a *Cortex M3* than on the remaining *Cortex M4* firmware. This might be due to the more complex instruction set or different compiler settings.

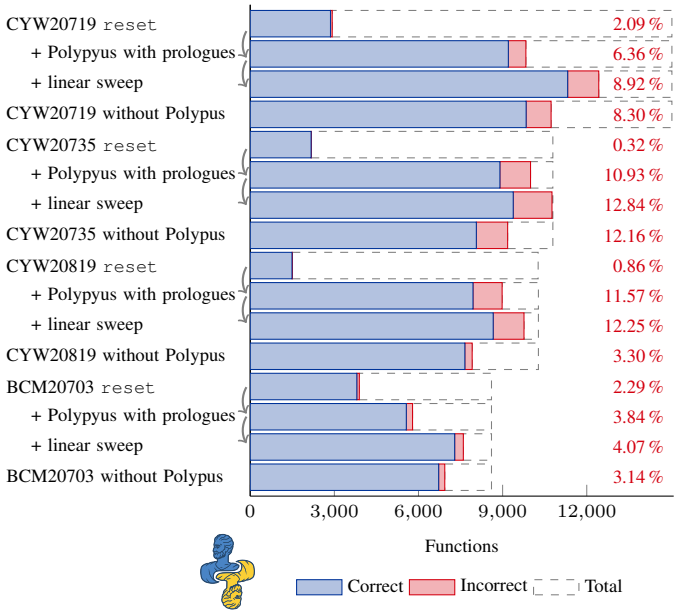

Fig. 5: Effectiveness of *Polypyus* import in *IDA Pro 7.4* including prologues, incremental steps marked with ⟶.

PDOMNode	PDOMNamedNode	PDOMBinding	PDOMCPPFunction
00 01 short factory_ID 00 07 short node_typ 00 11 ptr parent 9d 98	00 1f ptr name 49 6b	00 1f 49 7e ptr first_decl 00 00 00 00 ptr first_def 00 00 00 00 ptr first_ref 00 00 00 00 ptr local_to_file 00 00 00 00 ptr first_extref	00 00 00 02 int num_params 00 1f 49 79 ptr first_param 1f 00 1f 49 71 00 type function_type 99 3e 7a f1 int signature_hash 00 00 00 00 ptr exception_spec 00 42 short annotation 00 02 short required_arg_count 00 00 00 00 00 00 execution function_body 00 00 00 00 00 00 type declared_type

Fig. 6: PDOMCPPFunction node for btsnd_hcic_ble_remove_from_white_list C++ function definition.

V. Function Signature and Type Reconstruction

In addition to function starts, binary patching with C-based frameworks requires correct function signatures to pass arguments. *NexMon* is such a binary patcher supporting ARM [36]. Initially developed for *Broadcom* Wi-Fi chips, it has been ported to fitness trackers, vacuum cleaners, and more [8], [23]. As of now, each *NexMon* port has a `wrapper.c` file that defines selected functions required for supported patches. The wrapper is hand-written and based on manual reverse engineering. Thus, adding a new firmware version is a huge effort that we aim to automate in the following.

WICED Studio [13] contains Persistent Document Object Model (PDOM) files for Bluetooth, Wi-Fi and Zigbee chips. Not all of them are for sale and we do not have their raw firmware, however, we can still use the PDOM information to restore function signatures and custom types.

A. Eclipse and PDOM Purpose

The PDOM architecture consists of a client, an indexer and a database [15]. PDOM clients are *Eclipse* features like the C/C++ search page, searching for declarations and reference actions, content assist, and more. This information is stored in the PDOM database for fast access. In case it is not available, a slower direct access via the Document Object Model (DOM) is possible. Depending on the project size, a full indexer could index a project within seconds or hours. Thus, for large projects, a faster indexer that stores less information into the database is a common setting. All PDOMs analyzed in the following only contain partial information.

B. Utilizing PDOM in Reverse

The PDOM database contains valuable information about a project that exceeds the raw binary. Function signature and type information can be restored by using the database in the reverse direction. Since this is not the intended use case, we need to reverse-engineer the PDOM format. While it is sparsely documented with a few notes from 2006 [14], it is still maintained and open-sourced within the *Eclipse* project. Based on the *Eclipse* implementation, we create a custom PDOM database extractor that stores all information into a *SQLite* database. Even though the database extraction is slow, the *SQLite*-based import into *IDA Pro* is fast.

The database is structured by indexes. The file header contains offsets to the indexes, e.g., linkages, a file index, or an index of defective files [18]. In our case, only the linkage indexer is relevant. Each linkage is a top-level node of stored bindings. Two types of linkages differentiate between C and C++ nodes [20]. There are around 80 different bindings, but we only require a small subset to identify functions, parameters, and return types.

In the following, we analyze the function `btsnd_hcic_ble_remove_from_white_list` of the *CYW20735B1* PDOM file to provide an example for the data structure. Since it is a C++ function, it is based on the `PDOMCPPFunction` record [16]. This function is stored at offset `0xfa4b12` with its binary values shown in Fig. 6. We can follow its pointers to obtain further information by executing the PDOM library function `getRecPtr` [19]. For the function name pointer `0x1f496b`, `getRecPtr` returns the file offset `0xfa4b5a`, which contains the function name length and its name as string.

The number of parameters is directly encoded as `0x2`. Then, to retrieve the function parameter names, we follow the `first_param` pointer `0x1f4979` and apply the `PDOMCPPParameter` definition [17]. Since only the first parameter pointer is defined in a `PDOMCPPFunction`, each parameter contains a pointer to the next parameter. By following these pointers and applying names, we can extend the function signature as follows:

```
btsnd_hcic_ble_remove_from_white_list(addr_type, bda)
```

However, extracting a type requires further actions. A type can be direct, indirect, storable, or null [20]. In our case, the 6 B `function_type` definition starts with `0x1f`, which means that it is an indirect type. Indirect types are stored in an additional record, in our case stored at `0x1f4971`. The type is unpacked using the `unmarshalType` and `unmarshal` functions [22], [21]. Note that unpacking types can get rather complex. In our case, the function type is `BOOL32`. Furthermore, the function type record is directly followed by records containing information about the first and second parameter type. Applying all types results in the following final function signature:

```
BOOL32 btsnd_hcic_ble_remove_from_white_list(UINT8
    addr_type, UINT8* bda)
```

C. Function Signature Reconstruction Results

Tab. III shows that the PDOMs contained in *WICED Studio* can restore approximately 40 % of the function signatures of the previously extracted Bluetooth firmware. On the left-hand side, all PDOM databases and their project creation times are listed. For some chips, multiple projects exist with different amounts of indexed information.

If a function with the same name exists in both databases, we calculate their signature similarity based on the return value as well as parameter count, types, and names. Undefined types

TABLE III: PDOM history with function signature similarity benchmark.

Eclipse PDOM Input		CYW20735B1		CYW20719B1		CYW20706A2	
Chip	**Build Date**	**Same Names**	**Signature Similarity**	**Same Names**	**Signature Similarity**	**Same Names**	**Signature Similarity**
CYW20706A2	May 2 2016	2236	74.20 %	2377	72.44 %	1817	88.51 %
CYW20706A2	Aug 24 2016	1334	91.95 %	998	87.46 %	446	100 %
CYW20719B0	Aug 4 2016	2345	94.90 %	2595	93.94 %	1819	69.97 %
CYW20719B0	Aug 24 2016	767	95.56 %	820	95.79 %	382	70.53 %
CYW20719B0	Jun 28 2017	1138	98.42 %	863	97.58 %	410	76.55 %
CYW20719B1	Jun 7 2017	1271	97.75 %	994	100 %	431	80.35 %
CYW20721B1	May 30 2018	1040	97.94 %	788	99.19 %	235	95.03 %
CYW20729B0	Aug 23 2016	3476	81.24 %	5749	79.43 %	2603	69.49 %
CYW20735B0	May 2 2016	2255	80.31 %	2500	78.58 %	1805	68.73 %
CYW20735B0	Aug 24 2016	2132	97.34 %	2030	95.16 %	1211	76.78 %
CYW20735B1	May 30 2018	1249	100 %	991	97.35 %	417	78.66 %
CYW20739B0	May 2 2016	2294	80.21 %	2542	78.62 %	1811	68.81 %
20729B0 Zigbee	Aug 24 2016	79	93.16 %	263	90.81 %	43	93.65 %
20729B1 Zigbee	Jun 19 2017	171	93.28 %	376	92.71 %	50	94.56 %
20739B1 Zigbee	Jun 7 2017	1547	93.17 %	1007	95.33 %	439	72.75 %
43xxx Wi-Fi	5pm, Aug 31 2016	673	88.22 %	1153	88.46 %	281	89.55 %
43xxx Wi-Fi	8pm, Aug 31 2016	398	92.49 %	687	92.24 %	94	71.48 %
43xxx Wi-Fi	Jun 7 2017	1714	84.10 %	2354	87.32 %	1008	85.38 %
43012C0 Wi-Fi	Aug 9 2017	1011	96.06 %	707	93.00 %	311	91.55 %
		4594/11 810	91.07 %	6364/14 725	90.28 %	2833/8603	81.18 %
Combined Same Name/Total Functions		**38.90 %**		**43.22 %**		**32.93 %**	

Persistent Document Object Model (PDOM) files extracted from *Cypress WICED Studio 6.2* and *6.4*. Not all chips listed here are publicly available as evaluation kit, thus, some binaries are missing in the function identification analysis. Projects that were built multiple times contain partially redundant PDOM files.

are skipped in the comparison. Otherwise, each non-matching property marks a function signature as non-similar. For example, some functions exist in multiple firmware versions but got additional parameters over time and are not similar.

Even the Wi-Fi and Zigbee firmware contain useful inputs for the Bluetooth firmware because they partially share the same code base and vendor-specific library functions.

VI. Related Work

Polypyus is not the first tool leveraging binary similarity for diffing and function start recognition. Since it works disassembler-independent and finds matches in a large firmware history within seconds, it is nonetheless a useful tool for the reverse-engineering community. Moreover, the PDOM reconstruction provides closed-source function signatures.

Interestingly, even though there is plenty of related work about finding function starts [1], [4], [6], [38], a recent disassembler benchmark confirms our results that 25 % of function starts are missed [32]. *Polypyus* learns from two or more input binaries instead of huge pre-compiled known inputs, and still provides decent results on function starts and diffing.

IDA Pro implements *F.L.I.R.T.* signatures to recognize library functions [25]. However, these are limited to *IDA Pro*, optimized for libraries, and closed-source, while *Polypyus* is open-source and can be applied to arbitrary functions. In fact, many matches *Polypyus* finds are firmware-specific functions that are not part of public libraries.

VII. Conclusion

With *Polypyus*, we created a fast binary-only differ that integrates into the existing workflow with other binary analysis tools. History generation and function matching run within a few seconds, making it a suitable tool for projects with a large firmware set. Even though we did not evaluate *Polypyus* for other architectures than ARM *Thumb2*, matchers are binary-only. This means that it can be used for any architecture, including proprietary instruction sets, as long as there is some notation of functions that *Polypyus* can learn from.

The *Polypyus* function identification and diffing combined with the reconstruction of function signatures and types from *Eclipse* PDOM files will be helpful for C-based binary patching, such as *NexMon* for Bluetooth and Wi-Fi firmware.

Acknowledgment

We thank Christian Blichmann and Joxean Koret, the core developers of *BinDiff* and *Diaphora*, for their feedback. Moreover, we thank Jordan Wiens for the help with *Binary Ninja* and Eyal Itkin for the *Thumbs Up* support. We thank Ralf-Philipp Weinmann for feedback on this paper, Jakob Link for testing *Polypyus* with *Broadcom* Wi-Fi firmware, and Anna Stichling for creating the *Polypyus* logo.

This work has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

Availability

Polypyus is publicly available on *GitHub* [37]. As of now, it comes with a small firmware set of the binaries with symbols as well as the *Raspberry Pis*. Moreover, we will release the PDOM extractor as well as the extracted function signature and type databases.

REFERENCES

[1] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189. IEEE, 2017.

[2] Hugues Anguelkov. Reverse-engineering Broadcom Wireless Chipsets. https://blog.quarkslab.com/reverse-engineering-broadcom-wireless-chipsets.html, Apr 2019.

[3] Nitay Artenstein. Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets. https://blog.exodusintel.com/2017/07/26/broadpwn/, 2017.

[4] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, San Diego, CA, August 2014. USENIX Association.

[5] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, 2017.

[6] Andrew R. Bernat and Barton P. Miller. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, page 9–16, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Capstone. The Ultimate Disassembler. https://www.capstone-engine.org/, 2020.

[8] Jiska Classen and Daniel Wegemer. Fitbit Firmware Modifications. https://github.com/seemoo-lab/fitness-firmware, 2019.

[9] Cypress Semiconductor Corporation. Cypress to Acquire Broadcom's Wireless Internet of Things Business. https://www.cypress.com/news/cypress-acquire-broadcom-s-wireless-internet-things-business-0, June 2016.

[10] Cypress Semiconductor Corporation. *BCM4339: Single-Chip 5G WiFi IEEE 802.11ac MAC/Baseband/Radio with Integrated Bluetooth 4.1 and FM Receiver*, March 2017. Rev. *H.

[11] Cypress Semiconductor Corporation. *CYW20702: Single-Chip Bluetooth Transceiver and Baseband Processor*, November 2017. Rev. *M.

[12] Cypress Semiconductor Corporation. Bluetooth (BR + EDR + BLE) Connectivity Solution Families. https://www.cypress.com/products/ble-bluetooth, 2020.

[13] Cypress Semiconductor Corporation. WICED Software. https://www.cypress.com/products/wiced-software, 2020.

[14] Eclipse Foundation. PDOM & Indexing. https://wiki.eclipse.org/CDT/designs/PDOM, 2006.

[15] Eclipse Foundation. PDOM Overview. https://wiki.eclipse.org/CDT/designs/PDOM/Overview, 2008.

[16] Eclipse Foundation. PDOM CPP Function. https://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/pdom/dom/cpp/PDOMCPPFunction.java, Jan 2021.

[17] Eclipse Foundation. PDOM CPP Paramter. https://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/pdom/dom/cpp/PDOMCPPParameter.java, Jan 2021.

[18] Eclipse Foundation. PDOM Database Offset. https://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/pdom/PDOM.java, Jan 2021.

[19] Eclipse Foundation. PDOM getRecPtr. https://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/pdom/db/Chunk.java, Jan 2021.

[20] Eclipse Foundation. PDOM Linkage. https://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/pdom/dom/PDOMLinkage.java, Jan 2021.

[21] Eclipse Foundation. PDOM unmarshal. https://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/dom/parser/cpp/CPPFunction.java, Jan 2021.

[22] Eclipse Foundation. PDOM unmarshalType. https://git.eclipse.org/c/cdt/org.eclipse.cdt.git/tree/core/org.eclipse.cdt.core/parser/org/eclipse/cdt/internal/core/pdom/dom/cpp/PDOMCPPLinkage.java, Jan 2021.

[23] Dennis Giese. Not all IoT Devices are Created Equal: Reverse Engineering of Xiaomi's IoT Ecosystem. In *beVX*, 2018.

[24] Grant Hernandez and Marius Muench. Emulating Samsung's Baseband for Security Testing. *BlackHat USA 2020*, August 2020.

[25] Hex-Rays. IDA F.L.I.R.T. Technology: In-Depth. https://www.hex-rays.com/products/ida/tech/flirt/in_depth/, 2020.

[26] Hex-Rays. IDA Pro. https://www.hex-rays.com/products/ida/, 2020.

[27] Eyal Itkin. Thumbs Up: Using Machine Learning to Improve IDA's Analysis. https://research.checkpoint.com/2019/thumbs-up-using-machine-learning-to-improve-idas-analysis/, 2019.

[28] Joxean Koret. Diaphora. http://diaphora.re/, 2020.

[29] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. *The 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, July 2020.

[30] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. InternalBlue - Bluetooth Binary Patching and Experimentation Framework. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19)*, Jun 2019.

[31] National Security Agency. Ghidra. https://ghidra-sre.org/, 2020.

[32] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly but Were Afraid to Ask. In *2021 2021 IEEE Symposium on Security and Privacy (S&P)*, pages 194–212, Los Alamitos, CA, USA, May 2021. IEEE Computer Society.

[33] radare. radare2. https://rada.re/n/radare2.html, 2020.

[34] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36. USENIX Association, August 2020.

[35] Matthias Schulz. *Teaching Your Wireless Card New Tricks: Smartphone Performance and Security Enhancements Through Wi-Fi Firmware Modifications*. PhD thesis, Technische Universität, 2018.

[36] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. Nexmon: The C-based Firmware Patching Framework. https://nexmon.org, 2017.

[37] Secure Mobile Networking Lab. Polypyus on GitHub. https://github.com/seemoo-lab/polypyus, 2020.

[38] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626, Washington, D.C., August 2015. USENIX Association.

[39] Siguza. radiff2 -A doesn't work on raw binaries #13541. https://github.com/radareorg/radare2/issues/13541, 2019.

[40] Maddie Stone. What's Up with WhatsApp. https://github.com/maddiestone/ConPresentations/blob/master/Jailbreak2019.WhatsUpWithWhatsApp.pdf, 2019.

[41] Vector 35 Inc. Binary Ninja. https://binary.ninja/, 2020.

[42] zynamics. BinDiff. https://www.zynamics.com/bindiff.html, 2020.