IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 128–159. DOI:10.46586/tches.v2025.i1.128-159

Trace Copilot: Automatically Locating Cryptographic Operations in Side-Channel Traces by Firmware Binary Instrumenting

Shipei Qu[1,2], Yuxuan Wang[1,2], Jintong Yu[1,2], Chi Zhang[1,2✉], Dawu Gu[1,2✉]

[1] School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China {zcsjtu,dwgu}@sjtu.edu.cn

[2] State Key Laboratory of Cryptology, P.O. Box 5159, Beijing, 100878, China

Abstract. A common assumption in side-channel analysis is that the attacker knows the cryptographic algorithm implementation of the victim. However, many lab-setting studies implicitly extend this assumption to the knowledge of the source code, by inserting triggers to measure, locate or align the Cryptographic Operations (CO) in the trace. For real-world attacks, the source code is typically unavailable, which poses a challenge for locating the COs thus reducing the effectiveness of many methods. In contrast, obtaining the (partial) binary firmware is more prevalent in practical attacks on embedded devices. While binary code theoretically encapsulates necessary information for side-channel attacks on software-implemented cryptographic algorithms, there is no systematic study on leveraging this information to facilitate side-channel analysis. This paper introduces a novel and general framework that utilizes binary information for the automated locating of COs on side-channel traces. We first present a mechanism that maps the execution flow of binary instructions onto the corresponding side-channel trace through a tailored static binary instrumentation process, thereby transforming the challenge of locating COs into one of tracing cryptographic code execution within the binary. For the latter, we propose a method to retrieve binary instruction addresses that are equivalent to the segmenting boundaries of the COs within side-channel traces. By identifying the mapping points of these instructions on the trace, we can obtain accurate segmentation labeling for the side-channel data. Further, by employing the well-labeled side-channel segments obtained on a profiling device, we can readily identify the locations of COs within traces collected from un-controllable target devices. We evaluate our approach on various devices and cryptographic software, including a real-world secure boot program. The results demonstrate the effectiveness of our method, which can automatically locate typical COs, such as AES or ECDSA, in raw traces using only the binary firmware and a profiling device. Comparison experiments indicate that our method outperforms existing techniques in handling noisy or jittery traces and scales better to complex COs. Performance evaluation confirms that the runtime and storage overheads of the proposed approach are practical for real-world deployment.

Keywords: Side-channel analysis · Software/Hardware co-analysis · Binary instrumentation · Locating of cryptographic operations

1 Introduction

Physical side-channel attacks [Koc96] have been demonstrated to be an effective approach for breaking the confidentiality of embedded cryptographic devices. Instead of relying on weaknesses in the cryptographic algorithm itself, physical side-channel attacks exploit the

Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

dependency between the physical signals of the device (power consumption, electromagnetic radiation, etc.) and secret information in processing. For a successful attack, the evaluator must at least a) acquire one or more side-channel traces while the target device executes Cryptographic Operations (COs), and b) identify or speculate the type and implementation of cryptographic algorithms. To characterize the correlation between the secret information and side-channel patterns, the initial step in an attack typically involves appropriately locating and aligning the trace sections that correspond to the target COs.

When evaluating the source code implementation of cryptographic algorithms against side-channel attacks, which we refer to as the **`White-Box`** condition, locating COs in the traces can be accomplished by simply adding synchronization signals (e.g., GPIO triggers) to the code. Such an idealized setup is commonplace in academic research for theoretical attacks conducted under controlled laboratory settings [CZLG21]. However, the source code of real-world targets is typically restricted, particularly in the embedded system industry where many proprietary libraries and firmware are distributed only in closed-source form throughout the supply chain [JSO20].

On the other hand, for cryptographic devices whose internal implementation is completely unknown to an attacker, which we refer to as the **`Black-Box`** condition, performing side-channel analysis (SCA) is an inherently challenging task. As presented in various real-world attacks on black-box cryptographic chips [RLMI21, Hér20, Hér22], attackers have to repeatedly speculate on the implementation and/or protection of the target and try out the corresponding exploits until they succeed, which is time-consuming and labor-intensive. Some attacks identify unintended synchronization signals, such as communication packets with peripherals, to deduce the locations of cryptographic operations for fault injection attacks [Boo20, Wou22]. However, such *ad hoc* features are unscalable for general cases. An alternative approach is to obviate or automate the process of locating COs in the overall side-channel attack routine based on particular assumptions about the trace. Lu et al. [LZC+21] propose a deep learning architecture for end-to-end profiling attacks that does not require explicit COs locations in raw traces, but rather requires the target to be within a limited range ($\sim$100,000 sampling points). But the real-world cryptographic applications such as Secure Boot may run for several seconds with millions of sampling points. Trautmann et al. [TBW+22] designed a semi-automatic tool for locating COs on side-channel traces, assuming that the execution time of the target CO is fixed and known to attackers. However, measuring this parameter with reasonable accuracy is non-trivial without access to source code. In summary, automatically locating COs in side-channel traces remains a challenging task under the non-white-box condition.

We have, however, observed that in real-world attacks against embedded devices, obtaining part or all of their *binary* firmware is frequently possible and typically one of the earliest steps in overall security analysis. For example, Bellom et al. [MRBT21] analyzed the Pixel 3 filesystem to obtain the firmware for the Titan-M security chip. Other methods of obtaining the target binary include extracting from the upgrade kit [CLC+22], exploiting software flaws [WGP21], downloading from on-chip storage [NS], or performing a dump via fault injection attack [WGP22, Alt]. In addition, many hardware wallet vendors make their binary firmware public available to ensure transparency to their users [Tre, Key, One]. Since it is neither access to the full source code nor completely unaware of the cryptography implementation, we refer to it as the **`Gray-Box`** condition [1]. The gray-box binary firmware lacks the same ease of code modification and trigger addition as the white-box source code, but still contains all necessary information for side-channel analysis of the target device, including cryptographic algorithm implementation and input-output logic. Several studies have utilized binary information of single instruction [NSUH22] or basic blocks [VdHOGT21] to improve the efficiency of locating fault injection positions in side-channel

[1]Note that this is not equivalent to accessing all secrets inside the device, which may be generated internally or live in secure storage isolated from the code segment.

traces to bypass security access controls. Nevertheless, there is no systematic framework for integrating information from binary firmware to facilitate side-channel analysis against cryptography algorithms in embedded devices. Under this scenario, the existing methods for segmenting and aligning the raw side-channel traces work no differently than with the black-box condition, despite the availability of sufficient information.

In contrast to side-channel trace locating, numerous established techniques exist for automated CO code identification in compiled binaries. Some detection methods recognize cryptographic codes based on heuristic rules around the statistical features of the disassembled binary code, such as the entropy of the code [WJC+09], the ratio of bitwise instructions [MWLGP12], specific cryptographic algorithmic constants [Gui], etc. There are also advanced methods utilize graph-level information like Control-Flow Graph (CFG) and Data-Flow Graph (DFG) for detection, where even proprietary cryptographic algorithms are identifiable[XMW17, LGF15, MMW21]. Unfortunately, there is no existing method to leverage these results on locating COs in the side channel trace.

A promising approach is to patch the firmware directly to inject the signals for synchronization as in the source code, and the techniques for directly modifying binary code logic are often referred to as binary instrumenting or rewriting [SHC20, DBMP23, QZZG23]. Applying such an approach requires answering two questions: 1) where to inject the code for accurate CO locating within the side-channel trace, and 2) what code to inject to efficiently bring the information in binary code analysis into side-channel analysis. However, challenges could arise from the balance between minimizing disruptions to the side-channel characteristics of the original firmware and ensuring that the inserted code reveals sufficient information for accurate CO locating.

Our contributions

In this paper, we fill the gap of locating COs from side-channel traces under binary-only conditions by utilizing binary analysis and instrumentation techniques. We first map the binary code execution to the side-channel traces using a binary instrumentation strategy that we tailor for side-channel analysis. Through the instrumentation, we obtain well-labeled and segmented side-channel traces. After that, we convert the side-channel CO locating task into the cryptographic binary code detection problem. By integrating existing software analysis techniques with our CO analysis scheme tailored for side-channel analysis, we finally present a tool that automatically locates the target CO in side-channel traces relying only on binary information.

- We present a systematic framework to automate the precise location of cryptographic operations in side-channel traces under binary-only gray-box conditions, combining runtime binary information with physical side-channel signals via static binary instrumentation.
- We propose a scheme for fine-grained mapping binary code execution to physical side-channel traces on embedded devices. The scheme aligns the points on the trace with binary instructions and enables the co-analysis of binary characteristics with side-channel profiles.
- Based on the proposed scheme, we develop a method to automatically locate cryptographic operations in side-channel traces. Our method only requires the use of information from the binary firmware without necessitating further assumptions about the side-channel characteristics of the target. Further, we implement and open source[2] a prototype of the proposed solution for ARM Cortex-M, which is the most widely used architecture in embedded devices.

[2] https://anonymous.4open.science/r/anonymous-ches24-29-4E58

- We first demonstrate our tool by automatically locating AES rounds in long and noisy side-channel traces. Additional experiments demonstrate that the tool, with minimal user knowledge based fine-tuning, effectively scales to more complex CO implementations, including AES with random-delay countermeasures and elliptic curve scalar multiplication. Further, we evaluate its effectiveness in real-world with a commercial close-sourced cryptography application. The performance analysis and comparison experiments highlight our advantages over existing methods in noisy and jittery traces.
- We performed runtime and storage performance analysis of our tool on diverse devices against various cryptography libraries, including tinyAES [kok], MbedTLS [ARMc], WolfSSL [Inc]. The results indicate that our performance supports applications on real-world embedded devices and surpasses the efficiency of related tools.

The rest of this paper is organized as follows. Section 2 presents the background of the research problem, the existing tools we used, as well as our motivation and overview. In Section 3 we present our proposed framework in detail, including design considerations and challenges addressed. In Section 4, we introduce an application of the proposed framework on how to use binary information to automatically locate COs in side-channel traces. In Section 5, we evaluate our proposed tool on different hardware and cryptographic algorithm implementations and analyze the results. In Section 6, we provide a discussion around the limitations and possible future works of the method in this paper. Finally, we conclude this paper in Section 7.

2 Background

2.1 Locating COs in a side-channel trace

For a better understanding of our research questions and method, we formulate the problem to be addressed here. Define T_{sc} as the side-channel trace produced by the device:

$$T_{sc} = [\texttt{ADC}_{\mathrm{t_{start}}}, \cdots, \texttt{ADC}_{\mathrm{t-1}}, \texttt{ADC}_{\mathrm{t}}, \texttt{ADC}_{\mathrm{t+1}}, \cdots, \texttt{ADC}_{\mathrm{t_{end}}}] \tag{1}$$

where $\texttt{ADC}_{\mathrm{t}}$ represents the digital signal obtained by converting the amplitude of the side-channel measurement at sampling point t. Then locating a target CO in the side-channel trace can be defined as finding the segments $S_{co,i,s}$:

$$S_{co,i,s} = \left[\texttt{ADC}_{\mathrm{t_{CO,i,s}}}, \cdots, \texttt{ADC}_{\mathrm{t_{CO,i,e}}}\right] \tag{2}$$

where $\mathrm{t_{CO,i,s/e}}$ indicates the start and end moment of the ith round of CO execution. As mentioned earlier, it could be challenging to obtain this result automatically without additional information.

Since the proposed solutions that follow in this paper make extensive use of the binary instrumentation techniques, a necessary introduction to the techniques we use is provided in the next subsection.

2.2 Binary instrumentation on embedded devices

Binary instrumentation is a technique that involves directly modifying binary code to gain insights into its behavior or to insert extra code, which is desirable for addressing our challenges. Unlike on the x86 platform, there are quite limited binary instrumentation tools available for embedded platforms. We employ the PIFER framework [QZZG23] as our basic instrumenting tool, due to its open source and well support for the ARM Cortex-M, the embedded platform we focus on.

PIFER works by patching the target instruction to one that throws an exception when executed, and meanwhile hijacking the corresponding exception handler in the interrupt vector table to execute its own appended code (refer to Figure 3 for a visualization). After gaining the control flow, it will search for and jump to the user-defined hooking code corresponding to the address where the exception occurs. In order to allow hooking of (almost) arbitrary addresses, PIFER uses a complex mechanism to handle different instructions case by case.

However, it suffers from excessive time and storage overhead. Therefore, we thoroughly optimize PIFER for our purposes, which concentrate on COs and side-channel scenarios, by removing unnecessary instruction processing logic and significantly improving the efficiency of the target lookup algorithm (Section 3.2.1).

2.3 Overview

Motivation. Assuming that we obtain another trace T_{bin} for the execution of the binary code inside the target during the same time:

$$T_{bin} = [\texttt{PC}_{\mathrm{t_{start}}}, \cdots, \texttt{PC}_{\mathrm{t-1}}, \texttt{PC}_{\mathrm{t}}, \texttt{PC}_{\mathrm{t+1}}, \cdots, \texttt{PC}_{\mathrm{t_{end}}}] \tag{3}$$

where $\texttt{PC}_{\mathrm{t}}$ is the address of the the binary code being executed at moment t. The intuition is that by identifying the set of addresses associated with the CO (program counter $\mathrm{PC_t}$), we can precisely determine the indexes of T_{bin} that belong to the CO and derive the corresponding $\mathrm{t_{CO,i,s/e}}$ in Equation 2. However, two primary challenges must be overcome to achieve this goal:

1. Obtaining T_{bin}. It is equivalent to **mapping** the binary execution to the side-channel trace, i.e., identifying the running instruction at a specific point on the latter.

2. Identifying the binary instructions equivalent to the COs in the SCA perspective, in particular the **boundaries** corresponding to $\mathrm{t_{CO,i,s/e}}$.

Our method. After introducing the problem definition and the employed techniques, we offer a high-level overview of our solution here. Basically, the problem mentioned in Section 2.1 is divided into two subtasks according to the challenges identified:

- **Mapping the binary execution to the side-channel trace**: First, we address the challenge of obtaining T_{sc}. By designing a double instrumenting scheme, we can accurately obtain the correspondence between the binary program execution and the side-channel trace (Section 3).

- **Identify the boundary addresses of COs in a binary firmware**: Integrating the runtime mapping information and static binary analysis, we automatically locate appropriate boundary addresses within the cryptographic functions to further segment the COs in the side-channel traces. (Section 4).

Combining the results of the two subtasks, we can acquire high-quality side-channel trace segmentations of the COs from target devices for further analysis (Section 4.3.2). In the next two sections, we will dive into the details of each sub-task.

3 Mapping the program flow to the side-channel trace

In this section, we address the first subtask: mapping the binary execution flow to the side-channel trace. As illustrated in Figure 1 from a high-level perspective, our proposed approach consists of the following steps.

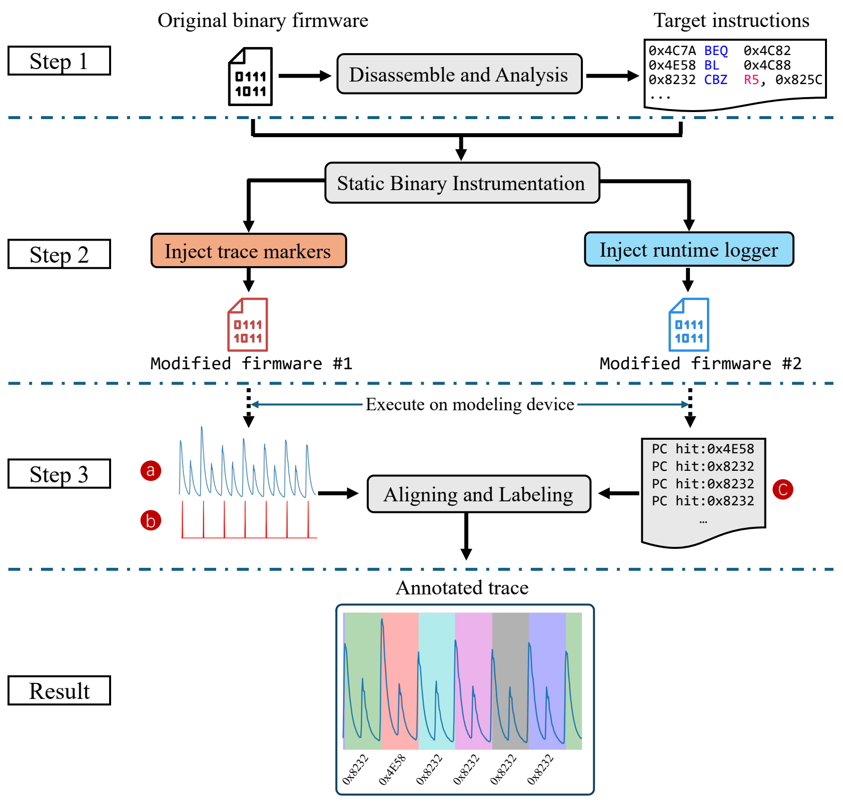


Figure 1: Mapping the program flow to the side-channel trace using binary analysis and instrumentation

1. Disassemble and analyze the firmware to extract the instructions of interest (Section 3.1). Hooking each instruction exhibits significant overhead and disrupts the original side-channel characteristics. Therefore, it is more practical to selectively trace a few critical instructions for SCA. For example, focus on the ones that make the Program Counter (PC) change non-linearly to depict the control flow.

2. Apply static binary instrumentation on these addresses to generate two patched binaries (Section 3.2):

   - **`firmware #1`**, where a code snippet that synchronously generates the segmentation signals (referred to as trace markers) is injected at all target instructions. For example, raising a GPIO pin before and after any AES round.
   - **`firmware #2`**, where the code injected at the target instructions will log runtime information such as the current PC or cycle counter. These logics usually take longer to process, hence they are separated from the trace markers in **`firmware #1`** to minimize the latter's impact on the original side-channel characteristics.

3. Execute two modified binaries on a controlled device to obtain:

   (a) The side-channel trace T_{f1} (e.g., EM or power) of the target running **`firmware #1`**. which contains the side-channel features of the original binary as well as our injected marker code.

   (b) The marker signals are triggered when executing to target addresses, which provide a merged version of adjacent $\mathrm{t}_{\mathrm{CO[i]}_{\mathrm{e/s}}}$. In other words, these signals indicate the execution timing of the target instructions, enabling alignment of binary code execution with the side-channel trace.

(c) The execution flow record of those addresses. By design, the marker signal only indicates that a target instruction is being executed, requiring additional information to identify which one it is. This is achieved by matching the occurrences of the marker signals one-to-one with the logged PC addresses.

4. Finally, we remove the noise introduced by the injected code in $T_{patched}$ and annotate the segments with the instruction addresses with the marker signals (Section 3.3).

In this way, we establish a precise mapping between binary code execution flow and side-channel traces, providing a basic framework for assisting side-channel analysis with binary information (or, conversely, side-channel analysis-assisted binary analysis). Next, we'll dive into the design details and challenges involved in each step.

3.1 Locate appropriate instructions

First of all, we need to parse and disassemble the raw binary firmware, which may be dumped from on-board FLASH memory or upgrade packages and is not in ELF format. There are quite a few tools and studies focusing on this problem, and we adopt Ghidra [Ghia] for its open-source license and high scalability for embedded architectures.

The practical challenge in this step is to select the appropriate target instructions to trace in a side-channel analysis context. If the trace is too coarse-grained (e.g., function level), it may fail to segment the side-channel trace well, especially for some massive cryptographic functions with possibly dozens of loops inlined. On the other hand, if we take a too-fine granularity (instruction level), the overhead is prohibitively high and the noise interference introduced by the tracker will completely ruin the side-channel information. In our design, we focus on all instructions that make non-linear changes to the PC registers, i.e. we are tracking the granularity of the basic block in the control flow graph. Taking the ARM Cortex-M architecture as an example, this includes unconditional jump instructions (`B`), conditional jump instructions (`B{cond}`, `CBZ/CBNZ`), function call (`BL`) and return instructions (`BX LR`). Choosing such basic blocks as the trace granularity has two advantages in the context of side-channel analysis:

- The instructions inside a basic block execute sequentially, with a fixed time duration most of the time (when there are no interrupts), facilitating precise mapping to side-channel trace.
- Instrumenting at these locations avoids disrupting arithmetic and data processing instructions, minimizing interference with information in the side-channel trace relevant to cryptographic operations of primary interest.

Specifically, we develop a Ghidra Python script to automate the processing of the binary firmware and filter out all PC-altering instructions in a pattern-matching way.

3.2 Static binary instrumentation for embedded firmware

In this step, we perform static binary instrumentation at the target addresses extracted in Step 1 to inject new code gadgets directly into the raw binary firmware. Specifically, those gadgets should help to precisely segment and label the side-channel traces, with the following design requirements:

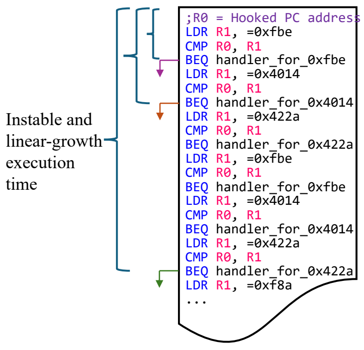
- **Correctness:** The annotation signal must accurately map the executing binary code block to the side-channel trace.

- **Efficiency:** There may be thousands of addresses to hook, therefore the performance is crucial. This rules out the option of using complex communications such as a hardware debugger.
- **Availability:** The way to send annotation information must be generic enough without dependence on specific hardware implementations (e.g. UART, MTB [ARMa]).
- **Low and removable noise:** Adding new code to the program execution will also emit physical side-channel information, hence minimizing and eliminating such noise from the trace is essential.

For correctness and efficiency, we first refine an existing binary instrumenting tool to fit our side-channel analysis scenario. For efficiency, availability, and noise issues, we propose a double-instrumentation scheme: one patch (`firmware #1` in Figure 1) focuses solely on efficiently segmenting traces in real-time, while the other (`firmware #2` in Figure 1) records the execution information of binary code. By combining the outputs generated by those two firmware, one can easily determine the executing code block at any given time.

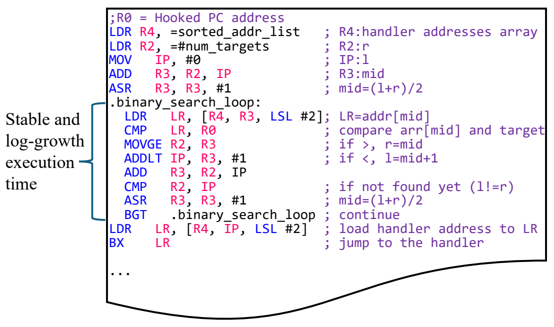
3.2.1 Refining binary instrumenting tool for side-channel analysis purpose

As discussed in Section 2.2, our basic instrumenting tool PIFER, as a generalized framework, is not designed for side-channel analysis. On the one hand, its performance overhead rises significantly with the number of hooking addresses. On the other hand, the linear lookup mechanism employed in the search for the hook handler corresponding to a target instruction, as depicted in Fig. 2(a), introduces instability in the instrumentation time overhead, hindering the removal of those noises from side-channel traces. In order to address the above issues and make it applicable to side-channel scenarios, we make two significant modifications to PIFER:



```
;R0 = Hooked PC address
LDR R1, =0xfbe
CMP R0, R1
BEQ handler_for_0xfbe
LDR R1, =0x4014
CMP R0, R1
BEQ handler_for_0x4014
LDR R1, =0x422a
CMP R0, R1
BEQ handler_for_0x422a
LDR R1, =0xfbe
CMP R0, R1
BEQ handler_for_0xfbe
LDR R1, =0x4014
CMP R0, R1
BEQ handler_for_0x4014
LDR R1, =0x422a
CMP R0, R1
BEQ handler_for_0x422a
LDR R1, =0xf8a
...
```

(a) PIFER's handling process



```
;R0 = Hooked PC address
LDR R4, =sorted_addr_list   ; R4:handler addresses array
LDR R2, =#num_targets       ; R2:r
MOV   IP, #0                ; IP:l
ADD   R3, R2, IP            ; R3:mid
ASR   R3, R3, #1            ; mid=(l+r)/2
.binary_search_loop:
  LDR   LR, [R4, R3, LSL #2]; LR=addr[mid]
  CMP   LR, R0              ; compare arr[mid] and target
  MOVGE R2, R3              ; if >, r=mid
  ADDLT IP, R3, #1          ; if <, l=mid+1
  ADD   R3, R2, IP
  CMP   R2, IP              ; if not found yet (l!=r)
  ASR   R3, R3, #1          ; mid=(l+r)/2
  BGT   .binary_search_loop ; continue
LDR   LR, [R4, IP, LSL #2]  ; load handler address to LR
BX    LR                    ; jump to the handler

...
```

(b) Our handling process

Figure 2: Improve and stabilize the time overhead for hook handler matching process.

1. Utilize bisection instead of linear search to reduce both the time and storage overhead, as illustrated in Figure 2(b). Since all target addresses are determined prior to firmware execution, they can be pre-sorted and employed for bisection search. Such an approach also maintains consistent overhead across different instrumenting targets, thus simplifying the removal of their noise signals from side-channel traces.
2. Eliminate processing logics that are not relevant to branch instructions to minimize runtime overhead. PIFER serves as a general-purpose tool that includes additional logic for instrumenting instructions such as `MOV` or `LDR`, which are unnecessary for our purpose.

Simple measurements using hardware have shown that our implementation executes much faster compared to PIFER's, with a total overhead of approximately 64 clock cycles when hooking a single address compared to PIFER's 300. Additionally, our binary searching algorithm results in an even shorter runtime for each hook as the number of hooking addresses increases.

3.2.2 Inject the trace marker

In **`firmware #1`**, we focus solely on efficiently segmenting the side-channel traces. Taking a conditional jump in a loop as an example, the injected trace marker system is illustrated in Figure 3. The PIFER framework takes over the control flow by patching the target instruction with an undefined instruction and hijacking the handler in the vector table. It also automatically preserves the context and determines the next value of the PC register to ensure the correct execution of the original program after the patch. To minimize the overhead, a marker signal is generated immediately when we gain control of the execution. The signal is typically a GPIO pin pull-up/pull-down, which is available on almost all embedded platforms. When the device requires GPIO pins' initial configuration before use, we hijack the reset handler to inject corresponding code at the very beginning of execution. After executing the corresponding hook function (inside the PIFER framework) which determines the next PC depending on the context, we reset the trigger pin before returning to the original control flow.

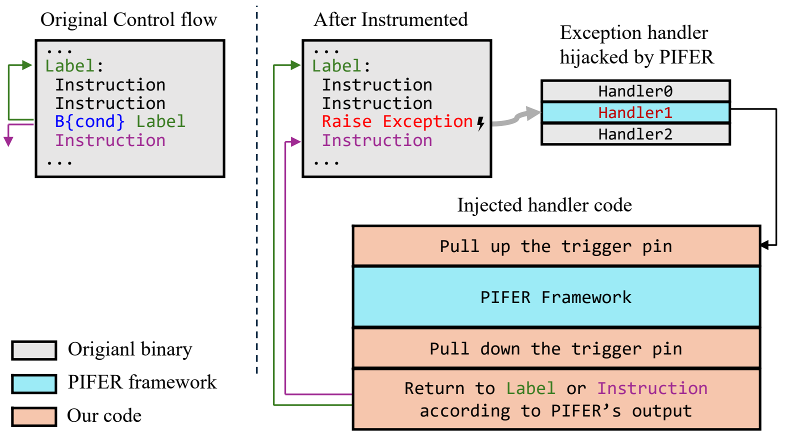


Figure 3: Marker signal injection example for a conditional jump instruction.

Note that to minimize interference with the original side-channel information, the hooking code above only sends the marker signals for trace segmentation, without recording associated instruction addresses. Nevertheless, it still introduces additional noise of hooking code execution into the original side-channel trace and makes the trace a bit "longer". In subsection 3.3, we will discuss methods of eliminating them.

3.2.3 Inject the runtime logger

In **`firmware #2`** we record the order of target address occurrences, a task that **`firmware #1`** omits due to noise overhead. A simple approach involves acquiring a debug trace utilizing a specialized performance evaluation tool, such as Segger J-Trace [Gmb], and searching for a subset of the target instruction addresses within it. However, such tools tend to be expensive and closed-source and require multiple target pins that are potentially redefined for other purposes. Therefore, we propose an alternative based on binary instrumentation and hardware breakpoints. Specifically, we first hijack the control flow at every target instruction following Fig. 3, without any additional operations. Next, we set a hardware breakpoint at the beginning of the hijacked handler using a hardware debugging interface (SWD, JTAG, etc.), and develop a GDB script on the host computer to read the source address each time the execution reaches the breakpoint, thereby recording the target instructions' execution flow. In other words, we *collapse* the execution of all target instructions to the same location through binary patching, effectively enabling tracing with a single hardware breakpoint (usually only 2-6 in total). The time overhead of communicating with the debugger and the host computer is non-critical, as only a single execution record is needed, and there is no concern about perturbing the side-channel trace, which is already handled in **`firmware #1`**.

3.3 Eliminate noise and annotate the trace

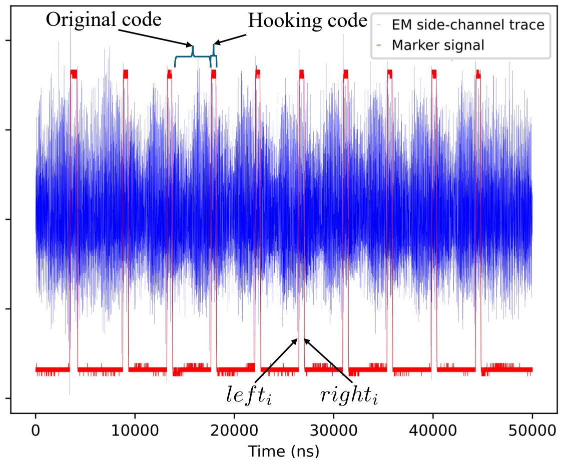


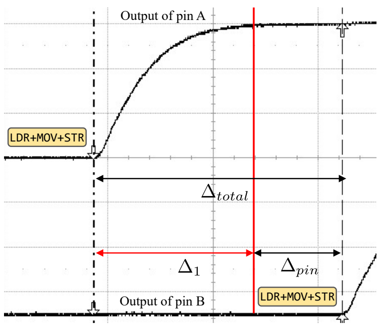
Figure 4: EM side-channel trace and marker signal collected from a device running an instrumented firmware which performs an AES-128 encryption.

As shown in Figure 4, there are extra sections in the side-channel trace after instrumenting due to the hooking code. We use the following procedure to eliminate those noise sections:

1. Extract all the edge pairs $[left_i, right_i]$ of the marker signals. Specifically, we scan the trace of the marker signal from left to right and record the offsets in the array where the amplitude exceeds or falls under a certain threshold (e.g., half of the maximum amplitude).
2. Measure interrupt response latency Δ_1 and interrupt return latency Δ_2.
3. For each edge pair, remove the $[left_i - \Delta_1, right_i + \Delta_2]$ part in the trace.

The interrupt response latency Δ_1 is the time between an undefined instruction's execution and the first instruction in the exception handler, while the interrupt return latency Δ_2 is the duration from the handler's final instruction to the next instruction in the original execution flow. Such delays are typically caused by the processor switching between contexts and are only related to the hardware device used. According to the hooking process depicted in Figure 3, the execution from $[left_i - \Delta_1, right_i + \Delta_2]$ is irrelevant to the original binary. Therefore, we must exclude these parts from the trace.

Figure 5 illustrates how the Δ_1 or Δ_2 is accurately measured using hardware metrics. Specifically, the device under test will pull up two different pins before (pin A) and after



(a) Measurement by the oscilloscope.

```
LDR   R1, =0x50000000
MOV   R3, #0x4000000
STR   R3, [R1,#0x508] ;pull up pin 26
UDF   #0xAA
...

handler:
LDR   R1, #0x50000000
MOV   R3, #0x8000000
STR   R3, [R1,#0x508] ;pull up pin 27
```

(b) Code used to measure Δ_1.

Figure 5: Measurement example for nRF52840 chip [Sem], where pin A set to P26 and pin B set to P27.

(pin B) the exception occurs, having an oscilloscope capture and measure their voltage level. The precise execution time of the interrupt response latency Δ_1 is calculated by subtracting the overhead of the pin control code Δ_{pin} (usually a combination of `LDR`+`MOV`+`STR` instructions) from the difference between the two rising edges Δ_{total}. The value of Δ_{pin} can be easily determined by measuring two successive pull-ups.

Finally, we iterate through the sequence of instructions produced by **`firmware #2`** and assign their addresses one by one to the side-channel locations corresponding to the marker signals. In conclusion, we successfully establish a mapping between side-channel traces and binary code. In the next section, we present a practical application of this hardware-software co-analysis technique: automatically locating cryptographic operations in side-channel traces using binary information.

4 Automatically locating cryptographic operations in side-channel traces using binary information

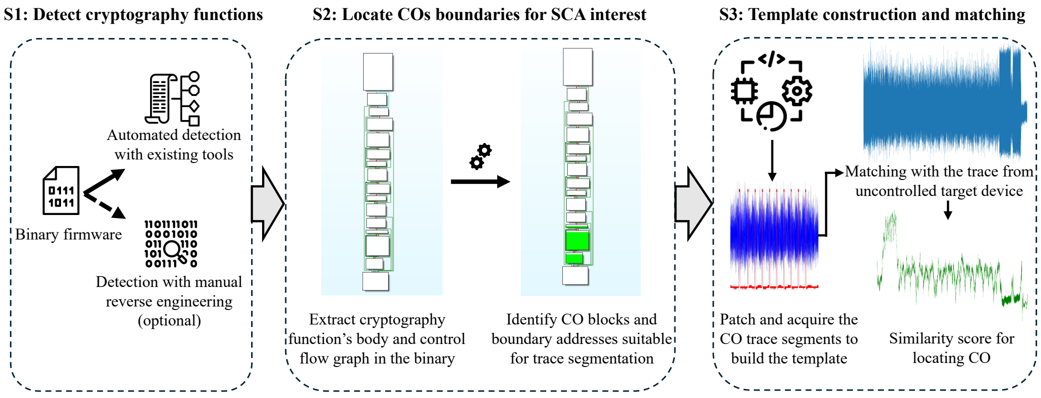

S1: Detect cryptography functions
Automated detection with existing tools
Binary firmware
Detection with manual reverse engineering (optional)
S2: Locate COs boundaries for SCA interest
Extract cryptography function's body and control flow graph in the binary
Identify CO blocks and boundary addresses suitable for trace segmentation
S3: Template construction and matching
Matching with the trace from uncontrolled target device
Patch and acquire the CO trace segments to build the template
Similarity score for locating CO

Figure 6: Automatically locating cryptographic operations in side-channel traces using binary information.

In this section, we first clarify our application scenarios and the threat model (Section 4). Then, we present our solution for automatically locating cryptographic operations in side-channel traces using binary information. As illustrated in Figure 6, the proposed approach consists of three steps:

- **S1:** Detect cryptographic functions in the binary firmware. Given that this long-studied problem in software analysis is not directly related to SCA, we just use existing tools for automation. As an alternative, manual reverse engineering may further improve the results but requires extra labor efforts. (Section 4.1)
- **S2:** In the function body, locate the boundary addresses of the COs suitable for trace segmentation. Based on the patterns of typical COs targeted in SCA, we build an efficient locating method that leverages both static analysis and dynamic statistical information generated by **`firmware #2`**. (Section 4.2)
- **S3:** Instrument at these boundary addresses as in Section 3 and obtain a set of segmented-by-CO traces from a profiling device. Then, build the templates to match unlabeled traces acquired from the target device. Finally, locate the CO positions in the target trace based on the matching scores. (Section 4.3)

Requirements and Application Scenarios. Before we dive into the details, we must clarify the application scenarios and assumptions of our method. We assume that the attacker is able to physically access the target device and capture multiple side-channel traces by triggering the CO execution inside. Following the common assumptions for side-channel attacks, we also assume that some basic level of user knowledge is known to the proposed method, including the architecture information of the target, the type of cryptographic algorithm to be attacked and the manner to invoke/trigger the execution of the cryptographic algorithm. We also assume that the user may supply more knowledge to fine-tune the tool to better handle sophisticated COs or countermeasures, as discussed in Section 5.2 and 5.3. Following typical real-world attack scenarios, we assume that the user can manage to acquire a controllable hardware of the same model (i.e., buying development kits), which we refer to as the profiling device. As we are focusing on the gray-box scenario, we also assume that the attacker has access to the binary code of the target cryptography implementation, but not to the key. For example, the secure boot or cryptography library in the SDK of the same chipset could be shared between different products, but the keys for encryption/decryption may vary.

4.1 Step 1: Function level cryptography code detection in the binary

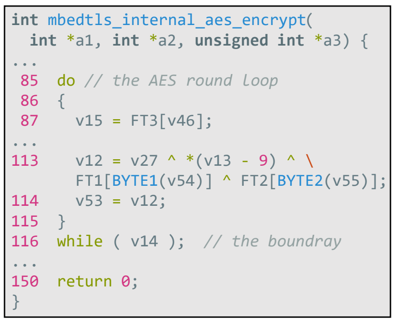
We employ two open-source tools to detect cryptographic functions in the binary firmware: a) **`where's crypto`** [MMW21], which uses data flow graph isomorphism to detect symmetric cryptography functions, and b) **`BinaryAI`** [JAH+24], which conducts binary-to-source similarity analysis to recognize cryptography library functions. Both tools can analyze stripped binary firmware without function names or other debug information, and we just pick the cryptographic functions of interest in their output. However, they mainly focus on function-level identification for reverse engineering or malware detection, while SCA requires finer-grained operations that correspond directly to some core cryptography structures, requiring a finer-grained definition and location for the CO. For example, marking the call/return of an ECDSA signing function extracts a side-channel trace containing all operations, such as memory management, random number generation, elliptic curve group arithmetic, and hashing. However, regular CPA or DFA attacks often need alignment to specific COs like scalar multiplication.

4.2 Step 2: Locating fine-grained COs boundaries for SCA interests

In our current implementation, we choose to use the **loop bodies** instead of function as a more reasonable basic unit of CO for SCA. The benefits are twofold: 1) the loop is more focalized on COs than the function body, providing significant information for SCA, and 2) it is a general structure that forms the majority of cryptography algorithms, giving us great generalizability. Another practical reason is that in memory-constrained embedded devices, loops are more popular than unrolled implementations for saving binary size. For example, the AES loop in MbedTLS [3] are never unrolled under ARM-GCC across different optimization levels. Nevertheless, it is possible to extend our tool to handle loop-unrolling COs, as discussed in the Future work in Section 6.2.

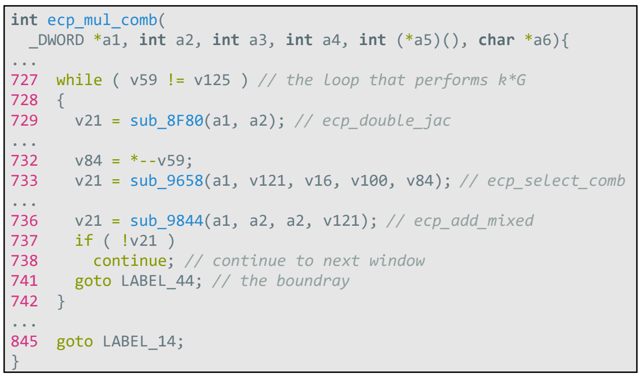
4.2.1 Identify the CO-related loops from others

Using Ghidra's scripts and headless analyzer [ghic], we automate the process of getting all the candidate loops in the given function[4]. In some simple cases, there is only one loop in the cryptographic function, such as the AES block encryption/decryption in MbedTLS (shown in Figure 7(a)). However, depending on the implementation and compilation options, the target function may include many non-CO loops, as exhibited in Figure 7(b).



```
int mbedtls_internal_aes_encrypt(
  int *a1, int *a2, unsigned int *a3) {
...
  do // the AES round loop
  {
    v15 = FT3[v46];
...
    v12 = v27 ^ *(v13 - 9) ^ \
    FT1[BYTE1(v54)] ^ FT2[BYTE2(v55)];
    v53 = v12;
  }
  while ( v14 );  // the boundray
...
  return 0;
}
```

(a) Decompiled AES function exhibits only one loop, which is easy to identify.



```
int ecp_mul_comb(
  _DWORD *a1, int a2, int a3, int a4, int (*a5)(), char *a6){
...
  while ( v59 != v125 ) // the loop that performs k*G
  {
    v21 = sub_8F80(a1, a2); // ecp_double_jac
...
    v84 = *--v59;
    v21 = sub_9658(a1, v121, v16, v100, v84); // ecp_select_comb
...
    v21 = sub_9844(a1, a2, a2, v121); // ecp_add_mixed
    if ( !v21 )
      continue; // continue to next window
    goto LABEL_44; // the boundray
  }
...
  goto LABEL_14;
}
```

(b) Decompiled elliptic curve scalar multiplication function. By inlining multiple sub-functions, it has over 20 loops and 800+ lines of pseudo-C, while only 1 loop with ~10 LoC is the CO we are interested in.

Figure 7: Analysis function samples in a compiled (-O3) MbedTLS library for Cortex-M4.

To identify the CO-related loops, we first filter out the non-cryptographic ones according to a set of heuristical rules as follows:

- **Infinite loop:** Contains basic block that unconditionally jumps to itself.
- **Idle/Peripherals waiting loop:** Contains special instructions such as `WFI` (Wait For Interrupt), `WFE` (Wait For Event), etc.
- **Tiny loop:** Contains very few instructions (e.g., less than 5) and no function calls in the whole loop body. They are usually inlined linear memory operations (`memcpy`/`memcmp`). In contrast, cryptographic operations typically have sub-routines or hundreds of arithmetic/bitwise instructions per loop.

Next, we further determine the CO candidates by exploiting the fact that a) COs generally consume more execution time than regular non-CO loops, and b) the number of loop iterations in COs is often predictable due to their constant-time implementation. For example, during ECDSA signing on a P-256 curve using MbedTLS's comb/window-based implementation (**`ecp_mul_comb, window size=5`**), we expect to see a time-consuming loop executed $\lceil \frac{256}{5} \rceil = 52$ times, corresponding to the scalar multiplication kG.

[3] https://github.com/Mbed-TLS/mbedtls/blob/c7569a8c4bc7525a4d4d435eb3cd04031d7f64bc/library/aes.c#L961

[4] https://anonymous.4open.science/r/anonymous-ches24-29-4E58/src/find_loops.py

[5] https://github.com/Mbed-TLS/mbedtls/blob/2ca6c285a0dd3f33982dd57299012dacab1ff206/library/ecp_curves.c

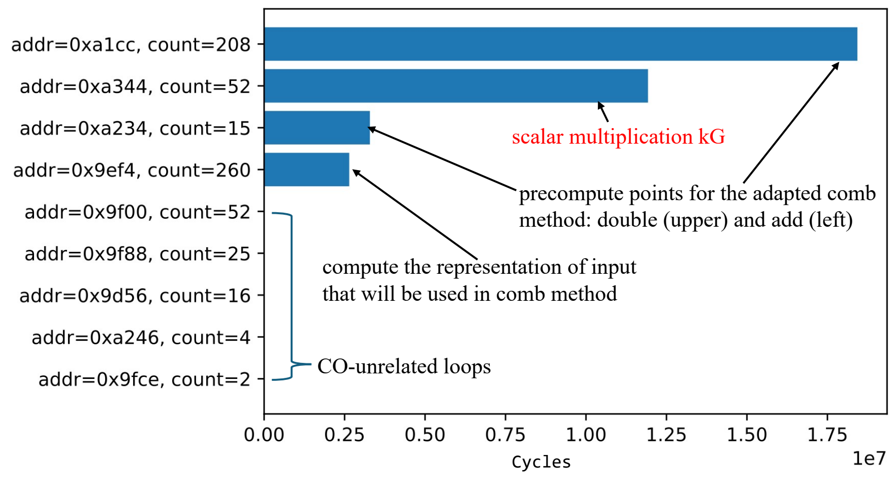


Figure 8: The remaining loops and their statistics after heuristic filtering in the **ecp_mul_comb** function in MbedTLS 2.6.10 from Figure 7(b), which performs a signing over the P-256 curve using comb method from scratch. In the latest version of MbedTLS, however, the precomputation tables for standard curves are **hard-coded**[5], making the scalar multiplication loop the most time-consuming one in the signing process.

```
0x824A   MOV     R1, R10  ; loop start
...
0x8282   CMP     R0, #0
0x8284   BNE.W   0x87EA   ; break early
...
0x828A   CMP     R5, R6
0x828A   BNE     0x824A   ; boundary: last conditional jump to loop start
```

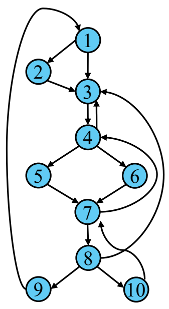
Figure 9: Select the instructions corresponding to the boundaries of the CO.

As illustrated in Figure 8, CO and irrelevant loops can be clearly distinguished by the timeshare and the number of loop executions. These statistics can be efficiently obtained by running **firmware #2** in Section 3.2.3 with minor modifications. Specifically, we instrument the control statement of each remaining candidate loop to log the clock cycle differences using the DWT Cycle Counter [ARMb]. For some special platforms without Cycle Counter support (Cortex-M0/M0+), we use **firmware #1** to hook at the same locations and calculate the markers' time differences as an alternative.

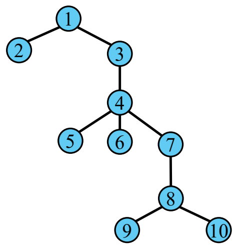
For each algorithm implementation, we define a configuration pair (N_{top}, C_{loop}) to determine the last selected loops, specifying the top N_{top} most time-consuming loops and their numbers of executions are in the set C_{loop}. Thus, even with the presence of false positives, the corresponding loops must occupy a significant portion of the trace, and segmenting them out would also be beneficial to the SCA. For example the AES-128 configuration is $(N_{top} = 1, C_{loop} = \{9, 11, 14\})$, while the ECDSA signature configuration, using the function in Figure 7(b), is $(N_{top} = 3, C_{loop} = \{52\})$. In summary, we comprehensively utilize the binary analysis tools like Ghidra as well as the outcome from Section 3.2.3 to automate the identification of the COs in the binary function.

4.2.2 Locate the boundary instructions for trace segmentation

To construct a robust matching template, we aim to include the loop body as fully as possible. Therefore, for each of the CO loops, we use the *last conditional branch* instruction



(a) An example control flow graph



(b) Dominator tree

Back-edges	Loop body
4→3	③④⑤⑥⑦⑧⑩
7→4	④⑤⑥⑦⑧⑩
8→3	③④⑤⑥⑦⑧⑩
9→1	①~⑩
10→7	⑦⑧⑩

(c) Back-edges and corresponding natural loop body

Figure 10: Finding loops using the classical dominator tree approach.

of the loop body as the boundary marker for the CO loop, as shown in Figure 9. To find such boundaries, we need to exploit the natural loop finding process, as shown in Figure 10, which involves the following steps:

1. Construct the dominator tree of the CFG: A node u of a CFG dominates node v if every path from the entry node to v must go through u. The dominator tree can be loosely understood as linking each node to its closest dominating parent. The construction of the dominator tree and its detailed definition are beyond the scope of this paper, and we obtained the dominator tree using Ghidra's built-in function [ghib]. Therefore, it is not discussed further.

2. Identify back edges: A back edge is an edge in the original CFG that connects a node to one of its ancestors in the dominator tree. Each edge represents a potential loop in the CFG.

3. For each back edge (u, v), where u is the source node and v is the target node:

    - Temporarily delete node v from the flow graph.
    - Find the node set $S_{u,v}$ that can reach u. Those nodes set $S_{u,v} \cup v$ forms the natural loop of (u, v).
    - Find the last instruction in node u. If a jump instruction is encountered, it is treated as the boundary of the CO; otherwise, a breadth-first search is initiated from it along the reversed edges in the loop body's subgraph until the jump instruction is reached.

4. Repeat Step 3 for all back edges in the CFG to identify all the natural loops and their boundaries.

Note that loop-finding algorithm itself is not our contribution; it is provided only to make the paper clearer and self-contained. However, we add the operations of finding CO boundary instructions to its framework. As depicted in Figure 9, tracking the execution of those boundary instructions, which usually control the loop execution and are executed as many times as the loop itself (with a maximum deviation of 1 if the loop is broken early), is equivalent to tracking the CO execution. For nested loops, we begin by marking solely the largest loop. If the resulting match is unsatisfactory, we can identify a smaller layer of loops and repeat as necessary.

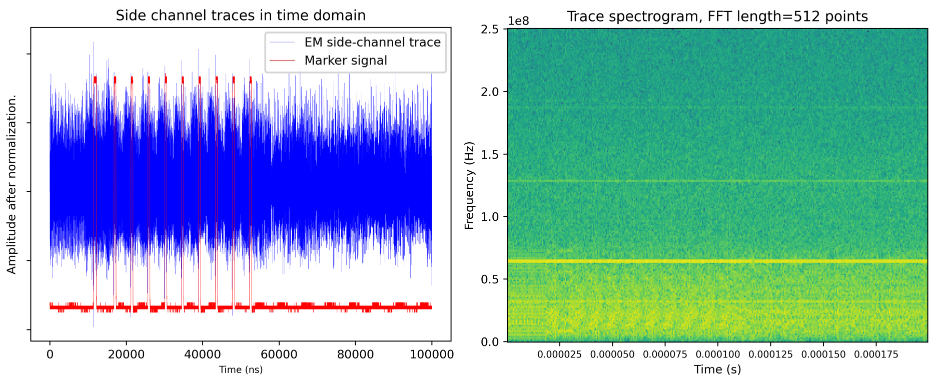
4.3 Template construction and locating in target traces

4.3.1 Mapping the binary COs to the side-channel trace

In Step 2, we instrument the CO-representing addresses obtained in the previous stage using the same method described in Section 3 to obtain **`firmware #1`** and **`firmware`**

#2. Subsequently, we executed the two firmware on a profiling device and combined them to produce annotated traces. The profiling device should not only be the same type of hardware as the target device, but also share the same clock frequency, physical environment, and parameters for side-channel trace acquisition (e.g., the relative position of EM probes). After obtaining the annotated traces, we extract the trace segments corresponding to the execution of CO by selecting the parts between each $right_{i-1}$ and $left_i$ in Figure 4. Note that this approach usually drops the first CO in a cryptographic algorithm sequence. Nevertheless, we can still use the patterns from the remaining rounds of the CO to match the first one in the unknown trace.

4.3.2 Pattern matching on traces collected from uncontrolled target device



(a) EM traces of an AES-128 encryption (b) Corresponding spectrum features

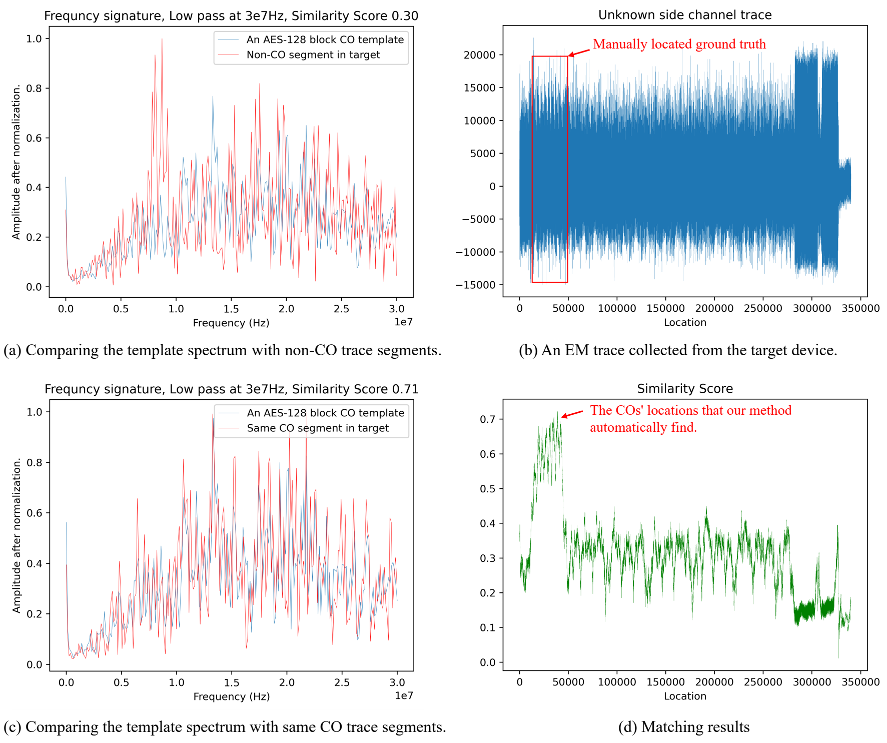
Figure 11: Frequency domain signature of CO.

After producing the well-segmented and labeled version of the side-channel trace for **`firmware #1`** on the profiling device, we can construct a fine-grained template of COs. By pattern matching, the template enables us to locate hidden COs in unknown traces that are collected from the unmodified target device. While more advanced methods, such as deep neural networks [BPS+20], exist for pattern matching, we implement an alternative solution that leverages spectral features for CO localization, which is more lightweight and serves as a proof-of-concept.

Using the frequency domain information of side-channel traces as a feature in CO pattern matching has been proven to be an effective method [LDMPT15]. As shown in Figure 11, the spectrum characterizes the energy distribution in the physical side-channel leakage. According to the underlying principles of side-channel attack, it is highly relevant to the operations and data performed in the device. Specifically, we build spectrum-based side-channel templates by the following steps:

1. Perform Fast Fourier Transform (FFT) and low-pass filtering based on spectral characteristics for the side-channel trace segments that map to any CO.
2. For multiple trace segments of the same CO, normalize their spectrum features.
3. Calculate the average width of remaining segments as W_{CO_i}, and use interpolation to transform all of these segments to the same length W_{CO_i}.
4. Average the remaining frequency domain distributions to produce a fusion template.

Figure 12 illustrates a frequency-based fusion template for detecting AES-128 block operations, which is constructed using only one profiling trace. It can be seen that the side-channel profile of the same AES-128 implementation (from another device, with unpatched



(a) Comparing the template spectrum with non-CO trace segments. (b) An EM trace collected from the target device.

(c) Comparing the template spectrum with same CO trace segments. (d) Matching results

Figure 12: The template of an AES-128 block CO and the matching results.

firmware) has a significantly higher similarity to it, whereas the common code (benchmark code from CoreMark) differs significantly. Finally, for each CO, we scan all segments of length W_{CO_i} on the target trace with an adjustable step (typically having $1 \sim 2\%$ points of W_{CO_i}), perform an FFT on them and compare the frequency characteristics with our template. Figure 3 illustrates the matching results of applying the template on a long, noisy trace, where similarity is measured using the Pearson correlation coefficient. Evidently, the similarity scores are significantly higher ($\sim$0.7) in the positions with correct cryptographic operations than in the other positions ($\sim$0.3).

4.4 Implementation and open source release

We implemented a prototype targeting the above method using about 3,400 lines of Python code. It generically supports the ARM Cortex-M platforms and works directly on the raw binary firmware. We use Ghidra scripts extensively for automated binary analysis of firmware and rely on keystone/capstone to assemble/disassemble. The implemented tool is released under an open-source license and can be found at https://anonymous.4open.science/r/anonymous-ches24-29-4E58.

5 Evaluation

In this section, our evaluation is conducted to answer the following research questions that support our early claims of our proposed tool:

- **Correctness:** Can it correctly locate cryptographic operations in long, noisy traces?
- **Practicality :** Does it scale to real-world, complex firmware for cryptographic applications?

- **Performance:** Does its runtime and memory overhead suffice for side-channel security analysis?

To answer the above questions, diverse measurements, and case studies are carried out in this section. We first evaluate the tool's accuracy on a series of long and low-quality traces generated based on the CoreMark benchmark suite [EEM09], with cryptographic operations representing less than 0.2% of the whole trace (Section 5.1). Next, we evaluate the scalability of the proposed method by integrating minimal user knowledge to handle the random delays as a side-channel countermeasure (Section 5.2). Further, by fine-tuning the template matching process, we investigate its effectiveness in locating long and complex COs with, e.g., an optimized elliptic curve scalar multiplication in ECDSA signing implemented by MbedTLS (Section 5.3). To validate the practicality of the locating method, we further evaluate the practicality of it on the official secure boot firmware of the STM32 platform, which is based on ST's closed-source binary cryptographic library X-CUBE-CRYPTOLIB [STMb] (Section 5.4). We also compare the results of the above experiment with the SEMI-LOC [TBW$^+$22], the existing state-of-the-art tool for automatically identifying COs without source code. SEMI-LOC relies on the continuous similarity between COs and requires knowledge of the width estimates of them in the trace. However, for binary-only cases, obtaining the width estimates itself is challenging except through our method described in Section 3. Finally, we comprehensively investigate its runtime performance and storage overhead using hardware measurements (Section 5.5).

5.1 Locating COs in long and noisy traces

Experiment settings. Our first experiment used tinyAES [kok] and the CoreMark benchmark suite [EEM09] on the nRF52840, an off-the-shelf Cortex-M4 chip with 32MHz operating frequency. The side-channel traces of the device are acquired by a Langer MFA EM probe [ET] and the sampling rate of the oscilloscope is set to 500MS/s. Each EM trace is approximately 32M samples long, while a call to the AES-128 encryption (including key expansion) for a single 16-Byte block takes ∼48K points. The CoreMark benchmark suite is extensively utilized to evaluate the efficiency of microcontrollers by utilizing a collection of typical workloads, such as matrix operations, state machines, linked list manipulations, and CRC checksum computations. In our experiments, we randomly insert an AES-128 encryption block into the test steps of CoreMark. The purpose of such a design is twofold: a) to increase the noise in the traces with representative code execution, thus simulating runtime interference in a real environment, and b) to investigate whether our tool can distinguish target COs from other computationally intensive operations like matrix multiplication.

Experiment steps. Specifically, the steps of our experiment are as follows:

1. Prepare the target firmware: at the source code level, randomly insert an AES-128 call with a static random key into CoreMark's test iterations, and add GPIO triggers marking at the beginning and end of AES function as ground truth.

2. Prepare the profiling firmware: Duplicate the target firmware and patch the AES key to null bytes as we do not know the real key. The binary code outputting the ground truth signals will be patched as a NOP operation. Then, use our tool to automatically analyze and instrument the profiling firmware.

3. Acquire the traces: The target and profiling binary firmware are flashed into separate nRF52840 development boards and side-channel traces are collected using an EM probe positioned identically for both. The ground truth signals are recorded with a second oscilloscope.

4. The collected traces are fed into our tool for template construction and matching. The matched results will be given as the similarity score of the CO template on the target trace.

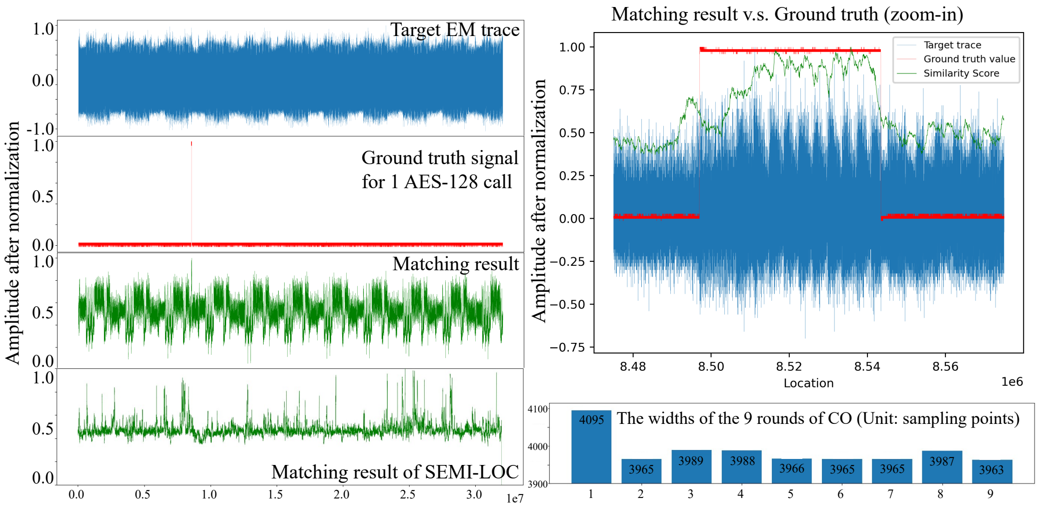


Figure 13: Find a CO in a long unknown target trace using 1 profiling trace and compare the result wtih the SEMI-LOC.

Results and analysis. Figure 13 presents the result obtained by matching the CO in a target trace using only one profiling trace. For a better presentation, this example includes only one AES-128 call, which constitutes about 0.15% of the entire side-channel, which is challenging to locate manually. Welch's t-test on the distributions of similarity scores inside and outside the ground truth region results in $p<0.0001$, demonstrating the significance of the matching results. It is worth noting that we do not perform any pre-processing on the profiling or target trace such as filtering or alignment, which means that our tool is robust to work in poor conditions.

Another noteworthy observation is that the normalized similarity result indicates other remarkable sections with a score greater than 0.8. Upon manual inspection, these sections are found to correspond to matrix operations. We hypothesize this result arises from the similarity between these operations and cryptographic algorithms which involve numerous arithmetic instructions. Nevertheless, our template-matching approach effectively differentiates the target CO from these operations.

To further validate the effectiveness of our work, we conducted a complete key recovery attack against the first round of the AES-128. As a baseline, we conduct the CPA attack directly on the raw long trace with basic correlation-based alignment. The results show that the AES key cannot be stably recovered even with more than 100,000 traces. In contrast, after reducing the alignment segment range to around 1.5 times the length of the target AES-128 encryption using our matching results, the same attack successfully recovered the 16-byte key with approximately 15,000 traces. As a comparison, under strict manual triggering alignment, the full key recovery requires $\sim$3,000 traces.

The bottom left corner of Figure 13 displays the results of matching the trace using SEMI-LOC after normalization to $[0, 1]$. We used 8 rounds instead of 9 in the setting to avoid significant width deviations introduced by the first round (width=4095), which could lead to incorrect results and unfair comparison. No other modifications were made to the open-source tool. The best width parameter is 3968, obtained by iterating from the smallest to the largest width with a step size of 5. Obviously, the result fails to accurately indicate the target CO and yields many false positives. We attribute this to clock jitter and other factors causing fluctuations in CO round execution times, which results in detection scores for real COs being similar to those of the normal loops in CoreMark. For instance, the lower right corner of Figure 13 shows the CO rounds with runtime variations larger than the working threshold (several cycles variations, e.g., ±2) for the SEMI-LOC tool. In contrast, our method utilizes the binary to precisely segment each CO, and therefore does not suffer from this problem.

5.2 Practicality in the presence of side-channel countermeasures

In this experiment, we investigate the performance of our approach in the presence of software side-channel countermeasure. Specifically, we choose the random delays countermeasure for its effectiveness in disrupting trace segmentation alignment, while masking or shuffling only complicates intra-CO analysis. It also invalidates the automatic CO locating method that searches for a sub-trace with periodic repeating signals in the side-channel trace. However, random delays are typically implemented as loops and thus can be detected and isolated from the template construction process in our approach.

```c
static unsigned char random_bytes[256] = {42};
unsigned char *RBP = random_bytes;
void refresh_rb(){
  for(int i=0;i<256;++i){
    random_bytes[i] ^= (
      1103515245*random_bytes[(i-1)%256] + 12345
    ) & 0x7fffffff;
  }
  RBP = random_bytes;
}
void inline random_delays(){
  for(int i=0;i<*RBP;i++){
    __asm__ volatile("nop");
  }
  RBP++;
}
```

(a) Random delay implementation

```c
void AES_init_ctx(struct AES_ctx* ctx, const uint8_t* key) {
  refresh_rb(); // generate new random bytes
  KeyExpansion(ctx->RoundKey, key);
}
static void Cipher(state_t* state, const uint8_t* RoundKey) {
  uint8_t round = 0;
  random_delays();
  AddRoundKey(0, state, RoundKey);
  for (round = 1; ; ++round) {
    random_delays(); // random delay
    SubBytes(state);
    ShiftRows(state);
    if (round == Nr)
      break;
    MixColumns(state);
    AddRoundKey(round, state, RoundKey);
  }
  AddRoundKey(Nr, state, RoundKey);
}
```

(b) Insert random delay between AES rounds

Figure 14: The evaluated implementation of random delay countermeasure in tinyAES library.

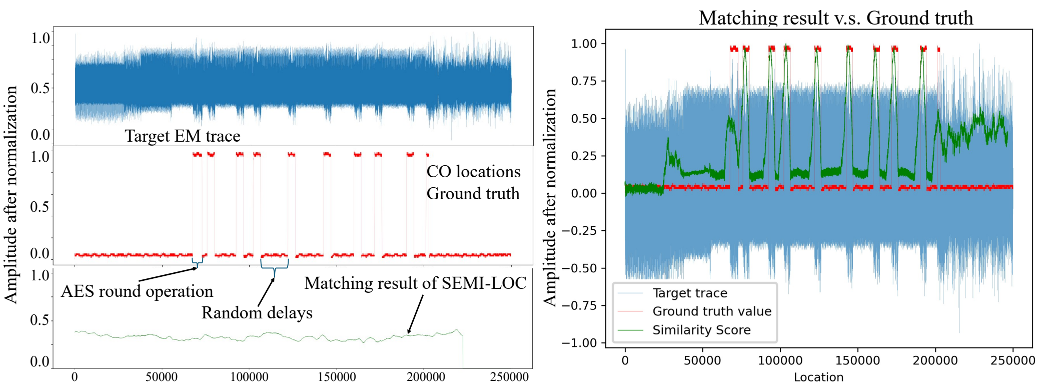


Figure 15: Find COs that are separated by the random delays.

Experiment settings. We observe that random delays are also commonly implemented as loops, and thus can be detected and isolated from the template construction process by our approach. As shown in Figure 14, we implemented a classical random delay scheme on the tinyAES library by inserting a random number of dummy instructions (`NOP`) between each round. The duration of the random delay is controlled by a pseudo-random number generator and will refresh each time the AES context is initiated (we only consider single-block AES-ECB encryption here). The hardware environment for trace acquisition remains the same as Experiment 5.1.

Experiment steps. The experimental steps are almost the same as Experiment 5.1, except that we insert ground truth triggers at the start (right after the **`random_delays`**) and end (after the **`AddRoundKey`**) of each AES round in the target firmware. Additionally, while generating profiling firmware, we made a small modification to the original code by temporarily adding the feature of random delay (a NOP-only loop) to the hooking targets. This ensures that our tool does not overlook it during heuristic filtering due to its small loop size. In the template construction, segments involved in the random delay loops will be eliminated, retaining only the CO parts. Note that since the above process involves knowledge of manual analysis of the random delay loop, we recognize that it is a semi-automated matching process here, treating it as a payoff for dealing with countermeasures.

Experiment analysis. Figure 15 demonstrates that our method effectively identifies the locations of most separated COs in traces with random delays. In contrast, SEMI-LOC struggles to efficiently identify matches due to its reliance on the continuous occurrence of COs. In fact, one advantage of our approach is that it can be combined seamlessly with manual binary-level analysis and patching, which is suitable for bypassing software countermeasures in the profiling phase.

5.3 Locating complex COs: A case study of ECDSA in MbedTLS

In our observation, few studies have considered automatically locating complex algorithms such as ECDSA in side-channel traces. A important reason is that the huge execution time of these algorithms makes their pattern easily identifiable even within long traces. However, a clear segmentation for these algorithms is still beneficial for SCA or fault injection attacks. Therefore, using the ECDSA P-256 implementation in MbedTLS as a case study, we investigate the performance of our scheme for locating its underlying CO: the scalar multiplication. Specifically, for P-256, MbedTLS eventually performs scalar multiplication using the **`ecp_mul_comb_core`** function. In a nutshell, it implements a optimized window-based (windows size $w = 5$ for P-256) scalar multiplication over Jacobian coordinates using a precomputed table, as shown in Figure 16.

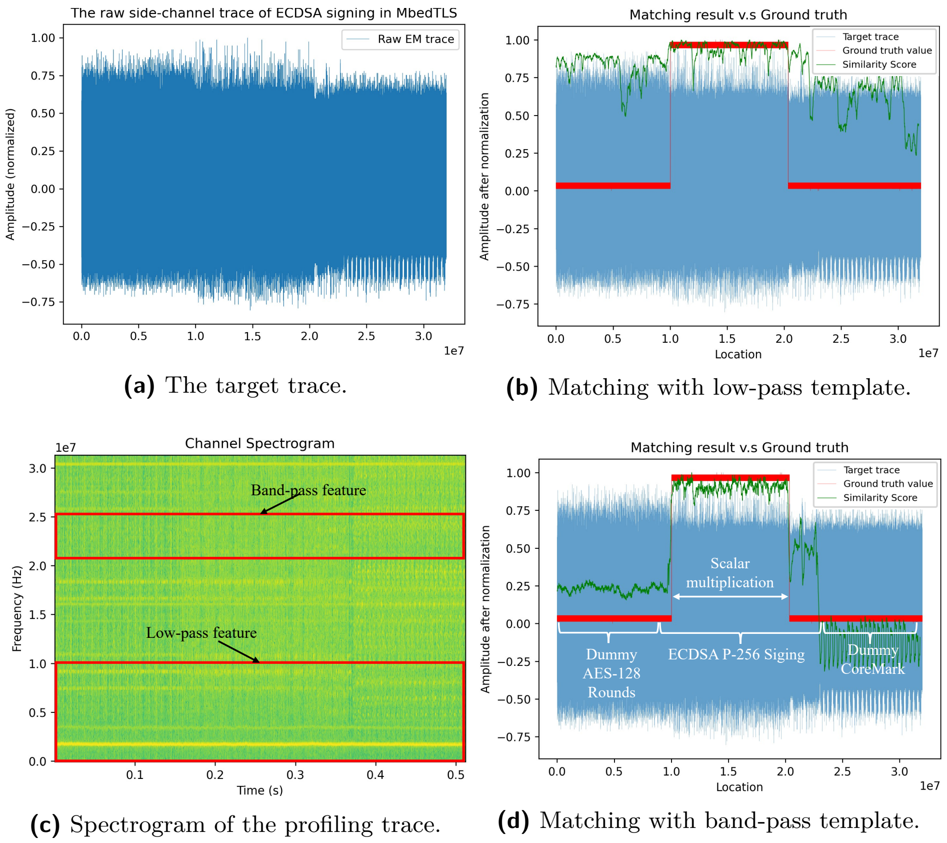
```
static int ecp_mul_comb_core(...) {
...
 while (i != 0) { // The CO loop
  MBEDTLS_ECP_BUDGET(MBEDTLS_ECP_OPS_DBL + MBEDTLS_ECP_OPS_ADD);
  --i;
  MBEDTLS_MPI_CHK(ecp_double_jac(grp, R, R, tmp));// double
  MBEDTLS_MPI_CHK(ecp_select_comb(grp, &Txi, T, T_size, x[i]));// table index
  MBEDTLS_MPI_CHK(ecp_add_mixed(grp, R, R, &Txi, tmp));// add
 }
...
}
```

Figure 16: Source code of the core CO for the ECDSA P-256 signing in MbedTLS.

Experiment settings. The hardware environment is the same as Experiment 1, except that the sampling rate of the oscilloscope is reduced to 31.25MS/s, allowing us to capture a much longer trace suitable for the ECDSA signing process. For the MbedTLS software, we utilize version 2.16.10 included in the official nRF SDK 17.1.0 for out-of-box development. However, we identified a performance issue in the MbedTLS wrapper for ECDSA signing in this latest nRF SDK version, where the wrapper always re-calculates the precomputed table unnecessarily. In the latest MbedTLS, precomputed tables for standard curves like P-256 are hard-coded in `library/ecp_curves.c`, eliminating this misuse problem. This bug not only invalidates the benefits of the optimized implementation but also makes the signing process too long to fit into the acceptable range of our laboratory oscilloscope. Fortunately, the issue can be easily resolved by integrating hard-coded precomputed tables during signature initialization, as done in the newer MbedTLS versions. Other than this fix and the insertion of ground truth triggers, we do not make any changes to the nRF SDK as well as the MbedTLS library.

To simulate the real operating environment, we add other cryptographic (dummy AES-128 operations) as well as non-cryptography operations (CoreMark benchmarking code) as noise around the ECDSA signing call (refer to Figure 17(d) for details).

Experiment steps. The experimental steps mirror those of Experiment 5.1, with the addition of ground truth triggers before and after the CO loops in Figure 16.



(a) The target trace.

(b) Matching with low-pass template.

(c) Spectrogram of the profiling trace.

(d) Matching with band-pass template.

Figure 17: The experiment result for locating the scalar multiplication CO within the target trace using a single profiling trace.

Results and analysis. Figure 17(a) presents a raw EM side-channel trace we collected from the target device, which contains not only the entire ECDSA P-256 signing process but also noise from the AES-128 and CoreMark code, making it difficult to clearly draw the boundary of the target CO. We first adopt the same frequency template matching strategy used in the previous experiment on AES CO, and the initial results are shown in Figure 17(b). However, it is hardly possible to locate the CO from the similarity score, especially for separating from AES-128 operations. To investigate we generated the spectrogram of a well-segmented trace captured on **`firmware #1`**, as shown in Figure

17(c). The spectrogram analysis reveals that the characteristics of scalar multiplication CO mainly appear in the high-frequency range. In the low-frequency domain, however, they are very similar to AES-128, making them difficult to distinguish. Therefore, we improve the method by focusing only on the bandpass-filtered ($2.1 \cdot 10^7 - 2.5 \cdot 10^7$Hz) frequency information in template construction and matching, and obtain the matching results in Figure 17(d). In the improved results, the scalar multiplication CO is well separated from other cryptographic or non-cryptographic operations, demonstrating the effectiveness of our method. Determining such starting/ending positions of scalar multiplication accelerates classical SCA on ECDSA, where the attacker typically recovers the highest or lowest bits of the nonce from the side-channel information near these positions. Moreover, it also enables the attacker to accurately set the trace acquisition timing and duration for the region of interest, further improve the overall analytical efficiency.

We also attempted to compare the performance of analyzing this side-channel trace with the SEMI-LOC tool. However, we directly encountered an Out-of-Memory error on our RTX3090 GPU (24GB memory). This issue likely arises from the excessive width of the ECC point double/add primitive, which is ~200,000 sample points long in the side-channel trace. The open-source implementation of SEMI-LOC is parallelized according to the points within the CO or with the whole trace, making it exceed the computing resource limits of available devices in our laboratory. In addition, the SEMI-LOC paper mentions that the time complexity of the method is $\mathcal{O}(w \cdot |t| \cdot r)$, where w is the CO's width, $|t|$ is the length of the whole trace and r is the rounds number of repeated COs. However, all of these parameters in the ECDSA signing are hundreds or thousands of times larger than those of AES evaluated in their experiments, further making this comparison infeasible.

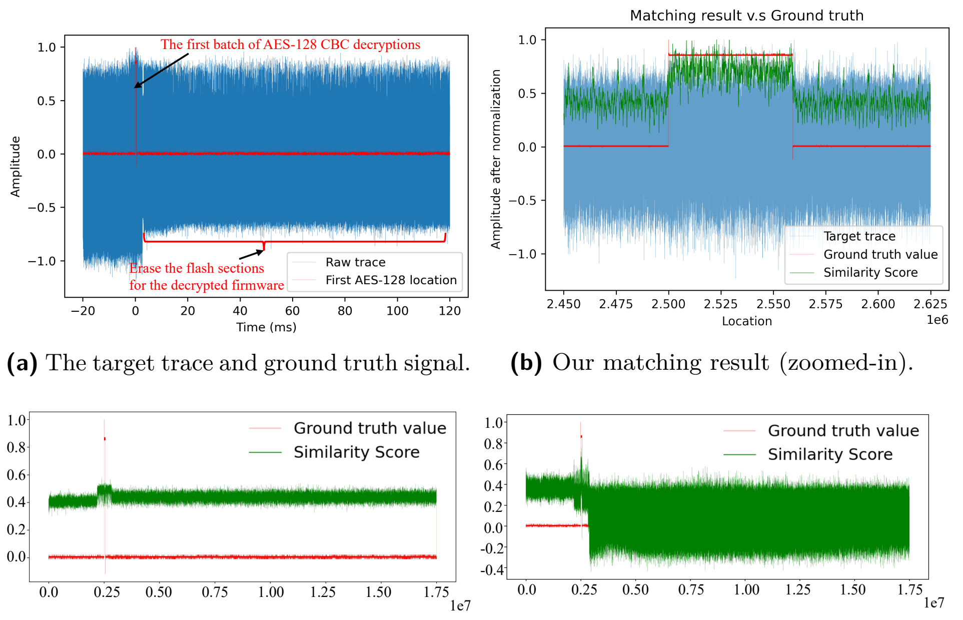
5.4 Assisting side-channel analysis of real-world cryptographic library without source code

Experiment settings. In this experiment, we examine the practicality of the proposed tool for real-world complex embedded cryptographic programs. We choose SBSFU [STMa], an official secure boot implementation of the STM32 platform for our analysis. It uses X-CUBE-CRYPTOLIB as the backend cryptographic engine, a closed-source cryptography library developed by STMicroelectronics. The hardware platform for our experiment is STM32F413H-DISCO, and we target the latest SBSFU (version 2.6.2) at the time of writing. The side-channel traces are obtained using the same setting as the previous experiment, with the exception of reducing the oscilloscope sampling rate to 125 MS/s, thereby enabling a longer sampling duration.
Experiment steps. We first analyze the behavior of SBSFU. With default settings, it provides two security applications:

- Secure boot: during a normal boot process, SBSFU checks the ECDSA signatures of the header and body of an application program stored in FLASH. If the verification succeeds, it will jump to the application for execution.
- Secure firmware update: in upgrade mode, SBSFU expects to receive a new application through the USB port using the Ymodem protocol [For86]. It will first verify the ECDSA signature and decrypt the application body using AES128-CBC with a pre-installed key.

Despite the availability of a newer release (4.1.0) of X-CUBE-CRYPTOLIB, SBSFU utilizes an outdated version (3.0.0). Our reverse engineering efforts revealed that the underlying AES implementation[6] in this version lacks side-channel protection. However, since the decryption is performed only after a long jittery USB communication, the location

[6] `AES_general_SW_enc` function in `libSTM32CryptographicV3.0.0_CM4_GCC_ot.a`



(a) The target trace and ground truth signal. **(b)** Our matching result (zoomed-in).

(c) Matching result of SEMI-LOC. **(d)** Matching result of our approach.

Figure 18: The experiment result for locating the first round of AES-128 in the upgrading process of the SBSFU using a single profiling trace.

on the side-channel trace is difficult to identify, as shown in Figure 18(a). For such a real-world challenge in side-channel analysis, our tool automatically locates the AES CO in it by following steps:

1. Prepare the target firmware: We use the official SBSFU demo that comes with the STM32F413H-DISCO board, the only change is to insert GPIO triggers before and after the call to the firmware decryption function[7] to annotate the ground truth.

2. Prepare the profiling firmware: We create firmware that links the X-CUBE-CRYPTO LIB library and invokes the `AES_CBC_Decrypt` function for performing dummy encryption.

3. Acquire the traces: For the target device, we developed a Python script to upload the encrypted firmware to SBSFU and notify the oscilloscope to acquire the side-channel signal. On the profiling device, we do exactly the same as in the previous experiment to obtain the annotated side-channel trace and build the CO templates.

4. Get the result: The collected target traces are fed into our tool for template-matching.

Results and analysis. In the experiment, we find that SBSFU decrypts in a 512-byte batch and initiates a FLASH erase operation upon completing the first batch to free up more space for subsequent decryption. Consequently, the side-channel traces exhibit segregation between the first and following AES blocks, interspersed with FLASH operations. Note that failing to find the first batch of AES encryptions could result in a wrong correlation to ciphertext and a complete failure in side-channel attacks. However, discovering it from long and noisy traces without binary-assisted analysis poses a challenging task, highlighting the importance of our study direction.

[7]The call to `SE_Decrypt_Append` in `sfu_fwimg_swap.c`

The matching result for the overall target trace is shown in Figure 18(c). The difference between the rest of the trace and the peak portion is not as significant as in the previous experiments. We attribute it to the fact that using basic spectrum correlation matching needs to be improved on noisy devices. Nevertheless, we still obtained recognizable results using only one trace from the profiling device. In contrast, the results given by SEMI-LOC do not precisely indicating the real location of the CO. Interestingly, the result for this real-world side-channel trace reveals a similar general trend for both schemes, indicating an inherent similarity in the two loop-oriented approaches.

5.5 Performance analysis

Experiment settings. In this section, we examine the performance of our implemented tool from the following two perspectives:

- Time consumption for CO extraction and instrumentation in binary: The runtime efficiency of performing binary analysis determines the complexity and scale of the firmware we can analyze, and therefore our scalability.
- Storage overhead of the modified firmware: The firmware will be larger than the original after instrumenting. Since embedded systems often exhibit storage constraints, the space overhead shapes our feasibility in the real world.

For the analysis efficiency, we evaluate our analysis tool against two industrial-level open-source cryptographic libraries, MbedTLS (version 3.4.0) and WolfSSL (version 5.6.4). Specifically, we choose four commonly employed cryptographic primitives (ECDSA, AES-128-CBC, AES-128-GCM, and SHA256) and created corresponding firmware. For each of the test firmware, we run our tool to find all possible CO loops in the binary, without specifying the target function list. All test firmware is compiled using Arm GCC 13.2.0 with the optimization level set to O2 and the target platform as Cortex-M4. The analyses are performed on a laptop computer with an AMD 4750U CPU and 32GB RAM.

To analyze storage space consumption trends in relation to instrumented loops count, we created a firmware that contains most of the cryptographic functions in MbedTLS 3.4.0, artificially limiting the number of COs that are instrumented and then recording the size of the patched binary. This benchmark firmware has an size of 379KB and runs on an nRF52840 development board with a FLASH size of 1MB. As a comparison, we also used the original PIFER tool to instrument at the same locations we extracted and checked the size of the generated binary.

Results and analysis.

Table 1: Processing time and binary size overhead for CO loops instrumentation.

Library	Index / Algorithm	CO loops instrumented	$T_{extract}$ (s)	$T_{instrument}$ (s)	Δ_{size} (KB)
MbedTLS	AES-128-CBC	10	21.305	2.711	0.63
	AES-128-GCM	33	24.173	2.789	1.79
	ECDSA	126	28.752	3.209	9.00
	SHA256	6	20.873	2.657	0.38
WolfSSL	AES-128-CBC	9	21.895	2.734	0.59
	AES-128-GCM	13	22.513	2.774	0.72
	ECDSA	74	28.109	3.115	3.98
	SHA256	7	21.068	2.765	0.47

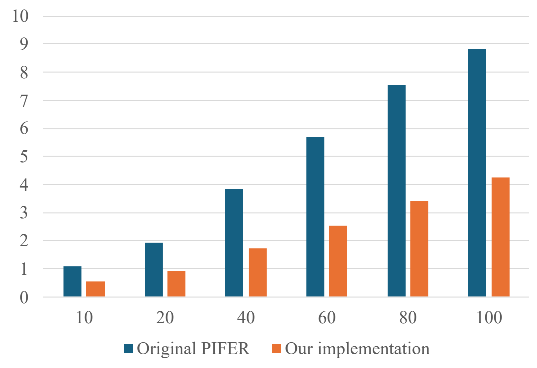


Figure 19: Storage space overhead from instrumenting different numbers of COs (KB).

Table 1 shows the results of the time overhead experiment for binary analysis of the firmware. From Table 1, we can observe that the extraction time $T_{extract}$ increases with more COs in the firmware. Overall, our tool can extract the COs corresponding to these algorithms within 30s and perform the instrumentation within a few seconds, demonstrating the capability of our tool to handle real-world cryptographic implementations. The growth in binary size after instrumentation Δ_{size} shows that our tool can handle real-world cryptographic primitives with storage overhead suitable for most Cortex-M platforms [STMc]. Figure 19 illustrates the storage overhead caused by the instrumentation, and our improved implementation reduces storage usage by approximately 50% compared to the original PIFER tool.

5.6 Conclusion of the evaluation

In general, the experimental results validate the effectiveness, flexibility, and robustness of our approach. Experimental results on long traces show that our tool is capable of reducing labor in locating and aligning COs, which are typically the initial steps in SCA. In the experiment on AES with random delay protection, our tool demonstrates the flexibility to address side-channel countermeasures collaborating with manual analysis. The ECDSA signing experiment indicates that our method can also be fine-tuned to locate significantly complex COs. Comparisons with SEMI-LOC illustrate that our approach performs better in the cases of jittery/noisy traces and CO discontinuities due to side-channel countermeasures. The experiment on the real-world closed-source cryptography library and the comprehensive performance analysis further verify the practicality of our implementation.

6 Discussion

Our proposed method allows the automatic localization of COs in the side channel traces of a target device under binary code-only conditions, considerably reducing manual efforts when performing SCA. However, as with many real-world tools, limitations exist. The following discussion aims to clarify its current scope of application as well as possible future directions.

6.1 Limitations

- **Code obfuscation:** Our approach aids side channel analysis mainly with the help of the information in the binary, but the code obfuscation is meant to hide it. Therefore it is ineffective against advanced obfuscation techniques(e.g., virtual machine protection [SBC+22]). Fortunately, such obfuscation is less common in resource-constrained embedded devices compared to x64 platforms.

- **The COs not implemented in loops:** Our approach currently does not address cases where CO is outside a loop in the binary code, such as loop unrolling for repeated symmetric cryptographic operations. Nevertheless, asymmetric COs are still typically in loop form, and embedded firmware projects often prefer not to unroll loops due to storage overhead.
- **Availability of the profiling device:** As in most side-channel analyses, we assume that a controllable device of the same model with similar side-channel characteristics is available. Consequently, our method is not applicable when such a device is not available at all or when there are significant differences between the two devices. However, performance analysis or reverse engineering assistance against the target binary firmware on a single controllable device is still feasible.

6.2 Future work

- **Handling of more general COs:** For the efficiency of the binary analysis and the space overhead of the instrumentation, we currently focus on COs that appear in loop form, which is common in memory-constrained embedded devices. Extending it for loop-unrolling COs requires two steps: 1) identify the starting and/or ending instruction of each unrolled round in the binary program as the boundaries, and 2) perform instrumentation at those non-jump instructions. Existing approaches [CFM12][XMW17][CSC+24] could assist with Step 1, though they may require dynamic analysis which is more challenging for embedded firmware. Step 2 is straightforward since PIFER already supports fine-grained instrumenting for almost all instructions, and one only need to restore the processing code that we removed for storage optimization. In summary, the main effort therein lies in pure software analysis, which we leave to future work to keep this paper focused on bridging the binary information to the side-channel traces.
- **Handling of read-only code.** In this paper we assist in side-channel analysis by injecting additional code into the binary firmware. However, read-only BootROMs are also frequently the subject of side-channel or fault injection analysis, which are not modifiable at all. Anyway, tracing and analyzing BootROM execution is inherently challenging. Future work may explore the possibility of utilizing the same hooking technique to relocate such read-only code to be executed at mutable memory.
- **Using side-channel information to supplement binary analysis:** In this paper we show how to use binary analysis to assist in side-channel analysis. However, the opposite direction is also an interesting issue. For example, we have seen cryptographic operations with unique side-channel information features, is it feasible to apply these features to enhance the identification of complex cryptographic functions (e.g., malware with obfuscation) in binary? The framework we propose in Section 3 that bridges binary code to side-channel traces makes it possible to answer this question, but that is beyond the scope of this paper.
- **Integration of more advanced template-matching:** As an open-source tool, we could integrate more advanced template-matching techniques into existing frameworks in the future.
- **Supporting more architectures:** Our implementation currently supports all ARM Cortex-M processors, the most widely used embedded platform. Extending support to architectures like RISC-V will require new binary instrumentation mechanisms,

posing an interesting yet challenging software engineering task. Nevertheless, the framework presented in this paper is theoretically architecture-independent, enabling effective migration to various embedded platforms.

7 Conclusion

In this paper, we focused on the challenges of side-channel analysis in real-world scenarios where only binary programs are available. We have introduced a novel framework that utilizes binary information to automate the process of locating COs in side-channel traces. Our solution involves mapping the execution flow of binary instructions onto the side-channel trace through static binary instrumentation and retrieving binary instruction addresses that correspond to the segmenting boundaries of the COs within the trace. By identifying the mapping points of these instructions, we can accurately label the side-channel data segments and locate the COs within traces collected from target devices.

We demonstrated the effectiveness of our proposed method on multiple different devices and widely used software cryptographic implementations. Experimental results indicated that we can accurately locate the COs in the trace collected from the target device using only binary information and a profiling device. The performance evaluation has confirmed that the runtime and storage overheads of the approach are practical for real-world applications. Our open-source release enables others to utilize and enhance this framework for more binary and side-channel co-analysis. Potential future work includes relocating read-only BootROM using similar instrumentation techniques and alternatively leveraging side-channel information to improve the detection of cryptographic functions within embedded binary programs.

Acknowledgements

This work is supported in part by the National Natural Science Foundation of China under Grant No.U2336210, No.62472286, and in part by the Startup Fund for Young Faculty at SJTU (SFYF at SJTU) under Grant No.24X010500123.

References

[Alt] Matthew Alt. Glitching in 3D Low Cost EMFI Attacks. https://voidstarsec.com/csw-2024/. (Accessed on 03/21/2024).

[ARMa] ARM MTB. https://developer.arm.com/documentation/100230/0004/jfa1432119346148. (Accessed on 03/21/2024).

[ARMb] Using DWT and other methods to count executed instructions on Cortex-M. https://developer.arm.com/documentation/ka001499/latest/. (Accessed on 06/27/2024).

[ARMc] ARM. An open source, portable, easy to use, readable and flexible TLS library. https://github.com/Mbed-TLS/mbedtls. (Accessed on 03/21/2024).

[Boo20] Jeremy Boone. There's A Hole In Your SoC: Glitching The MediaTek BootROM, 2020.

[BPS+20] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *Journal of Cryptographic Engineering*, 10(2):163–188, 2020.

[CFM12] Joan Calvet, José M Fernandez, and Jean-Yves Marion. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 169–182, 2012.

[CLC+22] Marco Casagrande, Eleonora Losiouk, Mauro Conti, Mathias Payer, and Daniele Antonioli. Breakmi: Reversing, exploiting and fixing xiaomi fitness tracking ecosystem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 330–366, 2022.

[CSC+24] Guoqiang Chen, Xiuwei Shang, Shaoyin Cheng, Yanming Zhang, Weiming Zhang, and Nenghai Yu. FoC: Figure out the Cryptographic Functions in Stripped Binaries with LLMs. *arXiv preprint arXiv:2403.18403*, 2024.

[CZLG21] Pei Cao, Chi Zhang, Xiangjun Lu, and Dawu Gu. Cross-device profiled side-channel attack with unsupervised domain adaptation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 27–56, 2021.

[DBMP23] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. ARMore: Pushing Love Back Into Binaries. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[EEM09] EEMBC. MCU Benchmark, CoreMark. https://www.eembc.org/coremark/, 2009.

[ET] Langer EMV-Technik. Langer mfa 01 set. https://www.langer-emv.de/en/product/. (Accessed on 03/21/2024).

[For86] Chuck Forsberg. Xmodem/Ymodem Protocol Reference. *http://www. commonsoftinc. com/Babylon_Cpp/Documentation/Res/KYModem. htm*, 1986.

[Ghia] Ghidra. https://ghidra-sre.org/. (Accessed on 04/20/2023).

[ghib] Ghidra documentation: Class dominator. https://ghidra.re/ghidra_docs/api/ghidra/util/graph/Dominator.html. (Accessed on 03/21/2024).

[ghic] Ghidra Headless Analyzer README. https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/RuntimeScripts/Common/support/analyzeHeadlessREADME.html. (Accessed on 05/24/2024).

[Gmb] SEGGER Microcontroller GmbH. J-trace streaming trace probes. https://www.segger.com/products/debug-probes/j-trace/. (Accessed on 03/21/2024).

[Gui] Ilfak Guilfanov. Findcrypt. https://hex-rays.com/blog/findcrypt/. (Accessed on 03/21/2024).

[Hér20] Olivier Hériveaux. Black-box laser fault injection on a secure memory. In *Symposium sur la sécurité des technologies de l'information et des communications-SSTIC 2020*, 2020.

[Hér22] Olivier Hériveaux. Triple exploit chain with laser fault injection on a secure element. In *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 9–17. IEEE, 2022.

[Inc] WolfSSL Inc. The wolfSSL library. https://github.com/wolfSSL/wolfssl/. (Accessed on 03/21/2024).

[JAH+24] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[JSO20] JSOF. 19 Zero-Day Vulnerabilities Amplified by the Supply Chain. https://www.jsof-tech.com/disclosures/ripple20/, June 2020.

[Key] KeystoneHQ. Keystone3: Best Open Source Cold Wallet and Hardware Wallet. https://github.com/KeystoneHQ/keystone3-firmware/releases/tag/1.3.4. (Accessed on 03/21/2024).

[Koc96] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[kok] kokke. Small portable AES128/192/256 in C. https://github.com/kokke/tiny-AES-c. (Accessed on 03/21/2024).

[LDMPT15] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. SoC it to EM: electromagnetic side-channel attacks on a complex system-on-chip. In *Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings 17*, pages 620–640. Springer, 2015.

[LGF15] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 203–214, 2015.

[LZC+21] Xiangjun Lu, Chi Zhang, Pei Cao, Dawu Gu, and Haining Lu. Pay attention to raw traces: A deep learning architecture for end-to-end profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 235–274, 2021.

[MMW21] Carlo Meijer, Veelasha Moonsamy, and Jos Wetzels. Where's crypto?: Automated identification and classification of proprietary cryptographic primitives in binary code. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 555–572, 2021.

[MRBT21] Damiano Melotti Maxime Rossi Bellom and Philippe Teuwen. Blackhat USA 2021: A Titan M Odyssey. https://i.blackhat.com/EU-21/Wednesday/EU-21-Rossi-Bellom-2021_A_Titan_M_Odyssey-wp.pdf, 2021.

[MWLGP12] Felix Matenaar, Andre Wichmann, Felix Leder, and Elmar Gerhards-Padilla. CIS: The crypto intelligence system for automatic detection and localization of cryptographic functions in current malware. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 46–53. IEEE, 2012.

[NS] Moritz Schloegel Nico Schiller. Unchained Skies: A Deep Dive into Reverse Engineering and Exploitation of Drones. https://mschloegel.me/slides/slides_recon23_drone_security.pdf. (Accessed on 03/21/2024).

[NSUH22] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno, and Naofumi Homma. Bypassing isolated execution on risc-v using side-channel-assisted fault-injection and its countermeasure. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 28–68, 2022.

[One] OnekeyHQ. Onekey hardware wallet firmware. https://github.com/OneKeyHQ/firmware/releases/tag/touch%2Fv4.7.0. (Accessed on 03/21/2024).

[QZZG23] Shipei Qu, Xiaolin Zhang, Chi Zhang, and Dawu Gu. Abusing processor exception for general binary instrumentation on bare-metal embedded devices. *arXiv preprint arXiv:2311.16532*, 2023.

[RLMI21] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A side journey to titan. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 231–248, 2021.

[SBC$^+$22] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3055–3073, 2022.

[Sem] Nordic Semiconductor. nRF52840 - Bluetooth 5.3 SoC. https://www.nordicsemi.com/products/nrf52840. (Accessed on 03/12/2024).

[SHC20] Majid Salehi, Danny Hughes, and Bruno Crispo. μSBS: Static binary sanitization of bare-metal embedded devices for fault observability. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 381–395. USENIX Association, 2020.

[STMa] STMicroelectronics. Secure boot and secure firmware update software expansion for STM32Cube. https://www.st.com/en/embedded-software/x-cube-sbsfu.html. (Accessed on 03/21/2024).

[STMb] STMicroelectronics. Stm32 cryptographic firmware library software expansion for stm32cube. https://www.st.com/en/embedded-software/x-cube-cryptolib.html. (Accessed on 03/21/2024).

[STMc] STMicroelectronics. STM32 family of 32-bit microcontroller. https://www.st.com/content/st_com/en/stm32-mcu-developer-zone/mcu-portfolio.html. (Accessed on 07/16/2024).

[TBW$^+$22] Jens Trautmann, Arthur Beckers, Lennert Wouters, Stefan Wildermann, Ingrid Verbauwhede, and Jürgen Teich. Semi-automatic locating of cryptographic operations in side-channel traces. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 345–366, 2022.

[Tre] Trezor firmware. https://github.com/trezor/data/tree/master/firmware/t2b1. (Accessed on 03/21/2024).

[VdHOGT21] Jan Van den Herrewegen, David Oswald, Flavio D Garcia, and Qais Temeiza. Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 56–81, 2021.

[WGP21] Lennert Wouters, Benedikt Gierlichs, and Bart Preneel. My other car is your car: compromising the Tesla Model X keyless entry system. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 149–172, 2021.

[WGP22] Lennert Wouters, Benedikt Gierlichs, and Bart Preneel. On the susceptibility of Texas Instruments SimpleLink platform microcontrollers to non-invasive physical attacks. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 143–163. Springer, 2022.

[WJC$^+$09] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Computer Security–ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings 14*, pages 200–215. Springer, 2009.

[Wou22] Lennert Wouters. Glitched on earth by humans: a black-box security evaluation of the SpaceX starlink user terminal. *DEF CON*, 2022.

[XMW17] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 921–937. IEEE, 2017.