Towards Cycle-Accurate Emulation of Cortex-M Code to Detect Timing Side Channels

Johannes Bauer
Department of Computer Science 1
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
joe.bauer@fau.de

Felix Freiling
Department of Computer Science 1
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
felix.freiling@fau.de

***Abstract*—Leakage of information through timing side channels is a problem for all sorts of computing machinery, but the impact of such channels is especially dramatic on embedded systems. The reason for this is that these environments allow attackers to exploit small timing differences down to clock cycle accuracy. On the defensive side it is therefore advisable to evaluate cautiously if security-critical code contains data dependent timing discrepancies. When working with real hardware, testing for such vulnerabilities is a tedious process. In order to reduce the burden of vetting, we study approaches that allow cycle-accurate behavioral emulation of relevant CPU behavior such as instruction pipeline flushes and bus contention. We show that our approach is feasible and efficient by implementing an emulator of the popular ARM Cortex-M core. Then we give an overview about the problems of cycle-accurate emulation and demonstrate our approach towards a cycle-accurate ARM Thumb-2 simulator. Finally, we show how this simulator can be integrated into the build process of firmware to check for the presence of timing side channels before the system is deployed.**

***Keywords*-timing side channel; cycle accurate; emulation; simulation**

I. Introduction

Security issues in real world systems do not only arise due to a flawed design, but also due to parasitic side effects which any computing machine exhibits. Differences in the power consumption, for example, lead to the presence of so-called *power emission side channels*. When electromagnetic emission of the hardware changes with the data used in a certain computation, we speak of *EM side channels*. The most intuitive class, however, are *timing side channels*. In these, some leakage is inadvertently generated by the fact that computations on secret data exhibit timing differences that depend on that data.

These timing side channels have been known since their first introduction by Lampson [1]. In modern systems they usually arise due to different optimizations within hardware or software. The reason why they are so prevalent on desktop computers, for example, is that desktop CPUs are using extraordinarily sophisticated techniques in order to aggressively optimize system performance. Such techniques include caching, instruction reordering or branch prediction. Unfortunately, these techniques give rise to data and code dependent timing behavior.

Traditionally, in embedded environments, these hardware optimization techniques were neither necessary nor were they particularly welcome: Since embedded systems and real-time computing often go hand in hand, predictability is of utmost importance to developers. Sophisticated mechanisms like branch prediction or caching were not prevalent at all. In more recent microcontroller architectures, however, these mechanisms slowly start to appear. In particular the popular ARM Cortex-M architecture, which has picked up significant momentum in recent years, shows features which previously only were present on highly sophisticated PC CPUs. One reason for their introduction is that internal MCU peripherals often cannot keep up with the high core clock speeds of modern microcontrollers. Without caching mechanisms, the slowest peripherals, such as flash ROM, would restrict the overall performance of the system significantly. With the introduction of these techniques, however, the same timing side channels which were also seen in PC environments before now also appear in microcontrollers.

One major difference, however, is the fact that in an embedded environment, timing side channels leak more information than in a PC environment: On a PC, an attacker usually needs to rely on imprecise measurements of time stamp counters and significant noise is present due to effects of the operating system. Such limitations are far less common in the embedded world. An attacker with physical access to a device is usually able to control the main clock source and has therefore the ability to slow down time to his liking. With today's mid-range hobbyist equipment, it is therefore possible to perform cycle-accurate timing measurements on such embedded systems. In many cases, the firmware runs directly on the *bare metal*, i.e. directly on the hardware with no operating system layer in between. Even if an embedded operating system is present, it will usually exhibit much more predictable timing characteristics than operating systems for the consumer market.

The motivation of such an attacker could be to gain access to an otherwise unavailable administration interface or extract information about the internal workings of such an device. For example, if an attacker would control a smart meter and extract the asymmetric private key, she could forge meter values and send arbitrary data to the utility company. Likewise, an attacker could find a master password for a manageable switch using timing attacks; although bad security practice, many vendors still ship their devices with such back doors in the hope that the protecting password remains secret.

A. Related Work

As with a lot of practical side channel work, Kocher [2] also pioneered the field of timing side channel analysis. He highlighted that timing differences in asymmetric cryptographic operations could lead to disclosure of private key data. Two years later, other implementations of his proposals emerged and were published by Dhem et al. [3], [4]. Around the same time, Kelsey et al. generalized on these side channels and showed their presence in popular algorithms such as IDEA, RC5 and DES [5]. Of particular interest to our work is the timing side channel they described in IDEA, where they exploited the timing difference present in a multiplication modulo $2^{16}+1$. In their example, a multiply-by-zero operation took significantly less time than a multiplication by a non-zero value.

Such side channels are common for algebraic field operations which are computed in software. At that time it was widely believed that an effective defense against this kind of timing attacks would be to use constant-time lookup tables for field operations. Page [6], however, showed how cache timing might cause these supposedly constant-time lookups to exhibit exploitable timing differences. Indeed, Tsunoo et al. [7] demonstrated the effects of leakage caused by caching effects to be relevant on real-world systems. Exploiting timing differences, they were able to break DES with a probability of over 90% with an astonishingly low amount of 2^{24} operations. Such attacks were later shown [8] to also apply to the more recent AES block cipher. A relatively well-known hack on the MSP430 mask ROM boot loader was presented by Goodspeed [9]: He exploited the timing differences of different control flow paths to find out the correct boot loader password of MSP430 devices.

Osvik et al. [10] proposed a general mitigation method for these problems based on different approaches such as normalization of cache timings, disabling of caches altogether or hiding mechanisms to obfuscate the leakage. Wang et al. [11] proposed hardware countermeasures, namely a new type of cache architecture, that claimed to solve side channel emission. However, Kong et al. [12] highlighted serious issues with this proposal, confirming yet again how difficult it really is to eliminate side channels in cached architectures.

Cycle-accurate simulation is a topic that is not only of interest for security research, but also for optimization purposes. Yourst [13] presented such a cycle-accurate simulator for the x86-64 architecture. Since the x86-64 is much more complicated than the Cortex-M architecture, their goal was to achieve 5% accuracy for all major simulated parameters; since the sophisticated optimizations of the x86-64 is not present in the Cortex-M architecture, the relative timing determinism of the Cortex-M enables much more accurate results in our case.

The Cortex-M uses internal SRAM for volatile storage, which is another contributing factor to the relatively straightforward implementation of an emulator. In contrast to DRAM, SRAM exhibits deterministic timing characteristics. To apply our results to DRAM, one could rely on complex simulations of DRAM timings, such as in the paper presented by Rosenfeld et al. [14]

If performance of the emulator is of interest – which for us was only a secondary objective – the paper of Ye et al. is also related. [15] They show which optimizations are applicable to a simulator while preserving its emulation accuracy. In their case they use it to improve the efficiency of a power consumption simulator.

Our approach relies on behavioral simulation of the architectural features. How processors can be modeled from a hardware perspective, for example during simulation of a synthesized FPGA, is covered by the work of Reshadi et al. [16].

B. Contributions

We present a practical approach to detect timing side channels in Cortex-M firmware. Our goal is to reliably detect such channels at implementation time. To do this, we built a behavioral Thumb-2 emulator with the specific purpose to exactly model timing behavior of the Cortex-M architecture down to clock cycle accuracy. We show how this emulator can help identify possible timing side channel leakage and how it can be incorporated into automatic vetting checks. To summarize, we make the following contributions:

- We describe our semi-automatic method of extracting hardware-dependent information about run time behavior from a real microcontroller and how to use this knowledge to create a cycle-accurate Cortex-M core emulator.
- We show how such an emulator can be integrated into the vetting process to prevent timing side channel leakage for embedded systems in an automated fashion.

C. Outline

This paper is structured as follows: Sect. II gives the necessary background to understand aspects of modern CPU design which are relevant for our topic. We then proceed to discuss the factors which influence run time behavior of our target platform in Sect. III and highlight important aspects in the design of a cycle accurate emulator. In particular we point out how real-world measurements on physical hardware can quickly be turned into a model for an emulator which we wrote. Our design is then evaluated in Sect. IV against real-world cryptographic algorithms and we show how integration of our work flow into a semi-automatic vetting process can look like. Afterwards, we discuss the results and give an outlook in Sect. V.

II. Background

In the following section, we give an overview about relevant concepts of modern CPUs in Sect. II-A. Then we

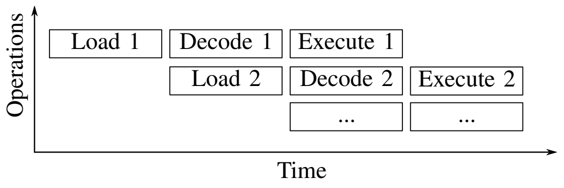


Figure 1. Pipelined instruction execution

proceed to show details about the particular CPU family (ARM Cortex-M) which we worked with in Sect. II-B.

A. Factors Influencing Execution Time in Modern CPUs

The internal construction of a modern CPU divides instruction execution into three phases: the fetch, decode and execution stages. At the first stage, one or more instructions are loaded from the position which the program counter points to. This is done via a read on the bus which fetches the data behind that address. In the next state, the instruction is decoded. This means the CPU evaluates which sub-components of the CPU need to be enabled to perform the action which is requested by the opcode. After this has been determined, the instruction is executed. In the process of execution there might again be access to a memory bus required, depending on the action that the opcode is supposed to perform.

Loading of instructions and decoding or execution can be parallelized. This is called *pipelining* and is shown in Fig. 1. While the CPU executes the first instruction, it can already – at the same clock cycle – concurrently fetch the next instruction. If the CPU has prefetched instructions and filled up the pipeline, but notices in the decoding stage that the prefetched instruction is not the next in line to be executed, the pipeline needs to be *flushed* and it needs to be refilled with the correct instructions. This is the case, for example, when a conditional jump takes control flow away from the instructions which the CPU had already prefetched.

There are also multiple factors which influence how long the execution stage of instructions takes. First and foremost is of course the actual computation that has been requested by the opcode itself. Some complex instructions need to be broken down by the CPU into smaller micro-instructions which are computed sequentially. For example, if there is a register indirect access with displacement, the CPU has first to compute the effective address and then execute the actual memory operation. The instruction is therefore broken into two parts: address calculation is performed on the *arithmetic logic unit* (ALU) after which comes the store operation. Both parts may or may not be pipelined, depending on the concrete architecture. The typical example of a complex instruction which takes a variable amount of clock cycles to execute is the integer division operation.

Another factor which influences run time is the dependency on a bus. Only one load or store can happen on a bus at any point in time. Access to the bus therefore has to be carefully coordinated. If multiple concurrent requests require bus access, *bus contention* occurs and the CPU must perform *arbitration* between the concurrent requests. The bus peripherals might also have some inherent latency associated with it. For example, typical external memory or internal flash ROM cannot serve data as fast as the internal CPU clock might require it for continuous operation. Therefore, the CPU has to wait some amount of time after the address has been put onto the bus before the data becomes valid. The time during which the CPU waits for a reaction from the bus is often measured by the number of successive *wait states* the CPU is in.

Lastly, *caching* is something that has major influence on the real world run time of a system. A cache miss is associated with the penalty to perform the actual read from the bus while access to cached data is typically faster by several orders of magnitude.

B. STM32 Cortex-M4 Specifics

The popular 32 bit Cortex-M architecture uses the ARM Thumb-2 instruction set. This is an instruction set in which opcodes are encoded either in a narrow 16-bit form or in a wide 32-bit form. The CPU core of the M4 uses an instruction pipeline which is 32 bits wide. Therefore, depending on the width of the instructions at the location of the program counter, either two narrow or one wide instruction is prefetched into the instruction pipeline [17]. Most instructions of the Cortex-M take either one or two clock cycles to execute, with the notable exception of the division unit which, depending on the processed data, takes anywhere in between 2 and 12 clock cycles for execution [18]. For operations which perform load/store actions, there is an additional penalty associated that is directly proportional to the amount of data that is to be loaded or stored.

While most System-on-Chips (SoCs) that target the embedded market go without any caches because it makes predictions about execution time much more difficult, the STM32 Cortex-M4 does have one instruction cache. This cache is referred to by STM as the adaptive real-time memory accelerator (ART). It caches access to the internal flash ROM memory which is unable to keep up with the core clock when the microcontroller unit (MCU) runs at high speeds. For example, at 3.3V the internal STM32F4 flash ROM can only provide zero wait state operation up to 30 MHz, but can require as many as 7 wait states at the low-voltage 1.8V operation when the CPU is clocked faster than 112 MHz [19], [17].

Like almost all architectures within the Cortex-M family, the M4 is based on a Harvard memory architecture. Concretely, this means that data and instructions are accessed via different buses. For normal operation, the text segment (i.e. where the executed instructions reside) is located within the flash ROM of the MCU and data is stored in the internal SRAM. Instructions are usually fetched via access to the instruction bus (I-Bus), but may also be fetched on the system bus (S-Bus), albeit less efficiently. Data access is performed on the data bus (D-Bus) or also via the S-Bus. The I-Bus and D-Bus can access only the lower 512 MiB of the 32 bit address space while the S-Bus can access almost all the remaining 3584 MiB [17], [18].

III. Cycle-Accurate Timing Simulation

We now focus on a concrete microcontroller and briefly describe the effects that make naive prediction of execution time difficult. We continue by showing how our emulator model is constituted and how it integrates into semi-automatic verification of code.

A. Execution Time Prediction

The standard attacker model for embedded systems places the system itself under full physical access of the attacker. This means an attacker is able to control the environment in which the microcontroller executes code. Part of the environment is a reference clock which is usually supplied externally in form of a quartz crystal. An attacker who has physical access to such a system can therefore modify the hardware itself (e.g., by changing this clock crystal) in order to force the system to slow down. This allows maximally precise, cycle accurate measurement with even mid-range commercial off-the-shelf hobbyist equipment. Any single clock cycle difference in timing can lead to an exploitable security vulnerability of such a system.

Consider the source code which is presented in Listing 1. It shows a `memcmp` function which differs from the standard `memcmp` in the way that no lazy abort is performed as soon as the first inequality is encountered between characters of the two supplied input buffers. While the overall result still is computed in a lazy fashion, the function always walks over the complete buffer in every case in an attempt to achieve constant execution time. This is something that a programmer who is aware of potential timing side channel leakage might do.

If you take a look at Listing 2 you will see how the GNU C compiler gcc 5.2.0 translated this code into ARM Thumb-2 assembly. You can see that the loop indeed covers all `len` bytes. However you might also notice that the compare-branch-if-not-zero instruction at `0x9b4` conditionally skips the following `subs` instruction if `result != 0`. This is the translated equivalent of the `if` condition. The code has therefore less work to do once `result != 0` and you might assume that it therefore executes a tiny bit faster whenever the `subs` is skipped.

To show the real-world effect of this code, we ran it on a STM32F407 microcontroller with variable input data. We then used the *embedded trace macro cell* (ETM) which we configured to monitor the executed cycle count using the CPU-internal *data watch point trigger* (DWT). The code to do this is shown in List. 3. Our results were checked for plausibility by connecting a Rigol DS2202 oscilloscope to the microcontroller. The code surrounding the profiling target generated a rising edge on a GPIO pin upon entry of the function and a falling edge on the return. We then triggered on the rising edge, but watched the *falling* edge of the signal with the infinite persistence function enabled. The different run times of the function can then clearly be seen on the oscilloscope.

When running the code from internal flash ROM, with a core clock of 8 MHz, the run time of a comparison in which the first characters differed was 196 cycles. For each byte at the head of the buffers which were equal, the routine became *faster* one clock cycle. Similar effects could be observed with code running from internal RAM: A comparison which differed at the first character took 279 clock cycles, but the routine also became faster for each correct heading character by *four* clock cycles.

Two aspects of this might seem odd and surprising: One is that the routine, no matter from where it is executed, becomes *faster* with an increasing count of equal heading characters. From the high level perspective, more work has to be done for each equal heading byte, so you might assume the routine to become *slower* for each match. Intriguingly, the opposite is the case. It also seems counterintuitive that the routine would run much faster from internal flash memory than when it runs from internal SRAM, since SRAM is typically much faster in terms of access times compared to flash ROM.

When taking a closer look at the architecture, however, both effects can be explained: That the routine becomes faster with each matching character stems from the fact that the compare-branch-if-not-zero instruction `cbnz` needs to skip the following `subs` by performing the conditional branch. The performance penalty which incurs with this taken branch is caused by the required instruction pipeline flush.

That flash ROM is actually faster than SRAM is also explainable: When instructions are loaded from internal flash ROM, the I-Bus is connected to the flash peripheral and the data access to RAM is performed via the S-Bus. As soon as both instructions and data come from RAM, however, the S-Bus has to be used for both: it is the only remaining bus that can access SRAM. This explains why performance decreases once instructions are served from SRAM. Bus contention and arbitration is leading to this degradation in performance.

Another effect that we would like to illustrate is the effect of wait states when retrieving data from flash ROM. Consider the code given in List. 4.

This code first XORs the first 10000 bytes of SRAM to get a steady baseline and then XORs n bytes of flash ROM on top of it. We executed that function and for each run determined the clock cycle differential in run time between the invocation with a byte count n and the subsequent invocation with byte count $n + 1$. Intuitively speaking, this is for every n the amount of clock cycles that the run additionally takes compared to the previous run. To not confuse issues, we ran our tests both with instruction and data caches enabled and later on again with all caches disabled. On actual hardware the timings we measured are shown in the plot in Fig. 2.

What can be seen easily is that the cache does have an effect on the total number of clock cycles, but it does not have an effect on the clock cycle differential. With all caches enabled, each new byte takes 10 more clock cycles except for the crossing of a 16-bytes boundary where this

```
int memcmp_cet(const uint8_t *a, const uint8_t *b, int len) {
    int result = 0;
    for (int i = 0; i < len; i++) {
        int char_result = a[i] - b[i];
        if (result == 0) result = char_result;
    }
    return result;
}
```

Listing 1. High-level memcmp routine which tries to achieve constant execution time

```
memcmp_cet:
 80009a4:  2300  movs   r3, #0                          ; r3 = 0 (i)
 80009a6:  b570  push   {r4, r5, r6, lr}
 80009a8:  4604  mov    r4, r0                          ; r4 = r0 = a
 80009aa:  4618  mov    r0, r3                          ; r0 = r3 = 0 (return value)
 80009ac:  4293  cmp    r3, r2                          ; if (i < len)
 80009ae:  da05  bge.n  80009bc <memcmp_cet+0x18>
 80009b0:  5ce6  ldrb   r6, [r4, r3]                    ; r6 = r4[r3] (a[i])
 80009b2:  5ccd  ldrb   r5, [r1, r3]                    ; r5 = r1[r3] (b[i])
 80009b4:  b900  cbnz   r0, 80009b8 <memcmp_cet+0x14>   ; if (r0 != 0) goto
 80009b6:  1b70  subs   r0, r6, r5                      ; r0 = r6 - r5 (return value)
 80009b8:  3301  adds   r3, #1                          ; r3 += 1 (i++)
 80009ba:  e7f7  b.n    80009ac <memcmp_cet+0x8>
 80009bc:  bd70  pop    {r4, r5, r6, pc}
```

Listing 2. Compiled memcmp routine in Thumb-2 assembly

```
ITM->LAR = 0xC5ACCE55;      // Instruction Trace Macrocell, unlock register access
COREDBG->DEMCR |= TRCENA;   // Debug Exception and Monitor Control, enable trace cell
DWT->CYCCNT = 0;            // Reset counter
DWT->CTRL |= CYCCNTENA;     // Set trace cell mode to cycle counting
```

Listing 3. Activation of cycle counting on the STM32F4

```
#define FLASH_ROM  ((volatile const uint8_t*)0x08000000)
#define RAM        ((volatile const uint8_t*)0x20000000)

int waitstate_test(int len) {
    uint8_t result = 0;
    for (int i = 0; i < 10000; i++) result ^= RAM[i];
    for (int i = 0; i < len; i++) result ^= FLASH_ROM[i];
    return result;
}
```

Listing 4. Code to demonstrate wait state influence

additional byte takes 15 clock cycles. For the case in which caches have been disabled, the same effect can be observed with the exception that the differential is usually 21 clock cycles and jumps to 26 clock cycles when crossing a 16-bytes boundary. The difference (5 clock cycles) is a direct consequence of the system's flash ROM wait states.

B. Architectural Modeling

In order to try to accurately predict timing, we developed a behavioral ARM core emulator. At the beginning we briefly looked into the option of modifying already existing emulation code (such as QEmu), but it soon became clear to us that most already existing code was written predominantly with performance in mind. Such code would likely have been difficult to turn into something that was usable for cycle-accurate simulation and we therefore developed our own emulator from the ground up in the C programming language.

In order to systematically determine execution time of code on hardware, we wrote platform evaluation code that allowed us to dynamically download code into the MCU's SRAM and have it execute with enabled ETM/DWT instrumentation. The result was a semi-automatic process in which the instructions which were of the greatest interest to us were evaluated in terms of their run time

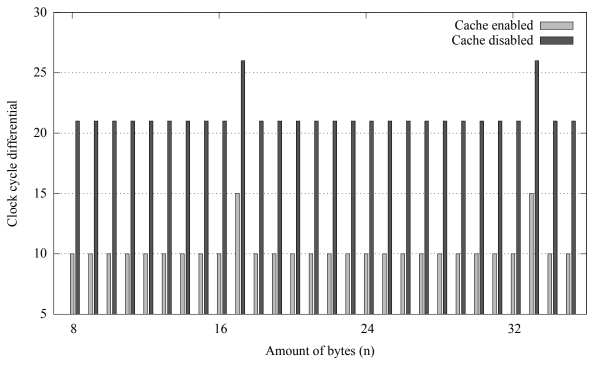


Figure 2. Pipelined instruction execution

performance. We omitted modeling of instructions which are not relevant for cryptographic purposes in general and for our use case in particular. Those were, among others, all floating-point unit (FPU) instructions the MCU offers. The run time information was collected by a host PC which was attached to our evaluation platform via RS232.

In our semi-automatic modeling process, the act of combining already evaluated instructions to form more complex code fragments was taken care of by a Python program. For all instructions which were not modeled, initially an execution time of zero was assumed – something that is deliberately wrong. We then randomly generated valid code snippets using a Python program which emitted increasingly complex sequences of instructions. These randomly generated code fragments were executed within the emulator and compared against the results returned by the actual hardware over the RS232 connection. In the first training stage it only emitted single instructions with no memory access and that could not fail (for example, division instructions weren't used in this stage since they can produce arithmetic errors when the dividend is zero). In a later stage, memory-transfer operations were added; even more complex were the snippets of code in which conditional branch instructions were probed. For the last group we had to give our code generator a coarse framework in which the branching instructions were to be embedded in order to avoid infinite looping or other undesired, undefined behavior.

Every produced code snipped was automatically generated, compiled and run locally by our emulator. As described, it was simultaneously downloaded on a STM32F4 and its execution was profiled on the real hardware. Whenever a discrepancy between the simulation and hardware arose, the code generator stripped the examples down to a minimal code fragment that still exhibited the issue. This allowed the developer in charge of modeling the device behavior to be able to exactly pinpoint erroneous instruction emulation and update the model accordingly. By this process we were able to achieve convergence towards an accurate model which simulated common cryptographic code (i.e. made heavy use of bit wise Boolean arithmetic such as AES or the SHA family) within just a few days.

The whole emulator is around 12000 lines of code

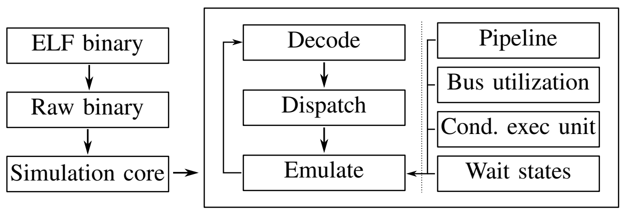


Figure 3. Model of our Thumb-2 emulator

(LOC). The greatest part of this, however, is taken up by instruction decoding (around 6000 LOC), while simulation core is around 1800 LOC long. Of the 233 non-FPU opcode variants, we emulate 114, i.e. around 49%. The instruction decoding code is generated using a self-written Python code generator from an XML architecture description. This description in turn was transcribed by us from the ARMv7-M Architecture Reference Manual [20].

A simplified model of the emulator architecture we developed is shown in Fig. 3. The code under test is compiled into an ELF binary first and the relevant data is extracted afterwards using standard tools into a binary that could be written into the flash ROM of a microcontroller. This binary file is then fed to the simulation core together with some metadata information. Such metadata could be a particular entry point or register configuration in order to suspend and later resume emulated execution. When a single instruction simulation step occurs, decoding and dispatching is performed by the simulation core to the particular unit responsible for that opcode variant. Since the decoding stub knows about particular data types and their respective encoding (e.g. immediate Thumb expansion or sign extension as explained in the ARMv7-M Reference Manual [20]) the handler functions can work on the already decoded data and do not need to care about specifics of their argument encoding.

That way, a particular decoded instruction is emulated while relying on a behavioral model of the architecture in the background. A global state is held which takes into account the aspects of the system previously described in Sect. III, such as memory wait states, bus utilization, pipeline fill and the state of the conditional execution unit of the CPU. All this impacts the actual run time of the operation within the execution and – depending on the performed operation – possibly updates the internal state again to reflect the changed state.

IV. Evaluation

In order to build up confidence in the accuracy of our emulator, we checked constantly during development that the model mimicked its physical counterpart closely. The way in which we used training code was explained previously in Sect. III-B. In this section we highlight the tests we conducted against sophisticated, real-world software in order to find out how closely our emulator is able to predict run time in these non-artificial pieces of software.

A. Experimental Setup

For our tests, we first created a monolithic ELF binary which contained all algorithms we wanted to test; these included public-domain variants of the AES and Camellia block ciphers as well as a public domain SHA256 implementation. We also evaluated the official Keccak-compact reference code in its version 3.2. Above those, some minor examples (`memcpy` and `sprintf`) were also added which used the embedded C library (newlib in our case). For all test code, small test stubs were written which invoked the respective functions. For example, the SHA256 testing function does the initialization of a SHA256 context, updates the context with argument-defined data length (we used the internal ROM as a data source in this case and just varied the length) and finalizes the context afterwards.

The main program then waits for commands on the RS232 interface, which was hooked up to a host PC. One offered command was to store received data into a program buffer, which was located in SRAM. Another command then invoked code execution of this program buffer and timed its execution time using the hardware-provided facilities described in Sect. III, particularly List. 3. This allowed us maximal flexibility because were able to execute arbitrary code and therefore have custom setup and tear down trampolines without having to re-flash the microcontroller every time. Instead, we compiled some code which called the functions we wanted to test on the host machine. For this process, we wrote another Python program which scanned to ELF binary for relevant entry symbols using `nm`, emitted Thumb-2 assembly code, compiled this code and sent it to the microcontroller's program buffer via RS232.

The whole process is illustrated in Fig. 4: At first, the monolithic ELF binary is compiled and flashed onto the microcontroller. From the ELF, we extract the address of a particular testing function which we would like to call using `nm`; in the shown case, this is the `test_sha` function at address `0x1234`. The trampoline stub is then generated which initializes the first parameter to `0x80` (i.e. hash 128 bytes using SHA256). Afterwards the register `r4` is set to `0x1235` (the least significant bit is always set, indicating to the processor that it is to use Thumb mode). Finally, a register indirect call is performed which jumps into flash ROM. This code is compiled on the host and its binary stream sent to STM32's SRAM program buffer. Execution of this program buffer now times the function (which is located in flash ROM) with the previously determined arguments.

By using empty testing functions stubs which merely returned, we could determine the amount of time spent in setup and tear down of our trampoline function in order to later subtract that amount of clock cycles from the measurements. This gave us the number of clock cycles which depended solely on the test function. All code snippets by default ran 512 times on the actual hardware. Our testing program ensured that all timing results were in agreement in order to avoid accidental mismeasurements.

B. Test Results

To verify the correct operation of our emulator, we performed tests using a variety of functions, many of which were of cryptographic nature. Among those were encrypting single blocks using the block ciphers AES128 and Camellia and hashing using the SHA256 and Keccak hash functions. Some other tests did non-cryptographic work; one example performs a call to `memcmp` and yet another issues a `sprintf` call which prints a format string of 24 bytes containing two integer substitutions (%d and %u).

The result of these tests is shown in Tab. II. Dynamic instruction count (i.e. the real number of instructions executed at run time) is shown as well as the amount of clock cycles on the real hardware and the predicted amount of clock cycles of the emulator. The host, an Intel Core i7-5930K, took a worst case of about 300 clock cycles to emulate one target clock cycle. There are, however, significant differences in speed depending on the type of code that is simulated. This can be explained mainly by code making use of the barrel shifter, something that's common in cryptographic computations and costly to emulate in software.

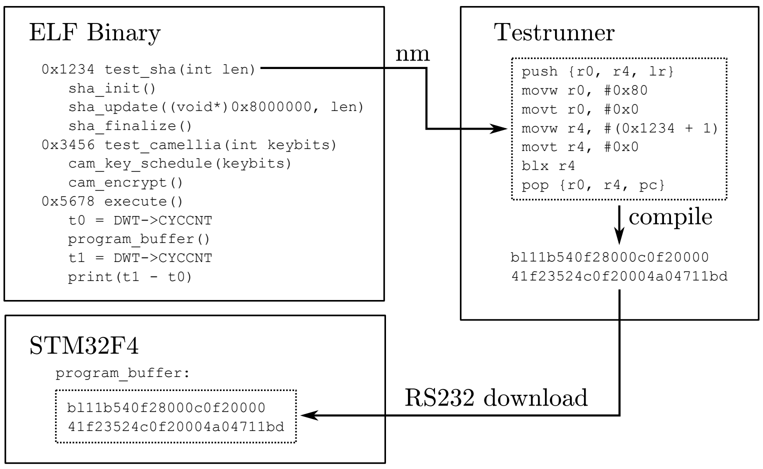
The amount of taken native clock cycles was estimated correctly by the emulator for most cryptographic algorithms; this is unsurprising as accurate modeling of cryptographic code was our primary objective and hence we put the most emphasis on that problem. Instructions which are seldom used in cryptographic code are not modeled with complete accuracy; in particular the divisions needed for the Keccak-call (`sdiv` opcodes) caused some slight discrepancies and similar issues arise at the `sprintf` example.

An estimate of how different the constitution of cryptographic and non-cryptographic code can be is given in Tab. III. In the AES128, SHA256 and `sprintf` examples we mentioned above we counted 108 unique opcodes that were executed. This includes different conditional variants of the same opcode after the occurrence of the Thumb-2 `it` "if-then" opcode as well as opcodes in their wide and narrow form. In order to do meaningful analysis with them, we grouped some of them into different categories as shown in Tab. I. It is immediately obvious that the `sprintf` example differs significantly and uses instructions and code which isn't required for cryptographic computations.

C. Semi-automatic vetting

In order to embed the emulator into a vetting process, we wrote a fuzzer using Python. A necessary prerequisite for functions which shall be tested is that they are runnable without any previous initialization right after a call to `main()`. If this isn't possible, the developer can additionally define initialization functions which will be called before executing the actual tests.

As can be seen in List. 5 the implementation of a fuzzing directive is done by implementing a method in a class. The test case generates two random byte arrays



```
0x1234 test_sha(int len)
    sha_init()
    sha_update((void*)0x8000000, len)
    sha_finalize()
0x3456 test_camellia(int keybits)
    cam_key_schedule(keybits)
    cam_encrypt()
0x5678 execute()
    t0 = DWT->CYCCNT
    program_buffer()
    t1 = DWT->CYCCNT
    print(t1 - t0)
```

```
push {r0, r4, lr}
movw r0, #0x80
movt r0, #0x0
movw r4, #(0x1234 + 1)
movt r4, #0x0
blx r4
pop {r0, r4, pc}
```

```
b111b540f28000c0f20000
41f23524c0f20004a04711bd
```

```
b111b540f28000c0f20000
41f23524c0f20004a04711bd
```

Figure 4. Work flow in the experimental setup

Group name	Description	Examples
`addsub`	Arithmetic addition and subtraction variants	`add`, `adc`, `sub`, `sbc`, `rsb`
`bcc`	Branch on condition code	any conditional branches as well as compare-then-branch instructions like the “compare branch if nonzero” instruction `cbnz`
`bitwise`	Bitwise operations	`and`, `bic`, `eor`, `orr`, `neg`
`ldr`	Family of load operations	`ldr`, `ldrb`, `ldrd`, `ldrsh`, `pop`, `ldmia`
`mov`	Family of move operations	`mov`, `movt`, `movs`, `movw`
`shift`	Bitwise shifting instructions	`lsl`, `lsr`
`str`	Family of load operations	`str`, `strb`, `strd`, `push`, `stmia`

Table I
USED INSTRUCTION GROUPING

Operation	Instructions	Native CCs	Predicted CCs	CC Diff.	Emulator CCs	CC ratio
`memcmp` 32 bytes	295	413	413	0	75 k	182
`memcmp` 100 bytes	907	1254	1254	0	227 k	181
16 bytes AES128 encrypt	8479	11634	11634	0	3.00 M	258
16 bytes Camellia-128 encrypt	1647	4469	4469	0	699 k	156
16 bytes Camellia-192 encrypt	2239	5962	5962	0	944 k	158
16 bytes Camellia-256 encrypt	2197	5892	5892	0	933 k	158
16 bytes hashing SHA-256	4091	5198	5198	0	1.53 M	294
32 bytes hashing SHA-256	4155	5299	5299	0	1.56 M	294
50 bytes hashing SHA-256	5227	5417	5417	0	1.59 M	294
32 bytes hashing Keccak-512	25866	40330	40326	4	8.19 M	203
64 bytes hashing Keccak-512	25938	40438	40430	8	8.22 M	203
256 bytes hashing Keccak-512	26370	41107	41075	32	8.37 M	204
`sprintf`	1088	2027	2010	17	367 k	181

Table II
MEASUREMENTS SHOWING THE DYNAMIC INSTRUCTION COUNT AND TAKEN CLOCK CYCLES (CCS)

Operation/Group	AES-128 (16)		SHA-256 (32)		sprintf	
addsub	1290	15.2%	999	24.1%	160	14.8%
`b`	144	1.7%	87	2.1%	28	2.6%
bcc	515	6.1%	227	5.5%	158	14.6%
bitwise	2116	25.0%	944	22.7%	15	1.4%
`clz`					1	0.1%
`cmp`	512	6.0%	218	5.2%	154	14.2%
`it`	298	3.5%			49	4.5%
ldr	1671	19.7%	579	13.9%	196	18.1%
mov	498	5.9%	833	20.1%	140	12.9%
shift	160	1.9%	43	1.0%	18	1.7%
str	535	6.3%	223	5.4%	152	14.0%
`tbh`					2	0.2%
`tst`	298	3.5%			1	0.1%
`umull`					8	0.7%
`uxtb`	442	5.2%				

Table III
DETAILED DYNAMIC INSTRUCTION BREAKDOWN OF TESTS WITH ASSEMBLY INSTRUCTION GROUPING

of 16 bytes each. A call to the `assert_crt` then actually triggers the verification. In this instance, `memcmp` is asserted to have constant run time independent of the content of the random byte arrays.

```
def tc_memcmp(self):
    array1 = self.randbytes(16)
    array2 = self.randbytes(16)
    self.assert_crt("memcmp",
        array1.addr, array2.addr, 16)
```

Listing 5. Python fuzzer test case

To come to a conclusion whether the assertion is true or false, the framework first takes the compiled ELF binary and simulates the code until `main()` is called. A snapshot of the memory and the CPU state is then generated as an optimization in order to be able to quickly switch back to this state later on for subsequent trials. The random byte arrays are generated at this point in time and mapped to a memory region which is otherwise unused by the hardware as to not interfere with the normal run time behavior. Then, `memcmp` is called with the appropriate memory addresses. The cycle count of the run time is recorded and the previous memory snapshot is restored. When the procedure is run again, it is verified that the second cycle count is equal to the first one. An arbitrary number of runs can be specified to get a reasonable amount of confidence that the probed function actually exhibits constant run time behavior. For the above example, a set of 1200 runs will ensure with a probability of at least 99% that the first byte in both random arrays was identical at least once. It would be equally possible to hard-code this, however, in Python, but we chose to keep the fuzzing example as simple as possible to illustrate our main point.

Integration of this Python fuzzer into the build process is trivial; in our case we simply added a `.PHONY` target called `check` into the `Makefile` which initiated the vetting procedure. If this target is defined as a dependency of, for example, the programming target, it can easily be assured that programming the microcontroller only then proceeds once the internal checks have passed (since the build process would abort otherwise).

If a discrepancy in the tested code arises, the user has the ability to re-run the emulator with tracing enabled. This gives a detailed breakdown at every executed instruction of where the cycle and program counter is at and allows to easily spot the sections which lead to unequal run time timing behavior.

V. Conclusion and Outlook

We have described theoretically and shown in practice that modern microcontrollers, such as those of the ARM Cortex-M family, exhibit timing phenomena which closely resemble behavior previously only seen on sophisticated desktop CPUs. In our explanations, it becomes clear how difficult it is to predict these effects to the naked eye. While the reasons for which these mechanisms have been incorporated into modern MCUs are beneficial for performance, they can lead to the presence of timing side channels. We explained how easy it is to exploit these tiny timing discrepancies by using mid-range commercial-off-the-shelf equipment. The described attacks are realistic for attackers with minimal hardware knowledge and physical access to the device.

To strengthen the defensive side, we have developed an emulator with the primary objective of emulating code with clock cycle accuracy. We have demonstrated the effectiveness of our approach and also describe how a streamlined process to improve the simulation model can look like. With the help of this emulator, it is possible for a software developer to regularly and semi-automatically probe code after each compilation in order to achieve continuous quality monitoring during the development process. It only has to be defined what results are expected from the function under test in order for the fuzzer to be able to do its work properly. If, for example by an upgrade of the compiler, the timing behavior changes in critical manners, this can be detected at a early stage within the development life cycle; timing analysis can then selectively performed on real hardware to confirm and pinpoint such irregularities.

Modern microcontrollers today are much more advanced than MCUs of previous generations. While this is a blessing on one hand, these quasi-magic performance boosters are a curse at the same time, since they are a major contributing factor to the presence of timing side channels. It is our hope that by accurately describing the effects present in these systems as well as releasing the complete source code of our project, our work contributes to both raising awareness and strengthening of mitigation strategies of timing side channels on embedded systems.

References

[1] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973. [Online]. Available: http://dl.acm.org/citation.cfm?id=362389

[2] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology—CRYPTO'96*. Springer, 1996, pp. 104–113.

[3] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," 1998.

[4] ——, "A practical implementation of the timing attack," in *Smart Card Research and Applications*. Springer, 2000, pp. 167–182.

[5] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," in *Computer Security—ESORICS '98*. Springer, 1998, pp. 97–110.

[6] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," *IACR Cryptology ePrint Archive*, vol. 2002, p. 169, 2002.

[7] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES implemented on computers with cache," in *Cryptographic Hardware and Embedded Systems—CHES 2003*. Springer, 2003, pp. 62–76.

[8] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[9] T. Goodspeed, "A side-channel timing attack of the MSP430 BSL," *Black Hat USA*, 2008.

[10] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology—CT-RSA 2006*. Springer, 2006, pp. 1–20.

[11] Z. Wang and R. B. Lee, *New cache designs for thwarting software cache-based side channel attacks*, ACM, 2007.

[12] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 2nd ACM workshop on Computer security architectures*. ACM, 2008, pp. 25–34.

[13] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, April 2007, pp. 23–34.

[14] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[15] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: A cycle-accurate energy estimation tool," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: ACM, 2000, pp. 340–345. [Online]. Available: https://doi.org/10.1145/337292.337436

[16] M. Reshadi and N. Dutt, "Generic pipelined processor modeling and high performance cycle-accurate simulator generation," in *Proceedings of the Conference on Design, Automation and Test in Europe—Volume 2*, ser. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 786–791. [Online]. Available: http://dx.doi.org/10.1109/DATE.2005.166

[17] ST Microelectronics, *RM0090 Reference manual STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx advanced ARM-based 32-bit MCUs*, September 2011.

[18] ARM Ltd., *Cortex-M4 Technical Reference Manual Revision r0 Part p0*, March 2010.

[19] ST Microelectronics, *PM0081 STM32F40xxx and STM32F41xxx Flash Programming Manual*, September 2011.

[20] ARM Ltd., *ARMv7-M Architecture Reference Manual DDI 0403E.b (ID 120114)*, December 2014.