Side-channel security of superscalar CPUs

Evaluating the Impact of Micro-architectural Features

Alessandro Barenghi
Politecnico di Milano – DEIB
Piazza Leonardo da Vinci, 32. 20133, Milan, Italy
alessandro.barenghi@polimi.it

Gerardo Pelosi
Politecnico di Milano – DEIB
Piazza Leonardo da Vinci, 32. 20133, Milan, Italy
gerardo.pelosi@polimi.it

ABSTRACT

Side-channel attacks are performed on increasingly complex targets, starting to threaten superscalar CPUs supporting a complete operating system. The difficulty of both assessing the vulnerability of a device to them, and validating the effectiveness of countermeasures is increasing as a consequence. In this work we prove that assessing the side-channel vulnerability of a software implementation running on a CPU should take into account the microarchitectural features of the CPU itself. We characterize the impact of microarchitectural features and prove the effectiveness of such an approach attacking a dual-core superscalar CPU.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Side-channel analysis and countermeasures**;

KEYWORDS

Side Channel Attacks, Superscalar microarchitecture

ACM Reference Format:
Alessandro Barenghi and Gerardo Pelosi. 2018. Side-channel security of superscalar CPUs: Evaluating the Impact of Micro-architectural Features. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3196112

1 INTRODUCTION

Side-channel Attacks (SCAs) are one of the most effective threats to breach the security of implementations of standardized, mathematically strong, cryptographic primitives. During the years, the target range for SCAs has steadily grown from smart-cards and dedicated accelerators, to micro-controllers [14], up to the point where single core CPUs have been shown to be vulnerable [8], and laptop grade CPUs have been shown to exhibit exploitable leakage, although with coarse grained models [13]. However, with an increasingly complex target, validating its vulnerability to SCAs as well as devising reliable and efficient countermeasures has also proven to be more and more difficult. In particular, considering software countermeasures, the theoretical assumption that no combination among given sets of intermediate values takes place, must be faithfully matched by the actual implementation [7]. Consequently, the current state-of-the-art in implementations assumes that countermeasures are realized preventing accidental value combinations such as the ones caused by register reuse, typically with careful assembly implementations. Whilst this approach serves well the purpose of smart-cards development environments, in the more recent Internet-of-Things (IoT) domain, (small) companies in need of either implementing or deploying side-channel secure cryptographic primitives, require tools and models able to provide a first assessment of the side-channel vulnerability of their software implementation running on a range of platforms. These platforms are likely to encompass scalar microcontrollers (e.g., ARM Cortex M0), superscalar microcontrollers (e.g., ARM Cortex M7), and full fledged dual core CPUs, depending on the application scenario [16]. To this end, providing a model of execution of the software from a side-channel leakage point of view, possibly in a form which can be integrated in static analysis tools as well as compilers, is a task which has gained interest also in the research community [16, 19]. Such tools and methods are expected to significantly reduce the delay of the feedback loop from security evaluation to development, integrating deeper security analysis as a part of the implementation process. For such tools to be developed, an understanding of how changes in the CPU microarchitecture map onto different side-channel leakage behaviors, is crucial. Indeed, while ensuring the correct execution requires only that the object code matches the underlying CPU at Instruction Set Architecture (ISA) level, we maintain that establishing the extent of the side-channel leakage of an implementation depends on both the ISA and the microarchitectural features of the underlying processor. While their impact on scalar μCs is limited, not taking into account the microarchitectural structure of the CPU executing the code may invalidate (partially or fully) the effectiveness of the countermeasures employed [10]. A critical case where this aspect comes into play is providing *portable* side-channel security, i.e., guaranteeing that a software side-channel resistant library preserves both its functional properties, and its side-channel security when executed on different, ISA-compliant, processors.

Contributions. In this paper, we analyze how the execution of a software primitive is performed by a superscalar CPU to highlight the importance of the microarchitectural features in pinpointing the sources of information leakage. In particular, we show how the sharing of internal pipeline buffers during the device activity is responsible for a critical information leakage, which cannot be recognized from an assembly representation of the program as it is not stemming from data dependencies among instructions. We also highlight how apparently innocuous changes to the register allocation in a secure implementation may lead to a practical vulnerability. As a case study, we consider the superscalar, partial-dual issue, 8 stages pipeline core CPU of an ARM Cortex-A7, providing a characterization of its information leakage sources. Moreover, we describe how to use the timing side-channel information stemming from the Clock cycles Per Instruction (CPI) index on a set of micro-benchmarks to infer a likely structure of the pipeline, a method

DAC '18, June 24–29, 2018, San Francisco, CA, USA
2018. ISBN 978-1-4503-5700-5/18/06.
https://doi.org/10.1145/3195970.3196112

which may be of independent use from the purpose of this work. We validate the effectiveness of our model via precisely justifying the success of an attack against an implementation of the AES block cipher, running both on the bare metal, and on a Linux distribution.

2 BACKGROUND AND RELATED WORK

SCAs exploit the link between the data being processed by a digital computing platform and one (or more) environmental parameters of the platform itself, such as its power consumption, Electro-Magnetic (EM) radiations or computing time. A differential power/EM attack works as follows [14]. Given the measurements, the attacker tries to deduce the correct value of the secret key employed by the cryptographic implementation, modeling the power consumption induced by a small portion of the computation, which in turn depends only on a small portion of the secret key, and comparing models with the actual measurements. As a result of the said comparison the attacker finds out the model which best fit the measurements, and consequently the actual value of the involved secret key bits. Since the side-channel measurements are affected by both random and systematic noise, the goodness of fit of a model to the actual device behavior is performed with statistical tests.

One of the points touted as an advantage of SCAs is that the power consumption model of the computation may be quite unaware of hardware/software stack details [2–4, 14] (e.g., a common model is the Hamming weight of an intermediate value of the algorithm). Indeed, such effectiveness was proven by successful attacks to complex computing platforms such as an unprotected software AES implementation running on a superscalar single core CPU clocked at 1 GHz and with full-fledged Linux OS [8]. In particular, in [8] the authors show how the high working frequency of the target device and the presence of an operating system with a preemptive scheduler offer hindrances which can be overcome by an attacker, while considering as a consumption model the Hamming weight of the outputs of the algorithm instructions. Similarly, exploiting the private-key dependent control flow of the code at hand, the authors of [13] extract the private-key of a digital signature algorithm running on both a mobile phone and a laptop. We note that the control flow dependent leakage can be suppressed more easily w.r.t. the data dependent leakage exploited in [8].

From a designer standpoint, countermeasures to SCAs can also be applied keeping into account a coarse leakage model [10, 11]. This is usually done since micro-architectural level specification of CPUs may not be freely available or known. However, two works sharing our research direction [18, 19] show unexpected leakage behaviors from software block ciphers implemented on an a AVR 8-bit microcontroller explainable only with a more detailed execution model. In [18, 19], the authors prove that the computing platform reveals information on the Hamming distance between the contents of two independent destination registers of two subsequent instructions due to register file write-port sharing. This leakage (from an ISA point-of-view) breaks provably secure countermeasure schemes implemented with state of the art assembly code [18].

We differentiate from the aforementioned ones in tackling a superscalar CPU architecture in our characterization, highlighting how its execution model, while semantically equivalent to a scalar one, inserts a non negligible amount of potential pitfalls for the sound realization of side-channel countermeasures. The results of our work can be fruitfully integrated into both static analysis tools, such as [16], and countermeasure checking tools, such as [11], to enhance their effectiveness. Striving toward the goal of providing portable side-channel security, our findings are also amenable to be integrated in compiler based approaches, such as [1]. Indeed, to provide a protected code emission matching the microarchitectural leakage model, constraints in the register allocation and the instruction scheduling backend passes can be added.

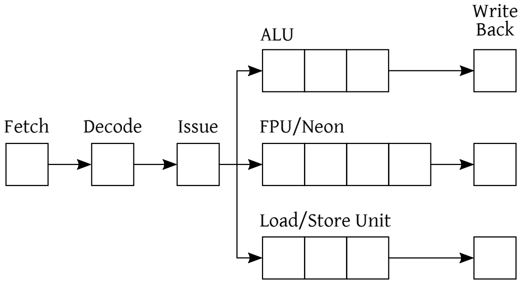

Figure 1: ARM Cortex A7 MPcore pipeline logical view [6]

3 MICROARCHITECTURAL EXPLORATION

In this section, we present the method employed to determine details of the Cortex-A7 CPU microarchitecture. This investigation is not required if a Hardware Description Language specification of the CPU is available, but that may not be the case due to its cost.

3.1 Logic-level Description of the Cortex-A7

The ARM Cortex-A7 MPCore [6] is a dual-core, partial dual-issue, in-order CPU, with an 8 stages pipeline. The reference manual reports a logic level view of the CPU pipeline as depicted in Figure 1, where each block in the picture represents a stage of the pipeline taking one cycle to execute. The CPU is described to be *partial dual-issue*, specifying that not all pairs of instructions are amenable to be executed simultaneously. We obtain a more detailed, description of which instructions are dual-issued through examining the *machine description* of the ARM Cortex-A7MPCore in the GCC backend [12], directly contributed by ARM. While this description adds more details to the outlook of the reference manual, it does not fully detail which dual-issue policy is implemented in the cores. Indeed, to understand the effects of the microarchitecture on the side-channel leakage we require the knowledge of which pipeline resources are shared by a pair of consecutive instructions, both in case of a dual-issue and a single issue execution. In particular, we need to determine: *i)* the number of ALUs present, and if they are all able to compute the same instructions; *ii)* whether or not units performing multiple cycle instructions (`mul`, `ldr`, `str` and `shift` according to [6]) are pipelined as Figure 1 would suggest; *iii)* how many data buses connect the pipeline stages both in paths going out of the Register File (RF) component, into the Decode (D) stage, into the EXecution/MEmory (EX/MEM) stage, and from the EX/MEM stage back into the RF.

3.2 Microarchitecture Characterization via CPI

To infer the microarchitectural details of the Cortex-A7 MPCore we employ the information leaked by the Clock cycles Per Instruction (CPI) achieved on a given instruction sequence. We compare the CPI indexes of different instruction sequences sharing the same

Table 1: Instruction pairs executed in dual-issue by the Cortex-A7 MPCore CPU. ALU indicates the set of arithmetic/logic operations save for the `mul`

	mov	ALU	ALU w/ imm	mul	shifts	branch	ld/st
mov	✓	✓	✓	✗	✓	✓	✗
ALU	✓	✗	✓	✗	✗	✓	✗
ALU w/ imm	✓	✓	✓	✗	✓	✓	✓
branch	✓	✓	✓	✓	✓	✗	✓
ld/st	✓	✗	✓	✗	✗	✓	✗
mul	✗	✗	✗	✗	✗	✓	✗
shifts	✗	✗	✓	✗	✗	✓	✗

opcodes, but differentiated by artificially induced Read-After-Write (RAW) hazards. Hazard free instruction sequences are issued at the best of the CPU capabilities, while hazard affected ones will not be dual-issued, allowing us to distinguish if dual-issuing takes place. To the best of our knowledge, this is the first time CPI data are employed to deduce the microarchitecture of a CPU.

We selected the Allwinner A20 System-on-Chip (SoC) [5], based on the Cortex-A7 MPcore as the target implementation of the CPU for our characterization. The Allwinner A20 was mounted on a commercial Olimex A20-OLinuXino-MICRO development board [17] which exposes a set of GPIOs connected to the SoC. Our microbenchmark for an instruction sequence was constituted of 200 repetitions of an instruction pair, preceded and followed by 100 `nop`s to fully flush the pipeline state. The benchmarks were written in assembly and ran directly on the bare-metal, started by the U-BOOT bootloader. For ease of measurement, the clock of the CPUs in the SoC was locked to 120 MHz setting the appropriate PLLs, and all the unused peripherals were clock gated. The time required for the benchmark execution was measured with a Picoscope 5203 sampling at 500 Msamples/s a GPIO asserted at the beginning of the computation and deasserted at the end. We derived the number of clock cycles elapsed between GPIO assertion and deassertion, subtracting the time required to execute 200 `nop`s and the GPIO toggling from the measure, and dividing by the clock period. This allowed us to determine directly the CPI of a given instruction pair dividing the aforementioned number of clock cycles by the number of non-`nop` benchmark instructions. The standard deviation in timing measurements was within the maximum precision allowed by our oscilloscope (i.e., ±2ns). Since the Allwinner-A20 is endowed with two level of caches, which may adversely affect deterministic execution, we iterated in an infinite loop the benchmark patterns so to warm them up and we measured the average execution time over 30 runs. This allowed us to exploit the caches to ensure a steady supply of data and instructions to the CPU, preventing unwanted stalls. We confirmed that a `mov` instruction sequence without hazards was running with CPI 0.5 indicating full dual-issuing.

The results of our CPI analysis are summarized in Table 1, showing which instruction pairs were dual-issued by the Cortex-A7. Concerning the investigation on the number of ALUs (as listed at point *i)* in Section 3.1), we deduce that two ALUs are present from the fact that two arithmetic/logic instructions can be dual-issued (provided one has an immediate operand), although the ALUs are not identical. Moreover, only one of the ALUs is endowed with a *barrel shifter* on one of the operands (employed to perform `shift-rotate` instructions in the ARM ISA) plus a multiplication unit: this statement can be deduced from the fact that `shifts` and `muls` are never dual-issued with a computational instruction.

Concerning the pipelined implementation of the units (point *ii)* in Section 3.1), we report that the CPI of a sequence of either `load` or `store` instructions (free from data hazards) resulted in a sustained CPI of 1, indicating that the Load Store Unit (LSU) of the Cortex-A7 is fully pipelined. The fact that the `load` instructions are dual-issued (CPI 0.5) with arithmetic instructions employing an immediate value, is coherent with the fact that address generation is performed in the Issue Stage (IS) as reported in [12] and thus does not clobber any ALU. We also observed that the multiplier in the ALU is fully pipelined, as a sequence of `muls` achieves CPI 1.

Finally, concerning the data buses structure (point *iii)* in Section 3.1), we are able to deduce from Table 1 that three data buses are present from the RF to the EX stage, as two arithmetic/logic instructions are dual-issued only when one of the two employs an immediate operand, but they are not whenever both require two operands from the RF. Since a sustained CPI of 0.5 is achieved when dual-issuing takes place, we assume that two buses connecting the output of the EX stage back to the RF are present, and that the RF is thus endowed with two write- and three read-ports. As a final note, we report that, albeit counter-intuitively, `nop` instructions are not dual-issued by Cortex-A7. The findings on microarchitecture of the Cortex-A7 allows to redraw the pipeline as in Figure 2, where the Fetch Unit is likely fetching two instructions per clock cycle, as it sustains the best-case CPI of 0.5 which was observed in practice.

4 SCA CHARACTERIZATION

To characterize the microarchitecture-driven leakage we consider that gates driving large (capacitive) loads are likely to be the main source of side channel leakage [15], with a power consumption which is well modeled by the Hamming distance of two values asserted on their outputs in subsequent clock cycles. In this scenario, employing Pearson's correlation coefficient between the measured power consumption and the predicted one was proven to be a statistically sound side channel distinguisher [9].

We analyzed our test case CPU modeling the possible leakages stemming from: the read-ports of the RF, the inter-stage buffer between the Issue and the EXecution stage (IS/EX), the inter-stage buffer between the EXecution stage and the Write-Back stage (EX/WB), the ALU output buffers, the barrel shifter output buffers, and the Memory Data Register (MDR) output. Moreover, we consider the possibility for the LSU to have an internal buffer where sub-word realignment is made (to support `load` instructions accessing a half-word or a single byte). Our measurement setup relies on the same Picoscope 5203 employed to perform the microarchitectural exploration (Section 3), and a custom loop probe made out of a 50Ω coax cable, exposing 2 mm of the coax core. The probe is connected to the oscilloscope via two Agilent INA-10386 amplifiers, and is placed close to the C62 ceramic decoupling capacitor placed on the supply line of the SoC. We devised a set of seven micro-benchmarks for the side channel leakage of the Cortex-A7, constituted of small (2 to 4) instruction sequences (see Table 2) and ran them employing randomly generated values at each execution asserting and deasserting a GPIO before and after to provide a trigger to the oscilloscope. We took care of eliminating the effect of the cache measuring the executions following the first one and avoided the possible boundary effects with the trigger handling instruction through inserting 100

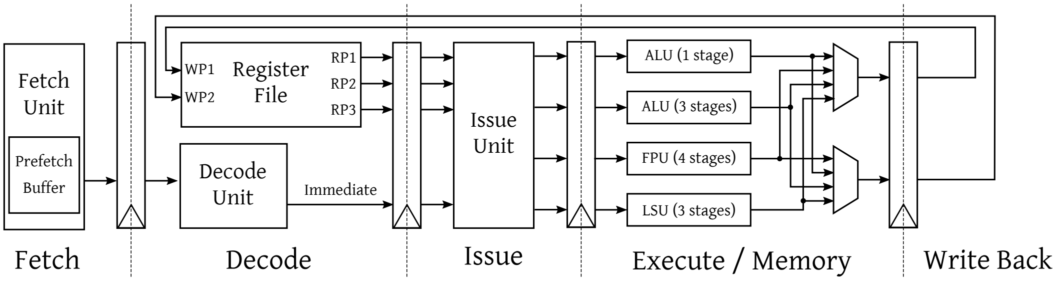

Figure 2: Alleged ARM Cortex A7 pipeline structure according to the deductions possible via CPI analysis

nops before and after the benchmark sequence. We recorded 100k power traces for each benchmark, each one obtained as the average of 16 executions of the benchmark with the same input. We considered a leakage to be present whenever its power model reported, in the correct clock cycle, a correlation distinguishable from zero with a statistical confidence >99.5%. To evaluate separately the leakage from the RF and the remaining pipeline stages, we pre-charged the destination register with the expected results of the computational instructions before each benchmark run.

4.1 Leakage Characterization

Table 2 reports the instruction sequences employed in each benchmark and the power models employed to detect the leakage of each component. The models for all the components but the RF were derived from switching activity of the component starting from the signal values taken when the benchmark is run. For the RF, we considered the leakage of reading the operands of the instruction onto the IS stage, as the power leakage arising from asserting the signals onto the write-port was ascribed by us to the EX/WB buffer output gates – following the common practice employed in EDA tools of ascribing the power consumption of a signal to its driving circuit. In Table 2 the models for which we detected a leakage are reported in red, while the ones for which we detected no significant value of the Pearson's correlation coefficient are reported in black.

Register File. We were able to determine that the RF does not appear to have statistically significant leakage with the considered number of traces and models (see 3rd column of Table 2). We ascribe this behavior to a short (capacitive) load on its read-ports, due to the IS buffers being the actual ones driving the execution units.

IS/EX Buffers. The outputs of the IS/EX buffers present leakage (see 4th column of Table 2), well modeled by the Hamming distance between the values of a source operand of an older instruction and the one of a younger instruction when single issued. Only Hamming distances between operands in the same source operand position (i.e., both first or both second operands) show significant leakage, since they share of the same bus. Moreover, interleaving two movs with a nop, forces them to be issued on the same ALU shows both the expected leakage due to the transition between the two operand values, and a leakage depending on the Hamming weight of the said operands. We infer the cause to be the implementation of the nop as a conditional instruction (set never to execute) with zero-valued operands. Such an implementation is also consistent with other leakage behaviors, in the EX/WB buffers. Concerning dual-issued arithmetic instructions (3rd row in Table 2) we note that no measurable leakage is present among their source operands. This is reasonable as the said operands are not sharing any resources before the result of the instructions are computed.

ALU and Shift Buffer. Concerning the leakage from the ALUs (see 5th and 6th columns of Table 2), we report a leakage dependent on the Hamming weight of the instruction result. We ascribe it to the way the ALUs were synthesized, which may assert the result on a previously zero-precharged set of signals. We confirm the presence of a leakage stemming from the buffer storing the result of the barrel shifter before it is fed into the ALU. A leakage proportional to the Hamming weight of the shifted value is present, albeit comparatively small: its absolute value in correlation is about $\frac{1}{10}$ of the average value for the other leakages.

EX/WB buffers. The information leakage of the EX/WB buffers mirrors the one of the IS/EX buffers, although it is related to the Hamming distance between the results of subsequent instructions whenever they are single issued (see 7th column of Table 2). We recall that such a leakage takes place regardless of the fact that the two instructions are sharing the destination register, and, more in general, that the two destination values are in any way related in the program data-flow. Moreover, we also report that the EX/WB buffer appears to be leaking also the Hamming weight of the result of the instructions. Taking into account the time instant when the leakage appears, we ascribe this effect to the presence of nop operations before and after the benchmarked instructions. In particular, we infer that a nop instruction resets the WB bus to zero as a consequence of an implementation choice. We confirm this fact repeating the benchmark more than once and observing that the aforementioned leakage is not present whenever the benchmarked instructions are not followed by nops. All the leakages which we deem due to this border effect are marked with a † sign in Table 2.

Memory Data Register (MDR) and Align Buffer. We consider the MDR as one of the potential leakage sources during load/store instruction sequences. In fact, we report the presence of an information leakage proportional to the Hamming distance of two subsequently loaded values, or two subsequently stored ones. In particular, even halfword and single byte loads/stores share the same leakage model, where the power consumption is proportional to the Hamming distance between full 32-bit words being loaded/-stored from/to the data cache as a consequence of a either 16- or 8-bit memory instruction. Such a behavior led us to consider the possible presence of a separate buffer in the LSU where the proper sub-word value is extracted during sub-word memory operations. We confirmed its presence (and the side channel leakage), testing for a leakage proportional to the Hamming distance between two byte sized values loaded by two ldrb instructions interleaved by

Table 2: Instruction micro-benchmark sequences employed to detect the main leakage sources in the Cortex-A7, and intermediate expressions employed to predict them. The expression in red have a statistically sound leakage with 100k measurements, expressions marked with a † represent leakage due to boundary effects of the nop instructions employed to flush the pipeline

Instruction Sequence	Dual Issued	Register File	Is/Ex Buffer	Shift Buffer	ALU	Ex/Wb buffer	MDR	Align Buffer

`mov rA, rB; nop`
`mov rC, rD`
	No	rB, rD	rB, rD,
			rB ⊕ rD
				–	–	rB†, rD†,
						rB ⊕ rD
							–	–

`add rA, rB, rC`
`add rD, rE, rF`
	No	rB, rC,
		rE, rF
			rB, rC, rE, rF,
			rB ⊕ rE, rC ⊕ rF
				–	rA, rD, rB,
					rC, rE, rF
						rA†, rD†,
						rA ⊕ rD
							–	–

`add rA, rB, rC`
`add rD, rE, n`
	Yes	rB, rC,
		rE, rF
			rB, rC, rE, rF,
			rB ⊕ rE, rC ⊕ rF
				–	rA, rD, rB,
					rC, rE, rF
						rA†, rD†,
						rA ⊕ rD
							–	–

`add rA, rB, rC, lsl n`
`add rD, rE, rF, lsl n`
	No	rB, rC,
		rE, rF
			rB, rC, rE, rF,
			rB ⊕ rE, rC ⊕ rF
				rC ≪ *n*,
				rF ≪ *n*
					rA, rD,
					rB, rE
						rA†, rD†,
						rA ⊕ rD
							–	–

`ldr rA, [rB]`
`ldr rC, [rD]`
	No	rB, rD	–	–	–	rA†, rC†,
						rA ⊕ rC
							rA ⊕ rC	–

`str rA, [rB]`
`str rC, [rD]`
	No	rB, rD	rA ⊕ rC	–	–	rA†, rC†,
						rA ⊕ rC
							rA ⊕ rC	–

`ldr rA, [rB]; ldrb rC, [rD]`
`ldr rE, [rF]; ldrb rG, [rH]`
	No	rA, rC
		rE, rG
			–	–	–	rA†, rC†, rE†, rG†
						rA ⊕ rC,rC ⊕ rE,rE ⊕ rG
							rA ⊕ rC,
							rC ⊕ rE,rE ⊕ rG
								rC ⊕ rG

32-bit load instructions `ldr` (row 7th in Table 2).

4.2 Superscalar Leakage Modeling

In our characterization we observed the presence of information leakage which combines values elaborated by potentially algorithmically independent instructions on the basis of four causes: *i)* the instruction scheduling order; *ii)* the position of the source operands; *iii)* the single or dual-issuing of the said instructions; *iv)* the potential data remanence in the LSU buffers.

A combination of factors *i)* and *ii)* was shown to be a practical source of leakage in scalar architectures in [18] even on masking schemes which were proven to be algorithmically secure: this continues to be an issue on superscalar architectures. In particular, even apparently harmless changes to an assembly codebase, such as swapping the source operands of a commutative operation (e.g., `xor`) may lead to the insurgence of side channel leakage due to their changed pipeline resource sharing. Such a codebase change will not be detected by tools considering only the semantics of the instructions and the register allocation. Moreover, it is also quite hard to deem harmful in a manual audit of the source code. The issues caused by point *i)* and *ii)* are worsened by the effects of dual-issuing (point *iii)*) as even leakage stemming from the combination of source operands of non consecutive instruction may take place in case the instruction in between them is dual issued with the older one. Such a point further states the importance of instruction scheduling among data independent instructions as a mean to prevent side channel leakage, an aspect which up to now was neglected. We note that dual-issuing may also be fruitfully employed to enhance the security of a software implementation of a masking scheme, providing the means to perform the computation of two shares in parallel, yielding a closer mimicry of a registered hardware computation of them.

The presence of side channel leakage caused by non contiguous instructions as the one exhibited in point *iii)* is further exacerbated by data remanence (point *iv)*) in the MDR and LSU buffers. In particular, the results of an entire computation performed in the registers may be accidentally combined with the last loaded/stored values, again giving rise to potentially harmful side channel leakage.

Finally, we also recall that inserting nop operations, while leaving the code semantically equivalent to the non modified one, may add leakage modes to it, in our Cortex-A7 implementation. It may be the case that the nop operations are semantically neutral, but not security neutral, which may be regarded as an unexpected fact.

However, despite the complexity of the leakage model of a superscalar CPU, we note that all the aforementioned considerations can be fruitfully integrated side channel resistant software development toolchain, yielding an effective benefit for developers.

5 EXPERIMENTAL VALIDATION

To provide a validation of our analysis we employ it to examine the results of an attack to a reference implementation of AES-128 running on our characterized platform and employing a non microarchitecture aware leakage model: the Hamming weight of an output byte from the SubBytes primitive of the cipher. Figure 3 reports the results of the attack performed on 100k power consumption traces, collected with the same setup described in Section 4, depicting the results for a portion of the first AES round. The first point showing leakage in time is within the SubBytes primitive and it corresponds to the `load` and subsequent `store` of the value from the AES substitution table during the computation of the look-up. The significant leakage matches our observations on the leakage of `store` operations, which was the highest among the detected ones. While computing the ShiftRows primitive, the same coarse grained leakage model matches instructions unrelated to the SubBytes, but employing its output values. The first detectable leakage in the ShiftRows, is when the output byte from the SubBytes is loaded into a register, followed by three leaking time instants where the said register is shifted progressively by one byte at once to compose the result of the primitive. Finally the result is stored back into memory causing a further leakage. The last leakage appearing in the ShiftRows primitive is when the MDR, which contains the last stored value, receives a zero value to be stored back into memory. Finally, the information leakage taking place during the

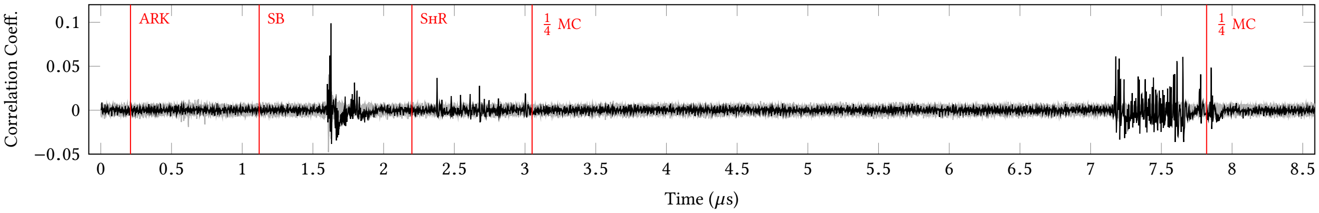

Figure 3: CPA against AES; leakage model: Hamming weight of the output of the SubBytes primitive. Beginning of the round primitives in red: AddRoundKey (ARK), SubBytes (SB), ShiftRows (ShR), MixColumns on a column ($\frac{1}{4}$ of the state) ($\frac{1}{4}$ MC)

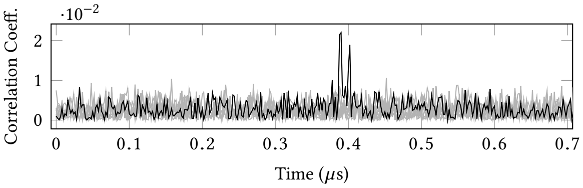

Figure 4: CPA against AES running on Linux, employing the Hamming distance between two byte-long stores SubBytes

MixColumns primitive is due to the load of the output byte from the ShiftRows and its manipulation performed by the function computing its product over $\mathbb{F}_{2^8}$ through a shift-reduce approach. Since the compiler did not inline the said function, additional leakage takes place due to spills and fills into the register file. All these information leakages match the Hamming weight of the value computed by the SubBytes primitive, as reported in Table 2. To further validate our microarchitectural leakage model, we exploit its accuracy to perform an attack on a realistic scenario where the AES block cipher is run as a userspace process in a full fledged Ubuntu 16.04 environment running on the Olinuxino board. In this scenario, we did not perform clock gating on any of the peripherals, nor prevent any processes of the Linux distribution from running, including the GUI, and the AES process runs with no CPU-affinity set or elevated priority. The only alteration with respect to a regular running environment was to limit the frequency of the CPU to 120 MHz due to the limitations of the sampling frequency of the oscilloscope. To provide a realistic noise for the environment, we installed the Apache 2.4.18 webserver on the board, set the maximum number of server processes to 100, and queried the board from a PC via Ethernet employing the HTTPerf performance testing tool at a rate of 1000 queries/s. We verified via the htop that both the CPU cores of the Cortex-A7 were at full load, and collected the AES power consumption in this environment. Figure 4 reports the result of performing a CPA employing as a leakage model the Hamming distance between two consecutively stored bytes in the SubBytes primitive on 100 power traces obtained each one as the average of 16 executions of AES on the same input. Despite the reduction in the absolute value of the Pearson correlation coefficient, the attack succeeds (the correct key is distinguishable from the best wrong guess with a statistical confidence > 99%). These results provide a validation of the reliability of a microarchitectural model to extract side channel leakage from a complex computing platform, even with strongly noisy environments which match real world scenarios. We foresee the possibility of integrating such a model within static analysis and code generation tools employing it as an enabler for more accurate analyses, easing and automating the mapping of countermeasure schemes onto implementations.

ACKNOWLEDGEMENTS

This work was supported in part by the EU grants: "SafeCOP" (ECSEL RIA) Grant agreement no. 692529, and "M2DC" (H2020 RIA) Grant agreement no. 688201.

REFERENCES

[1] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2014. A Multiple Equivalent Execution Trace Approach to Secure Cryptographic Embedded Software. In *DAC '14*. ACM, 210:1–210:6.

[2] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2015. Information leakage chaff: feeding red herrings to side channel attackers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 33:1–33:6.

[3] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2015. The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks. *IEEE Trans. on CAD of Integrated Circuits and Systems* 34, 8 (2015), 1320–1333.

[4] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2016. Encasing block ciphers to foil key recovery attempts via side channel. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*, Frank Liu (Ed.). ACM, 96.

[5] Allwinner Technology Co., Ltd. 2013. *Allwinner A20 User Manual 2013-03-22*. http://dl.linux-sunxi.org/A20/, last accessed 2017-20-11.

[6] ARM Ltd. 2011. *Cortex-A7MPCore Reference Manual*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0464f/, last accessed 2017-20-11.

[7] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. 2014. On the Cost of Lazy Engineering for Masked Software Implementations. In *CARDIS 2014 (LNCS)*, Vol. 8968. Springer, 64–81.

[8] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. 2015. DPA, Bitslicing and Masking at 1 GHz. In *CHES 2015*. 599–619.

[9] Nicolas Bruneau, Sylvain Guilley, Annelie Heuser, Damien Marion, and Olivier Rioul. 2017. Optimal side-channel attacks for multivariate leakages and multiple models. *J. Cryptographic Engineering* 7, 4 (2017), 331–341.

[10] Zhimin Chen, Ambuj Sinha, and Patrick Schaumont. 2013. Using Virtual Secure Circuit to Protect Embedded Software from Side-Channel Attacks. *IEEE Trans. Computers* 62, 1 (2013), 124–136.

[11] Hassan Eldib, Chao Wang, Mostafa M. I. Taha, and Patrick Schaumont. 2015. Quantitative Masking Strength: Quantifying the Power Side-Channel Resistance of Software Code. *IEEE Trans. on CAD* 34, 10 (2015), 1558–1568.

[12] Free Software Foundation and ARM. 2012. ARM Cortex-A7 description. (2012). https://github.com/gcc-mirror/gcc/blob/master/gcc/config/arm/cortex-a7.md, last accessed 2017-20-11.

[13] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs - Extended version. *J. Cryptographic Engineering* 5, 2 (2015), 95–112.

[14] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to Differential Power Analysis. *J. Cryptographic Engineer.* 1, 1 (2011), 5–27.

[15] S. Mangard and K. Schramm. Pinpointing the Side-Channel Leakage of Masked AES Hardware Implementations. In *CHES 2006 (LNCS)*, Vol. 4249. Springer.

[16] David McCann, Elisabeth Oswald, and Carolyn Whitnall. 2017. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In *USENIX Sec 2017*. USENIX Association, 199–216.

[17] OLIMEX Ltd. 2017. *A20-olinuxino-MICRO*. https://www.olimex.com/Products/OLinuXino/A20/, last accessed 2017-20-11.

[18] Hermann Seuschek, Fabrizio De Santis, and Oscar M. Guillen. 2017. Side-channel leakage aware instruction scheduling. In *CS2@HiPEAC 2017*. ACM, 7–12.

[19] Hermann Seuschek and Stefan Rass. 2016. Side-channel Leakage Models for RISC Instruction Set Architectures from Empirical Data. *Microprocessors and Microsystems* 47 (2016), 74–81.