ABBY: Automating the creation of fine-grained leakage models

Omid Bazangani
omid.bazangani@ru.nl
Radboud Univesity
Nijmegen, Netherlands

Alexandre Iooss
alexandre.iooss@student.ru.nl
Radboud University
Nijmegen, Netherlands

Ileana Buhan
ileana.buhan@ru.nl
Radboud University
Nijmegen, Netherlands

Lejla Batina
lejla.batina@ru.nl
Radboud University
Nijmegen, Netherlands

ABSTRACT

Side-channel leakage simulators allow testing the resilience of cryptographic implementations to power side-channel attacks without a dedicated setup. The main challenge in their large-scale deployment is the limited support for target devices, a direct consequence of the effort required for reverse engineering microarchitecture implementations. We introduce ABBY, the first solution for the automated creation of fine-grained leakage models. The main innovation of ABBY is the training framework, which can automatically characterize the microarchitecture of the target device and is portable to other platforms. Evaluation of ABBY on real-world crypto implementations exhibits comparable performance to other state-of-the-art leakage simulators.

KEYWORDS

side-channels, leakage simulator, neural networks, microarchitecture

1 INTRODUCTION

Technological advancements and twenty years of sustained effort by the cryptographic community significantly raised the workload required for successful key extraction. However, the problem of implementing a secure cryptographic algorithm on a given target device is not solved. Piling up countermeasures is not a solution as countermeasures come at the cost of resources. The challenge for a developer is to balance the presence of countermeasures against information leaks. As the product changes during development, it is important to understand if the changes are beneficial or on the contrary, if they compromise the security of the implementation.

The appeal of side-channel *leakage simulators*, which model the instantaneous power consumption of a device, is evident from the effort directed towards creating such tools [1]. A leakage simulator transforms high-level code into traces similar to the ones collected from the target architecture. When correct and informative, simulators are very advantageous during development and have a number of benefits compared to actually measured traces. The first benefit is *speed*, as generating traces is automated in the case of leakage simulators, while measuring traces requires the creation of a setup manually. Additionally, the traces generated by a leakage simulator contain no noise and suffer from no misalignment, while the measured traces require additional signal processing. The second advantage is *shorter time to market*, as transforming a cryptographic implementation into power traces is significantly more cost-effective and straightforward to operate than creating a physical setup.

Side-channel attack and defenses both rely on *leakage models* which are an abstraction of the physical implementation of the target device. It was shown that the effectiveness of masking, a common countermeasure for hardening a cryptographic implementation, is heavily influenced by the underlying hardware, as shown by the order-reduction theorem [10]. A *fine-grained leakage model* accurately captures the hardware by modeling the microarchitectural implementation of the target device and can be relied upon when verifying a masked implementation. Although the importance of fine-grained leakage models has been established [6, 8, 11], the microarchitecture implementation is considered a trade secret and therefore it is not captured.

While the methodology for building leakage simulators is known, the main limiting factor for their wide adoption is the limited number of supported target devices, a direct consequence of the effort required for reverse engineering the microarchitecture implementation. We consider ELMO [9] to be the most sophisticated fine-grained leakage simulator. As the main barrier to overcome for the widespread deployment of leakage simulators is the characterization of the target device, we ask the following question:
Can we automate the creation of fined-grained leakage models without reverse engineering the micro-architecture implementation?

Contribution. In this work, we answer the question in the affirmative. Specifically, we propose a new leakage simulator ABBY, which uses machine learning for the creation of fine-grained leakage models. We compare the performance in detecting leakage of ABBY with ELMO and show that its performance comes close without much optimization of the machine learning model. The advantage of ABBY is twofold: the first is that no reverse engineering of the target device is required, and the second is that ABBY can learn nonlinear fine-grained leakage models while existing solutions use linear leakage models, as there are some targets which exhibit nonlinear behavior, characteristic to small technology size or special logic styles.

Paper organization: Section 2 briefly introduces the background on side-channel attacks leakage detection and describes the hardware setup we used. Related works on leakage simulators are mentioned in Section 3. Section 4 discusses building the fine-grained leakage model for ELMO vs ABBY. Section 5 discusses the training of ABBY. Finally, Section 6 presents the experimental results while Section 7 concludes the paper.

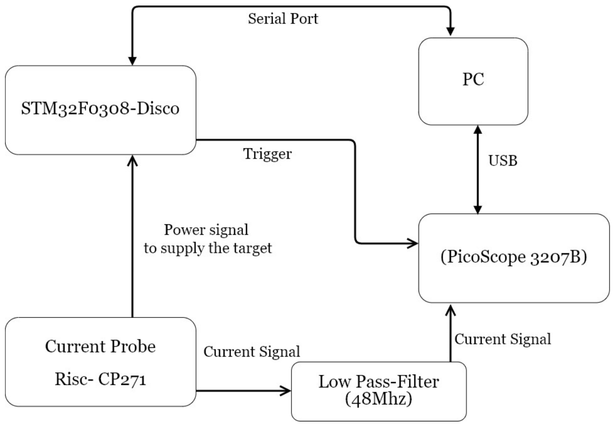


Figure 1: Setup Block Diagram

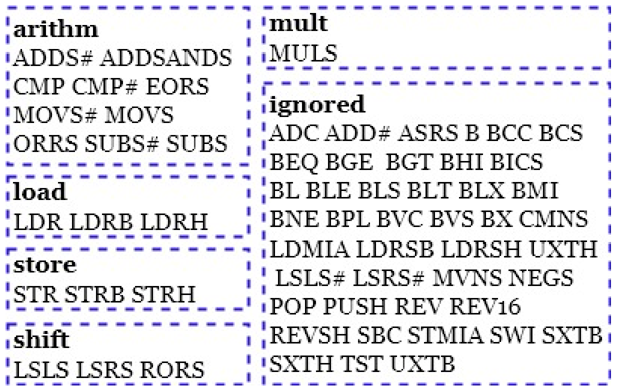
Implementation code: The implementation's of ABBY, will be available upon paper acceptance.

2 BACKGROUND

Side Channel-Leakage. The power consumption and EM signals emitted from a device correlate with the processed data and the executed instructions. The amount of power required to maintain a signal depends on the logical state of the signal. In CMOS technology, the predominant choice when manufacturing integrated circuits, changing the value of a bit requires a different power level than keeping the value of a bit constant. Therefore, the power consumption of a circuit directly correlates with the data circuit processes. Monitoring the physical properties of devices can reveal information about the operations they carry and the data they process. To perform a side-channel attack, an attacker attempts to correlate the physical properties with the secret values processed by the device. Other effects, such as variations in signal propagation time over the circuit, or cross-capacitance effects, all contribute to the instantaneous power consumption of the device and therefore correlate with the data processes.

Leakage detection. Test Vector Leakage Assessment (TVLA) [7] is one of the most popular methods for leakage detection due to its simplicity and relative effectiveness. It is based on statistical hypothesis tests and comes in two flavors: *specific* and *non-specific*. The 'fixed-vs-random' is the most common nonspecific test and compares a set of traces acquired with a fixed plaintext with another set of traces acquired with random plaintext. In the case of a specific test, the traces are divided according to a known intermediate value tested for leakage. In both cases, Welch's two-sample t-test for equality of means is applied for all trace samples. A difference between two sets larger than a given threshold is taken as evidence for the presence of a leak.

Instruction Emulators. To build a side channel leakage simulator, we need an emulator that outputs an instruction trace from the compiled machine code. Most emulators are only instruction-accurate and not cycle-accurate, i.e., ignoring the fact that one instruction may take more than one clock cycle. It is possible to reach cycle accuracy in emulation when detailed hardware description of all used peripherals is available. Verilator can convert Verilog hardware



arithm
ADDS# ADDSANDS
CMP CMP# EORS
MOVS# MOVS
ORRS SUBS# SUBS

load
LDR LDRB LDRH

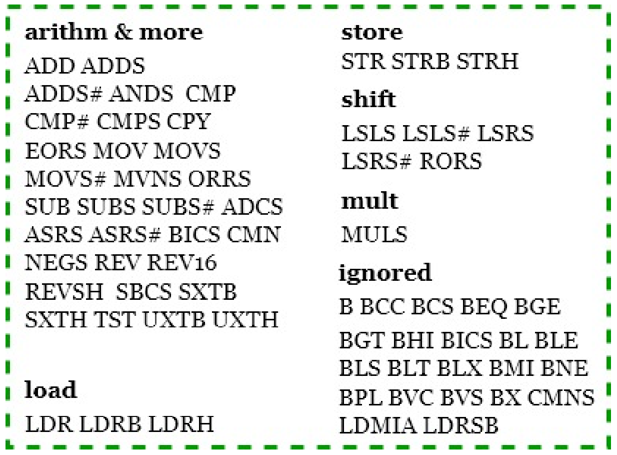
store
STR STRB STRH

shift
LSLS LSRS RORS

mult
MULS

ignored
ADC ADD# ASRS B BCC BCS
BEQ BGE BGT BHI BICS
BL BLE BLS BLT BLX BMI
BNE BPL BVC BVS BX CMNS
LDMIA LDRSB LDRSH UXTH
LSLS# LSRS# MVNS NEGS
POP PUSH REV REV16
REVSH SBC STMIA SWI SXTB
SXTH TST UXTB

(a) ELMO instruction clusters



arithm & more
ADD ADDS
ADDS# ANDS CMP
CMP# CMPS CPY
EORS MOV MOVS
MOVS# MVNS ORRS
SUB SUBS SUBS# ADCS
ASRS ASRS# BICS CMN
NEGS REV REV16
REVSH SBCS SXTB
SXTH TST UXTB UXTH

load
LDR LDRB LDRH

store
STR STRB STRH

shift
LSLS LSLS# LSRS
LSRS# RORS

mult
MULS

ignored
B BCC BCS BEQ BGE
BGT BHI BICS BL BLE
BLS BLT BLX BMI BNE
BPL BVC BVS BX CMNS
LDMIA LDRSB

(b) ABBY instructions

Figure 2: ELMO instruction clusters vs ABBY instructions

descriptions to cycle-accurate behavioral models. For ARM, these cycle-accurate models (Arm Fixed Virtual Platforms) are closed-sourced and do not typically describe other peripherals that might affect instruction execution speed. ELMO is instruction accurate and uses Thumbulator to emulate ARM Thumb-1. As a consequence of the discrepancy between instruction vs cycle accuracy the measured side-channel traces might not align with the instruction trace.

Hardware Setup. We use an ARM Cortex-M0 processor based on Armv6-M architecture, manufactured by ST Microelectronics with STM Discovery Boards. The target board has an STM32F0 (30R8T6) chip (similar to ELMO target) and an external crystal oscillator (8MHz). To filter the measurement noise, we modified the board by removing the power line capacitors and measuring the current through a current probe (Riscure CP271), which is used as a proxy for the target's power consumption. We use a PicoScope 3207B for data acquisition at a sampling rate of 500MS/s while the target is running at 8MHz. This oscilloscope can store up to 512Ms due to memory limitations. We use a physical 48MHz low-pass filter for the measurement. We are using two acquisition channels, power signal, and trigger signal. The trigger is fed by a GPIO of the target when the desirable segment of the code is running. The oscilloscope

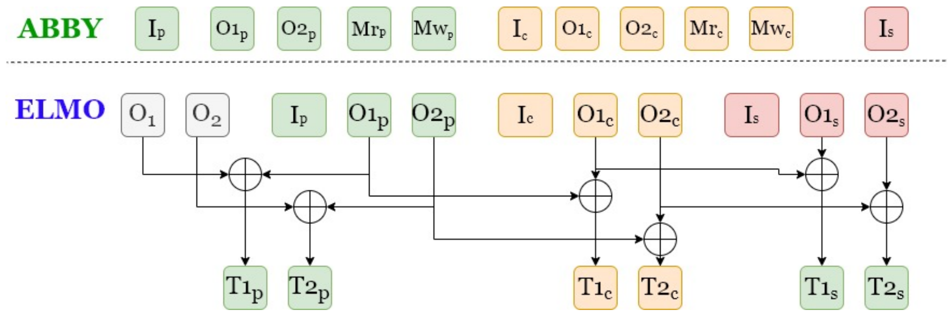


Figure 3: ABBY vs ELMO features to profile power consumption of the chip

will be armed for signal recording as soon as the trigger signal is detected. Figure 1 shows the setup block diagram.

3 RELATED WORKS

Leakage simulators. SILK [13], is the first open-source side-channel leakage simulator, which takes as input the source code of the cryptographic algorithm and some user-defined parameters (leakage function, number of leakage points, etc) and generates power traces. It captures no specific hardware architecture and targets data-dependent power consumption. It is suitable for the early stages of cryptographic algorithm design. Instruction level simulators (ISA), look at the machine code, compiled for a certain architecture, to predict side-channel leakage. ISA simulators use machine code that will be executed by a specific CPU, but do not require information about the processor or about the process technology. SAVRASCA [14] takes as input compiled binary code and using the tracing feature of the SimulAVR tool will output simulated power traces for AVR architecture. SAVRASCA was used to report a bug in the code of the implementation used for the DPAv4 trace set. ASCOLD [10] checks violations of the independent leakage assumption (ILA) for AVR architecture. It takes as input the assembly file of the masked implementation and a configuration file of the system and outputs the location of the leakage (line number) and the rule that was violated. The physical causes for the ILA breaching effects are device-specific (cannot be generalised) and counter-intuitive when related to the assembly description of the target.

Going one level lower are the simulators which capture some of the microarchitecture effects of the target. These simulators capture a more descriptive model of the target, at the cost of a bigger engineering effort. MAPS [2] is a power simulator designed for the ARM Cortex M3, which takes as input the source code of the masked implementation and outputs a simulated power trace. To capture the microarchitecture details, the authors used an HDL file of the target architecture and focus mostly on the leakage caused by the pipeline. In most cases, however, HDL files of the target are not available. We consider ELMO [9] to be the first genuinely fine-grained leakage simulator for the ARM-Cortex M0/M4 family. ELMO models the power consumption as a linear combination of bit values and bit changes. Most remarkably, the simulator was created without detailed information of hardware description or the target microprocessor. ELMO* [11] improves the leakage model of ELMO by capturing interactions that span multiple cycles. ROSITA [11] is a rule-driven code rewrite engine that patches the code automatically once leakage is detected. ROSITA starts with a (masked) implementation of a cryptographic algorithm, cross-compiled to produce both the assembly and the binary executable. A very compelling feature of ROSITA is that it extends an existing leakage detection tool, ELMO [9] to report instructions that leak secret information. The new detection framework (ELMO*), uses the binary file to detect leakage and identify the offending machine instruction; ROSITA then applies a set of rules that replace the leaky instruction with an equivalent one (functionally) that does not leak. ROSITA repeats the process until no more leakage is detected.

While the importance of microarchitecture details in a security analysis has been established [6], [8] access to its implementation is typically not available. The authors of ELMO had to reverse engineer the microarchitecture implementation of the target ARM Cortex M0 processor. The current state of the art allows reverse engineering a commercial ARM Cortex-M3 microprocessor [6]. The authors note that the current methodology involves intensive manual effort. However, it is worthwhile as it shows the importance of capturing microarchitectural effects.

4 BUILDING FINE-GRAINED LEAKAGE MODELS

4.1 Selecting salient features for the model

ELMO. The critical observation made by McCann et al. [9] when building the ELMO leakage model is that the power consumption of *the current instruction*, I_c depends on *the preceding instruction*, I_p and *the subsequent instruction* I_s [12]. ELMO is instruction-accurate, which has the advantage of allowing the quick identification of a leaky instruction. Following a cluster analysis to group "similar" instructions (i.e., which leak information in the same way), the authors identify five groups, see Figure 2. The groups correspond to the same processor component: ALU instructions in one group, shift instructions as another group, load, and stores that interact with the memory as two or more groups, and the `MULS` instruction with a distinct profile due to its single cycle implementation.

Figure 3 (bottom) is a visual representation of the interaction between the different instructions in the ELMO model. Transitions for instruction operands across the data bus are captured in the form of a 32-bit matrix (represented as $T1_p$, $T2_p$ for previous instruction, $T1_p$,$T2_p$ for current instruction and $T1_s$, $T2_s$) for the subsequent instruction).

ABBY. Similar to ELMO, ABBY captures the interaction between instructions in the pipeline registers, see Figure 3(top) and memory leakage. To simplify the collection of the training data, no assumptions about the operand interaction are made. ABBY leaves the modeling of the relation between operands to the machine learning model. ABBY also adds memory read/write values for the current and previous memory access to cover memory leaks, as these were shown to be important in [11].

As part of the effort to simplify training, we further remove the clustering of instructions. We keep the necessary Thumb instructions for crypto algorithms, which represent a total of 44 instructions for ARM Cortex-M0 including arithmetic, shift, store, load and multiplication operations. We do not profile branching, stack operations `PUSH`, `POP`, `LDR/STR sp`, or operations changing the `PC` register (Fig. 2) as these operations are not in use for implementing cryptographic algorithms. For store and load operations, we reserved register **r0** to put the memory address of an empty data section.

4.2 Data collection

ELMO. To reduce the profiling space McCann et al. [6] target only 21 ARM Thumb instructions that are commonly used in block ciphers. Collection of side-channel data is done in two steps. The first is the clustering of instructions in five groups as shown in (Figure 2). For each of the 21 instructions, 5000 traces are acquired by varying the instruction operands. The second is the support for sequence dependence, 1000 traces are collected for each all possible combinations of instructions ($5^3 = 125$) (i.e., five groups, 3 pipeline stages). ELMO requires ($21 \times 5000 + 5^3 \times 1000 = 230000$) data points.

Ignoring instructions such as branching `B`, `BEQ`, `BL` makes sense as block ciphers avoid to use them to be time constant and to prevent leaks from branch prediction. These clusters also represent the internal structure that we could expect from an ARM core as shift, multiplication, and arithmetic operations do not use the same part of the CPU.

ABBY. We generate a randomized assembly firmware including triplet Thumb instructions, to cover the 3-stage pipeline of the target. We analyze the output to ensure that the instructions are uniformly chosen and their operand value is uniformly distributed. As we do not have the clustering limitation (all instructions in one cluster), we keep all Thumb instructions typical for cryptographic implementations. We are not profiling the branching, the stack operations (`PUSH`, `POP`, `LDR/STR sp`), or the operations changing the `PC` register (Figure 2) as these instructions are not used for the implementation of cryptographic algorithms. The result is a group of 44 instructions for ARM Cortex-M0 including arithmetic, shift, store, load and multiplication operations. For the store and the load operations, we reserved register `r0` to put the memory address of an empty data section.

Using the generated firmware, we collect 1 000 triplets[1] (or pipeline states) in a single acquisition. By generating and flashing[2] the firmware 50 000 times, we collect for each triplet of instructions (with random operands) $\frac{50\,000 \times 1\,000}{44^3} \approx 587$ data points.

[1]A compromise between the oscilloscope memory and the duration of running Dynamic Time Warping for alignment ($t \sim O(n^2)$).

[2]We could run from RAM to preserve flash endurance, but it influences leaks and the number of cycles per instruction. Running the firmware from flash is closer to real-world scenarios.

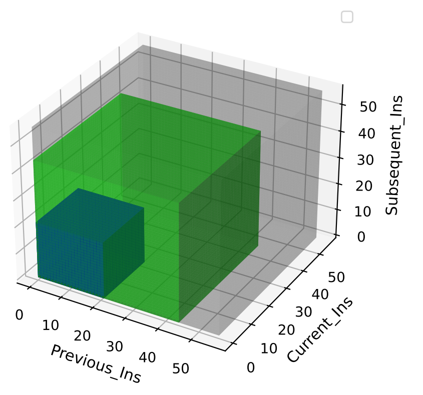


Figure 4: Possible instruction space in gray vs ABBY and ELMO in green and blue respectively

Instruction coverage. For a 3-stage pipeline microprocessor, and considering 56 possible Thumb-instructions for Cortex-M0, the entire instruction space is 56^3. Figure 4 shows the coverage of the instruction space of ELMO vs ABBY. We see that ABBY covers more instruction combinations compared to ELMO.

5 TRAINING ABBY

Model creation for ABBY is completely automatic and consists of two steps, dataset generation and model training. The dataset generation is controlled by a script which runs related scripts for firmware generation, feature extraction, trace annotation, and labels the generated dataset.

Figure 5 compares the model creation steps for ABBY with the creation of the ELMO model, which needs five different steps. Furthermore, as the Figure 5 shows, ELMO needs two steps of acquisition while ABBY only needs one.

5.1 Preparing the training dataset

We load the firmware with random instructions on the target device and measure the power while executing the firmware. To label the measured power samples, we need to identify the corresponding triplet of instructions. A challenge when annotating the measured traces with the executed instructions is that different instructions might take different cycles, depending on the optimizations made by the manufacturer.

Trace annotation. Our solution, dictated by the simplicity of the ARM Cortex M0 processor, is not perfect, but it is effective. We use ELMO as an initially estimated power consumption to compare with the feature annotation. Next, we replace the value generated by ELMO with the corresponding value extracted from the measured traces. ELMO is not cycle-accurate; the measured and estimated traces do not have the same number of samples. Our solution to align two time series of varying sizes is *Dynamic Time Warping*

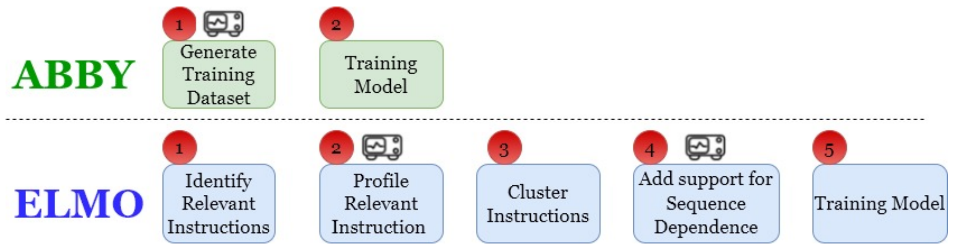
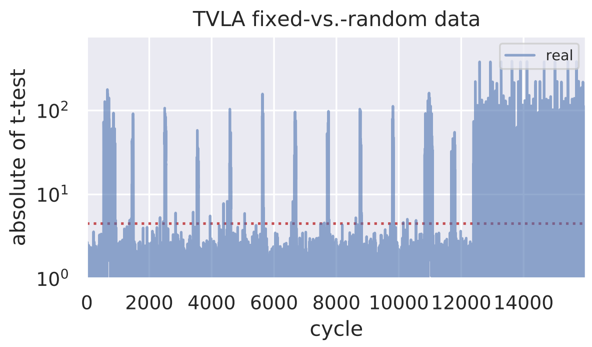
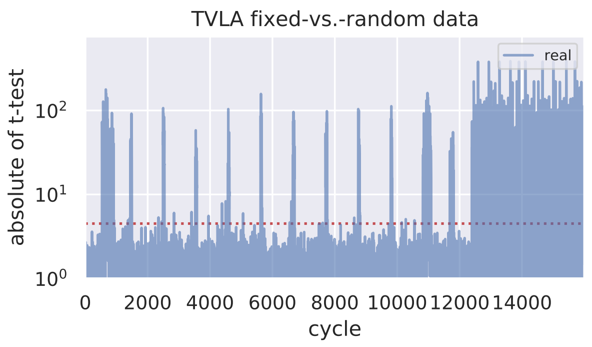

ABBY
1
Generate Training Dataset
2
Training Model
ELMO
1
Identify Relevant Instructions
2
Profile Relevant Instruction
3
Cluster Instructions
4
Add support for Sequence Dependence
5
Training Model

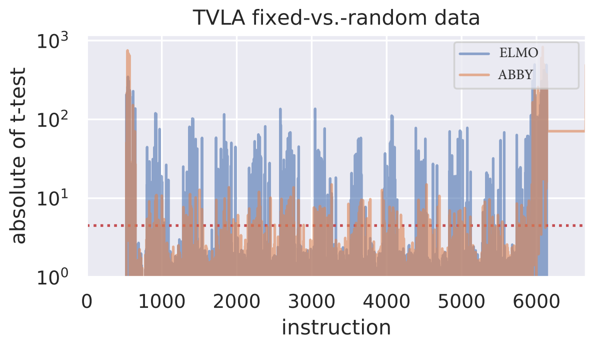
Figure 5: Model training steps ELMO vs ABBY



(a) Measured power trace

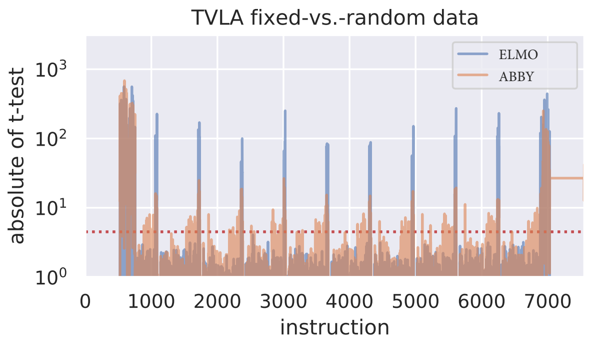


(a) Measured power trace



(b) ELMO model vs ABBY model

Figure 6: Xoodoo TVLA result on measured power trace and ELMO vs ABBY model for 100K traces(50K fix vs 50K random)



(b) ELMO model vs ABBY model

Figure 7: Byte-Masked-AES TVLA result on Measured power trace and ELMO vs ABBY for 100K traces(50K fix vs 50K random)

(DTW), a popular algorithm used for speech recognition. With this technique, we solve the signal alignment problem and find the corresponding power consumption value for each instruction triplet (Figure 8). DTW is not perfect, and alignment errors are possible. As a result of the first attempt to align the data, 22% of instructions are dropped. We also expect errors on the remaining data, so perform several passes. We observe that the alignment distance converges after four passes (there is no significant improvement with more iterations).

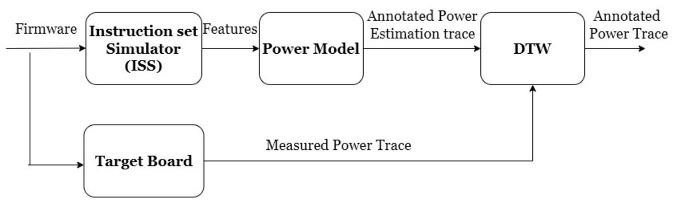


Figure 8: Data generation for ABBY, zoom in Figure 5, block 1 of ABBY.

Table 1: MLP hyperparameter description

Layer Types	Details
Input Fully-connected	#neurons 388, Relu
Hidden Fully-connected	#neurons 16, Relu
Hidden Fully-connected	#neurons 16, Relu
Output	#neurons 1, Linear

5.2 Fitting the ABBY model

We use a simple Multi-Layer Perceptron (MLP) model to train the ABBY. We use Tensorflow2 with Keras submodule to build a preprocessing pipeline and a MLP model with 2 hidden layers. The input layer reads the ABBY features in a normalized form. For normalizing operands and memory read/write values, we extended each value to a 32-bits binary array. For mnemonic assembly instructions which are not ordinal categorical features, we use one-hot encoding. The input layer consists of $(8 \times 32) + (3 \times 44) = 388$ neurons. Table 1 summarizes the model architecture.

After 341 training epochs, our MLP model reach $r^2 = 0.9951$ on a training set of $\approx$ 35 Millions samples with a test set of $\approx$ 15 Millions while 20% of training set used as validation. Although the r^2 metric evaluates the performance of the model, we apply standard side-channel evaluation metrics to determine its usefulness for leakage detection.

6 RESULT AND DISCUSSION

The main goal of any side-channel leakage simulator is to find the leaks, and determine their source such. There are different leakage detection methods in side-channel domain, but one of the most popular is TVLA, see Section 2 for a brief introduction. We applied TVLA test on two different cryptographic algorithms, AES [5] and Xoodoo [3]. For the AES implementation, we choose a first-order protected implementation by Yao et al.[15], which we refer to as *Byte Masked AES*. Although this implementation should be secure against first-order leaks, TVLA indicates leaks both on the measured power trace as well as on the simulated power traces by ELMO and ABBY models(Figure 7). As a proof that ABBY has learned the same leaks as ELMO, we notice how similar the t-trace scores produced by ELMO and ABBY are, with ABBY showing more leakage points compared to ELMO. Notice the difference in the x-axis between the labels of the measured traces (cycles) compared to the labels of the simulated traces (instructions).

To confirm that the positive results obtained for Masked AES are not just a lucky coincidence, we compare the t-test results of ELMO and ABBY on a different cryptographic algorithm, namely Xoodoo [3]. Xoodoo is the underlying permutation used in Xoodyak [4], one of the finalists in the NIST Lightweight Cryptography Standardization process. As shown in Figure 6, we confirm that also in this case the output of ABBY is comparable to ELMO model and the measured power traces.

7 CONCLUSIONS AND FUTURE WORK

We propose ABBY, the first machine learning-based leakage simulator. The main innovation of ABBY is its compact training framework, which does not require reverse-engineering the microarchitecture implementation of the target device. As a result, ABBY significantly reduces the human effort necessary for the creation of fine-grained leakage models. ABBY is scalable and can be transferred to different architectures. We did not investigate the optimizations for the machine learning model used by ABBY, and we used a simple MLP architecture. The most challenging aspect of porting ABBY to different platforms is the creation of the labeled dataset. We used the DTW algorithm, which, although not optimal, seems to work. As future work, we aim to improve the data generation, investigate how targeted microarchitecture benchmarks such as [8] and optimizations of the model architecture can further enhance the performance of ABBY.

REFERENCES

[1] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. 2021. SoK: Design Tools for Side-Channel-Aware Implementations. arXiv:2104.08593 [cs.CR]

[2] Yann Le Corre, Johann Großschädl, and Daniel Dinu. 2018. Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10815)*, Junfeng Fan and Benedikt Gierlichs (Eds.). Springer, 82–98. https://doi.org/10.1007/978-3-319-89641-0_5

[3] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. 2018. The design of Xoodoo and Xoofff. *IACR Trans. Symmetric Cryptol.* 2018 (2018), 1–38.

[4] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. 2020. Xoodyak, a lightweight cryptographic scheme. *IACR Trans. Symmetric Cryptol.* 2020, S1 (2020), 60–87. https://doi.org/10.13154/tosc.v2020.iS1.60-87

[5] J. Daemen and Vincent Rijmen. 2002. AES the Advanced Encryption Standard. *The Design of Rijndael* 26 (01 2002).

[6] Si Gao, Elisabeth Oswald, and Dan Page. 2021. Reverse Engineering the Micro-Architectural Leakage Features of a Commercial Processor. *IACR Cryptol. ePrint Arch.* (2021), 794. https://eprint.iacr.org/2021/794

[7] G. Goodwill, J.J.B. Jun, and P.Rohatgi. 2018. A testing methodology for side channel resistance validation. *NIST non-invasive attack testing workshop* (2018).

[8] Ben Marshall, Dan Page, and James Webb. 2021. MIRACLE: MIcRo-ArChitectural Leakage Evaluation. Cryptology ePrint Archive, Report 2021/261. https://ia.cr/2021/261.

[9] David McCann, Elisabeth Oswald, and Carolyn Whitnall. 2017. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 199–216. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann

[10] Kostas Papagiannopoulos and Nikita Veshchikov. 2017. Mind the Gap: Towards Secure 1st-Order Masking in Software. In *Constructive Side-Channel Analysis and Secure Design*, Sylvain Guilley (Ed.). Springer International Publishing, Cham, 282–297.

[11] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. 2019. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1445. https://eprint.iacr.org/2019/1445

[12] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. 1996. Instruction Level Power Analysis and Optimization of Software. In *VLSI Design*.

[13] Nikita Veshchikov. 2014. SILK: high level of abstraction leakage simulator for side channel analysis. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014, New Orleans, LA, USA, December 9, 2014*, Mila Dalla Preda and Jeffrey Todd McDonald (Eds.). ACM, 3:1–3:11. https://doi.org/10.1145/2689702.2689706

[14] N. Veshchikov and S. Guilley. 2017. Use of Simulators for Side-Channel Analysis. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 104–112. https://doi.org/10.1109/EuroSPW.2017.59

[15] Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. 2018. Fault-assisted side-channel analysis of masked implementations. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 57–64. https://doi.org/10.1109/HST.2018.8383891