A side-channel based disassembler for the ARM-Cortex M0

Jurian van Geest and Ileana Buhan

Radboud University, Digital Security Group
jurianvgeest@gmail.com, ileana.buhan@ru.nl

Abstract. The most common application for side-channel attacks is the extraction of secret information, such as key material, from the implementation of a cryptographic algorithm. However, using side-channel information, we can extract other types of information related to the internal state of a computing device, such as the instructions executed and the content of registers. We used machine learning to build a side channel disassembler for the ARM-Cortex M0 architecture, which can extract the executed instructions from the power traces of the device. Our disassembler achieves a success rate of 99% under ideal conditions and 88.2% under realistic conditions when distinguishing between groups of instructions. We also provide an overview of the lessons learned in relation to data preparation and noise minimization techniques.

Keywords: Side-channels · Disassembler · Machine-learning.

1 Introduction

The extraction of information using side channels is extensively studied in an adversarial setting, where the target of the attack is the implementation of a cryptographic algorithm. There are two classes of side-channel attacks. The first are *non-profiled* attacks, where the adversary can choose the input data, observe the encryption output, and monitor the side-channel information. The second is *profiled* attacks, where the adversary has access to a clone device to learn the behavior of the algorithm. Side channel attacks have a long history of success [1] in extracting key information from power or electromagnetic traces collected during the execution of a cryptographic algorithm.

However, monitoring the side-channel information of an embedded system has also been proven to be useful for defense purposes [2]. A *side-channel disassembler* monitors the control flow of an application at run-time by translating side-channel information, such as power traces, into assembly code consisting of instructions and operands. The applications of a side-channel disassembler are multiple. An example is the detection of security breaches, such as malware attacks. To detect malware using a side-channel disassembler, the signature information of the healthy device at run-time is collected. This information is used to verify the integrity of the code running on the device. A flag is raised if a deviation from normal operation mode is detected. Another application is reverse engineering of the firmware of a target device. Side-channel disassemblers have been shown to successfully recognize both the opcode and the operands for a given device and instruction set architecture.

Problem statement. Translating a power trace into a sequence of instructions is a challenge. The first challenge is that an instruction is typically executed once for an execution path, so there is relatively little information to use for identification. The second challenge is that the power signature of an instruction is influenced by other instructions in the pipeline [3], changing the side-channel signature of an instruction. The third challenge is that the implementation of the microarchitecture of an embedded device is a trade secret, and hidden storage elements influence the interaction of instructions [4].

Contribution. Building on previous work, we investigate the use of machine learning models for side-channel disassembly of instructions running on an ARM-Cortex M0 processor, which is a popular choice for IoT due to its ultralow gate count. Its side-channel leakage has been extensively studied in the context of *leakage simulators*. Unlike *side-channel disassemblers* which extract the assembly instruction from a power trace, *leakage simulators* aim to construct the power trace for a set of assembly instructions. This is the first attempt at modeling a 32-bit architecture with a 3-stage pipeline; previous work has focused on 8-bit processors with 2-stage pipelines. 32-bit architectures are more complex and typically add more components, increasing the difficulty of recognizing instructions in power traces. Using the information collected from the power traces, we performed experiments to identify the groups of instructions as suggested in [5] and individual instructions. Under ideal conditions, our side-channel disassembler reaches a success rate of 99%, while under realistic conditions, we observe a success rate of 88.2%.

Paper organization. Section 2 describes the related work. The experimental setup that we used to validate our results is presented in Section 3 and the datasets we collected are described in Section 4. The challenges of selecting mixed-instruction sequences are described in Section 5. Our results are presented in Section 6 and conclusions are presented in Section 7. The KL-based feature selection proposed in [6] is discussed in more detail in section A (appendix).

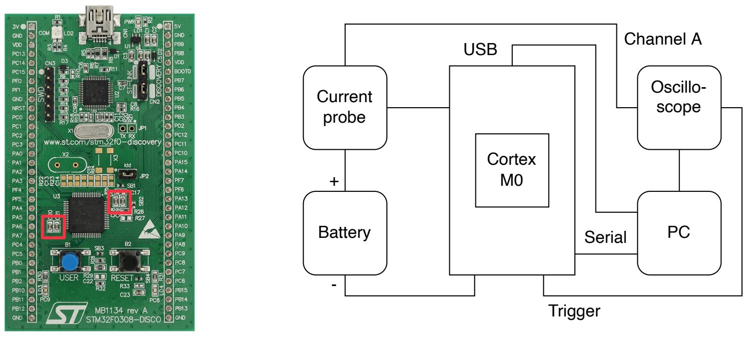
2 Related work

Side-channel disassemblers Park *et al.* [6] have created a side-channel disassembler targeting the Atmega 328P microcontroller and report a success rate of 99.03% in instruction identification. The first step of the disassembler is to collect power traces. Next, all instructions are divided into groups on the basis of their operands. Since the microcontroller used has a two-stage pipeline, the target instruction is preceded and succeeded by a random instruction to fill the pipeline. After the traces are collected, the difference between each trace and a reference trace containing only `nop`'s is computed to remove electrical noise. This work is the starting point for the results presented in this paper. We extensively experimented with the proposed special feature selection and combined it with several machine learning algorithms. Eisenbarth *et al.* [7] targets a PIC16F687 microcontroller, running at $1MHz$ that features a set of 35 instructions (most are 1 cycle instructions). Their goal is to reconstruct the instructions executed and their order from *a single measurement.* They use templates to model the power consumption of a single instruction. They also use instruction frequency analysis to determine the probability that instructions appear in a piece of code and feed this information to the distinguisher function. They report a recognition rate of 70%. Msgna *et al.* [8] targets an 8-bit ATMega163 microcontroller, running at $4MHz$, which features a set of 130 instructions. For the experiments, they only used 39 instructions and report a 100% recognition rate.

Side-channel leakage for the ARM-Cortex M0. McCann *et al.* [5] created ELMO, a leakage simulator for the ARM-Cortex M0/M4 family. ELMO is instruction-accurate, which easily allows the identification of a leaky instruction in the context of side channels. An exciting feature of ELMO is the support for *sequence dependency.* The critical observation is that the power consumption of different instructions depends on the instructions executed before. Following a cluster analysis to identify similar instructions (i.e., those that leak information in the same way), the authors identified five groups that correspond to the same processor component. In this work, we use the grouping of instructions proposed by [5]. In addition, the authors find remarkable consistency in the data-dependent leakage of different physical boards. Shelton *et al.* [4] improves the side channel model of ELMO by capturing interactions that span multiple cycles. ELMO [5] is augmented to account for storage elements, which play a critical role in the security of masked implementations. A novel feature of ELMO* [4] is a systematic battery of small code sequences that can be used to highlight the interaction of instructions through storage elements. The idea of finding hidden storage elements is generic and could be used for any other architecture. Bazangani *et al.* [9] propose a new leakage simulator ABBY, for the ARM-Cortex M0 architecture based on machine learning. The advantage of ABBY compared to ELMO is twofold. The first advantage is that no reverse engineering of the target device is required, and the second is that ABBY can learn nonlinear leakage models. Arora *et al.* [3] compare the manufacturing variability between different physical devices from the same manufacturer. The study targets an ARM-Cortex M0 architecture and shows that the power trace signature of a sequence of instructions depends on microarchitecture implementation. The implication of this work is that the existence of a generic side-channel disassembler, which is identified with high-accuracy instructions on ARM-Cortex M0 cores produced by different manufacturers, is improbable.

3 Experimental setup

For training or profiling, power traces are collected from an ARM-Cortex M0 microcontroller, STM32F0 Discovery (STM32F051R8). The CPU is clocked at 8 MHz. To improve the signal-to-noise ratio in the measured power traces, the capacitors between VDD and GND are removed, since they reduce the signal-to-noise ratio in the power traces (Figure 1a). The oscilloscope is set to 1.25GS, resulting in 156.25 samples per cycle. The power consumption of the board is measured using an AC current probe since it ignores the DC component, which can vary between different measurements. An overview of the setup can be found in Figure 1b.



(a)

(b)

Fig. 1: Overview of the setup used to collect the traces. (a) Frontal view of the STM32F030R8 board, where the removed capacitors: C18, C19, C20, C21 are highlighted. (b) Schematic overview of the setup used to collect traces.

4 The Datasets

The ARM-Cortex M0 implements the ARMv6-M instruction set, which consists of most of the 16 bit Thumb instructions and some of the 32 bit Thumb-2 instructions. For this project, we select the core instructions relevant for cryptographic operations, similar to [5], who observed that the power consumption of the selected instructions can be divided into five different groups by performing a cluster analysis. The resulting groups and instructions are shown in Table 1.

Group 1 (ALU)	`adds, ands, cmp, eors, movs, orrs, subs`
Group 2 (SHIFTS)	`lsls, lsrs, rors`
Group 3 (LOADS)	`ldr, ldrb, ldrh`
Group 4 (STORES)	`str, strb, strh`
Group 5 (Multiplications)	`muls`

Table 1: Overview of the division of instructions into groups.

Since the microcontroller board has a limited amount of memory, the data sets collected consist of multiple *acquisitions*. The *acquisitions* are created from multiples *programs*. A *program* is a sequence of assembly instructions. For our data sets, a *program* is constructed as follows: ten `nop`s, two *random instructions*, one *target instruction* followed by two *random instructions* and ten `nop` instructions. An example can be seen in Figure 2.

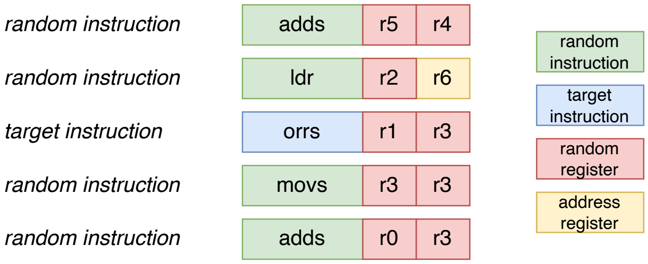


Fig. 2: Snippet of a *program*. Note that 10 `nop`s are executed before and after this fragment to ensure an accurate acquisition.

The `nop` instructions do not use operands. The operands used for the other instructions in *program* are random values in random registers. Since loads and stores instructions need an actual memory address to load from and store to, one register (`r6`) is reserved for this and filled with an existing memory address. The other free registers (`r0-r5`) are filled with new random 32-bit values before each program is executed.

Three datasets are created for different purposes. To be able to apply the proposed feature selection in [6], dataset A is created that matches the settings required for this special feature selection. Dataset B is created for the recognition of the five instruction groups, and dataset C is created for the recognition of instructions within the largest instruction group: group 1.

Dataset A contains sixty *programs* targeting three instructions. Two additional programs consisting of `nop` instructions are used as a baseline to remove electrical noise. The three *target instructions* are: `adds, ands` (from group 1) and `muls` (group 5). For each *target instruction*, we generate 20 programs by randomizing the *random instructions*. The *random instructions* are taken from groups 1, 2, and 5. For each *program* 300 power traces are acquired, with 6000 samples per trace. The collection of traces is done in one acquisition campaign.

Dataset B contains a total of 12,500 *programs*, with *target instructions* in the five groups. Each group contains 2,500 *programs*. The *target instruction* is randomly selected from all instructions in the group. For each *program*, 20 traces are collected, with 6000 samples each. An average is taken over these 20 power traces to reduce electrical noise. Only 500 *programs* fit into the memory of the board, so the data set consists of 25 different acquisitions.

Dataset C contains 17,500 *programs* targeting instructions from group 1. For each of the seven *target instructions* in the group, 2500 *programs* are created. For each *program*, 20 traces are collected with 6000 samples each. To reduce electrical noise, an average is taken over these 20 power traces. The data set is collected in 35 acquisition campaigns.

5 Selecting the mixed-instruction sequence

After acquiring the traces, we want to determine the samples in the trace related to the assembly code executed. Since the ARM-Cortex M0 has a three-stage pipeline, we selected three cycles (or the equivalent of 469 samples) since each of the stages can contribute to the power usage of the target instruction. The collected traces have 6000 samples, but we do not know at which samples our assembly is being executed. With our setup, it is not possible to calculate the time between the trigger and the moment our assembly code starts executing. In the power traces, we can see the influence of the executed assembly, but since we have a three-stage pipeline (Figure 3) we do not know which stage of the pipeline causes the change in power consumption or which instructions are in the pipeline at that exact moment.

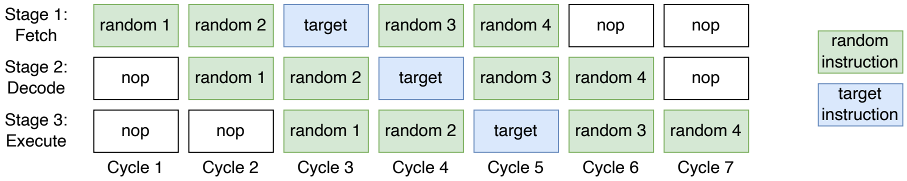


Fig. 3: Executed instructions in the pipeline. The target instruction is expected to influence the power trace in cycle 3-5.

To select the samples in the power trace corresponding to *target instruction*, we explore two techniques. The first is *sample-eviction*, which works by removing a window of three cycles from the traces by replacing them with zero values and then calculating the classification score. By evicting samples at different intervals in the trace, we expect the lowest score [1] to indicate the location of the most important samples. The second technique is *moving-window*, where we calculate the classification score in a moving window of three cycles. The highest score indicates the three cycles that contain the most useful information for the machine learning model. For both techniques, we chose to use a Multilayer Perceptron model as discussed in subsection 6.1.

The top graph in Figure 4 shows a processed power trace where noise and `nop` were subtracted to give a better visualization of where random and target instructions influence the power trace. The middle and bottom graphs in Figure 4 show the result of the *sample-eviction* and *moving-window* technique, respectively. Note that, for both techniques, the score for a given sample is calculated over the 469-sample window, which starts at that specific sample. The

[1] a poor classification score indicates the relevant samples are missing

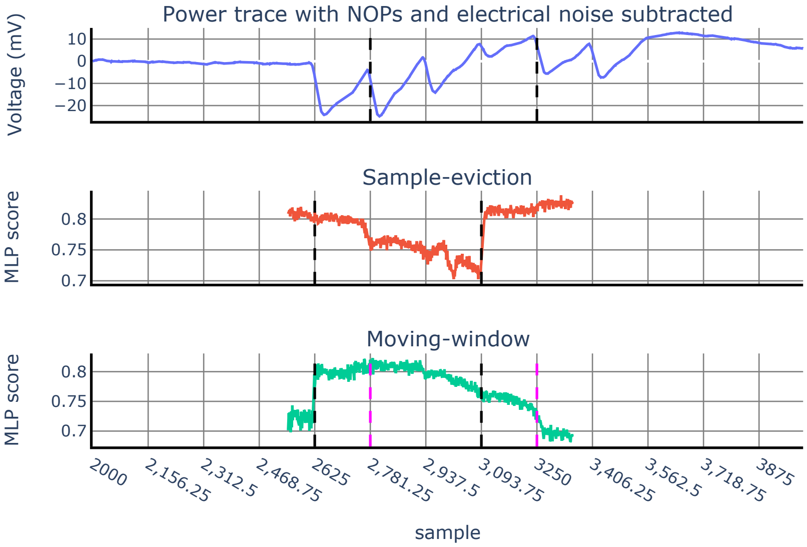


Fig. 4: Power-trace (top) and the applied techniques for selecting the samples to be used in further analysis (middle and bottom).

gray lines are plotted with intervals of one cycle, or 156.25 samples. The starting point of the cycle is not known, so the lines could be out of phase.

There are different options to interpret the available information. Looking at the top graph of Figure 4 there are six cycles clearly influenced by the executed assembly. An estimation could be that the pipeline fetch stage uses the least power, so the six cycles would be cycle 2-7 in Figure 3. The corresponding cycles where the target instruction is in the pipeline are marked with black dashed lines on the top graph of Figure 4.

In the middle graph Figure 4, *sample-eviction* the score is lowest just before sample 3100 and begins to increase rapidly after that. This could indicate that the most important cycles are happening before that moment (between the black dashed lines). Using the *moving-window* technique, we could use the three cycles just after the significant increase in the score (between black dashed lines) or the three-cycle window used to calculate the maximum score (between purple dashed lines).

These techniques and information do not give a clear location for the most important samples in the power trace. Since the machine learning algorithm used will receive a 469 sample input similar to *moving-window*, we take the maximum score for *moving-window* in sample 2780 as a starting point for our experiments.

To check whether other samples would contain additional information, we explore using more than 469 samples as input in subsection 6.2.

For the selection of KL divergence-based features, the location of relevant samples in the trace is followed by the continuous wavelet transform and KL divergence. In section A (Appendix), this is discussed in more detail.

6 Experimental results

6.1 Overview of algorithms used for training and classification

The machine learning models used are Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), Multilayer Perceptron (MLP), and Convolutional Neural Network (CNN). We use LDA and QDA to implement two of the models used in [6]. The package `sklearn.discriminant_analysis` is used for Python implementation. MLP and CNN models are often used for side-channel analysis. The models used are simple and created using the `tensorflow.keras` Python package. The details of MLP and CNN can be found in Table 2 and Table 3.

Layer type	Details
Dense	units=200, activation=selu
Dense	units=200, activation=selu
Dense	units=200, activation=selu
Dense	units=200, activation=selu
Dense	units=classes, activation=softmax

Table 2: MLP

Layer type	Details
Conv	filters=8. kernel size=20, activation=relu
Flatten	
Dense	units=128, activation=relu
Dropout	
Dense	units=128, activation=relu
Dense	units=classes, activation=softmax

Table 3: CNN

Before computing the MLP and CNN scores, the data have to be normalized first. This is done using `sklearn.preprocessing.StandardScaler`. For both the MLP and CNN models, they are set on 100 epochs, since the accuracy did not increase for more epochs. For all machine learning models, the scores are calculated using 5-fold cross-validation.

6.2 Choosing the configuration for the dataset

Most instructions take one cycle to execute; however, all loads and stores take two cycles. This means that the starting sample of *target instruction* can change depending on whether one or both of the two random instructions that precede the target instruction are `load` or `store`. To overcome this problem, while loading the traces, we check whether `load/stores` occurs before *target instruction* and increase the offset by the right amount of samples if there are. In Table 4 can be seen that for each of the machine learning models the score increases, for LDA and QDA there is even a significant improvement when adjusting the offset.

	LDA	QDA	MLP	CNN
Normal offset	65.9%	50.6%	80.4%	79.6%
Adjusted offset	84.4%	70.7%	85.6%	87.8%

Table 4: Dataset B. Offset vs. none

When different acquisitions are run, the power traces can be slightly different due to variables such as temperature. To check whether this influences the scores for our dataset, we ran scores for three different configurations. The first configuration uses only one acquisition file per group, resulting in 500 programs per group. The second configuration uses the complete datasets with five acquisition files per group, and the training and testing parts are taken randomly. For the last configuration, the training part consists of 4 acquisition files per group, and the last file is used for testing. The results for each configuration can be found in Table 5. Note that for the highlighted cell the QDA calculation warned that the variables are collinear, so this result should not be considered accurate. The partial data set performs worse than the complete dataset, so increasing the number of programs increases the accuracy despite adding multiple acquisition files. When using configuration 3 the scores are similar to configuration 2, so the influence of changing environmental variables on different acquisitions seems to be limited.

	LDA	QDA	MLP	CNN
Configuration 1: partial dataset	81.0%	31.3%	78.9%	76.9%
Configuration 2: complete shuffled	84.4%	70.7%	85.6%	88.3%
Configuration 3: complete	84.3%	71.0%	85.8%	88.1%

Table 5: Dataset B. Machine learning scores for different input configurations.

Since we do not have a perfect method for selecting the right samples in the power traces, we compare different amounts of samples in Table 6. Note that again the highlighted cell should not be considered accurate, since its calculation gave a collinearity warning. 469 samples selected using the sample-eviction and moving window, 781 samples using the previous selection and the cycle before and after that, and 3000 samples to ensure that all our assembly is in the selection. The differences between 469 and 781 samples are relatively small, which could indicate that our feature selection is close to the actual most important samples. The two additional instructions covered by the extra samples barely increase the scores. When taking a large 3000 sample selection, the scores are close to the other selections or even significantly lower in the case of MLP.

Samples	LDA	QDA	MLP	CNN
3000	84.9%	31.4%	79.6%	88.5%
781	85.4%	67.3%	88.8%	88.8%
469	84.4%	70.7%	87.9%	87.9%

Table 6: Dataset B. Using different amounts of samples.

For the next sections, we will use the all the acquisition files in the datasets with shuffled traces, an adjusted offset and a selection of 469 samples.

6.3 Amount of traces per program

In section 4 is indicated that for each program in dataset B and C an average trace is taken over 20 traces. In Table 7 can be seen what the scores are with different approaches than averaging. When taking only a single trace, the score drops significantly. If we use all 20 traces for machine learning, the score increases to 93.7%. However, in this scenario (*identical-program*) the traces for training and testing are taken randomly, which means that testing can be done on traces generated with the same program as some of the traces used for training. When making the train-test division based on program rather than traces, this problem is avoided, but the score drops to 77.8% (*different-program*). This means that averaging the 20 traces results in the best score (84.7%) for a realistic scenario.

Traces per program	Method	MLP score
1	-	79.5%
20	*identical-program*	93.7%
20	*different-program*	77.8%
20	averaged	84.7%

Table 7: Dataset B. Using different methods to divide all traces into training and testing sets.

6.4 Training and classification for groups of instructions

The scores for the classification of different groups are given in Table 8. Note that the CNN score for the first row is not given since the number of input variables after the selection of features is too low. The KL-based feature selection can only be applied to dataset A, for which the scores are very close to random guessing (50%). However, when we use the same samples for our analysis, the score increases to 99.9%. This means that there is enough information in the samples to get an almost perfect classification score, but the KL-based feature selection cannot extract this information. Possible reasons for this can be found in section A (appendix). However, this is a best-case scenario (*identical-program*), where two instructions from different groups are compared with a data set that contains power traces in its training and testing set that are based on the same program. To create a similar but more realistic scenario (*different-program*) for dataset B we took only two groups instead of all five in row three (*Dataset B (group 1 vs group 5)*), and this still gives a accuracy of 94.5%. When using the complete dataset, the accuracy drops to a maximum score of 88.2%.

Dataset	Feature selection	LDA	QDA	MLP	CNN
Dataset A (adds vs muls)	yes [6]	50.3%	50.2%	50.1%	-
Dataset A (adds vs muls)	no	69.2%	66.8%	99.9%	99.8%
Dataset B (group 1 vs group 5)	no	95.4%	77.2%	93.4%	95.4%
Dataset B (all groups)	no	84.4%	70.7%	86.4%	88.2%

Table 8: Dataset A vs. B groups.

The confusion matrix in Figure 5 shows the MLP result for all groups in dataset B. Since the training and testing data are randomly divided, the expected amount of traces per group, and therefore the maximum score in the matrix, is 500. It can be seen that the score is the worst for group 2 (shifts). The `loads` and `stores` can be distinguished best. Although they can be distinguished on the basis of their two-cycle duration compared to the one-cycle duration of the other instructions, loads are also not classified as stores or the other way around.

6.5 Training and classification results for individual instructions

The scores for the classification of different instructions are given in Table 9. Again, KL-based feature selection can only be applied to dataset A. The scores for KL-based feature selection are just above random guessing of instructions, but when machine learning is used on the same traces, the score increases to 99.9% (*identical-program*). This shows that although KL-based feature selection performs better for instructions than for groups, there is still a lot of information in the dataset. Moving to a more realistic scenario (*different-program*), however, in dataset C, the machine learning models perform significantly worse than for

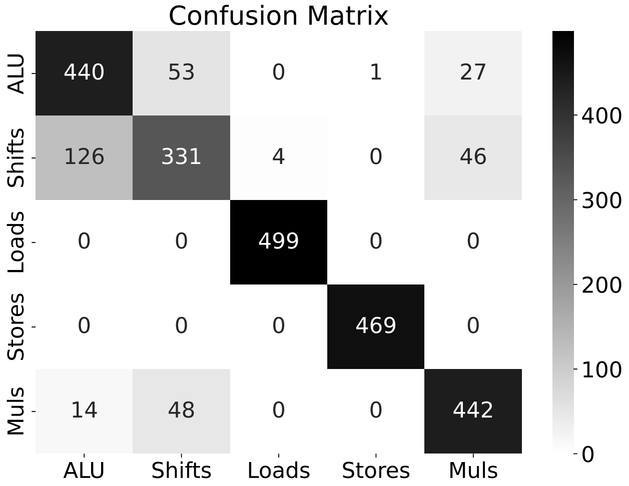


Fig. 5: Confusion matrix groups. Row indicates true label, column indicates predicted label.

groups with maximum scores of 58.1% for two classes and 25.5% for seven classes. Note that the expected score for random guessing is 50% and 14.3% for two and seven classes, respectively.

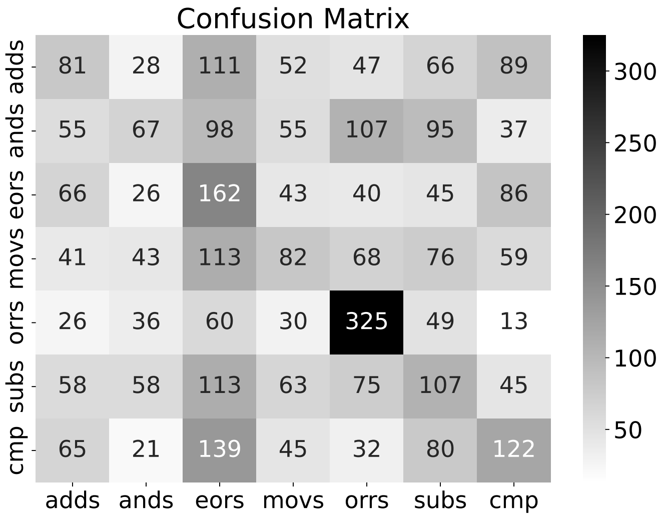
Dataset	Feature selection	LDA	QDA	MLP	CNN
Dataset A (adds vs ands)	yes[6]	56%	55.4%	51.6%	51.8%
Dataset A (adds vs ands)	no	88.1%	78.5%	99.9%	99.9%
Dataset C (adds vs ands)	no	54.7%	49.8%	58.1%	51.8%
Dataset C (all instructions)	no	20.1%	15.5%	25.5%	25.2%

Table 9: Dataset A vs. C instructions.

The confusion matrix in Figure 6 shows an MLP result for all instructions in dataset C. Since the training and testing data are randomly divided, the expected amount of traces per instruction and therefore the maximum score in the matrix is 500. Since the score is significantly lower than the score for groups, the matrix shows a lot of false positives and false negatives. The only instruction classified correctly in more than 50% of the cases is `orrs`.

6.6 Discussion

We use the term *identical-program* to describe the classification results obtained when traces of the same *program* are used to train the model and report the



	adds	ands	eors	movs	orrs	subs	cmp
adds	81	28	111	52	47	66	89
ands	55	67	98	55	107	95	37
eors	66	26	162	43	40	45	86
movs	41	43	113	82	68	76	59
orrs	26	36	60	30	325	49	13
subs	58	58	113	63	75	107	45
cmp	65	21	139	45	32	80	122

Fig. 6: Confusion matrix instructions. Row indicates true label, column indicates predicted label.

classification results. We use the term *different-program* to describe the classification results obtained when traces of different *programs* using the same *target instruction* are used to train the model and report the classification results. The *different-program* setting is more challenging compared to *identical-program* for classification, but is also more realistic.

Our side-channel disassembler reached a success rate of 99% in the *identical-program* setting, which is in line with most state-of-the-art results reported in the literature. However, we observe a decrease in the success rate (95.4%) when using the *different program* strategy. When we include all five groups, our model success rate reaches 88.2%.

We observe the same behavior when analyzing results related to instruction classification. Our side-channel disassembler reached a success rate of 99.9% in the *identical-program* setting when used to distinguish between two instructions (in the same group). The same classification task in the *different program* strategy results in a success rate of 58.1%. However, when we include all the instructions, our model success rate reaches only 25.5%.

7 Conclusions and future work

In this work, we present the first side-channel disassembler that targets the ARM-Cortex M0, a 32-bit microcontroller. Previous side-channel disassemblers target simple 8-bit architectures. We show that the training and classification strategies used have a substantial impact on the performance reported in the model. Under ideal conditions, our side-channel disassembler reaches a success rate of 99%, while under realistic conditions, we observe a success rate of 88.2%. To our surprise, the use of sophisticated methods for feature selection did not prove helpful, and the best results we obtained with unprocessed features. As a result, creating data sets is a simpler task. The present study only examined relatively simple deep learning models, which we did not optimize for the task. Therefore, we believe that more advanced deep learning architectures can improve the results.

8 Acknowledgments

This work received funding in the framework of the NWA Cybersecurity Call with project name PROACT with project number NWA.1215.18.014, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

References

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, M. J. Wiener, Ed., ser. Lecture Notes in Computer Science, vol. 1666, Springer, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25. [Online]. Available: https://doi.org/10.1007/3-540-48405-1%5C_25.

[2] J. Park, F. Rahman, A. Vassilev, D. Forte, and M. Tehranipoor, "Leveraging side-channel information for disassembly and security," *J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 1, Dec. 2019, ISSN: 1550-4832. DOI: 10.1145/3359621. [Online]. Available: https://doi.org/10.1145/3359621.

[3] V. Arora, I. Buhan, G. Perin, and S. Picek, "A tale of two boards: On the influence of microarchitecture on side-channel leakage," in *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*, V. Grosso and T. Pöppelmann, Eds., ser. Lecture Notes in Computer Science, vol. 13173, Springer, 2021, pp. 80–96. DOI: 10.1007/978-3-030-97348-3_5. [Online]. Available: https://doi.org/10.1007/978-3-030-97348-3%5C_5.

[4] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards automatic elimination of power-analysis leakage in ciphers," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/rosita-towards-automatic-elimination-of-power-analysis-leakage-in-ciphers/.

[5] D. McCann, E. Oswald, and C. Whitnall, "Towards practical tools for side channel aware software engineering: Grey box' modelling for instruction leakages," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17, Vancouver, BC, Canada: USENIX Association, 2017, pp. 199–216, ISBN: 9781931971409.

[6] J. Park, X. Xu, Y. Jin, D. Forte, and M. Tehranipoor, "Power-based side-channel instruction-level disassembler," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465848.

[7] T. Eisenbarth, C. Paar, and B. Weghenkel, "Building a side channel based disassembler," *Trans. Comput. Sci.*, vol. 10, pp. 78–99, 2010. DOI: 10.1007/978-3-642-17499-5_4. [Online]. Available: https://doi.org/10.1007/978-3-642-17499-5%5C_4.

[8] M. Msgna, K. Markantonakis, and K. Mayes, "Precise instruction-level side channel profiling of embedded processors," in *Information Security Practice and Experience*, X. Huang and J. Zhou, Eds., Cham: Springer International Publishing, 2014, pp. 129–143, ISBN: 978-3-319-06320-1.

[9] O. Bazangani, A. Iooss, I. Buhan, and L. Batina, *Abby: Automating the creation of fine-grained leakage models*, Cryptology ePrint Archive, Report 2021/1569, https://ia.cr/2021/1569, 2021.

A Discussion KL-based feature selection

In this section, we explain the feature selection method proposed by [6], with which we experimented extensively. After discussing the method in detail, we go over possible issues causing the bad performance. The KL-based feature analysis consists of two steps. The first step is pre-processing using the continuous wavelet transform, and the second step is feature selection using KL divergence. The input for KL-based feature selection is (number of traces, number of samples), resulting in output shape (number of traces, number of features).

A.1 Background

Continuous Wavelet Transform(CWT) is used to transform traces from the time domain to the time-frequency region. The wavelet used is a standard Ricker wavelet included in the `scipy.signal` package. The width used for the `scipy.signal.cwt` function is 50. The result is a two-dimensional array of shapes (50,469). This means that for each power trace, we end up with $50 * 469 = 23450$ data points. These data points are used as input for the next step: feature selection.

KL-based feature selection Kullback-Leibler (KL) divergence is the statistical distance between two probability distributions. This means that before we can use the KL divergence, the acquired data has to be transformed to probability distributions. This has to be done for each of the 23450 data points in the processed power traces. When comparing two programs or target instructions, for each data point the probability distribution is taken over the 300 (program) or 6000 (target instruction) traces. Computing the probability distributions is done with `numpy.histogram` using the Freedman-Diaconis rule for determining the bin width. For the actual KL divergence calculation we use the `scipy.special.kl_div` function.

The resulting two-dimensional array has the same shape (50,469) as all input power traces after the CWT. On the basis of the KL-divergence values at each of the sample points in this array, the features to be used in machine learning can be selected.

Not-varying feature points For each target instruction the KL divergence is computed for each unique combination of its programs. This results in 190 KL divergence arrays. To select points with a low KL divergence value, for each of the arrays a list of coordinates (*list*) is created that includes only the sample points that have a value below a certain threshold. The *not-varying feature points* are selected using Equation 1.

$$NVP_{target} = list_1 \cap list_2 \cap \cdots \cap list_{190} \tag{1}$$

Distinct points Between the different target instructions, the KL is also computed, using all the programs together instead of comparing the programs. For each of the 23450 sample points in the power traces, the probability distribution is taken over the $20 * 300 = 6000$ power traces. Since there is only one combination for which the KL divergence has to be computed, the result is just one array compared to the 190 for *not-varying feature points.* To avoid collinearity, only local maximum values are used instead of taking points above a certain threshold [6]. A list of the sample points that have a local maximum value is computed; this list is called $DP_{targetA\,vs.\,targetB}$.

The final result of the selection of features is a combination of *not-varying feature points* and *distinct points*:

$$feature\,points = NVP_{targetA} \cap NVP_{targetB} \cap DP_{targetA\,vs.\,targetB} \tag{2}$$

The selected points should not vary much when the same target instruction is executed, but should vary much when different target instructions are executed and therefore contain much information for classification.

A.2 Results of feature selection

The final amount of *feature points* is determined using Equation 2. The amount of *not-varying feature points* depends on the threshold used, while the amount of *distinct points* is fixed. The latter therefore is the limiting factor for the amount of resulting *feature points.* The different number of points to compute feature points for `adds` vs. `muls` and `adds` vs. `ands` is given in Table 10.

Threshold	`adds` vs. `muls`	`adds` vs. `ands`
Not-varying feature points	795	1540
Distinct feature points	155	435
Feature points	7	112
LDA score	50.3%	56%

Table 10: Results for KL-based feature selection

The first point of interest is the low amount of *feature points* for `adds` versus `muls`, but doubling the KL threshold to 0.8 only increases the amount of *feature points* to 8. The low amount of *feature points* could be an influence for the low results, however `adds` vs. `ands` does not perform much better with 112 *feature points*.

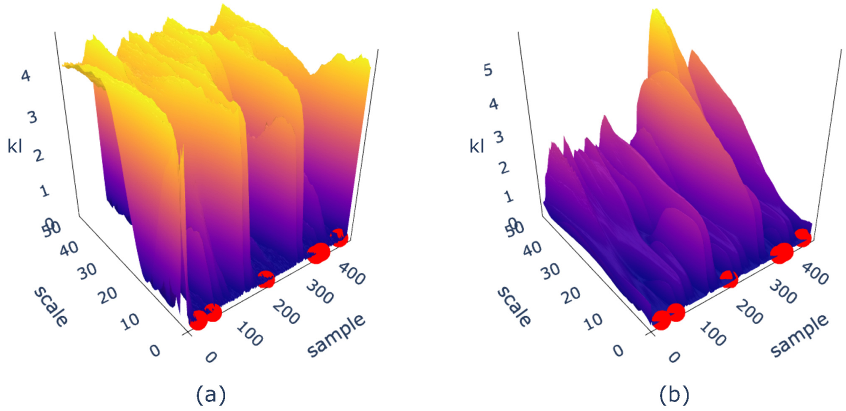


Fig. 7: KL divergence graphs with selected *feature points* for `adds` vs. `muls`. (a) `muls` program 1 vs. program 2. (b) `adds` vs. `muls`.

When plotting *feature points* in the KL divergence graphs, one of the possible causes for the low classification rates can be seen. The *not-varying feature points* are selected to be below a certain threshold and therefore should have a low KL value. This is true for both comparisons, as can be seen in Figure 7a and Figure 8a. However, when looking at *distinct points* (Figure 7b and Figure 8b) the selected *feature points* also have a very low KL value, whereas they should have a high KL value.

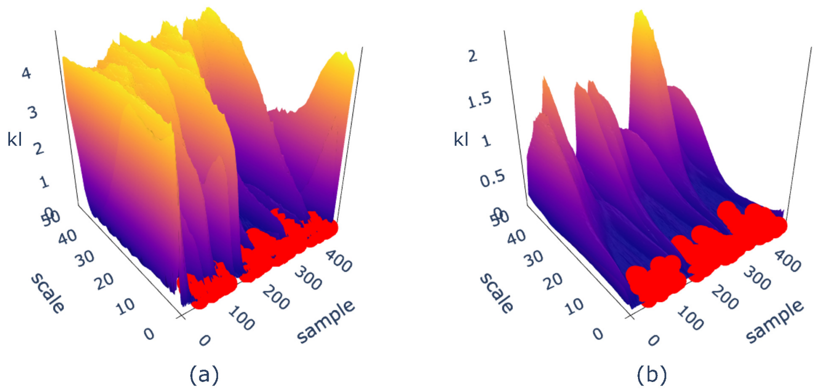


Fig. 8: KL divergence graphs with selected *feature points* for `adds` vs. `ands`. (a) `ands` program 1 vs. program 2. (b) `adds` vs. `ands`.

The low KL values for *distinct points* are possible because there is no threshold for *distinct points* to be above. The only requirement is that the selected points be a local maximum. This does not exclude points with a high KL value, but they are not present. When comparing these graphs with the results of [6], we notice that the shape of our figures is different. Whereas the high KL values for both comparing target instructions and comparing programs with the same target instruction are located mainly on the higher scales, this is different for the results in [6]. For their results, the low KL values for comparing different programs with the same target instruction are located at the same scales as the high values for comparing different target instructions. The cause of this could be related to the exact implementation, which the authors of [6] do not specify, or due to architectural differences between the different microcontrollers.