A Deep-Learning Approach to Side-Channel Based CPU Disassembly at Design Time

Hedi Fendri[*][†], Marco Macchetti[†], Jérôme Perrine[†] and Mirjana Stojilović[‡]
[*]Universita della Svizerra italiana, Lugano, Switzerland
[†]Kudelski Group, Cheseaux-sur-Lausanne, Switzerland
[‡]EPFL, Lausanne, Switzerland, Email: mirjana.stojilovic@epfl.ch

***Abstract*—Side-channel CPU disassembly is a side-channel attack that allows an adversary to recover instructions executed by a processor. Not only does such an attack compromise code confidentiality, it can also reveal critical information on the system's internals. Being easily accessible to a vast number of end users, modern embedded devices are highly vulnerable against disassembly attacks. To protect them, designers deploy countermeasures and verify their efficiency in security laboratories. Clearly, any vulnerability discovered at that point, after the integrated circuit has been manufactured, represents an important setback. In this paper, we address the above issues in two steps: Firstly, we design a framework that takes a design netlist and outputs simulated power side-channel traces, with the goal of assessing the vulnerability of the device at design time. Secondly, we propose a novel side-channel disassembler, based on multilayer perceptron and sparse dictionary learning for feature engineering. Experimental results on simulated and measured side-channel traces of two commercial RISC-V devices, both working on operating frequencies of at least 100 MHz, demonstrate that our disassembler can recognize CPU instructions with success rates of 96.01% and 93.16%, respectively.**

***Index Terms*—disassembly, deep learning, design time, embedded processors, power side channel**

I. Introduction

Thanks to the Internet of Things (IoT), modern embedded devices have pervaded many aspects of our lives. However, being deployed in places accessible to many people, these devices are at constant risk of a variety of attacks. For the designers of embedded systems, one of the greatest concerns is reverse engineering of their proprietary software (i.e., *disassembly*) for theft, piracy, or security breach [1], [2]. To protect their intellectual property, designers commonly use encryption and keep the code and data in tamper-resistant memories, except that, once the code is decrypted and starts running, the system becomes vulnerable to side-channel attacks.

Electrical side channels are inherent to all digital devices, due to the presence of mutual information (MI) between the executed code and the processed data, on the one hand, and the resulting power consumption and electromagnetic (EM) field emanations, on the other. Several techniques exist for extracting the information leaked through an electrical side channel: simple and differential power analysis [3], correlation power analysis (CPA) [4], template attacks [5], MI analysis [6], and machine learning (ML) [7]. A large body of research investigated side-channel analysis (SCA) for extracting secret keys from cryptographic primitives [3], leaving the side-channel disassembly as a considerably less-explored topic.

Deploying side-channel countermeasures at the device level is crucial, but doing it requires in-house security and integrated circuit (IC) design expertise. Evaluating the security of an IC is typically done at the silicon level (i.e., post manufacturing), when any discovered side-channel vulnerability causes significant setbacks—costly hardware redesign and considerably longer time-to-market. Consequently, for the IC design and security teams, it is of utmost importance to be able to reliably evaluate the efficiency of their countermeasures and the side-channel security of their devices *at design time*.

ML approaches have been shown particularly adapted for side-channel disassembly, whether for malicious purposes [8]–[15] or for system protection [16], [17], because the dimensionality of the problem and the size of the data to be recovered is much higher compared to cryptanalysis. Previous works mainly targeted simple microprocessors (e.g., MSP430, PIC16, ATmega), with a minimal number of pipeline stages and operating frequency in the order of several MHz only.

In this paper, we present a solution to side-channel disassembly at design time (Fig. 1). Firstly, we develop a general and portable framework that takes an integrated circuit design described in a hardware description language to model its static and dynamic power consumption at each clock cycle, and generate noise-free power-consumption traces at design time. Secondly, we develop a novel deep-learning-based disassembler that uses dictionary learning with sparse coding for feature selection and dimensionality reduction. For testing, unlike previous works, we choose a RISC-V processor (designed in-house) and perform experiments on simulated power side-channel traces. We achieve 93.16–96.01% accuracy for instruction classification and 88–100% F1 score for bit-level classification. Finally, we validate the suitability of deep-learning side-channel based disassembler using a GD32VF103 RISC-V core [18] and EM side-channel traces, achieving 93.16–94.5% accuracy in instruction classification.

II. Related Work

Quisquater and Samyde were among the first to demonstrate automated code disassembly on a Zilog Z80 processor, using power and EM traces [8]; their use of a Kohonen neural

This work is partially supported by Kudelski Group and the European Union Horizon 2020 research and innovation program under CPSoSAware project (grant No. 871738).

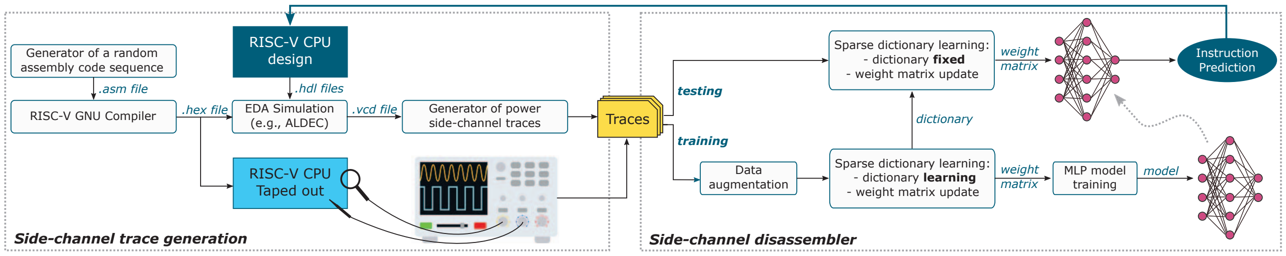

Generator of a random assembly code sequence
.asm file
RISC-V GNU Compiler
.hex file
RISC-V CPU design
.hdl files
EDA Simulation (e.g., ALDEC)
.vcd file
Generator of power side-channel traces
RISC-V CPU Taped out
Traces
Side-channel trace generation
testing
training
Sparse dictionary learning: - dictionary fixed - weight matrix update
weight matrix
Instruction Prediction
dictionary
Data augmentation
Sparse dictionary learning: - dictionary learning - weight matrix update
MLP model training
model
Side-channel disassembler

Fig. 1: System block diagram. To the left, trace collection (through simulations or measurements) is shown. To the right, our side-channel disassembler based on multilayer perceptron and dictionary learning with sparse coding is illustrated.

network (NN) was indicative of the observed trend of employing ML in subsequent research on side-channel disassembly. Eisenbarth et al. instead relied on hidden Markov models and Viterbi algorithm to build a power side-channel disassembler targeting PIC16F687; they reported 70% recognition rate on test and 58% on real codes [9]. Strobel et al. also targeted PIC16F687 [14]. Polychotomous linear discriminant analysis was used for preprocessing, to emphasize the particularly distinctive features in the observed data, which was classified into various classes by the k-NN classifier. They achieved up to 88% recognition rate on real codes; however, they used multiple EM antennas and decapsulated the chip package, thus leaving traces of the attack and increasing its complexity. Tsaguae and Twala [19] achieved 78.3% classification success rate on an ATMega163, using electromagnetic emanations, principal component analysis (PCA), and k-NN algorithm. Park et al. targeted AVR ATmega328p [10], where most of the instructions take one clock cycle. Taking advantage of continuous wavelet transform to observe differences between instructions not observable in the time domain, Park et al. then applied KL-divergence to select the important features, PCA for dimensionality reduction, and a hierarchical approach to ML classification. They reported almost 99.03% recognition accuracy on the test codes. Cristiani et al. [15] used many EM probe locations and PIC16F15376 as the target device. They focused on the fetch stage, while using quadratic and linear discriminant analysis (QDA and LDA, respectively) to extract trace samples that contain relevant information. They reported 99.41% recognition rate at bit-level and 95% with the full 14-bits instructions. Krishnankutty et al. found the optimal sequence of instructions, in terms of clock cycles per instruction on MSP430 chip, using empirically created templates together with dynamic programming [11]. To build templates, they aggregated data collected from several power pins of the chip. With support vector machine (SVM) classifier, they performed coarse-grained and fine-grained classification, and reported 86% recognition accuracy of the instruction opcode on real codes. Vafa et al. [12] built a hierarchical framework for instruction recognition on an 8-bit PIC16F690 and LPC1768, which includes ARM Cortex-M3 as a 32-bit platform (the core can operate at frequencies up to 100 MHz, but the authors do not say at which frequency they had it run in the experiments). They targeted 16-bit Thumb instructions, common in symmetric cryptography. Using continuous wavelet transform to map data from time to frequency domain, align traces, and remove noise, together with KL divergence and PCA, they obtained 98% recognition rate.

In this work, we target significantly more complex designs: RISC-V processor cores, working on frequencies of 166 MHz and 100 MHz, and having a six-stage and two-stage pipeline, respectively. Additionally, we do not profile individual instructions, but rather generate very long sequences of randomly selected instructions from the instruction set architecture, for profiling as well as for testing. Hence, our evaluation is comparable to testing on an example of a real code. Finally, we combine deep learning with sparse dictionary learning (SDL) to build a side-channel disassembler.

III. Threat Model

The threat model of a side-channel based disassembly attack (illustrated in Fig. 2) assumes an adversary who has access to two devices: one being a *template* device, which the attacker can manipulate to program an arbitrary code sequence, and one being a *target* device. The template and the target device are typically considered to be of the same series and by the same manufacturer, and assumed to have similar leakage characteristics. While the template device is running the code, the adversary measures and records a number of side-channel traces. With the acquired data set, the adversary then builds a side-channel disassembler (e.g., by training a deep-learning model to classify the instructions). Once the disassembler is ready (in our case, the deep-learning model is trained), the attacker turns to the target device and collects side-channel traces while it runs a confidential code. Finally, the adversary attempts to recover the sequence of the instructions being executed by the target device. This threat model is most relevant for easily accessible targets, such as IoT devices.

IV. Trace Collection at Design Time

Implementing side-channel attack countermeasures is a very challenging task, frequently faced by hardware security engineers. To assess the efficiency of the countermeasures, the hardware design is first manufactured (as ASIC) or implemented as a prototype (e.g., on an FPGA), and then subjected to extensive side-channel attack tests. Waiting for the ASIC manufacturing and side-channel analysis results takes months

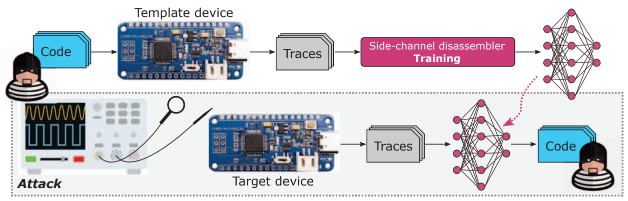


Fig. 2: Threat model of a side-channel disassembly attack.

and is, without doubt, very costly. Even if prototyping cuts the time and expenses, designers need the whole process to be even faster. In this context, a solution for closing the side-channel verification loop at design time, without the need to have the physical device at hand, is greatly appealing [20]. In this section, we present our solution to the above problem, in the context of RISC-V based designs: firstly, we discuss an automated approach for fast power side-channel trace simulation. Then, we describe how we collected the side-channel traces for a RISC-V processor design, which also served as our side-channel disassembler target device.

A. Power Modelling and Side-Channel Trace Simulation

For side-channel traces to be generated at design time, the target design described using a hardware-description language (e.g., VHDL, Verilog, etc.) needs first to be simulated. In this work, without loss of generality, we used ALDEC EDA tool to perform gate-level simulation with back-annotation of the target design and export the simulation results in a Value Change Dump (VCD) file. The VCD file contains values of all signals and states, at every simulated time step. To generate the power side-channel trace from a VCD file, we implemented a power-consumption simulator, which parses the file and, for every time step, computes an estimate of the static or the dynamic power consumption, applying Hamming weight or Hamming distance power-estimation models (commonly used in the SCA community). Being time-efficient, our approach is particularly suitable for use prior to considering advanced simulations (e.g., SPICE) or more complex models (e.g., CASCADE [20]). The simulator exports the traces in a Hierarchical Data Format (HDF), allowing for saving the power consumption traces and the metadata in the same file.

B. Data-Set Creation for Our RISC-V Design Target

As a target, we chose an in-house design of a RISC-V CPU core, having a six-stage pipeline and each stage lasting one clock cycle. Consequently, we will make a simplifying assumption that an instruction can be well characterized by a trace that is six clock cycles long. To compute the simulated power side-channel traces and their corresponding labels we apply the following set of steps.

1) Assembly code generation: We generate a sequence of *randomly* chosen instructions from the RISC-V instruction set architecture. Both the instructions and operands are chosen randomly, and our software tries to balance the number of different instructions. Implementing randomized selection and generating a long instruction sequence, solves the problem of having to run the side-channel disassembler on both the test and the real codes, common in previous works [10].

2) Code compilation: We use RISC-V GCC compiler from the RISC-V GNU toolchain to generate `.hex` files. Given that their format is not compatible with the format expected by the ALDEC EDA tool, we wrote a parser to read, adjust, and output an equivalent and compatible `.hex` file.

3) CPU programming: The `.hex` file is loaded to the design memory, for CPU to read and execute the code.

4) Simulation: We run the simulation until the entire code is executed. Then, the corresponding VCD file is exported.

5) Power side-channel trace generation: The VCD file is converted to a one-shot trace containing all the program instructions. We use dynamic power consumption model.

6) Synchronization and trace partitioning: At this point, the traces corresponding to individual instructions are to be extracted. To simplify the task of identifying which part of a trace corresponds to a specific instruction, we used an internal synchronization signal (as a trigger), active whenever a new instruction enters the execute stage of the pipeline, and saved three clock cycles before and after the activation of the trigger. Repeating the above in a sliding-window fashion provides us with traces for all instructions.

7) Label extraction: We used program counter values as trace labels, which we obtained by parsing the disassembly file generated using the RISC-V GCC compiler.

Overall, we generated 56,780 labeled traces from the one-shot trace corresponding to the same number of randomly selected instructions. Each trace contained 600 time-steps in total (each clock cycle contained 100 time samples).

V. Deep-learning based CPU disassembly

In this paper, the side-channel disassembly using deep learning is considered as a *supervised* classification problem. The data set corresponds to traces describing the simulated power consumption signatures of various CPU instructions. Each trace in the data set corresponds to a specific CPU instruction. Given the number K of instructions to be classified, we can label each instruction opcode as Y_i, $i \in \{0..K-1\}$. Initially, since we are in a supervised learning case, the traces are collected from a CPU, either through simulation (at design time) or via the measurements. These traces, where the correct class label is known, are then split into a training, validation, and testing set. The training set is used for the training phase, the validation set is used to tune the hyper-parameters of the deep learning model, and the testing set is used for the evaluation phase. The training set is denoted by $\mathbf{X}_{N\times M}$, where N is the number of time-steps (i.e., the features) and M is the total number of traces used for training.

The data profiling phase learns from the data a function $\mathbf{F}$, which assigns to each trace $\mathbf{x}$ an instruction prediction Y_i; the prediction can be compared to the ground truth to asses the performance of the model. In what follows, we discuss three phases of the classification problem: data augmentation, feature engineering, and the actual classification phase.

A. Data Augmentation

When a data set is relatively small, as in our case, data augmentation (DA) techniques can be used to increase its size. During DA, the training set is expanded by creating new data from the existing data. In general, using DA is a good practice, not only for increasing the size of the initial dataset when it is small, but also to avoid overfitting and to help improving the model performance. In this work, we doubled the initial training set by adding random jittering to the traces. To make the data set even larger, we doubled the training set again by adding random scale factors to the traces.

B. Preprocessing and Feature Engineering

The second phase consists of preprocessing and feature engineering. For traces generated at design time, no measurement noise is present, which should facilitate the preprocessing step. However, not all simulated time steps are important for classifying the executed instructions. Thus, we try to find the relevant samples so as to minimize the computational effort. In the literature, principal component analysis or linear discriminant analysis are used for feature extraction and dimensionality reduction. In this work, we opt for sparse dictionary learning instead. The reason is that, unlike decomposition based on PCA and its variants used in the literature, sparse dictionary learning does not impose the basis elements to be orthogonal; consequently, it allows more flexibility in adapting the data representation to the training data set.

Sparse dictionary learning is an unsupervised approach of feature encoding that allows learning a sparse representation of the training traces in terms of a linear combination of a few basic elements called atoms to reduce the original data's dimension. The atoms constitute the dictionary. Therefore, dictionary learning aims at finding a dictionary $\mathbf{D}_{N\times L} = \{\mathbf{d}_1, \mathbf{d}_1, ..., \mathbf{d}_L\}$ from the data, where L is the number of atoms, $\mathbf{d}_i \in R^N$, and N is the number of time steps.

Once the dictionary is learned, the data is mapped from a time-series space to a new space with a possibly lower dimension, defined by the atoms of the dictionary as follows:

$$\hat{\mathbf{X}} = \mathbf{D}\mathbf{H}. \tag{1}$$

Here, $\mathbf{H}_{L\times M}$ is the weight matrix containing the contribution of each atom of the dictionary for constructing the original trace. The weight matrix should be sparse, so that the number of dictionary atoms needed to represent our original signal is minimized. The process of learning a sparse weight matrix is referred to as sparse coding. The underlying idea is to learn the dictionary and the weight matrix from the training data, by minimizing the convex form of the reconstruction error between the original trace (in the time-series space) and the new trace (in the dictionary learning space).

Finally, the dictionary learning using sparse coding can be formulated as the following optimization problem:

$$\min_{\mathbf{H}}(\min_{\mathbf{D}} \sum_{i=1}^{M} \frac{1}{2} \left\|\mathbf{X}_{:,i} - \mathbf{D}\mathbf{H}_{:,i}\right\|^2 + \lambda \left\|\mathbf{H}\right\|_1) \textit{ s.t. } \left\|\mathbf{d}_i\right\| \leq 1, \tag{2}$$

where λ is a regularisation term corresponding to the trade-off between a good reconstruction error, on one side, and the sparsity of the weight matrix $\mathbf{H}$, on the other.

We solve the optimization problem in iterative fashion until convergence, by alternating between the two optimization problems present in Eq. 2. To do that, we try minimizing over one while keeping the other fixed as follows:

1) *Weight matrix update:* To solve the minimisation problem, we make the assumption that $\mathbf{D}$ is fixed and then employ Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) [21]. To initialize the dictionary in the first iteration, we take L random traces from the dataset.
2) *Dictionary update:* To update the dictionary $\mathbf{D}$, we solve Eq. (2), using the Block coordinate descent algorithm [22] and a fixed weight matrix from the previous iteration. In the first iteration, the weight matrix is initialized with zeros.

The final dictionary is used as an input to the classification (testing) phase, as illustrated at the right side of Fig. 1.

C. Classification

The third and last phase is the classification of the traces, i.e., predicting the class of given data points. For the training set, the labels are known; they represent the ground truth. Using an appropriate error function, the error between the model prediction and the ground truth is computed and used to update the model parameters.

After implementing the sparse dictionary learning on the training set, the DL model is trained using the weight matrix as an input. The traces that may belong to the same class will tend to use the same atoms of the learned dictionary. Other atoms will have a contribution close to zero due to the sparsity effect. For this reason, learning on the weight matrices will, in fact, allow us to distinguish between the traces that belong to different classes. As a DL model we choose multilayer perceptron (MLP), which we train and evaluate at various levels of difficulty:

1) Simple opcode detection: At first, to simplify the classification task, we want to distinguish between a limited number of classes. Hence, we target eight groups of instructions from the CPU RISC-V instruction set: arithmetic, set, shift, branch taken, branch not taken, load upper immediate, load, and store.

2) Advanced opcode detection: Then, we try to classify individual instructions.

3) Bit-level approach: Finally, we want to recover the entire 32-bit instruction word, which includes registers and data besides the opcode. Hence, we have a binary classification problem for each bit of the instruction.

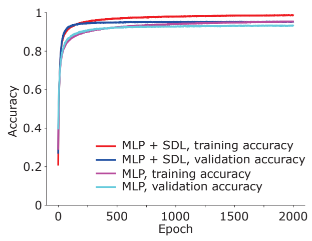
VI. Evaluation

A. Experimental Setup

In this work, we perform experimental evaluation on two design targets: the first is a RISC-V CPU core under design (not manufactured at the time of work), while the second is a Wio Lite RISC-V GD32VF103 development board. The operating frequencies of the two CPUs are 166 MHz and

100 MHz, respectively. The CPU architectures differ as well. As illustrated in the left side of Fig. 1, for the core that is not yet manufactured, we employ our automated simulation-based approach for generating power side-channel traces, while for the real device we conduct lab measurements. Since EM side-channel traces are known to provide superior results (thanks to capturing localized leakages), they are more attractive from the attacker perspective. Therefore, we chose to collect EM traces from the real device. To that purpose, we used the WaveRunner Lecroy oscilloscope (10 GS/s sampling rate) and an EM probe Langer EMV ICR HH100-27.

Before training a deep-learning model, we split the entire data set into training (63%), validation (7%), and testing subsets (30%). The model is trained on the training data set, then validated on the validation data set and, finally, tested on the testing data set.



(a) Simple classification.

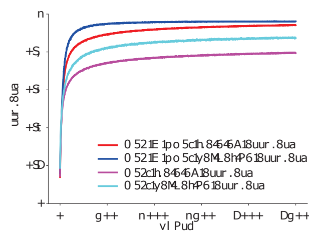
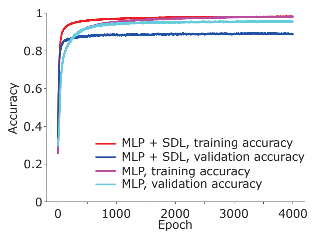
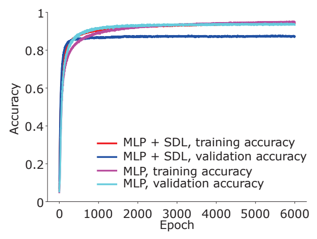

(b) Advanced classification.

Fig. 3: MLP training and validation accuracies on simulated traces.



(a) Simple classification.



(b) Advanced classification.

Fig. 4: MLP training and validation accuracies on measured traces.

B. Training and Classification with Simulated Traces

As described in Section IV-B, we generate 56,780 power side-channel traces, each having 600 time samples. Additionally, we augment the training data set by a factor of four (Section V-A), to obtain 143,088 traces. When applying the sparse dictionary learning on the training set, we fix the number of dictionary atoms to one half of the number of time steps ($L = 300$). Hence, the learned weight matrix of the dictionary, which will constitute the input layer of the MLP, has 143,088 data points and 300 features.

1) Simple opcode detection: The MLP is composed of five hidden layers with rectified linear unit (ReLU) activation function. The number of neurons per layer is 512, 256, 128, 64, and 16, respectively. A dropout layer of probability 10% is added after the first three hidden layers. The output layer has eight neurons with the softmax activation function, which corresponds to the number of instruction groups to classify.

2) Advanced opcode detection: The MLP is composed of five hidden layers with ReLU activation function. The number of neurons per layer is 512, 256, 128, 64, and 34, respectively. A dropout layer of probability 10% is added after the first three hidden layers. Here, the output layer has 34 neurons with softmax activation function, which corresponds to 34 instructions we want to classify.

3) Bit-level approach: Given that this is a binary classification problem, we trained 32 MLP models, one for each bit of the instruction. For every bit, we used the same MLP architecture: six hidden layers with ReLU activation function. The number of neurons per layer is 256, 128, 64, 32, 16, and 8, respectively. The output layer is a single neuron with sigmoid activation function, whose output can be interpreted as a probability of the bit being equal to logical '1' or '0'.

In the training phase, we chose cross-entropy loss function and ADAM optimizer with 0.001 learning rate.

C. Training and Classification with Real Traces

Given the CPU clock frequency of 100 MHz and 10 GS/s oscilloscope sampling rate, we obtain 100 samples per each clock cycle. To reduce the noise and interference, we make 100 recordings of each code sequence and average the traces, after synchronization. The total data set size is 183,000 traces.

The MLP models for simple and advanced opcode detection are identical to the models for simulated traces and, here too, we chose cross-entropy loss function and ADAM optimizer with 0.001 learning rate in the training phase.

D. Results

Since the datasets for simple and advanced opcode classification are balanced, we compute and report the classification accuracy. In the bit-level approach, however, the datasets are not balanced and, consequently, the accuracy cannot be used as it ignores the importance of false negatives and false positives. For this reason, we shall report a more appropriate metric: F1 score [23]. The F1 score is the harmonic mean of precision and recall, where the precision quantifies the number of correct positive predictions made over the total number of positive samples, and the recall quantifies the number of correct positive predictions made out of all positive predictions that could have been made.

Figures 3 and 4 show the training and validation accuracies over the number of epochs. Clearly, the validation performance curve follows the training one, indicating a good learning behaviour. In some cases, the validation accuracy outperforms the training one, this is caused by the random dropout layers added to the training process to prevent over-fitting.

1) Simulated traces: We obtained 96.05% and 96.01% accuracy for simple and advanced opcode detection, respectively. Table II shows the F1 score for each bit of the instruction word: it ranges from 88% to 100%, with an average of 95.16%.

2) Measured traces: We obtained 90.01% and 86.68% accuracy for simple and advanced opcode detection, respectively.

TABLE I: Comparison to related work.

	Eisenbarth et al. [9]	Strobel et al. [14]	Park et al. [10]	Vafa et al. [12]	This work
CPU	PIC16F687	PIC16F687	ATMega328P	LPC1768	**RISC-V**
Instruction	8 bits	8 bits	8 bits	16 bits	**32 bits**
Clock frequency	1 MHz	4 MHz	16 MHz	Not specified	**166 MHz,* 100 MHz****
Instructions	33	33	112	14	34
Recognition rate	70.1%	96.24%	99.03%	98.0%	96.01%,* 93.16%**
Bit-level	NO	NO	NO	NO	95.16%*(average F1 score)
Reduction	PCA, Ficher's LDA	Polychtomous LDA	PCA, KL-divergence	CWT, KL-divergence, PCA	Sparse dictionary learning
Classifier	Multivariate Gaussian	kNN	LDA, QDA, SVM, Naïve	Hierarchical	MLP
Side channel	Power	Multiple EM	Power	Power	Power,* EM**

* Simulated power side-channel traces.
** Measured EM side-channel traces.

TABLE II: F1 score for bit-level classification.

Bit index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F1 score (%)	95.8	93.8	94.9	94.3	93.4	94.4	94.5	95.2	94.3	93.2	95.1	95.0	97.6	93.8	94.9	95.2
Bit index	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
F1 score (%)	95.1	95.5	94.6	95.3	92.2	95.1	92.2	88	93.2	94.8	97.9	98.8	100	97.1	100	100

Bit-level classification failed to give satisfactory results, most likely due to the limited data set size and noise. Knowing that these factors may limit the performance of PCA, LDA, and dictionary learning techniques, we repeated the training using directly the collected traces. Indeed, for measured traces, the accuracies improved: 94.50% and 93.16%, respectively. At the same time, for simulated traces, skipping SDL lead to lower recognition accuracy: 93.22% and 88.04%, respectively.

Table I gives a comparison to previous works. Seeing that we target a 32-bit instruction word, a relatively complex RISC-V CPU, and that the device operating frequency is an order of magnitude higher than the frequencies reported in previous works, there is no doubt that deep learning (in particular, MLP) is very well suited for CPU disassembly attacks.

VII. Conclusion

In this paper, we present a novel side-channel disassembler based on multilayer perceptron and sparse dictionary learning, and complement it with an automated solution for collecting power side-channel traces at design time. The evaluation on two 32-bit RISC-V cores, operating at frequencies as high as 166 MHz and 100 MHz, shows that deep learning-based disassembly is highly effective—the instruction recognition accuracies on simulated and on measured traces reached 96.01% and 93.16%, respectively. With our approach, designers of modern embedded devices can effectively and in a time-efficient manner evaluate the vulnerability of their designs to the side-channel disassembly attacks.

References

[1] G. Canfora, M. D. Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, 2011.

[2] M. Ender, A. Moradi, and C. Paar, "The unpatchable silicon: A full break of the bitstream encryption of Xilinx 7-series FPGAs," in *USENIX*, 2020.

[3] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO '99*, 1999.

[4] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *CHES*, 2004.

[5] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *CHES*, 2002.

[6] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel, "Mutual information analysis," in *CHES*, 2008.

[7] B. Hettwer, S. Gehrer, and T. Güneysu, "Applications of machine learning techniques in side-channel attacks: A survey," *Journal of Cryptographic Engineering*, 2020.

[8] J.-J. Quisquater and D. Samyde, "Automatic code recognition for smart cards using a Kohonen neural network," in *CARDIS*, 2002.

[9] T. Eisenbarth, C. Paar, and B. Weghenkel, *Building a Side Channel Based Disassembler*. Springer, 2010.

[10] J. Park, X. Xu, Y. Jin, D. Forte, and M. Tehranipoor, "Power-based side-channel instruction-level disassembler," in *DAC*, 2018.

[11] D. Krishnankutty, Z. Li, R. Robucci, N. Banerjee, and C. Patel, "Instruction sequence identification and disassembly using power supply side-channel analysis," *IEEE Transactions on Computers*, Nov. 2020.

[12] S. Vafa, M. Masoumi, and A. Amini, "An efficient profiling attack to real codes of PIC16F690 and ARM Cortex-M3," *IEEE Access*, Dec 2020.

[13] P. Robyns, M. D. Martino, D. Giese, W. Lamotte, P. Quax, and G. Noubir, "Practical operation extraction from electromagnetic leakage for side-channel analysis and reverse engineering," in *WiSec*, 2020.

[14] D. Strobel, F. Bache, D. Oswald, F. Schellenberg, and C. Paar, "SCANDALee: A side-channel-based disassembler using local electromagnetic emanations," in *DATE*, 2015.

[15] V. Cristiani, M. Lecomte, and T. Hiscock, "A bit-level approach to side channel based disassembling," in *CARDIS*, 2019.

[16] Y. Han, I. Christoudis, K. I. Diamantaras, S. Zonouz, and A. Petropulu, "Side-channel-based code-execution monitoring systems: A survey," *IEEE Signal Processing Magazine*, Mar. 2019.

[17] J. Park, F. Rahman, A. Vassilev, D. Forte, and M. Tehranipoor, "Leveraging side-channel information for disassembly and security," *Journal Emerging Technologies in Computing Systems*, Dec. 2019.

[18] "Wio Lite Risc-V," Seeed Studio, 2021. [Online]. Available: wiki.seeedstudio.com/Wio_Lite_RISC_V_GD32VF103_with_ESP8266/

[19] H. D. Tsague and B. Twala, "An electromagnetic approach to smart card instruction indetification using machine learning techniques," in *SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*, 2017.

[20] D. Šijačić, J. Balasch, B. Yang, S. Ghosh, and I. Verbauwhede, "Towards efficient and automated side-channel evaluations at design time," *Journal of Cryptographic Engineering*, 2020.

[21] A. Beck and M. Teboulle, "Fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM J. on Imaging Sciences*, 2009.

[22] J. Mairal, F. Bach, J. Ponce, and G. Sapiro, "Online dictionary learning for sparse coding," *Proceedings of the 26th annual international conference on machine learning*, 2009.

[23] D. M. W. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," *International Journal of Machine Learning Technology*, 2011.