Machine Learning for Side-Channel Disassembly

Ognjen Glamočanin, Rania Islambouli and Dina Mahmoud
Department of Computer Science, EPFL, Switzerland

***Abstract*—Code execution on processors results in side effects such as electromagnetic emanations. Monitoring these effects can provide information about the code execution. However, extracting the information from the collected data is not a straightforward task and the information can be affected by noise and signal collection conditions. Therefore, in this project, we investigate the use of machine learning models for side-channel disassembly of instructions running on a processor. Building on previous work, we investigate the implementation of a hierarchical model to recognize the instructions and their operands with high accuracy. Our approach provides results close to simulation, highlighting the validity of our proposed approach.**

I. Introduction

From small embedded devices to IoT devices to cloud servers, central processing units (CPUs) are used for a variety of applications. Processor security is an ever-evolving field, with researchers continuously developing attacks and defenses. Among the well-known attacks are side-channel attacks, with power-based side-channel analysis proposed by Paul Kocher in 1996 [1]. While power and electromagnetic side-channels have been used to steal cryptographic keys, researchers have recently examined using them to track code execution [2], [3], [4]. On the one hand, this technique can be useful for an attacker to reverse-engineer proprietary code executed on a processor. On the other hand, this can be useful to track code execution without a debug interface to ensure integrity of the code.

Building on previous work, we train machine learning models to predict the executed instructions and their operands (registers and immediates) based on the electromagnetic (EM) emanation measurements of a RISC-V processor. In the previous work, power side-channel information was exploited from simulated power traces and from measured EM traces to train 34 models [5]. The time series obtained for each instruction are used to train the models. First, a model was trained to classify the instructions into 6 classes (arithmetic, set, logic, shift, store, load). Second, a model was trained to predict the actual instruction from 23 possible instructions (a large subset of all possible instructions). Finally, 32 models were used to recognize the bits of each instruction code, thus revealing the instruction and its operands. Using simulated traces without noise, the author achieved accuracies higher than 90% for the first two models, while the average accuracy of the 32 bit-wise models was 75% [5]. However, the power measurements collected from an ASIC were too noisy, resulting in the use of EM measurements instead. We use the same EM emanation measurements collected from the target RISC-V processor and modify the approach to obtain a high classification accuracy.

The rest of this report is structured as follows. We present our system design in Section II, followed by our training methodology in Section III. We present our results in Section IV. We discuss the reproducibility of our results in Section V. Finally, we consider possible future work and conclude the work in Section VI.

II. System Design

Based on the results of previous work, the easiest task is classifying executed instructions into classes while the most difficult task is predicting the bits of every instruction code [5]. These results are intuitive. Different classes of instructions typically utilize different hardware components and as a result, have different EM footprints. This variation in hardware utilization makes them easier to classify from the traces. Conversely, classifying the individual bits is more difficult as their inherent impact on the power consumption and EM emanations is very small, and the model needs to be able to extract more information from the traces.

Our idea is built upon the redundancy between the information used by the three different tasks. Therefore, we change the approach to a hierarchical one, where we first train a model to classify the instruction into one of the six classes. Then, we train six models, each one responsible for differentiating between the instructions in one class. Finally, for each of the instructions, we train 3-5 models that recognize the operands ((*src1*, *src2*, *dest*), (*src*, *dest*, *immediate*[3:0], *immediate*[7:4], *immediate*[11:8]), or (*src*, *dest*, *shift*)).

For the first stage, there is no difference between the hierarchical and non-hierarchical approaches. For the second stage, the task of each model is rendered easier as the number of output classes is greatly decreased. Finally, for the third stage, the task is reduced to recognizing the operands. Intuitively, this approach allows for better classification, as every level in the hierarchy focuses on extracting different information from the EM traces and can rely on information from previous level(s). If we assume that the model only predicts one class all of the time, then the accuracy of a model when it has six possible output classes will be higher than when it has 23 possible output classes. Other

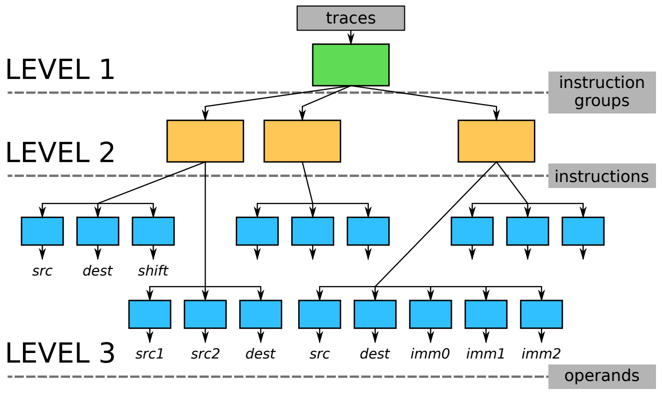

Figure 1. Diagram of the hierarchical classifier. Level 1 predicts the group of the instruction, Level 2 predicts the instructions within a group, while Level 3 predicts the operands of a given instruction.

researchers have studied the idea of using a classifier for instruction classes and then having other classifiers for the finer-grained instruction recognition within the classes [4]. A hierarchical approach results in the overall accuracy being the multiplication of the accuracies of each of the individual hierarchy levels. Therefore, if the accuracy is high enough at each of the levels, the overall accuracy of the models at the end of the hierarchy will be higher than the 75% accuracy obtained using simulated traces in previous work [5]. Furthermore, the number of levels and models could be adapted to obtain different results. Our aim is to show the validity of this approach.

III. Training and Evaluation Methodology

For building and training our models, we used the *keras* library [6]. This is a deep learning API written in Python, running on top of *TensorFlow*. It provides various models and functions allowing the user to run predefined models or build models using hidden layers and convolutional layers among others. It offers a variety of activation functions, evaluation metrics and stopping criteria for the training [6]. Finally, the code we built upon used *keras*, making the transition smooth [5].

We present below the data augmentation used and the training and evaluation methodologies for each of the hierarchy levels.

A. Data Augmentation

Since our dataset is relatively small, containing 45,827 datapoints in the first level, and 2,000 datapoints in the last level, we use data augmentation techniques in order to increase the size of our dataset. For each model in the third level, we first duplicate the dataset by appending the original data with the data convoluted with a flattop window of size 11, obtaining a 2× larger dataset, as shown in Fig. 2. To make the dataset even larger and more robust, we then duplicate the new dataset by appending data with added random noise, obtaining a 4× larger dataset, as shown in

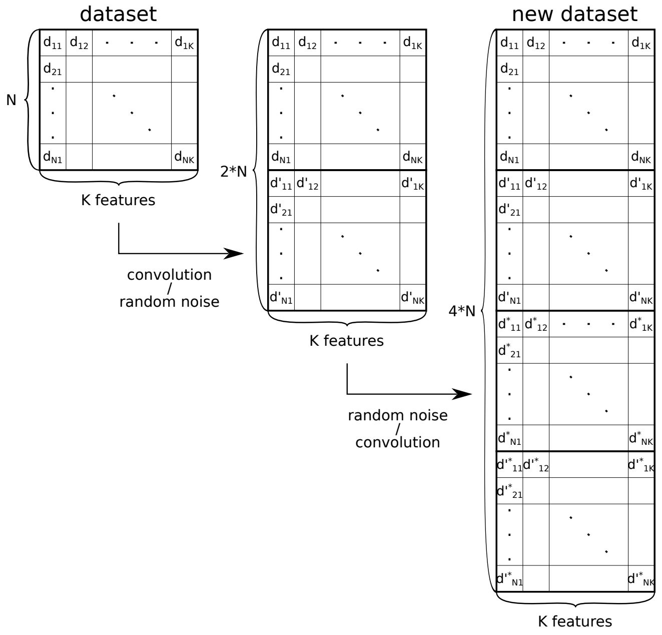

Figure 2. Data augmentation process. First, the data is augmented with noise (convolution) and appended to the dataset. Then, the new dataset is augmented with convolution (noise) and appended to the final dataset.

Fig. 2. For the models used for the first and second levels, we first duplicate the dataset by appending the original data with the data with random noise, obtaining a 2× larger dataset, as shown in Fig. 2. Furthermore, we duplicate the new dataset by appending the new dataset with data convoluted with a flattop window of size 11, obtaining a 4× larger dataset, as shown in Fig. 2. For data augmentation purposes, we use the *tsaug* library, which is specifically conceived for augmenting time series [7].

B. Simple Opcode Classification

The simple opcode classification represents both the simplest classification level and the most important one. It is simpler than the following levels because it only examines high level features and classifies into six classes. However, it is the most important because misclassifying at this level will definitely result in a wrong classification at the second level and possibly at the last level.

We start by evaluating the baseline model [5]. This gives a classification accuracy of 51.17%. We then augment our dataset, according to the methodology previously discussed. This increases the size of the dataset, but does not improve the accuracy significantly. Therefore, to extract important features from the dataset, we also perform principal component analysis (PCA) using *scikit* [8]. This is a standard technique used in time series classification, used in previous work [5], [3].

This improves the accuracy (77.9%) but not to a satisfactory level. Therefore, we increase the model size by using one of the well-known time series classification multi-layer perceptron (MLP) models [9]. This model consists of three hidden layers, all using *relu* activation function, each of 500 neurons. Before each layer, a dropout layer is used with rates respectively equal to 0.1, 0.2, 0.2 and 0.3. The final layer has a *softmax* activation function and a number of neurons equal to the output classes (six, in this case). The training is carried out for 6,000 epochs, with a batch size of 256. Some earlier experiments involved a larger number of epochs. However, the larger number of epochs did not improve the accuracy.

C. Instructions Classification

For classifying the instructions, we follow the approach with the best results from level 1: augmenting the data, carrying out PCA and using the three hidden layers MLP. For each of the three classes of instructions for which we trained the models, we change the number of neurons in the output layer; three for the *arithmetic* class, four for the *set* class and six for the *logic* class. This does not cover all possible instructions and all possible classes, but is rather presented as a proof-of-concept for our approach. As in the previous level, we train the models for 6,000 epochs with a batch size of 256.

The classification accuracies at this level are relatively high, as presented in Section IV. Therefore, we do not explore other models.

D. Operands Classification

The final level of the hierarchy consists of multiple models per instruction, as shown in Fig. 1. The goal of this level of hierarchy is to predict the operands of every instruction. Almost all operands are 5-bit values representing the address of the corresponding registers in the CPU register file. An exception to this are the immediate values for the shift instructions, which are still 5-bit values, and the immediates for arithmetic and logic instructions, which are 12-bit values. Depending on the position of the operands in the instruction code, the instructions can be split in three groups. One group contains instructions with operands *src1*, *src2*, and *dest* (*add*, *slt*, *sltu*, *and*, *or*, *xor*, *sll*, *srl*, *sub*, *sra*). Another group contains instructions with operands *src*, *dest*, and *shift* (*slli*, *srli*, *srai*). The final group contains instructions with operands *src*, *dest*, and a 12-bit immediate *imm* (*andi*, *ori*, *xori*, *addi*, *slti*, *sltiu*). Starting from the same multi-layer perceptron (MLP) architecture, one model was trained for each of operands of every instruction, except for the *imm* operand. The large number of possible values represented by a 12-bit immediate causes the output layer of the corresponding MLP to be $2^1 2$ neurons, making the neural network too large to be trained using a local GPU. Consequently, we split each immediate into three "operands": *imm 3:0*, *imm 7:4*, and *imm 11:8*, and train a different model for each immediate of each instruction.

The MLP trained for the operands has 512 neurons in the first layer, 256 in the second, 128 in the third and fourth, 64 in the fifth and sixth, 34 neurons in the seventh layer, and 32 neurons in the output layer. All layers use the *relu* activation function except the output layer, which uses *softmax*. The *keras* optimizer was set to *adam*, and the models were trained on a batch size of 256 during 6,000 epochs. The MLP trained for the 4-bit immediates has 512 neurons in the first layer, 256 in the second, 128 in the third and fourth, 64 in the fifth and sixth, 34 neurons in the seventh layer, and 16 neurons in the output layer. Again, all layers use the *relu* activation function except the output layer (*softmax*). The *keras* optimizer was set to *adam*, and the models were trained on a batch size of 256 during 6,000 epochs.

The dataset was augmented using the methods presented in Section III-A, by first introducing more datapoints using convolution and then adding random noise. The test-train ratio is set to 20%, while the validation-train ratio is set to 30%.

Table I
RESULTS

		Min (single/hier.)	**Avg** (single/hier.)	**Max** (single/hier.)
Level 1	*group*	0.9069	0.9069	0.9069
Level 2	*instruction*	**0.948**/0.860	**0.955**/0.866	**0.968**/0.878
Level 3	*src1*	**0.886**/0.762	**0.906**/0.785	**0.933**/0.819
	src2	**0.881**/0.757	**0.906**/0.785	**0.937**/0.823
	src	**0.866**/0.745	**0.912**/0.790	**0.943**/0.828
	shift	**0.923**/0.794	**0.934**/0.809	**0.954**/0.837
	dest	**0.869**/0.747	**0.906**/0.785	**0.934**/0.820
	shift	**0.924**/0.794	**0.934**/0.809	**0.954**/0.838
	imm 3:0	**0.887**/0.763	**0.917**/0.794	**0.938**/0.823
	imm 7:4	**0.887**/0.763	**0.923**/0.800	**0.947**/0.831
	imm 11:8	**0.880**/0.757	**0.919**/0.796	**0.939**/0.824

IV. RESULTS

We present the test accuracies of our trained models in Table I. The table shows the minimum, maximum and average for the accuracies at each level (at the first level, they are all the same value because there is only one model). Furthermore, we also show the accuracy for the hierarchical model. We obtain the hierarchical accuracies by multiplying the level's accuracy with the accuracy of the level(s) above it. Therefore, the accuracy at level 1 is also unchanged.

From Table I, it is clear that our models manage to recognize the instructions and their operands with a high accuracy, highlighting the viability of our approach. The final accuracies are on a comparable level with the accuracies obtained using simulated traces without noise. Moreover, the accuracy of the first level is better than the previous results using the same EM traces [5].

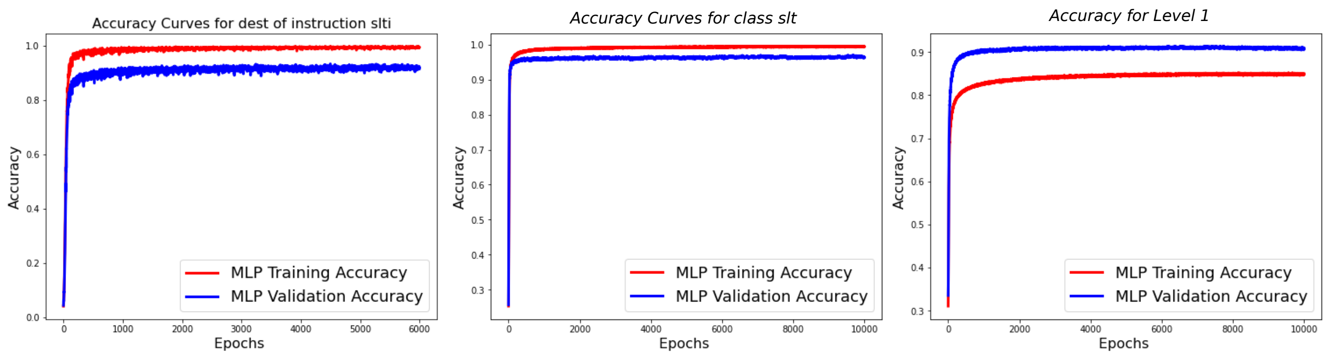

Figure 3. Training and validation accuracies for models in each hierarchical level. On the right, for the model in the first level. In the middle, for the model in the second level that classifies instructions in the *set* class. On the left, for the model in the third class that predicts the destination register of the *slti* instruction.

Fig. 3 shows the progression of the training and the validation accuracies during the training of models at different levels of hierarchy. The accuracy curves have the same shape for all other models in levels two and three. It can be observed that the test accuracy closely follows the curve of the train accuracy, meaning that the model does not overfit on the data.

V. Reproducibility

As there is significant randomness involved in the functions used for the training and evaluation of the models, we discuss here how to ensure the reproducibility of our results. The *keras* APIs run on top of *TensorFlow* and use *NumPy* functions. Therefore, we set the random seeds for both *TensorFlow* and *NumPy*. Furthermore, for all functions with a random seed or a random state, we set their values. Finally, we also enable the manual variable initialization for the backend of *keras*.

In Table I, the bold entries correspond to those models generated with the above described methods, and whose results are, therefore, reproducible. The other models require retraining, and we leave that as part of future work, discussed in the following Section. Due to the first level requiring retraining to obtain reproducible results, the final accuracies are not in bold in Table I.

VI. Conclusion and Future Work

In this work, we have demonstrated the training of models to be used for hierarchical classification of instructions executed on a processor based on collected EM traces. Our proposed system design achieves high accuracies for all levels (more than 90%) and good accuracies when chaining the models together (more than 74%). For future work, the first step would be to retrain all of the models to ensure the reproducibility of their results. Based on our results, we would also like to extend the models to cover all the possible instructions of the RISC-V processor. This requires collecting more traces, adapting the existing models and training new ones for instructions whose classes were not explored in this work (e.g. branch instructions).

Having all the models covering all possible code can be followed by chaining the models and evaluating the accuracy of our approach on more traces. In this case, the inference code would take the trace, get its classification result from the first classifier and based on it, forward it to one of the classifiers at level 2. The result from level 2 would control to which models at level 3 the trace is sent.

Finally, other methods have been proposed for time series classification [9]. Testing the same approach with other techniques and comparing their accuracies, advantages and disadvantages would be essential for the deployment of this approach.

References

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology — CRYPTO '96*, ser. Lecture Notes in Computer Science, N. Koblitz, Ed. Springer, 1996, p. 104–113.

[2] T. Eisenbarth, C. Paar, and B. Weghenkel, *Building a Side Channel Based Disassembler*, ser. Lecture Notes in Computer Science. Springer, 2010, p. 78–99.

[3] J. Park, X. Xu, Y. Jin, D. Forte, and M. Tehranipoor, "Power-based side-channel instruction-level disassembler," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, Jun 2018, p. 1–6.

[4] 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*, vol. 69, no. 11, pp. 1639–1653, Nov. 2020.

[5] H. Fendri, "ML-based side-channel analysis and disassembly of hardware root of trust," Master Thesis, 2020.

[6] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[7] *tsaug*. Arundo Analytics, 2019.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: a review," *Data Mining and Knowledge Discovery*, vol. 33, no. 4, p. 917–963, Jul 2019.