Portability of Deep-Learning Side-Channel Attacks against Software Discrepancies

Chenggang Wang[*]
Auburn University at Montgomery
Montgomery, AL, USA
cwang2@aum.edu

Mabon Ninan
University of Cincinnati
Cincinnati, OH, USA
ninanmm@mail.uc.edu

Shane Reilly
University of Cincinnati
Cincinnati, OH, USA
reillysp@mail.uc.edu

Joel Ward[*]
Cedarville University
Cedarville, OH, USA
joelbenward@gmail.com

William Hawkins
University of Cincinnati
Cincinnati, OH, USA
hawkinwh@ucmail.uc.edu

Boyang Wang
University of Cincinnati
Cincinnati, OH, USA
boyang.wang@uc.edu

John M. Emmert
University of Cincinnati
Cincinnati, OH, USA
john.emmert@uc.edu

ABSTRACT

Deep-learning side-channel attacks can reveal encryption keys on a device by analyzing power consumption with neural networks. However, the portability of deep-learning side-channel attacks can be affected when training data (from the training device) and test data (from the test device) are discrepant. Recent studies have examined the portability of deep-learning side-channel attacks against hardware discrepancies between two devices.

In this paper, we investigate the portability of deep-learning side-channel attacks against software discrepancies between the training device and test device. Specifically, we examine four factors that can lead to software discrepancies, including random delays, instruction rewriting, optimization levels, and code obfuscation. Our experimental results show that software discrepancies caused by each factor can significantly downgrade the attack performance of deep-learning side-channel attacks, and even prevent an attacker from recovering keys. To mitigate the impacts of software discrepancies, we investigate three mitigation methods, including adjusting Points of Interest, domain adaptation, and multi-domain training, from the perspective of an attacker. Our results indicate that multi-domain training is the most effective approach among the three, but it can be difficult to scale given the diversity of software discrepancies.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

Side-channel analysis, deep learning, software discrepancies

ACM Reference Format:
Chenggang Wang, Mabon Ninan, Shane Reilly, Joel Ward, William Hawkins, Boyang Wang, and John M. Emmert. 2023. Portability of Deep-Learning Side-Channel Attacks against Software Discrepancies. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'23), May 29-June 1, 2023, Guildford, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3558482.3590177

1 INTRODUCTION

Side-channel attacks [22] can recover encryption keys from a device, such as a microcontroller or an FPGA (Field-Programmable Gate Arrays) by analyzing correlations between power consumption and intermediate values of encryption, such as AES (Advanced Encryption Standard). Recent studies [10, 20, 27] show that machine learning can offer new advantages in side-channel attacks compared to traditional attacks, such as Correlation Power Analysis [9] and Template Attacks [12]. For instance, attacks [6, 31] built upon deep learning can recover keys from raw power traces with (no or less) pre-processing and can even defeat countermeasures, including masking and random delays.

On the other hand, deep-learning side-channel attacks still face challenges in terms of *portability* [7, 15]. Portability indicates the effectiveness of *profiling side-channel attacks* when training and test data are collected from two devices. According to recent studies [7, 11, 33, 39, 43], when training and test data are discrepant due to hardware discrepancies between the training device and test device, an attacker needs a greater number of traces to recover keys, and may even fail to compromise keys.

In this paper, we investigate the portability of deep-learning profiling side-channel attacks against *software discrepancies*, which have not been well-studied in the current literature. Put differently, we examine the cases where training data and test data are discrepant and these discrepancies are *primarily* caused by different

[*] The work was done when the authors were at the University of Cincinnati.

WiSec'23, May 29-June 1, 2023, Guildford, United Kingdom

ACM ISBN 978-1-4503-9859-6/23/05…$15.00
https://doi.org/10.1145/3558482.3590177

software settings between two devices. We examine the following two open research questions.

- ***RQ1:*** *To what degree can software discrepancies impact the portability of deep-learning profiling side-channel attacks?*
- ***RQ2:*** *How can an attacker overcome/mitigate software discrepancies and still recover keys successfully?*

Our Contributions. To answer the two research questions, we make the following contributions:

- We investigate four factors that can lead to software discrepancies, including *random delays*, *instruction rewriting*, *optimization levels*, and *code obfuscation*. Specifically, we simulate random delays by randomly shifting power measurements. We examine instruction rewriting at the assembly level and also over ELF (Executable and Linkable Format) files by leveraging a reverse engineering tool `Ghidra` [1]. We investigate four optimization levels, including Os, O1, O2, and O3 given a cross-compiler. We explore three code obfuscations offered by a code obfuscation tool `Tigress` [2].
- To facilitate our investigation, we collect a large-scale dataset, named `SoftPower` dataset. It consists of more than 3.2 million power traces (187 GBs) of AES-128 encryption from two types of microcontrollers, including AVR XMEGA (8-bit RISC) and ARM STM32 (32-bit Cortex-M4), using ChipWhisperer [3] and various software settings associated with the four discrepancy factors we examine.
- Experimental results suggest that every software discrepancy factor we examine leads to attack performance drops. Specifically, it takes a much greater number of test traces for a CNN (Convolutional Neural Network) to reveal encryption keys. Moreover, discrepancies caused by instruction rewriting, optimization levels, or code obfuscation can even result in the failure of recovering keys using a CNN.
- To mitigate the impacts of software discrepancies, we explore three mitigation methods, including (I) *adjusting Points of Interest (POI)*; (II) *domain adaptation*; and (III) *multi-domain training*, from the perspective of an attacker. Each method requires lowering the attack assumption to some degree. Specifically, adjusting POI requires *a small amount of unlabeled traces from the test device* to identify leakage points through statistical analysis, e.g., Normalized Inter-Class Variance [8]. Domain adaptation requires *a large amount of unlabeled traces from the test device.* Multi-domain training requires *large amounts of labeled traces of multiple software settings from the training device.*
- Our results suggest that adjusting POI can improve attack performance against discrepancies caused by instruction rewriting, optimization levels, or code obfuscation. Domain adaptation is only effective in few examples associated with random delays. Multi-domain training can overcome discrepancies caused by every factor and effectively reveal keys. However, it has to significantly demote the attack assumption by requiring an attacker to know every possible software setting in advance. Although this is potentially feasible, it could be difficult to scale in practice given the diversity of software discrepancies between the training device and test device. We highlight our findings in Table 1.

Table 1: Our Main Findings (○ indicates a method is not effective at all; ◒ indicates a method is effective to some degree or in some cases; ● suggests a method is effective.)

	Adjusting POI	Domain Adaptation	Multi-Domain Training
Random Delays	○	◒	●
Instruction Rewriting	◒	○	●
Compiler Optimization	◒	○	●
Code Obfuscation	◒	○	●

Reproducibility. The source code and datasets of this study are made publicly available at [4].

2 BACKGROUND

2.1 System and Threat Model

System Model. The system model (shown in Fig 1) of machine-learning profiling side-channel attacks includes two devices, including the *training device* and the *test device*. An *attacker* aims to reveal an *unknown but fixed* key on the test device. As in previous studies, we assume that this attacker has control of the training device to assist her to learn a profile (e.g., a classifier). Specifically, this attacker knows the key on the training device and can capture power traces and associated plaintexts (i.e., inputs of the encryption) from the training device. On the other hand, this attacker does not have control of the test device but can passively capture power consumption and associated plaintexts from the test device.

A machine-learning profiling side-channel attack includes two phases, the *training phase* (a.k.a., *profiling phase*) and the *test phase* (a.k.a., *attack phase*). In the profiling phase, this attacker trains a classifier with labeled power traces from the training device. In the attack phase, the attacker acquires unlabeled power traces from the test device and tries to recover the key on the test device by leveraging the classifier. We focus on deep-learning profiling side-channel attacks, where the classifier is a neural network.

Cross-Device Scenario. We focus on the cross-device scenario, where the training device and test device are of the same type but are distinct (e.g., two STM32 microcontrollers). Both of the devices run the same encryption algorithm but the two encryption keys on the two devices are different. We also assume that hardware discrepancies exist between the two devices due to imperfection/variations in manufacturing.

Software Discrepancies. *In addition to key and hardware discrepancies*, we assume that the two devices also carry software discrepancies. Specifically, we assume that the initial source code of the encryption remains the same but the binaries running on the two devices are different. For instance, given the same AES C code, the binary on the training device was compiled with O1 optimization but the binary on the test device was compiled with O2 optimization. These discrepancies could happen when an attacker has knowledge of the initial source code but does not have full knowledge regarding how the code was further modified or compiled for the test device.

Generation of Binaries with Cross-Compilation. To facilitate the discussions later, we briefly describe the cross-compilation process for a binary (i.e., a hex file) running on a microcontroller, specifically with ChipWhisperer – the hardware platform we leverage for data acquisition. Cross-compilation means generating a

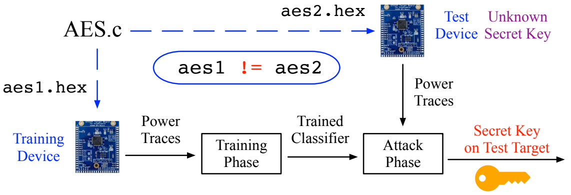


Figure 1: The system model of machine-learning profiling side-channel attacks with software discrepancies.

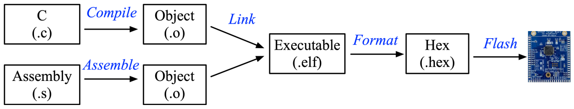


Figure 2: High-level overview of the generation of a hex file from source code with cross-compilation.

binary for a platform other than the one on which the compiler is running. For ChipWhisperer, given a C code (or an assembly code), it is first compiled (or assembled) as object files using a cross-compiler on a PC. Next, multiple object files are linked together to produce an ELF file. Finally, the ELF file is further formatted as a hex file on a PC and this hex file is flashed to a microcontroller.

2.2 Notations and Leakage Model

A power trace t is denoted as a vector $t = (t[1], ..., t[l])$, where $t[i]$ is the measurement of power consumption of a device at time i and l is the number of measurements. Let $\mathcal{M}$ be the plaintext space and $\mathcal{K}$ be the key space. Given a plaintext $m \in \mathcal{M}$ and key $k \in \mathcal{K}$, a power trace t is collected when a device runs encryption with plaintext m and key k. We use $z = \varphi(m, k)$ to denote an intermediate value of encryption, where function $\varphi(\cdot)$ is a leakage step.

Encryption Algorithm. We focus on attacks on AES-128 encryption, where a side-channel attack reveals one key byte each time. As in previous studies, *our description of side-channel attacks focus on recovering one byte*, where we assume key k, plaintext m, or intermediate value z only has one byte. We use $k_1^*, k_2^*, ..., k_{256}^*$ to denote all the possible 256 key values. We use the SubBytes of the 1st round of AES as the leakage step $\varphi(\cdot)$.

Leakage Model. We leverage *Hamming Weight (HW) model* [6, 31] to formulate side-channel leakage. The HW model assumes that there are correlations between the power consumption of an intermediate value and the Hamming weight of this intermediate value. The label of a power trace t is $\mathtt{HW}(z)$ — the Hamming weight of the intermediate value $z = \varphi(m, k))$ given plaintext m and key k. There are 9 possible Hamming weights (i.e., 0~8) as we assume intermediate value z has one byte.

Points of Interest (POI). Given a power trace, Points of Interest are power measurements associated with the leakage step, which is the SubBytes of the 1st round of AES-128 in this study. Given a device and a software setting, we identify POI in advance as in [39]. Specifically, given an AES implementation (in C or assembly), we insert multiple consecutive `NOP`s (No Operations), before and after SubBytes of the 1st round of AES to produce obvious gaps in power consumption. With this approach, we can *loosely* locate the start and end of SubBytes in a (raw) power trace. For instance, given a (raw) power trace t, if POI=[1200, 2200], only power measurements $(t[1200], ..., t[2200])$ are used for the side-channel attacks.

Training Phase. Given power traces $T = (t_1, ..., t_N)$, plaintexts $M = (m_1, ..., m_N)$ and key k from a training device, where m_i is associated with t_i, an attacker obtains intermediate outputs $Z = (z_1, ..., z_N)$, where $z_i = \varphi(m_i, k)$, and computes $h_i = \mathtt{HW}(z_i)$ as the label of power trace t_i. An attacker trains a classifier F with $(T, H) = \{(t_1, h_1), ..., (t_N, h_N)\}$.

Attack Phase. An attacker captures traces $T' = (t_1', ..., t_{N'}')$ and plaintexts $M' = (m_1', ..., m_{N'}')$ from a test device running key k', where m_i' is associated with t_i'. Given power trace t_i', an attacker obtains a score for each Hamming weight from classifier F and tries to infer the key k' on the test device.

Evaluation Metric. We utilize ***key rank*** (a.k.a. *guessing entropy* [31]) as the metric to measure the attack performance of side-channel attacks. Specifically, given a power trace t_i', classifier F outputs a *HW score vector* $(s_i[0], ..., s_i[8])$, where $s_i[j]$ is the score for Hamming weight j. Next, an attacker obtains a *key score vector* $(r_i[k_1^*], ..., r_i[k_{256}^*])$ by calculating

$$r_i[k_g^*] = s_i[j], \quad \text{if} \quad \mathtt{HW}(\varphi(m_i', k_g^*)) == j \qquad (1)$$

for $1 \le g \le 256$, where $r_i[k_g^*]$ is the score of possible key k_g^* according to trace t_i' and plaintext m'. The *aggregated key score vector* $(r[k_1^*], ..., r[k_{256}^*])$ over N' power traces are computed as

$$r[k_j^*] = \sum_{i=1}^{N'} r_i[k_j^*], \ \mathit{for}\ 1 \le j \le 256 \qquad (2)$$

The aggregated key scores $(r[k_1^*], ..., r[k_{256}^*])$ are further sorted in descending order based on the scores.

Key rank is denoted as w, where $w \in [0, 255]$, if correct key k' is ranked as the $(w+1)$-th key among all the possible keys $k_1^*, ..., k_{256}^*$ based on the aggregated key scores. A key rank of 0 over N' traces suggests that an attacker can recover key k' with N' traces. We use ***KRC*** to indicate the number of test traces that an attacker needs for key rank to converge to 0. If key rank converges to 0 with a lower number of traces, it indicates that an attack is more effective.

Normalized Inter-Class Variance (NICV). NICV [8] can identify leakage points with high correlations between power consumption and intermediate values of encryption. Specifically, let Y be a random variable of power measurements at time i over N traces and Z^* be a random variable of intermediate values of the leakage step associated with the N traces, NICV at time i over the N traces can be computed as $\frac{\mathtt{Var}[\mathbb{E}[Y|Z^*]]}{\mathtt{Var}[Y]}$, where $\mathbb{E}$ denotes mean and `Var` denotes variance. We utilize NICV in this study, but other methods, such as Signal-to-Noise Ratio [21] or Test Vector Leakage Assessment [34], can also locate points with high correlations.

3 SOFTWARE DISCREPANCIES

We examine four discrepancy factors, including random delays, instruction rewriting, optimization levels, and code obfuscation.

① *Random delay* is an effective way to mitigate side-channel attacks, where POI across traces are misaligned. Random delay can be implemented in software. ② *Instruction rewriting*, in the context of this study, refers to *manually* modifying machine instructions of AES implementation without affecting its correctness. ③ When compiling a C code, a cross-compiler can choose an *optimization*

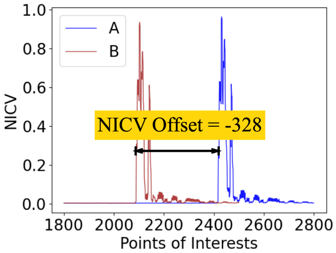


Figure 3: The NICV offset is -328 from dataset A to dataset B. If the training over A uses POI=[1800, 2800], then testing over B with offset=-328 means POI=[1472, 2472].

level. Different optimization levels offer different tradeoffs among compilation time, file size, and memory usage. ④ *Code obfuscation* transforms a C code to an obfuscated version, in which the functionalities remain the same but the code is difficult for reverse engineering.

Since discrepancies produced by these factors depend on the architecture of microcontrollers, hardware platforms, etc. We will discuss the details of each factor in the evaluation section.

4 MITIGATION METHODS

We investigate three mitigation methods in order to still recover keys against software discrepancies.

4.1 Mitigation I: Adjusting POI

Since software discrepancies fundamentally lead to different instructions, which result in different power patterns of AES on a device (see examples in Appendix) and shifts in terms of leakage points. We believe that adjusting POI based on statistical analysis, such as NICV, could be helpful.

Specifically, given training data from the training device and test data from the test device, we first compute the NICV of each dataset. Next, we compare the high peaks of NICVs between the two datasets and obtain *a NICV offset*. This NICV offset is defined as the offset between the first high peak of NICV of the training dataset and the first high peak of NICV of the test dataset. Finally, we align the POI of the test data with the POI of the training data by applying the NICV offset (*or offsets close to this NICV offset*). An example of NICV offset is illustrated in Fig. 3.

In essence, given a vector of NICV values $(n_A[1], ...n_A[l])$ from dataset A and a vector of NICV values $(n_B[1], ..., n_B[l])$ from dataset B, the NICV offset from A to B can be formulated as the offset α such that the distance between the two vectors is minimized, where the distance is defined as below:

$$\arg\min_{\alpha} = \sum_{i=1}^{l} \left| n_A[(i+\alpha) \mod l] - n_B[i] \right| \quad (3)$$

The intuition of this mitigation is that if a classifier can learn side-channel leakage and recover keys from training data, the measurements associated with high NICV peaks contribute significantly to the predictions of the classifier. If we align the NICV peaks of the test data with the NICV peaks of the training data, it could help the classifier to mitigate data discrepancies.

Assumptions of Mitigation I. To apply this method, we need to assume that the NICV peaks can be observed. This requires two assumptions for the attacks: (1) the number of test traces from a test dataset is sufficient (e.g., a few hundred or thousand traces depending on the noise level of a dataset); (2) random delay or masking is not implemented such that leakage points are not preserved.

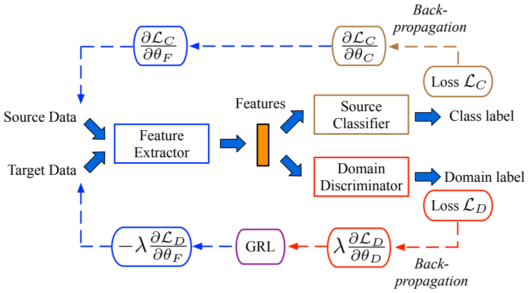

$\frac{\partial \mathcal{L}_C}{\partial \theta_F}$
$\frac{\partial \mathcal{L}_C}{\partial \theta_C}$
Back-propagation
Loss $\mathcal{L}_C$
Source Data
Target Data
Feature Extractor
Features
Source Classifier
Class label
Domain Discriminator
Domain label
Loss $\mathcal{L}_D$
$-\lambda\frac{\partial \mathcal{L}_D}{\partial \theta_F}$
GRL
$\lambda\frac{\partial \mathcal{L}_D}{\partial \theta_D}$
Back-propagation

Figure 4: The structure of a domain adversarial network [16]. GRL stands for Gradient Reversal Layer.

4.2 Mitigation II: Domain Adaptation

When training data and test data are discrepant, i.e., training data and test data are from two different domains, domain adaptation [37] is a common machine learning approach to mitigate discrepancies. We particularly leverage Adversarial Domain Adaptation (ADA) [16, 36], which has shown promising results to mitigate domain discrepancies in other areas, such as image recognition.

Background of ADA. In domain adaptation [37], there are two datasets, including a source dataset and a target dataset. Domain adaptation addresses domain discrepancies between the two datasets by mapping source data and target data into a domain-invariant feature space. *In the context of side-channel attacks, data from the training device are considered as source data while data from the test device are treated as target data.*

ADA [16, 36] learns a domain-invariant feature space by training a domain adversarial network. It leverages generative adversarial learning [19] and outperforms traditional domain adaptation relying on Maximum Mean Discrepancy [24]. As illustrated in Fig. 4, a domain adversarial network consists of three parts, including a Feature Extractor F, a Domain Discriminator D, and a Source Classifier C. Each part is a neural network. During the training of a domain adversarial network, the Feature Extractor takes *labeled* source data and *unlabeled* target data as inputs and outputs domain-invariant features. The Domain Discriminator aims to distinguish whether an output of the Feature Extractor is generated from the source domain or the target domain. The Source Classifier minimizes the loss on predicting the correct labels of source data.

Let θ_F, θ_D, and θ_C be the parameters of the Feature Extractor, Domain Discriminator and Source Classifier, respectively. Given the loss function $\mathcal{L}$ of the domain adversarial network, the Feature Extractor and Source Classifier minimize the loss $\mathcal{L}$ while the Domain Discriminator maximizes the loss $\mathcal{L}$. The loss function $\mathcal{L}$ can be computed as

$$\mathcal{L}(\theta_F, \theta_D, \theta_C) = \mathcal{L}_C(\theta_F, \theta_C) - \lambda \mathcal{L}_D(\theta_F, \theta_D) \quad (4)$$

where $\mathcal{L}_C$ is the loss function of the Source Classifier, $\mathcal{L}_D$ is the loss function of the Domain Discriminator, and λ is a pre-defined trade-off parameter shaping features [16]. After the training, the

Feature Extractor F and the Classifier C can be extracted and used to perform classifications over target data.

Assumptions of Mitigation II. This method requires that an attacker possesses *a large number of unlabeled traces of the target dataset collected from the test device*, such that Domain Discriminator can potentially assist the entire domain adversarial network to learn a robust domain-invariant feature space.

4.3 Mitigation III: Multi-Domain Training

Multi-domain training trains a classifier with data from multiple domains and test data from each domain individually. For instance, an attacker can train a CNN with power traces collected from multiple optimization levels, including O1, O2, and O3, using the training device. Then, this attacker can test the CNN with traces collected from the test device, where it assumes the power traces are generated with a binary compiled with either O1, O2, or O3.

Assumptions of Mitigation III. This method downgrades the attack to a *closed-world problem.* Closed-world means that an attacker needs to know all the possible software settings on the test device and can capture large amounts of labeled traces from each software setting on the training device.

5 DATA COLLECTION SETUP

Hardware Setting. We collect power traces of AES-128 from devices, including AVR XMEGA and ARM STM32F3 microcontrollers with ChipWhisperer. Specifically, we leverage two desktops (PC1 and PC2) and two sets of ChipWhisperer Level 1 Kits (CL1 and CL2) to collect power traces. Both desktops run Ubuntu 18.04. A figure of the setup is presented in Fig. 10 in Appendix.

We examine data from two XMEGA (X1 and X2) and two STM32F3 (S1 and S2). We use PC1 and CL1 to collect data from X1 (S1) and use PC2 and CL2 to obtain data from X2 (S2). We use X1 (S1) as the training device and leverage X2 (S2) as the test device. We use different keys between the training device and the test device. We use the default sampling rate from ChipWhisperer.

We use the format of `Target_Key_ExtraInfo` to name each dataset from our data acquisition. `Target` indicates which microcontroller it is, `Key` suggests which key is used, `ExtraInfo` contains additional information of the data collection. For instance, `X1_K1_O1` indicates that a dataset is collected from X1 using key K1, and the binary is compiled with O1. For each dataset, unless specified, we always collect 200,000 traces from X1 (S1) and 100,000 traces from X2 (S2). Each trace consists of 5,000 measurements.

Basic Software Settings. We leverage two implementations of AES-128 as the basic software settings. The first one is tinyAES [3], which is an unmasked AES written in C. The second one is a low-security masked AES (version 1) [5] written in assembly for XMEGA. For the cross-compiler, we use `avg-gcc` (5.4.0) for XMEGA and `arm-none-eabi-gcc` (6.3.1) for STM32. The default optimization level is Os. All the software discrepancies evaluated later are based on basic settings by modifying certain specifics.

6 EVALUATION

Experiment Setting. All the experiments are carried out on a desktop equipped with Ubuntu 18.04, Intel i7 CPU, NVIDIA Titan RTX GPU, and 64GB memory. For all of the experiments, we always report attack results on the **third byte** of an AES key. We use Hamming Weight model to formulate the leakage.

Table 2: Keys Involved in Our Data Collection

K1	`0x2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c`
K2	`0xaa 80 d8 a7 84 d3 3f 5c 0b 90 a9 85 20 8e ff 4a`
K3	`0xd2 d5 01 68 82 83 91 43 96 9e e9 a2 53 a7 52 e1`
K4	`0xe6 de 35 a9 a5 23 19 df c6 cc bb ba c1 36 c3 bf`

Table 3: Datasets Associated with Random Delays

XMEGA	X1_K1	X2_K2, X2_K2_RD100,	X2_K2_RD50 X2_K2_RD200
STM32	S1_K1	S2_K3, S2_K3_RD100,	S2_K3_RD50 S2_K3_RD200

We utilize the CNN from [6], which was used for side-channel attacks over ASCAD datasets. The CNN is implemented with Tensorflow 2.3.2. The hyperparameters of this CNN are presented in Table 18 in Appendix. We use the same CNN (without the output layer) as the Feature Extractor of ADA. Source Classifier of ADA contains 1 convolutional layer with 1024 neurons and 1 output layer with 9 neurons. Domain Discriminator of ADA contains 3 convolutional layers (No. of neurons: {1024, 512, 256}) and 1 output layer with 2 neurons. The ADA is implemented with PyTorch 1.8.1[1].

For a CNN, we always use 40,000 traces for training and 10,000 traces for testing[2]. If a dataset is involved in multiple experiments, the same training data (or test data) are used across experiments for fair comparisons. For ADA, we use 40,000 traces of a source dataset and 40,000 (unlabeled) traces of a target dataset for training, and 10,000 traces of a target dataset for testing. We train 100 epochs for a CNN or ADA. Key ranks are reported on average by running each test 5 rounds and shuffling the order of testing traces per round.

Experiment 0: Impacts of Hardware Discrepancies. We first demonstrate the discrepancies introduced by hardware and keys in our data acquisition. *We show that these discrepancies have impacts on the attack results but are (relatively) minor among our datasets.*

Specifically, we collect 4 datasets, including `X1_K1`, `X2_K2`, and `S1_K1`, and `S2_K2`. The details of each key can be found in Table 2. As shown in Table 4, it only takes 1 trace for a CNN to recover the key when both training and test data are from `X1_K1`. On the other hand, it takes 5 traces for the same CNN to recover the key when the training data are from `X1_K1` and test data are from `X2_K2`. We have similar observations from STM32. We use POI=[1800, 2800] for XMEGA and POI=[1200, 2200] for STM32.

6.1 Impacts of Random Delays

Evaluation Scope. Similar to previous studies [6, 10], we simulate random delays by delaying power measurements of each trace with a random number of r, where $r \in [0, RD]$ and r is uniformly distributed. RD is denoted as the *max delay parameter.* We focus on the cases where the training data and test data are generated by different max delay parameters running tinyAES.

Datasets. We generate three datasets from `X2_K2` by applying max random delay $RD = \{50, 100, 200\}$ respectively. We repeat the

[1] ADA in PyTouch outperforms ADA in Tensorflow in our evaluation.

[2] In general, training with more than 40,000 traces does not necessarily improve attack performance of a CNN in our evaluation. Therefore, we choose 40,000 traces for training. Additional traces are available in our datasets for others to expand our findings.

Table 4: Attack Performance of CNN and ADA; Random Delay; CNN: Train with 40k (Source), Test with 10k (Target); ADA: Train with 40k (Source) and 40k (Target), Test with 10k (Target); XMEGA POI=[1800, 2800]; STM32 POI=[1200, 2200]. NA: Not Applicable; ⊥: key is not revealed.

Source	Target	KRC (CNN)	KRC (ADA)
X1_K1	X1_K1	1	NA
	X2_K2	5	NA
	X2_K2_RD50	170	486
	X2_K2_RD100	786	1,531
	X2_K2_RD200	4,115	3,831
S1_K1	S1_K1	5	NA
	S2_K3	6	NA
	S2_K3_RD50	127	c
	S2_K3_RD100	754	⊥
	S2_K3_RD200	4,201	⊥

same process for S2_K2. Datasets associated with random delays can be found in Table 3.

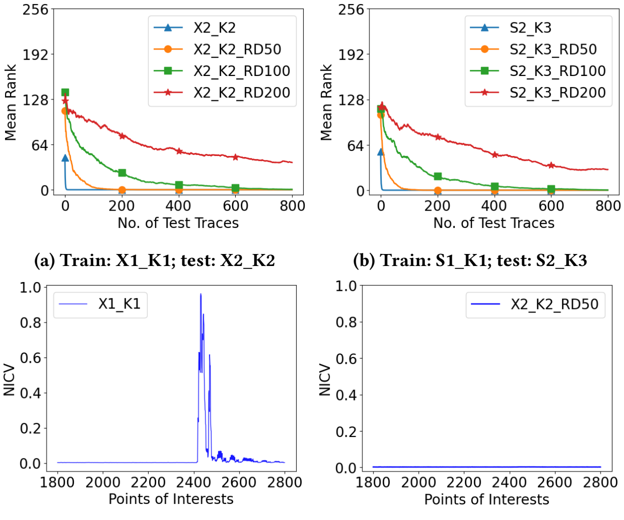
Experiment 1.1: Impacts of Random Delays. We train a CNN with data from X1_K1 and then test it with data from X2 with different max random delays. As shown in Table 4 and Fig. 5, the attack performance decreases with an increase of max random delay *RD*. We have a similar observation from STM32.

The performance drops are expected as different random delay parameters cause discrepancies between training and test data. Fig. 5 shows that a dataset with random delays does not contain NICV peaks due to misalignment across traces. Despite no NICV peaks observed from test data, CNN can still recover keys. This is likely because (1) there are distribution overlaps between training and test data in the case of random delays and (2) the convolutional layers can automatically overcome random delays to some degree. This is consistent with the results of CNNs over traces with random delays in existing studies [6, 10].

Observation 1.1: *Our results suggest that discrepancies caused by random delays can downgrade attack performance, where a CNN needs a greater number of test traces to reveal keys.*

Experiment 1.2: Mitigating Discrepancies Caused by Random Delays. As datasets with random delays do not leak NICV peaks, adjusting POI is not suitable. We first explore domain adaptation to mitigate discrepancies caused by random delays. As shown in Table 4, ADA can recover keys on XMEGA but it requires even more test traces than CNN, except when the target dataset is X2_K2_RD500. In addition, it fails to recover keys on STM32. In other words, CNN fails to improve attack performance. Note that when there are no software discrepancies between two datasets (e.g., X1_K1 and X2_K2), there is no need to apply ADA as CNN can effectively recover keys already.

Next, we perform multi-domain training. Specifically, we generate an additional dataset S1_K1_RD200 from S1_K1 by using $RD = 200$. As random delays are uniformly distributed, S1_K1_RD200 includes traces from $RD = \{0, 50, 100\}$. Therefore, we leverage S1_K1_RD200 as a dataset containing data from multiple domains to train a CNN, and test the CNN over datasets with different max random delays. As shown in Table 5, the CNN can recover keys within 8 traces regardless which dataset is tested.



(a) Train: X1_K1; test: X2_K2

(b) Train: S1_K1; test: S2_K3

(c) NICV of X1_K1

(d) NICV of X2_K2_RD50

Figure 5: Key Ranks and NICV, Random Delays

Table 5: Attack Performance of CNN; Random Delay; Train: 40k (Source), Test: 10k (Target), Source: S1_K1_RD200, POI=[1200, 2200];

Target	KRC	Target	KRC
S2_K3	8	S2_K3_RD100	6
S2_K3_RD50	6	S2_K3_RD200	6

Observation 1.2: *Our results suggest that ADA fails to promote attack performance against discrepancies caused by random delays. On the other hand, multi-domain training can effectively mitigate discrepancies caused by random delays.*

6.2 Impacts of Instruction Rewriting

Evaluation Scope. We focus on the scenarios where the training data are from an original implementation but the test data are from a rewritten implementation. We examine two cases that an original implementation can be rewritten, including (1) rewriting at the assembly level on .s files; (2) rewriting on ELF files.

Case 1: Assembly, XMEGA. We utilize the masked AES in assembly [5] on XMEGA. We focus on instructions associated with AddKey (line 880 – line 914 in .s file) and SubBytes (line 353 – line 442 in .s file). We identify 5 types of instructions (summarized in Table 6) that can be rewritten without affecting the correctness of AES encryption. Overall, 23 lines in the original .s file were rewritten to derive the modified version.

Datasets from Case 1. We collect one dataset X1_K1_ASM from the original assembly code on XMEGA. We collect another dataset X2_K4_ASM_RW by running a hex file generated by the modified assembly code on XMEGA. As the 1st round of SubBytes in the assembly code does not happen within the first 5,000 measurements, an offset of 17,500 during the data acquisition (i.e., recording measurements from index 17,500 to index 22,500) is used to collect measurements associated with the 1st round of SubBytes

Case 2: ELF, STM32. We compile tinyAES C code into an ELF file by leveraging the Makefile from ChipWhisperer repository. We then leverage Ghidra to locate the SubByte function in the ELF file.

Table 6: Rewritten Instructions for Assembly on XMEGA

Original Instructions	Instruction Purpose	Modified Instructions	No. of lines modified
`CLR r0`	Clear register r0	`EOR r0, r0`	13 lines
`DEC r0`	Decrease the value at register r0 by 1	`SUBI r0, 0x01`	2 lines

`MOV r0, r1`	Copy data at register r1 to r0	`EOR r0, r0`
		`ADD r0, r1`
			3 lines

`MOVW r0, r1`	Copy data at register r1 to r0	`EOR r0, r0`
		`ADD r0, r1`
			4 lines

`ADIW r0, 16`	Add 16 to the value at register r0	`EGR r0`
		`SUBI r0, 16`
		`EGR r0`
			1 line

We find that two bytes (`0x00bf`) are redundant towards the end of the SubByte function. These 2 bytes can be removed without affecting the correctness of encryption.

As presented in Fig. 9 in Appendix, we make the following modifications on the ELF file by using a hex editor `HxD`. (1) We insert 2 bytes `0x0032`, which is a 2-byte dummy instruction `adds r2 #0x0`, at address `0x080014ee`. The instruction adds 0 to register r2, which does not affect the correctness of AES. (2) We move the original instructions starting at `0x080014ee` down by 2 bytes. (3) We remove the two redundant bytes `0x00bf`. (4) We update the offset `f7` to `f6` for bne instruction at address `0x080014f2` and offset `f3` to `f2` for bne instruction at address `0x080014f8`. We verify that the outputs of AES remain correct after these modifications.

Note that the above modifications are not the only way to rewrite the ELF file. For instance, we can insert the 2-byte dummy instruction at other addresses before `0x080014fa`. Moreover, we can use other 2-byte instructions (e.g., `0x003d`, which is instruction `sub r5 #0x0`) rather than `adds r2 #0x0`.

Datasets from Case 2. For the original ELF file, we already have dataset `S1_K1`. We collect another dataset `S2_K3_RW` by running a hex file generated by the rewritten ELF file on STM32.

Experiment 2.1: Impacts of Instruction Rewriting. We use `X1_K1_ASM` as the source and use `X2_K4_ASM` and `X2_K4_ASM_RW` as the target respectively. As shown in Table. 8, CNN recovers keys within 3 traces if the traces are both collected from the original assembly code. However, when we test over traces from the rewritten assembly code with the same CNN, the CNN fails to recover the keys. We have a similar observation from the case of ELF files on STM32, where discrepancies caused by instruction rewriting lead to performance drops as presented in Table. 9.

***Observation 2.1** Our results suggest that discrepancies in instruction rewriting can downgrade attack performance, where a CNN may not even recover the keys.*

Experiment 2.2: Mitigating Discrepancies Caused by Instruction Rewriting. As illustrated in Fig. 6, instruction rewriting causes shifts of NICV peaks, which is likely the reason for performance drops. The rewritten assembly on XMEGA leads to a NICV offset of 398 according to Fig. 6a.

We apply adjusting POI to recover keys. Specifically, when a CNN is trained with `X1_K1_ASM` with POI=[1600, 4500], we test the CNN with `X2_K4_ASM_RW` by using different POI offsets. For instance, the attack can recover keys with 189 test traces given offset 363 (i.e., POI=[1963, 4863]) as presented in Table 9. We only report the offset with the lowest KRC in Table 9 and more results from different offsets are highlighted in Fig. 6c.

Table 7: Datasets Associated with Instruction Rewriting

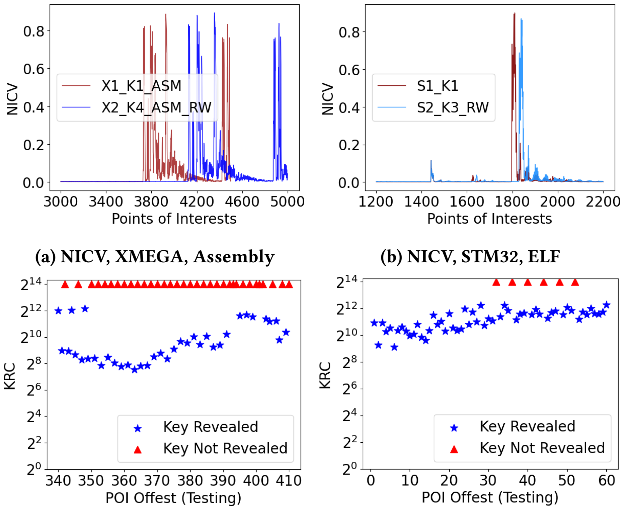
XMEGA	X1_K1_ASM	X2_K4_ASM, X2_K4_ASM_RW
STM32	S1_K1	S2_K3, S2_K3_RW

Table 8: Attack Performance of CNN; Instruction Rewriting, Assembly; Train: 40k (Source), Test: 10k (Target), Source: X1_K1_ASM, POI=[1600, 4500], ⊥: key is not revealed.

Target	POI Offset	KRC
X1_K1_ASM	0	3
X2_K4_ASM	0	3
	0	⊥
X2_K4_ASM_RW	**363 (best)**	**189**
	398 (NICV)	⊥

Table 9: Attack Performance of CNN; Instruction Rewriting, ELF; Train: 40k (Source), Test: 10k (Target), Source: S1_K1, POI=[1200, 2200].

Target	POI Offset	KRC
S2_K3	0	6
	0	2,001
S2_K3_RW	**6 (best)**	**552**
	32 (NICV)	⊥



(a) NICV, XMEGA, Assembly

(b) NICV, STM32, ELF

(c) Train: X1_K1_ASM (POI=[1600, 4500]), test: X2_K4_ASM_RW

(d) Train: S1_K1 (POI=[1200, 2200]), test: S2_K3_RW

Figure 6: NICV and KRC, Instruction Rewriting.

In general, we observe that the attack performance first increases but then decreases (i.e., KRC first decreases but then increases — a "U" shape) when we incrementally update the POI offset towards the NICV offset during the attack. In addition, not every offset reveals the key. Enumerating offsets that are close to the NICV offset can be helpful to identify the offset with lowest KRC. While the offset achieves the lowest KRC is often not the NICV offset itself, it is close to the NICV offset. In other words, identifying the NICV offset can inform us how we should adjust POI in general.

Table 10: Datasets Associated with Optimization Levels

STM32	S1_K1	S2_K3, S2_K3_01, S2_K3_02, S2_K3_03

Table 11: Attack Performance of CNN; Optimization Levels, STM32, Train with 40k (Source), Test: 10k (Target). Source: S1_K1 POI=[1200, 2200]; ⊥: key is not revealed.

Target	POI Offset	KRC
S2_K3	0	6
S2_K3_01	0	⊥
S2_K3_02	0	⊥
S2_K3_03	0	⊥
S2_K3_01	**-94 (best)**	**1,683**
	-136 (NICV)	⊥
S2_K3_02	**-172 (best)**	**204**
	-176 (NICV)	253
S2_K3_03	-756 (NICV)	⊥

Observation 2.2 *Our results suggest that adjusting POI based on NICV can mitigate the discrepancies caused by instruction rewriting and recover keys. On the other hand, even adjusting POI can improve attack performance, the number of test traces needed to recover keys is still (much) higher than the one obtained from the cases with no software discrepancies between training and test data.*

6.3 Impacts of Optimization Levels

Evaluation Scope. We focus on the cases where training data and test data are generated by binaries built from the same C code of AES but with different optimization levels. Specifically, we examine four optimization levels, including Os (default in ChipWhisperer), O1, O2 and O3, with tinyAES on STM32.

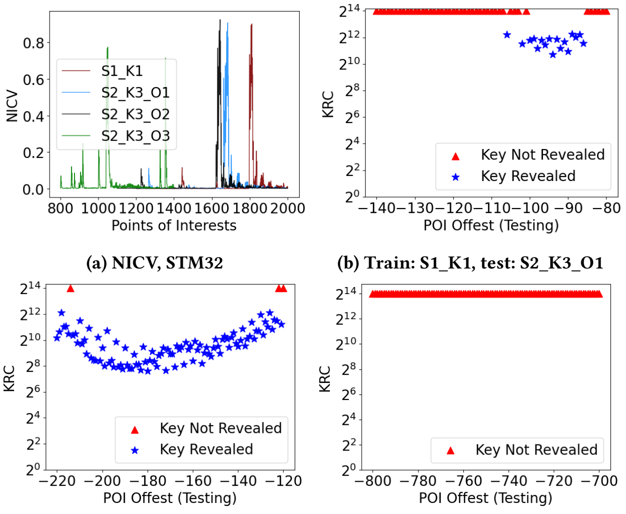
Datasets. We leverage dataset S1_K1 and S2_K3, which were collected using Os in Experiment 0. In addition, we collect three datasets with S2 and key K3, using O1, O2, and O3 respectively.

Experiment 3.1: Impacts of Optimization Levels. As summarized in Table 11, when training data and test data are generated by the same optimization level (e.g., Os), a CNN can recover keys easily. However, when the training data and test data are generated by different optimization levels, a CNN does not recover keys given the same POI. This is expected as different optimization levels lead to different power patterns of AES (as shown in Appendix).

Observation 3.1 *Our results suggest that discrepancies in optimization levels between two devices can significantly downgrade attack performance, where a CNN does not even recover keys.*

Experiment 3.2: Mitigating Discrepancies Caused by Optimization Levels. Different optimizations fundamentally lead to different instructions. It is likely that the leakage points remain present but are shifted. Therefore, we first adjust POI based on NICV. We examine the NICV across different optimizations on STM32 in Fig. 7a. As shown in Table 11 and Fig. 7, when we apply offsets to data from O1 or O2 optimization, the CNN can recover keys. However, adjusting POI is not effective for O3. Failing to mitigate software discrepancies for O3 even by adjusting POI is expected as the NICV peaks are very different compared to the other three optimization levels according to Fig. 7a.

We further examine domain adaptation with ADA. To make it easier for ADA to mitigate discrepancies, we also apply offsets to the POI of each target dataset when we train and test with ADA. Unfortunately, we did not observe improvements in attack performance as shown in Table 12.



(a) NICV, STM32

(b) Train: S1_K1, test: S2_K3_O1

(c) Train: S1_K1, test: S2_K3_O2

(d) Train: S1_K1, test: S2_K3_O3

Figure 7: NICV and KRC, Optimization Levels

Table 12: Attack Performance of ADA, Optimization Levels, STM32, Train: 40k (Source) and 40k (Target); Test: 10k (Target), Source: S1_K1 POI=[1200, 2200], ⊥: key is not revealed.

Target	POI Offset	KRC
S2_K3_01	-94	⊥
S2_K3_02	-172	⊥
S2_K3_03	-756	⊥

Table 13: Attack Performance of CNN, Optimization Levels, STM32, Train: 40k (Source), Test: 10k (Target). Source: {S1_K1, S1_K1_O1, S1_K1_O2, S1_K1_O3}, POI=[800, 2000];

Target	KRC	Target	KRC
S2_K3	9	S2_K3_02	8
S2_K3_01	19	S2_K3_03	7

Next, we collect three additional datasets, including S1_K1_01, S1_K1_02, S1_K1_03, with different optimizations on S1 to perform multi-domain training. Each dataset includes 100,000 power traces. We train a single CNN with a total number of 40,000 traces with 10,000 traces per dataset collected on S1. Then, we test this CNN with 10,000 test traces from each dataset collected on S2. We select POI=[800, 2000] for the training and all the tests as it covers measurements associated with the third byte of SubBytes across all four optimization levels. As presented in Table 13, the CNN can easily recover keys within 20 test traces for each optimization.

Observation 3.2 *Our results suggest that adjusting POI can mitigate the discrepancies caused by optimization levels in some cases. Multi-domain training can overcome discrepancies caused by all optimization levels and recover keys.*

6.4 Impacts of Code Obfuscation

Evaluation Scope. We investigate the cases where training data are generated by a C implementation of AES but the test data are

Table 14: Datasets Associated with Code Obfuscation

STM32	S1_K1	S2_K3, S2_K3_FLA,	S2_K3_ENC, S2_K3_SPL

Table 15: Attack Performance of CNN, Code Obfuscation, STM32, Train: 40k (Source), Test: 10k (Target), Source: S1_K1, POI=[1200,2200], ⊥: key is not revealed.

Target	POI Offset	KRC
S2_K3	0	6
S2_K3_ENC	0	⊥
S2_K3_FLA	0	⊥
S2_K3_SPL	0	⊥
S2_K3_ENC	**466 (best)**	**131**
	464 (NICV)	285
S2_K3_FLA	**1,550 (best)**	**116**
	1,544 (NICV)	247
S2_K3_SPL	440 (NICV)	⊥

generated by an obfuscated version of it. Specifically, we leverage tinyAES as the original C implementation of AES. We utilize Tigress to generate obfuscated C code and associated binaries.

There are overall 32 transformations provided by Tigress [2]. Given the original code and a given transformation, we obfuscate only the SubBytes function. Due to the compatibility with ChipWhisperer and the architectures of microcontrollers, we are able to examine three transformations, including Enc.Arithmetic, Flatten, and Split on STM32. Enc.Arithmetic replaces integer arithmetic with complex expressions. Flatten removes structured flow. Split breaks a large function into smaller pieces.

Generating Obfuscated Code. Given a transformation, such as Flatten, we run the command below to generate an obfuscated file obf_aes.c from aes.c by obfuscating SubBytes function only.

```
$ tigress --Environment=armv7:Linux:Gcc:4.6 --Transform
=Flatten --Functions=SubBytes --out=obf_aes.c aes.c
```

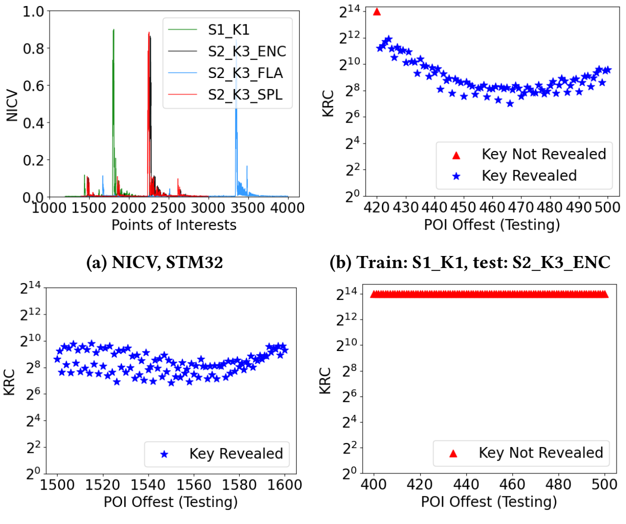
Note that the above command does not directly derive an obfuscated code that can be compiled and flashed successfully to a microcontroller with ChipWhisperer. Additional customized modifications are needed in the obfuscated file obf_aes.c. Details regarding these modifications can be found on our GitHub repository [4]. We test the encryption is still correct after each transformation.

Datasets. We collect datasets of power traces based on each transformation as shown in in Table 14.

Experiment 4.1: Impacts of Code Obfuscation. We train a CNN with data from S1 with no obfuscation and then test it with data from S2 with obfuscation. As we can observe from Table 15, when there are obfuscation discrepancies between the two devices, a CNN cannot recover keys given the same POI.

Observation 4.1 *Our results suggest that discrepancies caused by code obfuscation between two devices can significantly downgrade attack performance, where a CNN does not even recover keys.*

Experiment 4.2: Mitigating Discrepancies Caused by Code Obfuscation. As code obfuscation also rewrites instructions, we first mitigate the discrepancies by adjusting POI based on NICV. We identify the NICV offsets based on Fig. 8a and perform the testing by adjusting POIs. As shown in Table 15 and Fig. 8, the CNN can recover keys for Flatten and Enc.Arithmetic with updated POIs. However, for Split, the CNN still fails to recover keys.



(a) NICV, STM32

(b) Train: S1_K1, test: S2_K3_ENC

(c) Train: S1_K1, test: S2_K3_FLA

(d) Train: S1_K1, test: S2_K3_SPL

Figure 8: NICV and KRC, Code Obfuscation

Table 16: Attack Performance of ADA, Code Obfuscation, STM32, Train: 40k (Source) and 40k (Target); Test: 10k (Target), Source: S1_K1, POI=[1200, 2200], ⊥: key is not revealed.

Target	POI Offset	KRC
S2_K3_ENC	466	⊥
S2_K3_FLA	1,550	⊥
S2_K3_SPL	440	⊥

Table 17: Attack Performance of CNN; Code Obfuscation, STM32, Train: 40k (Source), Test: 10k (Target), Source: {S1_K1, S1_K1_FLA, S1_K1_SPL, S1_K1_ENC}, POI=[1500, 3700];

Target	KRC	Target	KRC
S2_K3	5	S2_K3_FLA	29
S2_K3_ENC	7	S2_K3_SPL	8

We further leverage ADA to mitigate discrepancies caused by code obfuscation on STM32. We also apply offsets to the POI of each target dataset to ease the discrepancies between source and target data. However, ADA does not recover the keys as presented in Table 16. Next, we leverage multi-domain training. We collect three additional datasets, including S1_K1_ENC, S1_K1_FLA, and S1_K1_SPL, on S1. Each dataset includes 100,000 power traces. We train a single CNN with a total number of 40,000 traces with 10,000 traces per dataset from S1. We test this CNN with 10,000 test traces per dataset from S2_K3, S2_K3_ENC, S2_K3_FLA, or S2_K3_SPL separately. We select POI=[1500, 3700] for the training and all the tests. As presented in Table 17, the CNN can reveal keys within 30 test traces for all obfuscations.

Observation 4.2 *Our results indicate that adjusting POI can mitigate the discrepancies caused by code obfuscation in some cases. Multi-domain training can overcome discrepancies caused by all the obfuscations we investigated.*

7 DISCUSSIONS AND LIMITATIONS

More Findings on XMEGA. We also evaluate optimization levels (O1, O2, and O3) and code obfuscations (Split and Flatten) on

XMEGA. We find that adjusting POI based on NICV can always recover keys. Some of the results are presented in Appendix.

Hardware Discrepancies v.s. Software Discrepancies. Given a power trace in a two-dimensional space, as the ones in Fig. 11 in Appendix, x-dimension is the measurement index (or the time) and y-dimension is the power consumption. Hardware and key discrepancies lead to shifts in y-dimension and software discrepancies (alone) result in shifts in x-dimension according to our observations. The aggregation of key, hardware, and software discrepancies lead to distribution shifts in both dimensions, which raise challenges for neural networks to recover keys. Despite these distribution shifts that neural networks need to address, we would like emphasize that these discrepancies should not be considered as countermeasures against side-channel attacks as they do not preserve leakage.

Better Methodology for Studying Software Discrepancies. Besides the four factors we examine, there are other factors that can lead to software discrepancies. For instance, *the version of a cross-compiler* can lead to different machine instructions, and therefore, shifts in terms of leakage points. Discrepancies caused by combinations of multiple factors (e.g., random delays + optimization levels) lead to more severe discrepancies between training and test data.

A better methodology would be first summarizing and categorizing all the primitives/factors that can cause software discrepancies (in essence, different ways of modifying machine instructions) and then identifying which factors should be investigated with a higher priority. In addition, quantifying the distribution shifts (or side-channel leakage shifts) caused by each software discrepancy factor is also an important problem to explore in future work.

More Neural Networks. We only examine one CNN architecture in our evaluation. Other CNN architectures or other neural networks, such as Multi-Layer Perception, that are also effective in side-channel attacks [6], can be investigated against software discrepancies in future studies.

Scalability of Multi-Domain Training. While multi-domain training seems to be the most effective way among the three methods, we would like to point it out that it is extremely difficult for an attacker to scale (in terms of data acquisition and training time) in order to cover all the potential software settings in advance. We do not believe the number of software settings is infinite, however, it is obviously challenging to enumerate all of them.

Improving Domain Adaptation. Domain adaptation has shown promising results in other research areas, unfortunately, it is not helpful in mitigating domain discrepancies caused by software discrepancies in side-channel attacks based on our results. On the other hand, we would like to acknowledge that the ADA we implement is certainly not optimal and there are also other approaches to perform domain adaptation rather than ADA. How to further optimize domain adaptation to address software discrepancies in side-channel attacks remains open. We leave it as future work.

8 RELATED WORK

In the following, we discuss studies that are mostly related to this work. More detailed summaries on machine-learning side-channel attacks can be found in [28, 31].

Deep-Learning Profiling Attacks. Cagli et al. [10] demonstrated that CNNs can recover keys when traces are misaligned due to jittering. Benadjila et al. [6] leveraged Multi-Layer Perceptrons and CNNs in side-channel attacks and demonstrated that CNNs can defeat masking. Maghrebi et al. [27] showed that deep learning can outperform Template Attacks [12]. Several recent studies further investigated the explainability [38], imbalance of data with Hamming Weight model [30], robustness of a neural network [26, 29, 41, 42], as well as hyperparameter tuning [32] in the context of deep-learning profiling side-channel attacks.

Portability of Deep-Learning Profiling Attacks. Several studies [7, 13–15, 17, 18, 33, 40, 43, 44] examined the portability of deep-learning side-channel attacks in the cross-device scenario, where the training device and test device are not identical. These studies focus on cases where the discrepancies between training and test data are primarily caused by hardware discrepancies. Specifically, studies in [7, 14] propose to utilize multi-device training. Pre-processing over feature space [18] or in the frequency domain [44] has shown promising results in overcoming hardware discrepancies. Rioja et al. [33] investigated how to quantify discrepancies of power traces across devices with Dynamic Time Warping. Studies in [11, 43] demonstrate that transfer learning, such as meta-transfer learning and unsupervised domain adaptation with Maximum Mean Discrepancy, can also be utilized to address hardware discrepancies in profiling side-channel attacks. *However, none of these studies investigates the impact of software discrepancies on the portability of deep learning profiling side-channel attacks.*

Deep-Learning Non-Profiling Attacks. Studies in [23, 25, 35] leverage deep learning to perform non-profiling attacks, in which all the traces are captured from a single device but these traces are unlabeled. For instance, Timon [35] proposed differential deep learning analysis, which labels power traces with all the 2^8 possible keys and trains 2^8 neural networks accordingly. The sensitivity of the neural network obtained from the correct key is distinguishable from the sensitivity of others, which allows an attacker to reveal the correct key.

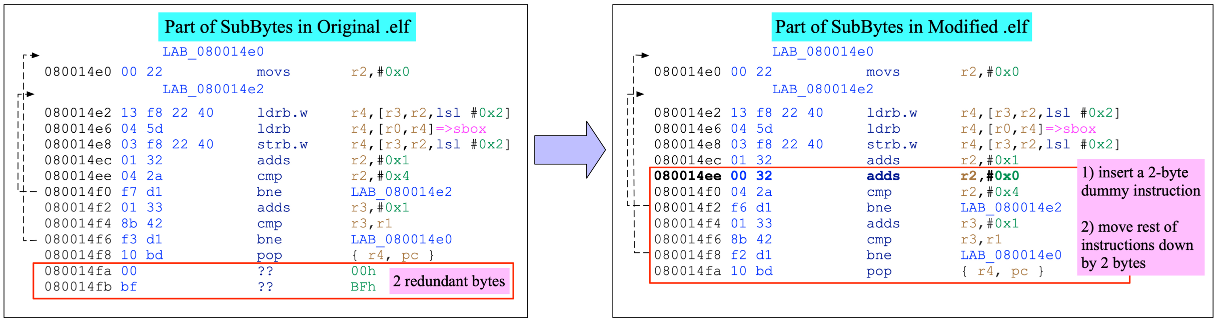
Since all the traces are assumed to be collected from a single device with a fixed key, discrepancies caused by keys, hardware, and software are not a concern in the context of non-profiling attacks.

9 CONCLUSION

In this study, we show that software discrepancies between two devices can significantly affect the portability of deep-learning profiling side-channel attacks. While adjusting POI and multi-domain training have shown promising results in terms of promoting the portability against software discrepancies, each of them still faces limitations. Moreover, there are many problems that remain open related software discrepancies in deep-learning profiling side-channel attacks. We hope that our findings and datasets can serve as a stepping stone for the community to further expand the research in this direction.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers and the shepherd for their comments and suggestions. This work was partially supported by National Science Foundation (CNS-2150086) and NSF IUCRC Center for Hardware and Embedded System Security and Trust (CHEST) under Grant 1018660.



Part of SubBytes in Original .elf

```
                  LAB_080014e0
080014e0 00 22              movs      r2,#0x0
                  LAB_080014e2
080014e2 13 f8 22 40        ldrb.w    r4,[r3,r2,lsl #0x2]
080014e6 04 5d              ldrb      r4,[r0,r4]=>sbox
080014e8 03 f8 22 40        strb.w    r4,[r3,r2,lsl #0x2]
080014ec 01 32              adds      r2,#0x1
080014ee 04 2a              cmp       r2,#0x4
080014f0 f7 d1              bne       LAB_080014e2
080014f2 01 33              adds      r3,#0x1
080014f4 8b 42              cmp       r3,r1
080014f6 f3 d1              bne       LAB_080014e0
080014f8 10 bd              pop       { r4, pc }
080014fa 00                 ??        00h
080014fb bf                 ??        BFh
```

2 redundant bytes

Part of SubBytes in Modified .elf

```
                  LAB_080014e0
080014e0 00 22              movs      r2,#0x0
                  LAB_080014e2
080014e2 13 f8 22 40        ldrb.w    r4,[r3,r2,lsl #0x2]
080014e6 04 5d              ldrb      r4,[r0,r4]=>sbox
080014e8 03 f8 22 40        strb.w    r4,[r3,r2,lsl #0x2]
080014ec 01 32              adds      r2,#0x1
080014ee 00 32              adds      r2,#0x0
080014f0 04 2a              cmp       r2,#0x4
080014f2 f6 d1              bne       LAB_080014e2
080014f4 01 33              adds      r3,#0x1
080014f6 8b 42              cmp       r3,r1
080014f8 f2 d1              bne       LAB_080014e0
080014fa 10 bd              pop       { r4, pc }
```

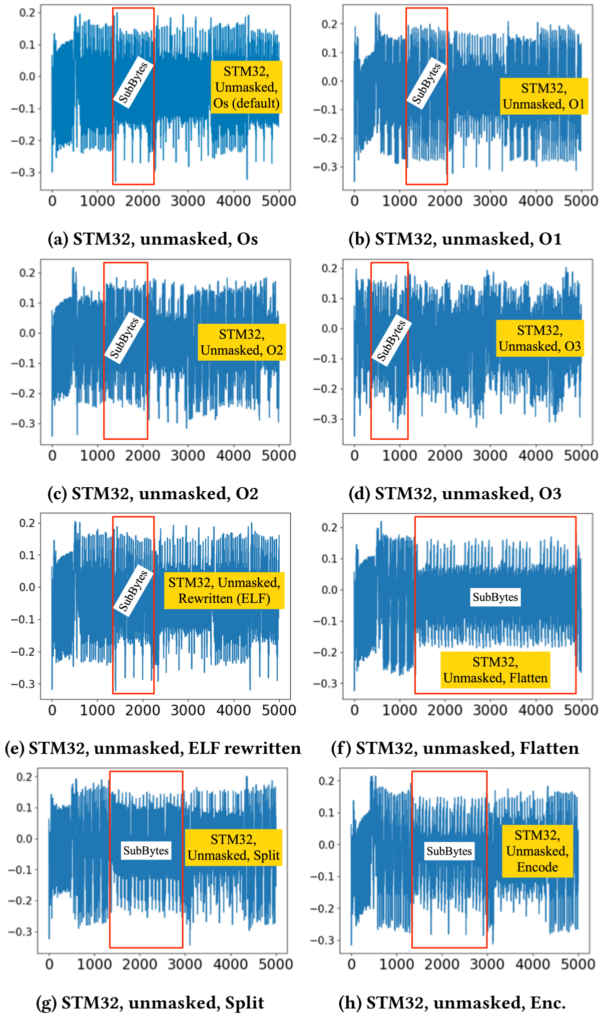
1) insert a 2-byte dummy instruction

2) move rest of instructions down by 2 bytes

Figure 9: Instruction rewriting on the ELF file of tinyAES for STM32.

REFERENCES

[1] [n. d.]. https://ghidra-sre.org/
[2] [n. d.]. https://tigress.wtf/
[3] [n. d.]. https://github.com/newaetech/chipwhisperer
[4] [n. d.]. https://github.com/UCdasec/SoftPower
[5] [n. d.]. https://github.com/ANSSI-FR/secAES-ATmega8515
[6] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas. 2020. Deep learning for side-channel analysis and introduction to ASCAD database. *Journal of Cryptographic Engineering* 10, 2 (2020).
[7] S. Bhasin, A. Chattopadhyay, A. Heuser, D. Jap, S. Picek, and R. R. Shrivastwa. 2020. Mind the Portability: A Warriors Guide through Realistic Profiled Side-channel Analysis. In *Proc. of NDSS'20.*
[8] S. Bhasin, J. Danger, S. Guilley, and Z. Najm. 2014. NICV: Normalized inter-class variance for detection of side-channel leakage. In *2014 International Symposium on Electromagnetic Compatibility.*
[9] E. Brier, C. Clavier, and F. Olivier. 2004. Correlation Power Analysis with a Leakage Model. In *Proc. of CHES'04.*
[10] E. Cagli, C. Dumas, and E. Prouff. 2017. Convolutional Neural Networks with Data Augmentation Against Jitter-Based Countermeasures. In *Proc. of CHES'17.*
[11] P. Cao, C. Zhang, X. Lu, and D. Gu. 2021. Cross-Device Profiled Side-Channel Attack with Unsupervised Domain Adaptation. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 4 (2021), 27 – 56.
[12] S. Chari, J. R. Rao, and P. Rohatgi. 2002. Template Attacks. In *Proc. of Cryptographic Hardware and Embeeded Systems (CHES 2002).*
[13] J. Danial, D. Das, A. Golder, S. Ghosh, A. Raychowdhury, and S. Sen. 2022. EM-X-DL: Efficient Cross-device Deep Learning Side-channel Attack with Noisy EM Signatures. *ACM Journal on Emerging Technologies in Computing Systems* 18, 1 (2022), 1–17.
[14] D. Das, A. Golder, J. Danial, S. Ghosh, A. Raychowdhury, and S. Sen. 2019. X-DeepSCA: Cross-Device Deep Learning Side Channel Attack. In *Proc. of 56th ACM/IEEE Design Automation Conference (DAC'19).*
[15] M. Elaabid and S. Guilley. 2012. Portability of templates. *Journal of Cryptographic Engineering* 2 (2012), 63–74.
[16] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky. 2016. Domain-Adversarial Tranining of Neural Networks. *Journal of Machine Learning Research* (2016).
[17] C. Genevey-Metat, A. Heuser, and B. Gerard. 2021. Train or Adapt a Deeply Learned Profile?. In *Proc. of International Conference on Cryptology and Information Security in Latin America (Latin Crypt'21).*
[18] A. Golder, D. Das, J. Danial, S. Ghosh, S. Sen, and A. Raychowdhury. 2019. Practical Approaches Towards Deep-Learning Based Cross-Device Power Side Channel Attack. *IEEE. Trans. on Very Large-Scale Integration (VLSI) Systems* 27, 12 (2019).
[19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. 2014. Generative Adversarial Networks. In *Proc. of the International Conference on Nerual Information Processing Systems (NIPS 2014).*
[20] G. Hospodar, B. Gierlichs, E. D. Mulder, I. Verbauwhede, and J. Vandewalle. 2011. Machine Learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering* 1, 4 (2011), 293–302.
[21] V. Immler, R. Specht, and F. Unterstein. [n. d.]. Your Rails Cannot Hide From Localized EM: How Dual-Rail Logic Fails on FPGAs. ([n. d.]). https://eprint.iacr.org/2017/608.pdf.
[22] P. Kocher, J. Jaffe, and B. Jun. 1999. Differential Power Analysis. In *Proc. of CRYPTO'99.*
[23] D. Kwon, H. Kim, and S. Hong. 2021. Non-Profiled Deep Learning-based Side-Channel Preprocessing with Autoencoders. *IEEE Access* (2021).
[24] M. Long and J. Wang. 2015. Learning transferable features with deep adaptation networks. In *Proc. of ICML'15.*
[25] X. Lu, C. Zhang, and D. Gu. 2021. Attention - Based Non-Profiled Side-Channel Attack. In *Proc. of 2021 Asian Hardware Oriented Security and Trust Symposium (AsianHost).*
[26] Z. Luo, M. Zheng, P. Wang, M. Jin, J. Zhang, and H. Hu. [n. d.]. Towards Strengthening Deep Learing-based Side Channel Attacks with Mixup. ([n. d.]). https://arxiv.org/abs/2103.05833.
[27] H. Maghrebi, T. Portigliatti, and E. Proff. 2016. Breaking cryptographic implementations using deep learning techniques. In *Proc. of International Conference on Security, Privacy and Applied Cryptography Engineering (SPACE'16).*
[28] M. Panoff, H. Yu, H. Shan, and Y. Jin. 2022. A Review and Comparison of AI-enhanced Side Channel Analysis. *J. Emerg. Technol. Comput. Syst.* (2022).
[29] G. Perin, L. Chmielewski, and S. Picek. 2020. Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020).
[30] S. Picek, A. Heuser, A. Jovic, and F. Regazzoni. 2019. The curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 1 (2019), 209–237.
[31] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina. 2022. Sok: Deep Learning-based Physical Side-channel Analysis. *ACM Computing Surveys* (2022).
[32] J. Rijsdijk, L. Wu, G. Perin, and S. Picek. 2021. Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021).
[33] U. Rioja, L. Batina, and I. Armendariz. 2020. When Similarities Among Devices are Taken for Granted: Another Look at Portability. In *Proc. of AFRICACRYPT 2020.* 337 – 357.
[34] T. Schneider and A. Moradi. 2015. Leakage Assessment Methodology – A Clear Roadmap for Side-Channel Evaluations. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES).*
[35] B. Timon. 2019. Non-Profiled Deep Learning-based Side-Channel Attacks with Sensitivity Analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019, 2 (2019), 107–131.
[36] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell. 2017. Adversarial Discriminative Domain Adaptation. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*
[37] E. Tzeng, J. Hoffman, N. Zhang, K. Saenko, and T. Darrell. 2014. Deep domain confusion: Maximizing for domain invariance. (2014). https://arxiv.org/pdf/1412.3474.pdf.
[38] D. van der Valk, S. Picek, and S. Bhasin. 2020. Kilroy Was Here: The First Step Towards Explainability of Neural Networks in Profiled Side-Channel Analysis. In *International Workshop on Constructive Side-Channel Analysis and Secure Design.*
[39] C. Wang, J. Dani, S. Reilly, A. Brownfield, B. Wang, and J. M. Emmert. 2023. TripletPower: Deep-Learning Side-Channel Attacks over Few Traces. In *Proc. of IEEE HOST'23.*
[40] H. Wang, M. Brisfors, S. Forsmark, and E. Dubrova. 2019. How Diversity Affects Deep-Learning Side-Channel Attacks. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC).*
[41] P. Wang, P. Chen, Z. Luo, G. Dong, M. Zheng, N. Yu, and H. Hu. [n. d.]. Enhancing the Performance of Practical Profiling Side-Chanel Attacks Using Conditional Generative Adversarial Networks. ([n. d.]). https://arxiv.org/abs/2007.05285.
[42] L. Wu, G. Perin, and S. Picek. 2022. The Best of Two Worlds: Deep Learning-assisted Template Attack. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 3 (2022), 413–437.
[43] H. Yu, H. Shan, M. Panoff, and Y. Jin. 2021. Cross-Device Profiled Side-Channel Attacks using Meta-Transfer Learning. In *Proc. of the 58th ACM/IEEE Design Automation Conference (DAC'21).*
[44] F. Zhang, B. Shao, G. Xu, B. Yang, Z. Yang, Z. Qin, and K. Ren. 2020. From Homogeneous to Heterogeneous: Leveraging Deep Learning based Power Anlysis across Devices. In *Proc. of 57th ACM/IEEE Design Automation Conference (DAC'20).*



(a) STM32, unmasked, Os

(b) STM32, unmasked, O1

(c) STM32, unmasked, O2

(d) STM32, unmasked, O3

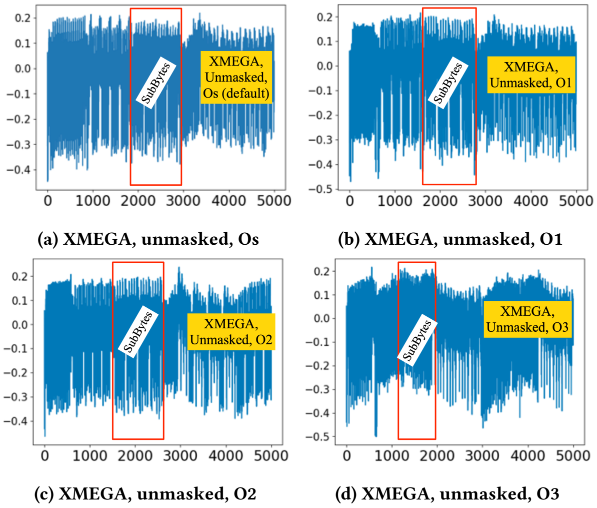
(e) STM32, unmasked, ELF rewritten

(f) STM32, unmasked, Flatten

(g) STM32, unmasked, Split

(h) STM32, unmasked, Enc.

Figure 11: Pattern of Power Traces on STM32



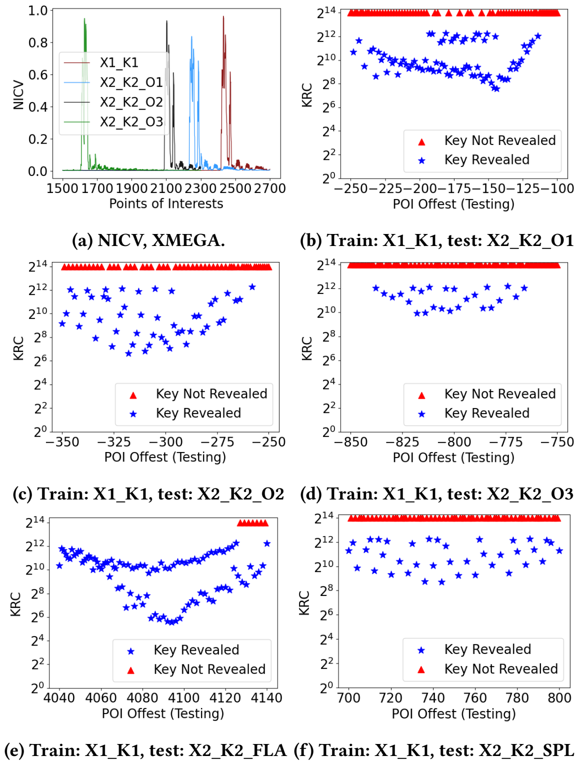
(a) XMEGA, unmasked, Os

(b) XMEGA, unmasked, O1

(c) XMEGA, unmasked, O2

(d) XMEGA, unmasked, O3

Figure 12: Pattern of Power Traces on XMEGA.



(a) NICV, XMEGA.

(b) Train: X1_K1, test: X2_K2_O1

(c) Train: X1_K1, test: X2_K2_O2

(d) Train: X1_K1, test: X2_K2_O3

(e) Train: X1_K1, test: X2_K2_FLA

(f) Train: X1_K1, test: X2_K2_SPL

Figure 13: XMEGA, KRC, Optimization & Obfuscation

APPENDIX

Power Pattern. In Fig. 11 and Fig. 12, we present the pattern of power traces of AES-128 on STM32 and XMEGA. For XMEGA, due to space limitation, additional pattern of power traces obtained from instruction rewriting and obfuscation can be found at [4].

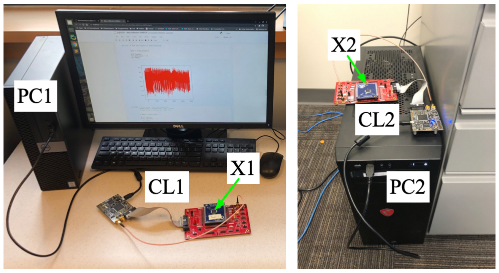


Figure 10: Our data collection setup (was also used in [39])

Table 18: Hyperparameters of CNN [6]

Conv 1	filters: 64; kernel size: 11; stride: 2; Relu
Conv 2	filters: 128; kernel size: 11; stride: 2; Relu
Conv 3	filters: 256; kernel size: 11; stride: 2; Relu
Conv 4~5	filters: 512; kernel size: 11; stride: 2; Relu
AvgPool 1~5	pooling size: 2; stride: 2
Dense 1~2	No. of neurons: 4096; Relu
Output	No. of neurons: 9; softmax