Generalized Power Attacks against Crypto Hardware using Long-Range Deep Learning

Elie Bursztein¹, Luca Invernizzi², Karel Král², Daniel Moghimi¹, Jean-Michel Picod² and Marina Zhang¹

¹ Google, Sunnyvale, USA, scaaml@google.com
² Google, Zurich, Switzerland, scaaml@google.com

Abstract. To make cryptographic processors more resilient against side-channel attacks, engineers have developed various countermeasures. However, the effectiveness of these countermeasures is often uncertain, as it depends on the complex interplay between software and hardware. Assessing a countermeasure’s effectiveness using profiling techniques or machine learning so far requires significant expertise and effort to be adapted to new targets which makes those assessments expensive. We argue that including cost-effective automated attacks will help chip design teams to quickly evaluate their countermeasures during the development phase, paving the way to more secure chips.

In this paper, we lay the foundations toward such automated system by proposing GPAM, the first deep-learning system for power side-channel analysis that generalizes across multiple cryptographic algorithms, implementations, and side-channel countermeasures without the need for manual tuning or trace preprocessing. We demonstrate GPAM’s capability by successfully attacking four hardened hardware-accelerated elliptic-curve digital-signature implementations. We showcase GPAM’s ability to generalize across multiple algorithms by attacking a protected AES implementation and achieving comparable performance to state-of-the-art attacks, but without manual trace curation and within a limited budget. We release our data and models as an open-source contribution to allow the community to independently replicate our results and build on them.

Keywords: Deep Learning · Side-Channel Analysis · AES · ECC

1 Introduction

Cryptographic co-processors, which are widely used to perform security-sensitive operations, can be vulnerable to side-channel attacks. These attacks aim to extract the secrets these chips safeguard, such as AES [KJJ99] or RSA keys [Koc96]. Broadly speaking, side-channel attacks recover secret data by collecting signals such as timing, power consumption [MOP08], and electromagnetic emissions [QS01] while the chip runs computations. These signals are later processed using statistical methods [CRR03] or machine learning techniques [MPP16] to recover the targeted secret information (e.g., the AES key). Attackers can then use the recovered secrets to bypass vital security features such as secure boot,¹ remote attestation [Fid15, MSEH20], and identity protection.²

In recent years, side-channel attacks assisted with machine learning (SCAAMLS) [PB19] due to their superior accuracy, reduced manual work, and lesser need of domain knowledge, have started to replace profiling attacks such as Template attacks [CK14, CRR03]. In particular, SCAAML attacks have proved to be effective at recovering AES keys, even

¹https://source.android.com/docs/security/features/verifiedboot
²https://source.android.com/docs/security/features/biometric/measure
when facing strong masking countermeasures [ERR+18, MPP16, CDP17].

Despite their clear effectiveness against specific targets, several obstacles limit SCAAMLs broad adoption during the product development cycle [PPM+23], where engineers need to be notified of potential leaks in a matter of hours or days to stick to the production schedule, including:

- **Lack of cross-algorithm generality**: SCAAMLs have not been reported to generalize beyond a single protected cryptographic algorithm [LBM15, MPP16, CDP17, ERR+18, MS23, WHJ+21, HGG19, ZS20, WP20, PP21, WAGP20, AGF23, WPP22, ZBH20, PCP20, RBA20, ZSX+20, BCH+20, LZC+21]. There is currently no known machine learning technique that is able to attack both highly protected AES and ECC for example.

- **Lack of cross-implementation generality**: Current attacks rely on custom ML architectures tailored to a very specific target [PP21, WAGP20, AGF23, WPP22, ZBH20, PCP20]. Targeting a different implementation that uses different countermeasures requires to manually modify the attack.

- **High-expertise requirements**: So far SCAAMLs require expertise to not only modify the neural network architecture and its objectives but most of them also require expert manual pre-processing of the traces (e.g., [LZC+21]) so they can be used by the neural network.

In this paper, we provide the first step toward addressing these limitations by proposing a novel deep-learning architecture, GPAM (Generalized Power Analysis Model), that is able to perform fully automated power side-channel attacks against multiple protected algorithms, namely ECC and AES, countermeasures, and implementations. GPAM is designed to work on raw traces straight from the oscilloscope doing away with the expensive requirement of expertly preprocessing traces before performing attacks. Our novel architecture, presented in Section 5, combines temporal patchification [LMW+22] to process the very long traces generated by algorithms incorporating countermeasures, state-of-the-art Transformer encoder blocks [HDLL22] to efficiently identify long-range trace data relationships, and multi-task learning [Rud17] to allow the model to attack masked implementations.

We demonstrate GPAM effectiveness by carrying out power analysis attacks against four protected ECDSA implementations in Section 6. These implementations counter-measures range from a simpler-to-defeat constant-time countermeasure to masking protections that are considered resistant to side channels attacks [Cor99, PCBP21, GRV17, RLM21]. Specifically, GPAM is able to recover the four most significant bits of the secret scalar with an accuracy between 71.86% to 96.39% depending on the targeted implementation. At that level of accuracy, combining the model predictions confidence with a lattice attacks is enough to recover the full secret key [HGS01, NS02], as demonstrated in Section 6.8. To the best of our knowledge, this is the first time that these highly-protected ECDSA implementations have been proven to be vulnerable to power side-channel attacks, demonstrating that GPAM architecture is not only general but also highly effective at attacking state-of-the-art hardware defenses. We note that generalized models, such as GPAM, fulfill a different need than custom attack models. Custom attack models excel at uncovering vulnerabilities in high-value targets, but require the expertise of side-channel specialists. Instead, generalized models empower non-experts, such as implementation engineers, to evaluate the side-channel security of their designs without specialized attack knowledge. As such, they complement, and not supersede, custom attack models.

In Section 7, we show-case GPAM’s ability to generalize to multiple cryptographic algorithms without architectural changes by successfully recovering masked AES keys. When compared with state-of-the-art attacks that rely on manual trace pre-processing and hand-tuning (which GPAM does not require), GPAM achieves comparable performance.

Last but not least in Section 6.8, we demonstrate that GPAM generalization capabilities
also extend beyond white-box attacks by demonstrating its ability to recover hardware masked ECC scalar in a black-box settings.

Overall, the sum of our experimental results highlights that this new generation of generalized automated attacks is competitive with algorithm-specific state-of-the-art approaches for evaluating power leakage countermeasures. Moreover as discussed in Section 5.3, the operational costs of adapting GPAM to a new target via automated hyper-tuning, a few hours of GPU time, is considerably lower than hiring side-channel experts. Our attack generality, speed, and cost-effectiveness move us closer to more secure chips by empowering design teams to incorporate automated countermeasure testing as part of the development process.

To allow the community to independently replicate our results and get us closer to the standardization of fully-automated side-channel leakage evaluations we open-source both our models and datasets under the Apache 2 Licence at [anonymized].

Ethics This research was intentionally performed on research implementations, not production ones. Accordingly, these results do not warrant a coordinated responsible disclosure.

2 Background

This section provides the key background information on cryptography, side-channel attacks, and deep learning needed to understand the paper.

2.1 Elliptic-Curve Cryptography

Elliptic-curve cryptography (ECC), which supports both key exchange and digital signatures, comprises public parameters, and a public/private key pair. Public parameters include an elliptic curve \( E \), a point \( G \) on the curve, and the integer order \( n \) of \( G \) over \( E \). The secret key \( d \) is a random integer satisfying \( 1 < d < n - 1 \). The public key is calculated as \( Q = d \times G \) (\( \times \) is the scalar multiplication operation supported by curve \( E \)). As relevant to this paper, to generate a signature for a message hash \( h \), ECDSA algorithm chooses a random secret \( k \) such that \( 1 < k < n - 1 \), computes \((x, y) = k \times G, r = x \mod n\), and \( s = k^{-1}(h + r \cdot d) \mod n\), and outputs \((r, s)\) as the signature pair.

It is critical for the private key \( d \) and the per-message random secret \( k \) to remain secret. An attacker who acquires one instance of \( k \) for a known signature can simply calculate the private key as \( d = r^{-1}(s \cdot k - h) \mod n \). An attacker who can recover part of \( k \) can apply lattice-based cryptanalysis to recover the private key from partial knowledge of \( k \) from several signature-generation operations [HGS01, NS02].

2.2 Side-Channel Attack and Defense

Side-channel attacks (SCA) target the execution of cryptographic algorithms [MOP08, Koc96]. During the execution, certain physical signals may be generated by intermediate computations that depend on the secret data bits being processed. The attacker can build a distinguisher that identifies which signals are related to which secret bits. This is typically accomplished by attacking each segment of the key separately. For instance, we build a distinguisher that can find the correct key byte for AES, which can be repeated for each of the 16 different key bytes of the 128-bit AES key.

There are two primary scenarios for constructing a distinguisher: In direct attacks,
such as SPA [MOP08] or DPA [KJJ99], the attacker attempts to retrieve the key from traces without prior modeling of the target. In profiling-based attacks, e.g., Template attacks [CRR03], the attacker first constructs a model based on previous observations of the target (or a similar one). We focus on white-box profiling-based attacks, which are valuable for assessing the security of an implementation against a strong, well-informed attacker.

One can use masking countermeasures to mitigate side-channel attacks by disrupting the statistical correlation between intermediate values and the physical signal, e.g., power consumption. To achieve this, implementations can generate a random value and combine it with secret parameters and intermediate values during computation. As a result, the computation is carried out using blinded secrets instead of cleartext ones.

A common protection for ECC implementations is to randomize the secret integer \( d \) or \( k \) during scalar multiplication [Cor99]. For this, implementations can add a random multiple of the curve order \( n \) to the private integer \( k \) as \( k' \rightarrow k + r \cdot n \). Later on, when computing the scalar multiplication \( k' \times G \) as in the signature generation, it results as \( (k + r \cdot n) \times G = k \times G + r \cdot n \times G \). Since \( n \times G \) is equal to the point at infinity (the identity element), the expression simplifies to \( k \times G \). Randomizing the secret integer can also be achieved using the euclidean division \( k = [k/r] \cdot r + (k \mod r) \), or the secret integer can be divided into multiple random shares for extra security \( k = k_1 + \ldots + k_m \).

In this paper, we evaluate the security of ECC masking with single and double shares that are considered secure when the random share \( r \) is chosen in a way that \( \|r\| \geq \|n\|/2 \) (see [RLMI21, RIL20, GRV17]) where \( \|n\| \) stands for the bit-length of a natural number \( n \). These implementations include hardware-accelerated constant time scalar multiplication (CM0), additive masking (CM1), multiplicative masking (CM2), and a combination of the previous two (CM3). For details, see Section 6.1.

2.3 Deep Learning

Throughout the paper, we assume a certain familiarity with standard deep-learning terms such as layer, activation function, and loss. Those terms are defined in widely available textbooks (e.g., [GBC16] and [Cho21]).

To process long traces efficiently GPAM borrows ideas from the recent advances in image patchification techniques which were introduced in vision transformers to efficiently process images [LMW22].

In terms of architecture, GPAM leverages the Transformer architecture [VSP+17] which is at the heart of the recent breakthroughs in deep-learning including large language models (LLMs) such as chatGPT and Gemini. What makes the transformer architecture well-suited to side-channels attacks is its use of self-attention [VSP+17], which enables the model to efficiently understand long-range dependencies and capture contextual information. These abilities are key to build a generic and efficient side-channel attack model as it allows it to exploit complex data leaks that occur through interconnected relationships between distant data points. In our work, we use a high-performance transformer block called GAU (Gated Attention Unit) which was introduced by Hua et al. [HDLL22]. GAUs enhance the Transformer encoder block by replacing the vanilla attention and feed-forward network with a combined gating mechanisms that improve data representation and computation speed, leading to faster training and improved accuracy.

Activation functions play a key role in model accuracy by introducing different form of non-linearity. Different functions are better suited to different type of data and use-cases. As suggested by [XTG+20], we use swish [RZL17], a smooth activation function which is more robust in the presence of counter-measures. Last but not least GPAM heavily relies on using multi-task learning [Car98] to converge and generalize.
3 Threat model

We assume the attacker has access to a clone of the targeted hardware and the tools to collect power traces following side channel attacks’ standard assumptions [CRR03].

Our main threat model is the white-box model, even though, as illustrated in Section 6.8, it is also able to perform well in a black-box setting. Following SCA standard model, we also assume attacks are carried in two phases: the training phase during which the attacker collects data using the cloned hardware to train their attacks and the attack phase where the attacker is attempting to recover secret from the targeted device.

During the attack phase, the model exclusively processes the raw traces from the targeted device to recover the targeted secret without any access to the countermeasure parameters regardless of the threat model considered. We ensure our dataset collection process is consistent with this modus operandi by using two different chips to collect our data: one chip is used for creating the training and testing data while the other one is used to collect the holdout dataset used for attack evaluations.

During training, we consider two threat models:

1. Black-box threat model: In this threat model, the attacker has no knowledge of the deployed countermeasures, only knows the input and output of the cryptographic primitive and can only control the inputs (chosen text attack).

2. White-box threat model: In this threat model, the attacker has full knowledge of the countermeasures used and they control all the protection parameters during the training phase. This is the main threat model used in this paper because it mimics the level of access that chip development teams or certification evaluators have.

4 Related Work

Machine-learning (ML) side-channel attacks. Machine learning has been repeatedly shown to be an effective approach to SCA. For example, Lerman et al. [LBM15] outperformed template attacks [CRR03] in recovering keys from masked implementations of AES, leveraging classical ML algorithms such as support–vector machine (SVM). Maghrebi et al. [MPP16] later applied deep–learning algorithms, including CNNs and LSTMs, to attack AES. Bursztein et al. [B19, BP19] then proved the feasibility of full-trace attacks using deep learning. These attacks tend to require a higher number of traces than classical attacks, though other researchers also adopted MLPs and CNNs to reduce the number of traces required [CCC19, ZBH21, CDP17].

To improve upon these early results, research is still needed to overcome the following three challenges:

(1) Trace preprocessing: This remains a costly endeavor done by experts. There are works that develop techniques to address this issue. Notably, Won et al. [WHJ21] developed a framework based on a multi-scale CNN to enable the integration of user-defined preprocessing phases. Hettwer et al. [HGG19] explored various image-classification metrics for finding points–of–interest in the signal. Zhou and Standaert [ZS20] proposed a technique based on residual networks for aligning SCA traces. Wu and Picek [WP20] used autoencoders to filter out noise added by mitigations such as clock jitter and random delays. Transformations of one dimensional traces into two dimensional images and using established network architectures for images have been studied by [HHGG20].

Direct use of the whole traces remains rare when the traces are long. Even very recent publications [HCM24] suggest that using raw traces of tens of thousands of points is still not a solved problem. To the best of our knowledge, the following are the only papers which directly target traces of at least 100k points using ML (the threshold is somewhat arbitrary since for each threshold there are papers which almost make it, e.g., [GJS20].
with 65k point traces). [MBC+20] target AES implementations automatically protected by code polymorphism (traces up to 160k points). Lu et al. [LZC+21] developed an ML architecture (autoencoders and attention mechanism) acting directly on raw traces of up to 300k samples to target AES implementations from public datasets. [GLS22] directly use traces of length up to 219k samples from the CHES 2020 contest. Our model has improved the result of [LZC+21] on the ASCADv1 variable key dataset. Our approach improves on prior art as it does not require trace preprocessing and can support very long traces, up to 16 million samples and up to 1 million points on a public dataset.

(2) Generalizability: Prior art has mainly focused on identifying the optimal network architecture for each device, implementation, and crypto algorithm [PP21, WAGP20]. This is an effective strategy to find an optimal solution for a specific attack configuration, but it is not clear how well it serves embedded engineers trying to identify leaks in a new product. Some works are addressing this issue, by searching for the right ML architecture based on various tools such as Information Theory [AGF23], Bayesian Optimization [WPP22], and Gradient Visualization [ZBH20]. Pernin et al. [PCP20] take a different approach, using an ensemble of ML models based on average class probabilities to improve generalization. Whereas there are works that target multiple implementations (e.g., [WCPB21]), to the best of our knowledge, no prior work has studied generalization across multiple algorithms (e.g., AES, ECC, RSA). Our approach differs from prior art as we find a single architecture capable of generalizing across devices (with identical model weights), implementations, and algorithms (using the same tunable architecture), thus reducing training costs and heading toward a fully-automated SCA leakage evaluation for hardware certifications.

(3) Portability: Identical hardware devices, even when originating from the same production line, exhibit minute physical variations that result in differences in their power traces. ML models that generalize across devices are superior, as during training, they do not need access to the devices they will eventually attack. Prior research has looked into incorporating device-to-device variation into SCAAML training [RBA20, ZSX+20, BCH+20]. Our holdout datasets are also captured on different physical chips.

Side-channel attacks on ECC. Single-trace side-channel attacks on ECC aim to recover most of the secret bits in one execution of scalar multiplication. This is ideal for signature schemes, like ECDSA, which performs scalar multiplications on a fresh integer every time. However, these attacks [SI11, JB17, NC18, HIM+14, WPB19] have only been successful when scalar blinding has low entropy. Most recently, Pernin et al. [PCBP21] showed ML’s effectiveness in unsupervised attacks, recovering 90% of the secret bits when scalar blinding is performed with 32 and 64 randomly-generated bits. [RIL20] leveraged ML to attack ECC key generation by collecting multiple traces from scalar multiplication for the same secret.

Prior work has also shown lattice-based attack’s effectiveness in recovering keys from partial leakage collected on real cryptographic chips when no masking countermeasure is applied [MSEH20, JSSS20, RLMI21]. However, the effectiveness of SCAAML in assisting lattice attacks and bypassing stronger countermeasures is unknown. Goudarzi et al. [GRV17] combine lattice attacks with side-channel (using hypothetical SNR analysis) and show that attacks are probable when $\|r\| \geq \|n\|/2$, $\|r\| \geq 128$ bits in our case, and even for higher-order masking with more than one share.
5 GPAM

In this section, we present the GPAM (Generic Power Analysis Model) architecture and discuss how we train it. We start by discussing the training objective along with the metrics used to evaluate convergence. Next, we detail GPAM model architecture. Finally, we discuss relevant implementation details.

5.1 Training objective and metrics

Training objective GPAM’s training objective is to predict the value of a specific key byte $k_i$. Following standard machine learning practices, we cast this problem as a classification problem where the model is tasked to produce the correct byte value out of 256 possibilities. In practice, this translates to the model outputting softmax probabilities $P$ for every possible key candidate $c$, i.e., $P[k_i = c]$ for $c = 0x00, \ldots, 0xFF$, each indicating the likelihood that the predicted key byte $k_i$ is equal to $c$. We use categorical cross-entropy as our loss function.

Metrics We use the following metrics to evaluate GPAM performance:

- **Accuracy** is our main metric. It is defined as the categorical accuracy of the output, meaning the ratio of which the model predicts the correct output. The baseline accuracy for a random guess is $1/256 = 0.39\%$. We will indicate it in the rest of the paper with the $\sigma$ symbol.
- **Rank** is the position of the correct value in the ranking of predicted byte values, sorted in descending order by probability. A rank of zero is assigned to the highest probability and 255 is assigned to the lowest probability, given that a byte has 256 possible values.
- **MaxRank** is the maximum rank over a set of model predictions for a batch of examples. The baseline MaxRank is 255. A value less than 255 implies that the key space required to bruteforce the correct value, in the worst case, has been successfully reduced, as each correct prediction is contained in a smaller range of values.
- **MeanRank** is the average rank of predictions. The baseline MeanRank for a random guess is 127.5. A lower value implies that the key space was successfully reduced and that on average, the correct value is within a smaller range of values.
- **Confidence** reflects the standard margin sampling confidence [BJSR22] the difference between the highest and second-highest probabilities (i.e., the value $prediction.sort(); prediction[-1] - prediction[-2]$). Intuitively, this metric is one of the most informative way to measure how much the model is confident that the predicted value is correct.

These metrics are well suited to inform an evaluator about the presence of leakage. For instance, one such indicator is the accuracy rising over the threshold given by the $3\sigma$ rule. However, the leakage detected by these metrics does not imply a successful attack. To account for this, we also evaluate models as part of end-to-end attacks from trace collection to recovered key to understand their real-world performance. For example, we attack ECDSA end-to-end in Section 6.8, and AES in Section 7.1.

Following machine learning best practices, we conduct experiments with the GPAM model on the test split (called validation split by some authors), then pick the best model according to the metrics discussed above, and evaluate only once on the holdout (which is collected on a different chip) by carrying the end to end attack.

5.2 Architecture

At a high-level, GPAM is composed of three functional components, as depicted in Figure 1: a temporal patchification stem designed to group traces into a sequence that preserves the
The last component of GPAM is its multi-headed DAG (directed acyclic graph). This to do so, GPAM uses a trunk made of three state-of-the-art GAU transformer encoder blocks, the trunk also includes a combiner module made of latent representation of the traces needed by the heads to predict the targeted values. Multi-task learning, i.e., predicting multiple values at once.

5.2.1 Temporal Patchification

The temporal patchification stem has two goals:

1. Preserve the temporal inductive bias while making the sequence faster to process by transformer encoder blocks by grouping adjacent points – recall here that attention computation cost is quadratic in the length of its input sequence. To achieve this we reshape the trace into blocks of N contiguous non-overlapping chunks, or “patches”. This approach, while slightly different, is inspired by state-of-the-art image patchification techniques [LMW+22].

2. Potentially provide positional information to allow the transformer encoder block to perform efficiently. This is done by injecting global positional encoding information to the sequence [CTB+21].

5.2.2 Trunk

The trunk’s main function is to attend to the patchified sequence by extracting the latent representation of the traces needed by the heads to predict the targeted values. To do so, GPAM uses a trunk made of three state-of-the-art GAU transformer encoder blocks [HDLL22] that are able to isolate and process long-range interacting features. In addition to the transformer blocks, the trunk also includes a combiner module made of convolutional layers that is meant to combine the output of the three encoder blocks into a unified latent representation. Combining the output of the encoder blocks instead of using the output of the last one, as traditionally done in NLP, is useful because each block extracts features at a different level of “abstraction”. Those multi-level representations are commonly used in other signal processing applications such as speech recognition [CZH+21].

5.2.3 Heads and Relational Outputs

The last component of GPAM is its multi-headed DAG (directed acyclic graph). This component is designed to achieve two goals:

1. **Allow multi-task learning**: GPAM relies on multi-task learning [Car98] to perform efficiently against masked implementation, as reported in Section 6.4. Multi-task learning is accomplished by not only predicting the targeted byte value but also predicting intermediate values such as the mask and random nonce values.

2. **Inject domain expertise**: Standard multi-task learning involves jointly predicting
values without establishing relation between the outputs. We found out, as reported in Section 6.4, that we can increase GPAM performance by representing the outputs as a DAG where intermediate outputs feed into the byte prediction output, as depicted in Figure 1. This allows the model to benefit from expert understanding and makes it easier for it to learn which intermediate values are useful for computing a given output. We note that defining those relations is fully configuration driven and does not require to change the model architecture or fiddle with the code.

5.2.4 Heads design

![Figure 2: Single head output.](image)

Unlike standard transformer architectures, where the output is a single layer, we discovered during our architecture search that having a deeper head architecture improves model performance. As visible in Figure 2, GPAM head architecture comprises several dense layers and a single dropout layer. Extensive initial testing during the architectural development phase revealed that adding normalization layers, residual connections, or more dropout layers did not seem to improve model performance or convergence speed.

5.3 Hyper-parameter tuning

GPAM is designed to be automatically hyper-tuned (using Keras-tuner) to quickly adapt to new cryptographic implementations. To minimize GPAM production costs, we focus on reducing the tuner search space to a minimum. We perform a one-time extensive architecture search to isolate which parameters should be hyper-tuned and which should be considered canonical. We emphasize that this search is a one-time cost paid by this research, and will not be run when deployed in an automated testing environment. For example, the activation function used (Swish), the number of layers per head, and the number of GAU blocks (3) all proved to be close to optimal choices across algorithms and implementations and therefore were excluded from hyper-tuning.

<table>
<thead>
<tr>
<th>Table 1: Model hyper-parameters for each targeted implementation used in this paper.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Batch size</strong></td>
</tr>
<tr>
<td>Batch size</td>
</tr>
<tr>
<td>Steps per epoch</td>
</tr>
<tr>
<td>Epochs</td>
</tr>
<tr>
<td>Target learning rate</td>
</tr>
<tr>
<td>Merge filter 1</td>
</tr>
<tr>
<td>Merge filter 2</td>
</tr>
<tr>
<td>Trace length</td>
</tr>
<tr>
<td>Patch size</td>
</tr>
</tbody>
</table>

All in all, at the end of the architecture search, GPAM only requires 8 parameters to be tuned for each new application. The values of those 8 parameters for all the ECC and AES implementations targeted in this paper are reported in Table 1.

- **Batch size** depends purely on the hardware available for training, and should be
maximized as much as the GPU memory allows (as a standard machine learning practice, doubling \textbf{batch size} implies halving \textbf{steps per epoch} and doubling \textbf{target learning rate}).

- \textbf{Trace length} and \textbf{patch size} depend on the size of the dataset, and the capture sample rate. The Patch size is the square root of trace length (rounded).
- \textbf{Target learning rate} and \textbf{number of epochs} depends on the difficulty for the optimizer to learn. Learning rate is to be searched for in logarithmic scale (good values are 0.001, 0.0001, 0.00001).
- \textbf{Merge filter 1} and \textbf{Merge filter 2} are used to make the trunk output length more manageable. Merge filter 1 and 2 are searched among the powers of two, and as a rule of thumb merge filter 2 is half of the merge filter 1.

We emphasize that hyper-tuning the 8 parameters is straightforward and automated. For example, we hyper-tuned GPAM to attack ASCADv2 using NVIDIA RTX 4090 GPUs. The training totaled 29 days (cumulative across GPUs), which amounts to 7 hours spent training for each parameter configuration. We note here that during hyper-tuning we do not train fully, instead we only use 10% to 30% of the number of epochs used for the final full training. This amount of compute being enough to figure out which parameter values are the best.

Using a server with eight GPUs, a standard number when using NVIDIA SMX technology, this hyper-tuning search can be done under 3.6 days. For example running this training on a similar configuration on Google Cloud Compute costs under $2.5k^3$. This hyper-tuning can alternatively be performed on a local server with equivalent features if a local deployment is preferred, albeit for a higher startup cost. This is a comparatively small price to pay to avoid having experts manually preprocess traces and manually tweak attack parameters, whose cumulative hourly rate can quickly surpass this dollar value. Also, note that this hyper-tuning price has to be paid only once per targeted combination of hardware platform, algorithm, and countermeasure.

5.3.1 Weight Initialization Influence

Training instability is a well-known issue in deep learning in general, and for transformers in particular, where an unlucky initialization can lead to model collapse or sub-par performance (see for instance [NS19]). Following machine-learning practices, we mitigate this issue by using a custom learning-rate scheduler. We start from a low initial learning rate value, then steadily increase it to reach its target value, and finally decrease it with a cosine decay as the model converges [LH16]. During the warm-up phase, a low learning rate allows the model to perform smaller steps along the gradient, reducing the influence of the weight initialization.

5.4 Implementation

We implement the GPAM architecture and conduct training using TensorFlow [AAB+15] and the Keras API [C+15]. We use Keras Tuner [OBL+19] to hyper-tune GPAM automatically. The temporal patchification code was specially designed and implemented for GPAM. The GAU layer is a custom implementation based on the pseudo-code provided in [HDLL22]. The relational output technique is our own contribution and its implementation unique to GPAM. All in all, the GPAM code is about 1,000 lines of Python.

The estimated training times for attacking the various datasets used in the paper, using

\footnote{This price is calculated in September 2023 with Google Cloud Pricing Calculator (https://cloud.google.com/products/calculator) running a \texttt{a2-highgpu-8g} host for 3.6 days. In reality, the GPUs offered by that configuration (A100 80GB) are slightly more performant than our 4090 cards (benchmark https://lambdalabs.com/gpu-benchmarks), so this price tag estimate is an upper bound.}
Table 2: Training time estimates.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CM0</th>
<th>CM1</th>
<th>CM2</th>
<th>CM3</th>
<th>ASCADv2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training time [hours]</td>
<td>1</td>
<td>24</td>
<td>48</td>
<td>150</td>
<td>20</td>
</tr>
</tbody>
</table>

an NVIDIA RTX 4090 as reference GPU, are reported in Table 2. In practice, we use multiple servers with various GPU configurations, as the total computation required to complete all our experiments presented in this paper requires over 1 year of computation. As mentioned throughout the paper, we did our best to avoid running unnecessary experiments to minimize carbon emissions.

6 Generic attacks against hardware-protected ECC

In this section, we evaluate GPAM’s ability to generalize to multiple hardware-protected implementations by performing side-channel attacks against scalar multiplication on four distinct implementations of ECDSA. The targeted hardware implementation includes a constant-time implementation and three distinct algebraic masking implementations that are considered state-of-the-art protections [Cor99, BR23]. We start by describing the implementations targeted, then describe our collection process. Next, we evaluate GPAM’s performance against those datasets. Finally, we discuss how we can attack the ECDSA signature scheme by combining the partial nonce $K$ recovered via GPAM and a lattice attack.

Here we answer the following questions:

1. Does GPAM generalize to multiple hardware implementations? (Section 6.3)
2. Is Multi-task learning needed to detect leakage in a protected implementation, and if yes, which tasks are needed? (Section 6.4 and Section 6.5)
3. Are related outputs only learning a function of other outputs, or are they also using the latent representation provided by the trunk? (Section 6.6)
4. How many traces are needed for GPAM to successfully detect leakage in various implementations? (Section 6.2)

6.1 Targeted hardware implementations

Given our goal to evaluate GPAM against highly-protected hardware implementations, we use the NXP K82F dedicated cryptographic accelerator (LTC – LP Trusted Crypto) as a base for all implementations to perform constant-time hardware-accelerated scalar multiplication and point addition. Relying on this accelerator ensures that all our implementations are not vulnerable to timing attacks or software-based leakages. All our ECDSA implementations use the elliptic curve FRP256\(^4\) from ANSSI, but our results apply equally to other curves (e.g., NIST P-256), as the scalar multiplication algorithm is typically the same for all Weierstrass curve implementations.

6.1.1 Countermeasures

We evaluate the following four implementations to highlight the effectiveness of GPAM against increasingly stronger protections. Here, we use \(\text{\texttt{\#}}\) to denote computation done in software and \(\text{\texttt{\#}}\) to indicate computation done on the chip. The $k \leftarrow \delta_L(N)$ notation indicates the generation of a random number with $N$ bits where each bit is selected independently and uniformly at random. The four implementations considered in this section are:

1. Constant-time execution (CM0). A simple countermeasure effective against

\(^4\)https://neuromancer.sk/std/anssi/FRP256v1
timing attacks, but not power side channels. It exclusively relies on our chip’s constant-time accelerated scalar multiplication, without randomizing the secret multiplier.

2. **Additive masking (CM1):** This implementation is significantly more resistant to power side channel attacks compared to CM0, thanks to the addition of multiplier masking. A random integer $r$ is added to $k$ so the scalar multiplication executes on the blinded secret scalar. More formally, it

(a) Chooses an independent 256-bit random mask $r$.
(b) Computes the difference $km$ between the secret multiplier $k$ (the ECDSA nonce) and the mask $r$.
(c) On chip, computes $P_{km} = km \times G$ and $P_r = r \times G$.
(d) On chip, computes $P_{km} + P_r$ and returns this value.

Here is the pseudo code used to implement this scheme:

\[
\begin{align*}
    k &\leftarrow \delta_L(256) & &\text{(secret multiplier)} \quad \\text{(CM1 random mask)} \\
r &\leftarrow \delta_L(256) & &\text{(random mask)} \quad \\text{(CM1 random mask)} \\
km &\leftarrow (k - r) \mod n & &\text{(scalar multiplication)} \quad \\text{(CM1 random mask)} \\
P_{km} &\leftarrow km \times G & &\text{(scalar multiplication)} \quad \\text{(CM1 random mask)} \\
P_r &\leftarrow r \times G & &\text{(scalar multiplication)} \quad \\text{(CM1 random mask)} \\
Result &\leftarrow P_{km} + P_r & &\text{(equal to $k \times G$)} \quad \\text{(CM1 random mask)} 
\end{align*}
\]

3. **Multiplicative masking (CM2):** In this implementation, we use a Euclidean division rather than addition computation, which is another canonical way to mask the secret scalar [BR23, Cor99]. Formally, it:

(a) Chooses an independent 128-bit random mask $r$.
(b) Computes $km$, the quotient of the division of the secret multiplier $k$ and $r$. It also computes the remainder.
(c) On chip, computes $P_{km} = km \times G$ and $P_r = \text{rem} \times G$.
(d) On chip, computes $P_{km2} = r \times P_{km}$, then returns $P_r + P_{km2}$ (equal to $k \times G$).

Here is the pseudo code used to implement this scheme:

\[
\begin{align*}
    k &\leftarrow \delta_L(256) & &\text{(secret multiplier)} \quad \\text{(CM2 (a) instead of $P_{km} = km \times G$)} \\
r &\leftarrow \delta_L(128) & &\text{(random mask)} \quad \\text{(CM2 (a) instead of $P_{km} = km \times G$)} \\
k &\leftarrow \delta_L(256) & &\text{(secret multiplier)} \quad \\text{(CM2 (b) instead of $P_{km} = km \times G$)} \\
r &\leftarrow \delta_L(128) & &\text{(random mask)} \quad \\text{(CM2 (b) instead of $P_{km} = km \times G$)} \\
km &\leftarrow (k - r_1) \mod n & &\text{(CM2 (a) instead of $P_{km} = km \times G$)} \\
rem &\leftarrow k \mod r & &\text{(CM2 (b) instead of $P_{km} = km \times G$)} \\
P_{km} &\leftarrow km \times G & &\text{(CM2 (a) instead of $P_{km} = km \times G$)} \\
P_r &\leftarrow \text{rem} \times G & &\text{(CM2 (b) instead of $P_{km} = km \times G$)} \\
P_{km2} &\leftarrow r \times P_{km} & &\text{(CM2 (a) instead of $P_{km} = km \times G$)} \\
Result &\leftarrow P_r + P_{km2} & &\text{(CM2 (b) instead of $P_{km} = km \times G$)} \quad \\text{(CM2 (b) instead of $P_{km} = km \times G$)} 
\end{align*}
\]

4. **Combined countermeasure (CM3):** This combines CM1 and CM2 techniques in an attempt to further increase security with higher-order masking.

\[
\begin{align*}
    k &\leftarrow \delta_L(256) & &\text{(secret multiplier)} \quad \\text{(CM1 random mask)} \\
r_1 &\leftarrow \delta_L(256) & &\text{(CM1 random mask)} \\
r_2 &\leftarrow \delta_L(128) & &\text{(CM2 random mask)} \\
r_3 &\leftarrow \delta_L(128) & &\text{(CM2 random mask)} \\
km &\leftarrow (k - r_1) \mod n & &\text{(CM2 (a) instead of $P_{km} = km \times G$)} \\
\text{rem} &\leftarrow km \mod n & &\text{(CM2 (a) instead of $P_{km} = km \times G$)} \\
P_{km1} &\leftarrow \text{rem} \times G & &\text{(CM2 (a) instead of $P_{km} = km \times G$)} \\
P_{km2} &\leftarrow r_2 \times \text{rem} & &\text{(CM2 (b) instead of $P_{km} = km \times G$)} \\
P_r &\leftarrow r_3 \times \text{rem} & &\text{(CM2 (b) instead of $P_{km} = km \times G$)} \\
P_{km1} &\leftarrow P_{km1} + P_{km2} & &\text{(CM1 (c) instead of $P_{km} = km \times G$)} \\
P_r &\leftarrow P_{km1} + P_{km2} & &\text{(CM1 (c) instead of $P_{km} = km \times G$)} \\
Result &\leftarrow P_r & &\text{(CM2 (b) instead of $P_{km} = km \times G$)} 
\end{align*}
\]
Random masks selection  We discuss the choice of the random mask $r$ (or $r_1, r_2, r_3$ in the case of CM3). For additive masks, we choose $r$ in a way that $\|r\| = \|n\|$, i.e., 256 bits. However for the multiplicative masks, we have to choose $r$ in a way that $\|r\| < \|k\|$, otherwise, the mask would not be effective. We choose $r$ as $\|r\| = \|n\|/2$, i.e., 128 bits, to ensure that it meets this property, but it also achieves the level of resistance to side-channel attacks, as suggested by prior work [GRV17, RLMI21]. We also choose each byte of $k$ and $r$ (or $r_1, r_2, r_3$ for CM3) independently and uniformly at random for each computation to ensure we are performing attacks against implementations that do not have a bias in their random entropy.

6.2 Power trace collection

Our capture setup consists of a Chipwhisperer CW308 UFO board with a CW308T-K82F target board connected to it. The firmware of the target chip was solely responsible for curve addition and scalar multiplication. We did not rely on any software implementation for curve arithmetic. Scalar multiplication and point addition operations were performed by sending an integer and a point, or two points, respectively, to the LTC. For creating labels for model training, we additionally record on the host computer the secret multiplier and random parameters used for masking each trace.

Capture setup  We collect power measurements using the Teledyne LeCroy WavePro 404HD-MS [Tel] oscilloscope connected to the embedded resistor shunt on the ChipWhisper NAE-CW308T-K82F [New] target board. The oscilloscope probe is hooked to the test point TP5 of the CW308 UFO board to measure the current, while digital channel D0 is connected to the GPIO4/TRIGGER pin to get the trigger signal, which starts the capture. We insert a 7.37MHz crystal into the X1 socket and adjust the clock source selection jumper J3 to the CRYSTAL position to provide the clock signal. This configuration ensures that there is no correlation between the target chip clock and the oscilloscope sampling clock, resulting in asynchronous measurements. The oscilloscope channel is set to AC coupling with a bandwidth limited to 200MHz. The first scalar multiplication is always aligned using a trigger signal for each operation, and there is no additional alignment performed. The trigger signal was configured to stay high during each operation performed by the LTC. We use the first rising edge of the trigger signal as the oscilloscope trigger to start capturing.

Experimental leakages  We ensure that no UART communication is leaking by conducting one experiment where we replace captured points in the training set with Gaussian noise when the trigger signal is low. After training a model, we observe no performance loss when compared to training on raw traces, indicating that no discernible UART leakage is occurring in the replaced points. Note that all other experiments are conducted with raw traces.

Datasets Collected  Table 3 provides a technical summary of the datasets generated using the implementation discussed in Section 6.1. We use the SCAAML ([B+19]) dataset library to store our traces and attack point values as TFRecord files. We ensure that no key is reused between the splits by tracking which keys were previously used. Overall each dataset collection process takes anywhere from several weeks (CM0) to months (CM3) to complete.

Note that to ensure that the attack generalizes between chips of the same family despite potential subtle hardware variations, we use a different chip to collect the training/test splits and the holdout split. The holdout splits, following machine learning best practices, are never used to tune the models or during experimentation. Instead, they were reserved for the final evaluation presented in Section 6.3.

Note that Table 3 also reports the ASCADv2 dataset that we use to evaluate GPAM
generalization across multiple algorithms in Section 7. This dataset was made public in [ERR+18]; we simply convert it to the SCAAML dataset format. Since there was no apparent restriction on how to divide the samples into splits (train, test, holdout), we took a portion of consecutive samples for train, a portion for test, and a portion for holdout.

Table 3: List of ECC evaluation datasets used in this study to evaluate GPAM generalization to multiple hardware implementations. The names refer to protected implementations described in Section 6.1. The table also includes the ASCADv2 dataset collected in [ERR+18] that is used in Section 7 to evaluate GPAM generality across multiple algorithms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Trace length</th>
<th>Train length</th>
<th>Test length</th>
<th>Holdout length</th>
<th>File size [TB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECC CM0</td>
<td>1.6M</td>
<td>57,344</td>
<td>8,192</td>
<td>8,192</td>
<td>0.2</td>
</tr>
<tr>
<td>ECC CM1</td>
<td>5M</td>
<td>194,544</td>
<td>8,192</td>
<td>8,192</td>
<td>1.5</td>
</tr>
<tr>
<td>ECC CM2</td>
<td>10M</td>
<td>122,880</td>
<td>8,192</td>
<td>8,192</td>
<td>2.1</td>
</tr>
<tr>
<td>ECC CM3</td>
<td>17.5M</td>
<td>122,880</td>
<td>8,192</td>
<td>8,192</td>
<td>3.7</td>
</tr>
<tr>
<td>ASCADv2</td>
<td>1M</td>
<td>640,000</td>
<td>80,000</td>
<td>80,000</td>
<td>0.9</td>
</tr>
</tbody>
</table>

6.3 Generalization over multiple implementations

Overall, GPAM can successfully attack all four ECC hardware implementations in white-box settings using multi-task relational outputs training as reported in Table 4. These results are computed on the holdout splits which, as discussed previously, are captured on a different chip of the same family and were not used for model tuning or any other experiments discussed later in this section. As discussed in the threat model section (Section 3), white-box setting means that the model had access to the intermediate values (masks and random values) during training.

Note that we only evaluate GPAM on the initial (most-significant byte \( k_0 \)), middle (\( k_{15} \)), and last (least-significant byte \( k_{31} \)) byte of each implementation, as those bytes are representative of GPAM performance against those implementations. We make this choice because these experiments are resource intensive, and the goal of GPAM is to identify leakage.

Table 4: GPAM white-box key byte recovery success rate on the four ECC hardware protected implementations holdout splits.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Attack point</th>
<th>Accuracy [%]</th>
<th>MeanRank</th>
<th>MaxRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM0</td>
<td>( k_0 )</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CM0</td>
<td>( k_{15} )</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CM0</td>
<td>( k_{31} )</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CM1</td>
<td>( k_0 )</td>
<td>78.80</td>
<td>0.75</td>
<td>192</td>
</tr>
<tr>
<td>CM1</td>
<td>( k_{15} )</td>
<td>93.20</td>
<td>0.31</td>
<td>253</td>
</tr>
<tr>
<td>CM1</td>
<td>( k_{31} )</td>
<td>92.98</td>
<td>0.24</td>
<td>218</td>
</tr>
<tr>
<td>CM2</td>
<td>( k_0 )</td>
<td>66.22</td>
<td>1.40</td>
<td>254</td>
</tr>
<tr>
<td>CM2</td>
<td>( k_{15} )</td>
<td>0.30 $\infty$</td>
<td>127.29</td>
<td>255</td>
</tr>
<tr>
<td>CM2</td>
<td>( k_{31} )</td>
<td>11.31</td>
<td>8.76</td>
<td>233</td>
</tr>
<tr>
<td>CM3</td>
<td>( k_0 )</td>
<td>8.60</td>
<td>19.77</td>
<td>255</td>
</tr>
<tr>
<td>CM3</td>
<td>( k_{15} )</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CM3</td>
<td>( k_{31} )</td>
<td>0.37 $\infty$</td>
<td>127.89</td>
<td>255</td>
</tr>
</tbody>
</table>

As expected, as the strength of the protection increases, the model accuracy decreases to the point where for CM3, only the initial byte can be attacked successfully. Note that an accuracy of 0.4% is close to random chance. We hypothesize that increasing GPAM’s performance against stronger countermeasures requires more training data, not increased model capacity. This is empirically supported by our experiment in Section 6.7, which looks at model accuracy as a function of the number of training traces, showing that
Table 5: GPAM black-box key byte recovery success rate on the first three ECC hardware protected implementations using the holdout splits.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Attack point</th>
<th>Accuracy [%]</th>
<th>MeanRank</th>
<th>MaxRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM0</td>
<td>$k_0$</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CM1</td>
<td>$k_0$</td>
<td>0.29 $\infty$</td>
<td>127.5</td>
<td>255</td>
</tr>
<tr>
<td>CM2</td>
<td>$k_0$</td>
<td>22.81</td>
<td>5.55</td>
<td>231</td>
</tr>
</tbody>
</table>

GPAM’s accuracy against CM3 only starts to rise past 100k traces. Furthermore, looking at the MaxRank results, it is clear that the model did not fully generalize for CM1, CM2 and CM3, as it is close to its 255 upper bound.

In Table 5, we report GPAM results under black-box settings on the holdout splits. Unlike when using the white-box settings, GPAM is not always able to successfully recover even the initial byte $k_0$. Interestingly, GPAM fails to recover CM1 $k_0$, but is able to recover CM2 $k_0$, which is surprising given that in the white-box setting, CM1 appears to be easier than CM2. We hypothesize that CM1 is harder in the black-box setting because its random mask uses 256 bits whereas CM2 only uses 128 bits. Having access to intermediate values seems to make the size of the random mask irrelevant in the white box setting but a strong defense in the black box setting.

6.4 Multi-task effectiveness evaluation

In the following set of experiments, we study whether using multi-task learning improves GPAM’s accuracy. In particular, we are interested in understanding which additional tasks beyond the key byte prediction improve model accuracy, if any. To not taint our holdout splits, the results reported in this section are computed on the test splits. Once again we only perform experiments on representative bytes to limit the computation time, namely the initial bytes ($k_0, k_1, k_2$), middle one ($k_{15}$) and final ones ($k_{29}, k_{30}, k_{31}$). For that same reason, we also only target the middle of the road dataset CM2 and reserve the study of CM1 for the ablation study discussed later in the paper in Section 6.5.

Overall, there are two types of additional tasks that can be included in the training: adjacency predictions and intermediate predictions. Adjacency predictions ask the model to predict the key bytes on the left and the right of the targeted bytes with the hope it helps with carry issues and generalization. Intermediate predictions involve the model predicting the value of intermediate computation points, including mask and random nonces values.

The model can operate in two modes when performing multi-task learning using intermediate values: the multi-outputs mode and relational outputs mode. In the multi-outputs mode, the model outputs all the asked values without any interaction between outputs. This is the classical form of multi-task learning used by many models to boost generality and accuracy. In the relational output mode, as illustrated in Figure 1, we create a directed acyclic graph between the heads to model expert knowledge of how the outputs relate to each other according to the protecting algorithm (CM1-CM3). Obviously this type of knowledge is only available in white box testing conditions.

Notation To make the results tables easier to understand, we are using the following visual convention to distinguish between the various relation conditions:

- Circles are byte indexes centered at the column index (if the column byte index is $i$ then the circles represent $[i-1, i, i+1]$).
- A white circle ◯ at position $j$ means not leveraging multi-task learning.
- A gray circle ● at position $j$ means using multi-task learning.
- A black circle ● means using relational outputs learning.

Here are a few examples of such notations for the column $k_2$ of Table 6:
O O O Outputs: $k_2$, relations: [ ]. Byte $k_2$ is targeted and no multi-task learning is used.

O O O Outputs: $k_2, km_2, r_2$, relations [ ]. The model operates in the multi-task learning mode and predicts the values of $k_2, km_2$, and $r_2$.

O O O Outputs: $k_2, km_2, r_2$, relations $[km_2 \rightarrow k_2, r_2 \rightarrow k_2]$. The model uses relational outputs to predict the values of $k_2, km_2$, and $r_2$. The values of $km_2$ and $r_2$ are fed into $k_2$.

O O O Outputs: $k_2, km_1, r_1, km_2, r_2$, relations $[km_1 \rightarrow k_2, r_1 \rightarrow k_2, km_2 \rightarrow k_2, r_2 \rightarrow k_2]$. The model uses relational outputs and adjacency relations to predict the values of $k_2, km_2$, and $r_2$. The values of $km_1$, $r_1$, $km_2$ and $r_2$ are fed into $k_2$.

Table 6: GPAM accuracy in % on CM1 test dataset when trained using various forms of multi-task learning.

<table>
<thead>
<tr>
<th>Relations</th>
<th>$k_0$</th>
<th>$k_1$</th>
<th>$k_2$</th>
<th>$k_{15}$</th>
<th>$k_{29}$</th>
<th>$k_{30}$</th>
<th>$k_{31}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>O O O O</td>
<td>0.39</td>
<td>0.30</td>
<td>0.20</td>
<td>0.20</td>
<td>0.39</td>
<td>0.78</td>
<td>0.50</td>
</tr>
<tr>
<td>O O O</td>
<td>81.50</td>
<td>79.20</td>
<td>85.45</td>
<td>43.36</td>
<td>86.62</td>
<td>86.13</td>
<td>92.68</td>
</tr>
<tr>
<td>O O O</td>
<td>85.35</td>
<td>71.88</td>
<td>88.09</td>
<td>85.64</td>
<td>85.64</td>
<td>87.21</td>
<td>92.58</td>
</tr>
<tr>
<td>O O</td>
<td>87.40</td>
<td>74.80</td>
<td>83.98</td>
<td>63.67</td>
<td>82.03</td>
<td>87.70</td>
<td>93.65</td>
</tr>
<tr>
<td>O O</td>
<td>85.38</td>
<td>80.96</td>
<td>41.11</td>
<td>84.08</td>
<td>88.48</td>
<td>91.78</td>
<td></td>
</tr>
</tbody>
</table>

Results Overall, we observe that multi-task learning is needed for the attack to succeed as reported on CM1 in Table 6. Without any relations, denoted using the O symbol, the model predictions are unable to exceed random chance ($k_0, k_1, k_{15}, k_{29}$) or barely exceed it ($k_{30}$ and $k_{31}$). Using the simplest form of multi-task learning, denoted using the O symbol, the model obtains high accuracy for almost all the bytes except $k_{15}$. Using relational-output, denoted using the • symbol, the model accuracy improves further overall compared to using multi-task learning.

Conversely, the effectiveness of adjacency relations is marginal. As reported in Table 6, adjacency relations introduce model instability with the accuracy slightly increasing on some bytes (e.g., $k_0, k_{30}, k_{31}$) but decreasing significantly on others (e.g., $k_{15}$). Given our goal to have a stable fully automated attack, we decided against using adjacency relations. We leave it as future work to make better use of them.

![Figure 3: ECC validation accuracy for $k_0$ when using relational outputs.](image)

Effect of multi-task learning on model convergence The positive impact of multi-task learning is best visualized by looking at how each of the output accuracies improve as training progresses. Regardless of the hardware implementation, we observe that outputs start to converge one after the other. For example, as visible in Figure 3(a), we observe that the mask prediction ($km_0$) accuracy rises before the random nonce ($r_0$) prediction accuracy improves, and the key value ($k_0$) prediction starts converging only after both the mask and the random nonce have reached a high accuracy. The same effect is observed for CM2 (Figure 3(b)) and CM3 (Figure 3(c)).

Additionally, we observe that in each case the model first learns to predict the mask...
values and then the random nonces. This behavior is consistent with the hypothesis that multi-task learning is critical to create generalized SCAAML attacks as it allows models to learn to "unpack" protections one step at a time. It also supports the hypothesis that black-box attacks are significantly harder, because models greatly benefit from the extra information. Last but not least, this behavior seems to confirm the effectiveness of higher-order masks against advanced side channel attacks such as SCAAML.

6.5 Multi-task ablation study

In this section, we perform an ablation study to better understand which intermediate values are needed for the attacks to succeed on CM1 and CM2. We exclude CM0, as there are no intermediate values. We also exclude CM3, as GPAM’s relatively low accuracy on this dataset makes it hard to confidently separate the results, and CM3 experiments would take roughly 16 months of computation.

<table>
<thead>
<tr>
<th>Target</th>
<th>Dependency</th>
<th>Accuracy [%]</th>
<th>MeanRank</th>
<th>MaxRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_0$</td>
<td>$km_0$</td>
<td>86.26</td>
<td>0.17</td>
<td>9</td>
</tr>
<tr>
<td>$k_0$</td>
<td>$r_0$</td>
<td>0.39</td>
<td>0.14</td>
<td>4</td>
</tr>
<tr>
<td>$k_0$</td>
<td>Nothing</td>
<td>0.29</td>
<td>127.90</td>
<td>255</td>
</tr>
<tr>
<td>$k_{15}$</td>
<td>$km_{15}, r_{15}$</td>
<td>44.00</td>
<td>1.07</td>
<td>58</td>
</tr>
<tr>
<td>$k_{15}$</td>
<td>$km_{15}$</td>
<td>0.48</td>
<td>125.60</td>
<td>255</td>
</tr>
<tr>
<td>$k_{15}$</td>
<td>$r_{15}$</td>
<td>0.29</td>
<td>126.20</td>
<td>255</td>
</tr>
<tr>
<td>$k_{15}$</td>
<td>Nothing</td>
<td>0.48</td>
<td>125.60</td>
<td>255</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$km_{31}, r_{31}$</td>
<td>92.87</td>
<td>0.09</td>
<td>15</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$km_{31}$</td>
<td>0.48</td>
<td>123.46</td>
<td>255</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$r_{31}$</td>
<td>0.29</td>
<td>126.94</td>
<td>255</td>
</tr>
</tbody>
</table>

For CM1, as reported in Table 7, removing the prediction of the mask ($r_*$) at training time results in the model being unable to successfully attack CM1. Removing the prediction of the random nonce ($km_*$) drastically reduces the accuracy of the model for the middle key byte but has no effect on the initial and last byte. We are not sure why this happens.

<table>
<thead>
<tr>
<th>Target</th>
<th>Dependency</th>
<th>Accuracy [%]</th>
<th>MeanRank</th>
<th>MaxRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_0$</td>
<td>Nothing</td>
<td>18.26</td>
<td>8.44</td>
<td>235</td>
</tr>
<tr>
<td>$k_0$</td>
<td>Outputs: $r_0, km_{16}, rem_{16}$</td>
<td>52.44</td>
<td>1.78</td>
<td>235</td>
</tr>
<tr>
<td>$k_0$</td>
<td>$r_0, km_{16}, rem_{16}$</td>
<td>66.50</td>
<td>1.99</td>
<td>252</td>
</tr>
<tr>
<td>$k_0$</td>
<td>$r_0, km_{16}$</td>
<td>66.50</td>
<td>2.14</td>
<td>254</td>
</tr>
<tr>
<td>$k_0$</td>
<td>$km_{16}, rem_{16}$</td>
<td>39.16</td>
<td>4.47</td>
<td>244</td>
</tr>
<tr>
<td>$k_0$</td>
<td>$km_{16}$</td>
<td>32.40</td>
<td>4.21</td>
<td>223</td>
</tr>
<tr>
<td>$k_0$</td>
<td>$r_0$</td>
<td>39.26</td>
<td>4.43</td>
<td>247</td>
</tr>
<tr>
<td>$k_0$</td>
<td>$km_{16}$</td>
<td>36.23</td>
<td>2.63</td>
<td>160</td>
</tr>
<tr>
<td>$k_0$</td>
<td>$rem_{16}$</td>
<td>11.72</td>
<td>15.37</td>
<td>203</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>Nothing</td>
<td>0.39</td>
<td>125.60</td>
<td>255</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>Outputs: $r_{15}, km_{31}, rem_{31}$</td>
<td>9.00</td>
<td>6.74</td>
<td>192</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$r_{15}, km_{31}, rem_{31}$</td>
<td>10.40</td>
<td>10.00</td>
<td>238</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$r_{15}, km_{31}$</td>
<td>10.74</td>
<td>9.90</td>
<td>142</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$r_{15}, rem_{31}$</td>
<td>8.49</td>
<td>10.33</td>
<td>207</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$km_{31}, rem_{31}$</td>
<td>8.59</td>
<td>8.58</td>
<td>195</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$km_{31}$</td>
<td>9.27</td>
<td>8.99</td>
<td>155</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$rem_{31}$</td>
<td>1.36</td>
<td>41.38</td>
<td>215</td>
</tr>
<tr>
<td>$k_{31}$</td>
<td>$rem_{31}$</td>
<td>8.59</td>
<td>9.90</td>
<td>165</td>
</tr>
</tbody>
</table>

We only target $k_0$ and $k_{31}$ in Table 8, as the model is only able to target the most and least significant bytes of CM2.
6.6 Trunk ablation study

In this ablation study, we evaluate whether the relational outputs benefit from the trunk output. To validate this hypothesis, we try to predict $k_{15}$ but with only using the output of the $km_{15}$ and $r_{15}$ heads not the output of the trunk. Additionally we prevent the model from cheating and encoding $k_{15}$ prediction information in the intermediate outputs by applying a stop gradient on the $km_{15}$ and $r_{15}$ output layers. For this ablation study, the head outputting $k_{15}$ looks like the one in Figure 2 when one removes the “Trunk” input.

Table 9: CM1 ablation of trunk output for the $k_{15}$ head.

<table>
<thead>
<tr>
<th>Trunk</th>
<th>Accuracy [%]</th>
<th>MeanRank</th>
<th>MaxRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>85.64</td>
<td>0.52</td>
<td>198</td>
</tr>
<tr>
<td>no</td>
<td>30.6</td>
<td>1.53</td>
<td>126</td>
</tr>
</tbody>
</table>

When the key byte prediction head is not directly connected to the trunk, the model exhibits an accuracy drop of 55 percentage points as reported in Table 9. Those results strongly support the hypothesis that the transformer encoder blocks’ latent representations and compute power are critical for accurate prediction.

6.7 Dataset size impact evaluation

In this section, following the insights of [KMH+20] where the authors showed that transformer models are bound by capacity, compute, or data, we attempt to determine which of these factors is limiting GPAM performance. We already know, thanks to the experiment ran in Section 6.4, that GPAM is not bounded by compute since the performance plateaus after a few hundred epochs - see Figure 3(a) for example. Accordingly, to decide whether the model is capped by the amount of data available or the model capacity/architecture, we trained the same model on an increasing number of traces from CM1, CM2, and CM3. We take 10%–100% of the 122,880 available examples and learn to predict $k_0$.

We found out, as visible in Figure 4(a) and Figure 4(b), that GPAM is most likely bounded by the lack of data since the accuracy keeps rising as the amount of training examples increases. In particular, for CM3, GPAM starts to generalize only when at least 110,000 examples are used, suggesting that increasing the dataset size would most likely lead to significant accuracy gain. However, the CM3 dataset already requires 3.7TB of storage, so we decided against increasing the dataset size further, as GPAM is already able to successfully attack CM3, and increased accuracy does not bring significant additional
benefits.

### 6.8 ECDSA attack

Due to the presence of highly-secure countermeasures, GPAM is not able to recover all bytes of the multiplier at once with sufficient confidence. Thus, a single-trace attack similar to [WCPB21] seems out of reach. This is expected, as custom attack models have an edge over generalized models in this regard. However, a multi-trace attack is feasible, as we shall demonstrate. Specifically, the partial leakage we obtained can be combined with a lattice attack to recover the key from about 8000 traces, as demonstrated by [MSEH20, RLM121].

We apply a standard lattice attack [HGS01, NS02] to leverage the partial leakage from ECC scalar multiplication protected by either CM1, CM2 (even blackbox), or CM3 and recover the private key from ECDSA ([JMV01]). To simulate a realistic attack, we take our holdout split (captured on a different physical chip than the training data), treat it as the nonce multiplication of ECDSA $k \times G$ explained in Section 2.1, and predict the most significant byte. One caveat is that our multipliers in the holdout split are chosen to test the deep learning model so that each byte is chosen independently and uniformly at random between 0 and 255. Since for ECDSA we require the nonce $k$ to be $1 \leq k < n$, where $n$ is the size of the elliptic curve, we exclude all measurements that have the multiplier outside of this range, after which we are left with roughly 7,800 examples (depending on the dataset – CM1, CM2, CM3). We try to closely simulate an attacker with a profiling device (i.e., the chip the attacker uses to collect training and validation splits) and traces from the device under attack. The attacker is free to leverage the model’s outputs as they wish, as long as they are able to recover the secret key in reasonable time. The way of using prediction confidence described below is specific to our attack configuration (model and dataset), and we do not claim that it directly transfers to other attack configurations.

The lattice attack requires all predictions to be correct for it to work, or as shown by [DDME+18], it can tolerate a very small number of noisy signatures. After trial and error with the predictions, our intuition is that if we turn byte predictions into predictions of the four most significant bits (MSBs), we will achieve the highest accuracy. To compute the probability of the 4 MSB of the nonce for a given signature, we sum the probabilities of the 16 possible LSB values with the same 4 MSBs. Table 10 shows the prediction accuracy of the four most significant bits. As we can see, we have a much higher accuracy in this case.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Accuracy [%]</th>
<th>MeanRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM1</td>
<td>94.83</td>
<td>0.06</td>
</tr>
<tr>
<td>CM2</td>
<td>96.39</td>
<td>0.07</td>
</tr>
<tr>
<td>CM2 black-box</td>
<td>85.75</td>
<td>0.20</td>
</tr>
<tr>
<td>CM3</td>
<td>71.86</td>
<td>0.87</td>
</tr>
</tbody>
</table>

A lattice attack using the four MSBs of the nonce requires 80 signatures [JSSS20, MSEH20] predicted by the model. Still, random sampling of 80 signatures will have a high chance of having erroneous signatures, especially when the accuracy is still under 90%. However, we notice that in cases where the correct key 4 MSBs are detected with high accuracy, the prediction value has a higher confidence (highest probability - second highest probability). We use the confidence of predictions as weights when randomly sampling (using the parameter weights of random.choices in Python). This approach retrieves the secret key after several retries for CM1 and CM2.

For the case of CM3, we employ the following heuristics to succeed in the attack. First, we give more weight to samples with higher confidence; we achieve this by using confidence to the power of eight as weights when sampling for the subset used in the lattice attack. The constant eight is chosen by profiling the attack on the validation set (also roughly
8,200 examples). Second, we discard samples with too high confidence (more than 0.25, chosen by trial and error) when predicting the byte value, discarding roughly 10% of examples. This excludes all (and thus also the wrong) predictions that are too confident and would occur in most random samples.

We use the lattice construction of [BVdPSY14] and [NS02] with the [The20] implementation of the BKZ algorithm [SE94]. The attacks take a few minutes. The longest is the black box attack on ECDSA using CM2, which takes roughly 15 minutes on a desktop computer with AMD Ryzen 9 CPU. This is due to several retries needed to get all 80 samples correct.

**Additional evaluation on a public ECC dataset** To better compare to prior work, we evaluate GPAM against the REASSURE ECC dataset introduced by [Chm20]. This datasets consists of roughly 6,000 unprotected traces subdivided into 255 sub-traces that are aligned to expose the corresponding cswap bit of the Montgomery ladder. Each of the sub-traces consists of 5,500 points.

The model described in [NCOS16] predicts the cswap bits with 99.6% accuracy and therefore recovers the whole key from a single trace. GPAM achieves comparable accuracy 99.57% without the need for hypertuning, showcasing once again its ability to generalize across datasets and use-cases.

**Evaluating prior-art models on our datasets** Finally, to evaluate the difference in generalization capability between GPAM and previous work, we train the LSTM model proposed in [LZC21] on ECC CM0, targeting $k_0$. This model achieves 91.4% accuracy but fails at attacking CM1. This demonstrates its limitations in generalizing past simple constant-time defenses to strongly protected implementations. On the other hand it proves our point that a single model can be used for multiple algorithms ([LZC21] were targetting AES).

We also train a CNN model similar to [WCPB21] to evaluate convolution networks generalization potential. We had to halve the number of filters and lower the batch size to 32 to be able to run it on our longer ECC traces. This model achieved 100% accuracy on CM0 $k_0$ but, similarly to [LZC21], failed to show significant leakage on CM1 $k_{m0}$ (mean rank 118). This highlights a similar limited ability to generalize against strong defenses.

We note that those finding are consistent with our extensive internal experiments before landing on GPAM, during which we were not able to get even state-of-the-art convolutional architecture such as ConvNeXt [LMW22] to achieve results as good as GPAM’s.

### 7 Generalizing GPAM to AES

In this section, we show that GPAM generalizes across cryptographic algorithms by demonstrating its effectiveness in attacking an AES software-protected implementation, namely the publicly available ASCADv2 dataset [MS23], in an end-to-end manner without trace processing. Additionally we evaluate GPAM on two other publicly available AES datasets namely ASCADv1 [ERR18] and CHES 2023 challenge to assess its ability to generalize to various AES implementations and compare its performances to previous specialized approaches.

**AES attack evaluation** Leakage from a single trace is often not enough to uncover the secret key in AES. In this case, the attacker may combine information from multiple traces with the same key but variable plaintext. Predictions of key byte values are then combined over a set of attack traces to a maximum likelihood score vector (their logarithms are summed). The index in this vector with the largest value is the predicted value of the key byte. More generally, *Guessing Entropy* (GE) [SMY06] is the average number of entries larger than the one corresponding to the correct value (also mean rank of the correct value
in the score vector). When the GE of all key bytes is less than one we call the attack successful.

When a sensitive value \(s\) (e.g., value of a byte of the S-BOX input) is split into two shares, bytes \(x, y\), we may target \(x\) and \(y\) separately and then from their probabilities we compute \(P[s = b] = \sum_{i=0}^{255} P[x = b \oplus i]P[y = i]\) for any byte value \(b \in \{0x00, \ldots, 0xFF\}\). Analogous formulas for other types of masking (shuffling and affine masking) are derived by [MS23].

### 7.1 ASCADv2 dataset

The ASCADv2 dataset [MS23] comprises 800,000 power traces collected from a Cortex M4 microcontroller manufactured by ST Microelectronics (STM32F303RCT7) while it was performing AES-128 encryptions. The firmware implements affine masking and shuffling to protect the AES encryption computation from side-channel attacks. More details about the dataset can be found in [BKPT20].

**Attack Scenario** We replicate the “First Threat Scenario” described in [MS23]. We target the following equation from the AES S-BOX masked operation [MS23]:

\[
c[i] = r_m \times Sbox[pt[i] \oplus k[i]] \oplus r_{out}
\]

where \(\times\) and \(\oplus\) stand for multiplication and addition in the Rijndael finite field [DR01], \(Sbox\) is the AES S-BOX, \(r_m\) and \(r_{out}\) are affine mask bytes, \(pt[i]\) is the permutation index, \(pt[i]\) is the byte of the plaintext, and \(k[i]\) is the byte of the AES round key.

Critically, unlike previous work [MS23], we perform an attack by using all 1,000,000 points of each trace without preprocessing, instead of using an SNR (signal-to-noise ratio analysis) analysis to use only 15,000 points (1.5%) out of the total trace. Additionally, we do not modify the GPAM architecture to perform the attack and solely rely on hyperparameter tuning to adjust the model hyperparameters to this new target.

**AES attack evaluation** For \(c[i]\), we report the best of 7 runs (due to the influence of initialization weights, see Section 5.3.1). That is, we train a model seven times, and for each attack point, we pick the model with the highest accuracy on the validation set. We then use that to evaluate the performance on the holdout split. Our model targets the intermediate value \(c[i]\), mask bytes \(r_m, r_{out}\), and the permutation \(p[i]\) (for \(i = 0, \ldots, 15\)). Table 11 shows a comparison of ML metrics of GPAM and the results of [MS23]. We then estimate GE to also compare attack results from the acquired intermediate values. Following [MS23] we simulate a fixed key (all zero) by replacing \(pt\) by its XOR with the corresponding random key. Since \(r_m\) has very high accuracy we use it directly instead of the whole probability distribution. We sample 10,000 times to estimate the GE and need roughly 80 traces to have GE of all key bytes under 1, comparable with the 60 traces needed by [MS23] (but use heavy preprocessing).

**Table 11**: ASCADv2 results \((c[i]\) is the best out of 7 runs) compared with results of [MS23], measured on holdout dataset with 80,000 examples. For the sake of brevity we average results when there are multiple indexes.

<table>
<thead>
<tr>
<th>Attack point</th>
<th>Accuracy [%]</th>
<th>MeanRank</th>
<th>MaxRank</th>
<th>Acc [MS23]</th>
<th>MeanRank [MS23]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r_m)</td>
<td>100.00</td>
<td>0</td>
<td>0</td>
<td>99.2</td>
<td>–</td>
</tr>
<tr>
<td>(r_{out})</td>
<td>18.25</td>
<td>4.78</td>
<td>65</td>
<td>21.1</td>
<td>–</td>
</tr>
<tr>
<td>(c[i]) (average)</td>
<td>1.18</td>
<td>80.65</td>
<td>255</td>
<td>1.6</td>
<td>80</td>
</tr>
<tr>
<td>(p[i]) (average)</td>
<td><strong>95.07</strong></td>
<td>0.055</td>
<td>4.44</td>
<td>88.9</td>
<td>–</td>
</tr>
</tbody>
</table>
7.2 ASCADv1 variable key

ASCADv1 the precursor of ASCADv2 is comprised of two electromagnetic emission datasets captured of an ATmega8515 micro-controller running a masked AES implementation. Following [EST+22] we target the dataset with variable key and compare to the SOTA attacks [LZC+21] and [HCM24]. For this attack we hypertune the patch size (possible values [100, 200, 400, 625, 1000, 2000]) and merge filter 1 (possible values [0, 4, 8, 16, 32, 64]) Each tuning run comprise 50 epochs of 500 steps and use a batch size of 256. The learning rate is set to 0.0005 and we use the full trace length. A total of 49 experiments run over two days with the best model having a merge filter 1 equal to 0 and a patch size of 625. This configuration was fully trained for 500 epochs and achieves a 95.94% accuracy on the third byte of S-BOX input which is the standard target as the first two bytes in the dataset are unprotected due to the mask being always zero. With 95.94% GPAM significantly outperforms both [LZC+21] which only achieve 6% accuracy and [HCM24] neither of which achieves a single trace attack. Full training of GPAM required twice as long as [LZC+21] on similar hardware but reached 90% validation accuracy halfway through.

7.3 CHES 2023 SMAesH challenge

We evaluate GPAM on the the CHES 2023 conference challenge consisting of two protected AES datasets A7 (Artix-7 FPGA) and S6 (Spartan-6 FPGA). GPAM uncovers leakage but does not achieve top results.

S6 dataset: we target bytes 1, 6, 11, and 12 as suggested by the winner. GE with 250k traces was under 15 (0.85 in one case). Other bytes did not leak enough for an attack.

A7 dataset: a single model targets msk Plaintext[i] ⊕ msk key[i] and msk Plaintext[i + 16] ⊕ msk key[i + 16] (since the XOR of these is the XOR of the original key and plaintext). These targets reach 124 mean rank. For some i one of those values did not converge and prevented the recovery of the corresponding key byte (re-training might be beneficial). Best GE at 290k traces (current winner) was 23.7 (for i = 2) which is still too high for an attack.

8 Conclusion

In this paper we presented GPAM, the first deep-learning architecture that can perform power side-channel analysis against multiple protected cryptographic algorithms, namely ECC and AES. We demonstrate GPAM’s ability to generalize by successfully attacking several highly protected ECC implementations without changing the model architecture, and verify its effectiveness on multiple devices. To enable reproducibility of these results, we open-source our models and datasets. This research moves us one step closer to a fully-automated side-channel attacks and leakage detection system that could be used as part of a hardware product release process and to test new countermeasures. Our results also suggest that some advanced countermeasures that are currently considered sufficient to thwart side-channels attacks can be defeated. Accordingly, there is a pressing need to devise new countermeasures that are resilient to deep-learning attacks.

References


[HCM24] Suvadeep Hajra, Siddhartha Chowdhury, and Debdeep Mukhopadhyay. Es
tranet: An efficient shift-invariant transformer network for side-channel
analysis. *IACR Transactions on Cryptographic Hardware and Embedded

[HDLL22] Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. Transformer quality in


[HGS01] Nick A Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital

[HHGG20] Benjamin Hettwer, Tobias Horn, Stefan Gehrer, and Tim Güneysu. Encoding
power traces as images for efficient side-channel analysis. In *2020 IEEE
International Symposium on Hardware Oriented Security and Trust (HOST)*,

[HIM+14] Johann Heyszl, Andreas Ibing, Stefan Mangard, Fabrizio De Santis, and
Georg Sigl. Clustering algorithms for non-profiled single-execution attacks

[JB17] Kimmo Järvinen and Josep Balasch. Single-trace side-channel attacks on

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital
signature algorithm (ECDSA). *International journal of information security*,

[JSSS20] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sys. Minerva:
The curse of ECDSA nonces: Systematic analysis of lattice attacks on noisy
leakage of bit-length of ECDSA nonces. *IACR Transactions on Cryptographic
Hardware and Embedded Systems*, 2020.


[KMH+20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin
Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA,

[LBM15] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. A machine
learning approach against a masked AES: Reaching the limit of side-channel

[LH16] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with

[LMW+22] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhoefer, Trevor

[LZC+21] Xiangjun Lu, Chi Zhang, Pei Cao, Dawu Gu, and Haining Lu. Pay attention
to raw traces: A deep learning architecture for end-to-end profiling attacks.
*IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021.


