Activation Functions Considered Harmful: Recovering Neural Network Weights through Controlled Channels

Jesse Spielman
School of Computer Science
University of Birmingham
Birmingham, UK
jxs1366@bham.ac.uk

David Oswald
School of Computer Science
University of Birmingham
Birmingham, UK
d.f.oswald@bham.ac.uk
School of Computer Science
Durham University
Durham, UK
david.f.oswald@durham.ac.uk

Mark Ryan
School of Computer Science
University of Birmingham
Birmingham, UK
m.d.ryan@bham.ac.uk

Jo Van Bulck
DistriNet
KU Leuven, Belgium
Leuven, Belgium
jo.vanbulck@cs.kuleuven.be

***Abstract*—Recent advancements in hardware-based enclaved execution environments, such as Intel SGX, aim to protect critical model parameters in high-stakes machine learning applications increasingly moving to end-user or cloud environments. However, this also introduces the risk of privileged side-channel attacks, traditionally aimed mainly at cryptographic targets.**

In this paper, we develop a novel attack methodology that exploits input-dependent memory access patterns in common neural network activation functions to extract hidden model parameters. In several case studies using the SGX-Step attack framework and the Tensorflow Microlite library, we demonstrate complete recovery of first-layer weights and biases, along with partial recovery of deeper layer parameters under specific conditions. Our novel attack technique requires only 20 queries per input per weight to obtain all first-layer weights and biases, with an average absolute error of less than 1%, improving over prior model stealing attacks. Furthermore, a broader ecosystem analysis reveals the widespread use of activation functions with input-dependent memory access patterns in popular machine learning frameworks and maths libraries. Our findings highlight the limitations of deploying confidential models in SGX enclaves and emphasise the need for stricter side-channel validation of machine learning implementations, akin to the vetting efforts applied to secure cryptographic libraries.

***Index Terms*—Machine Learning Security, Side Channel Attacks, Trusted Execution, Confidential Computing, SGX**

I. Introduction

Training and deploying Machine Learning (ML) models, in particular Neural Networks (NNs), often requires a massive investment in computational resources. Hence, the threat of model stealing attacks is significant, because they allow an attacker to abscond with an ML application, causing substantial financial damage as well as security and privacy risks. Attacks could be used for industrial espionage (in the case of a network that solves a difficult problem, e.g., a sophisticated automated trading system), or to ease the creation of adversarial examples to defeat a classification system, e.g., for spam filtering or intrusion detection. An application that incurs a fee to access per interaction could also be duplicated to remove this restriction. While existing model stealing attacks show how to duplicate NNs essentially via brute force, they scale poorly for increasingly complex networks and require detailed access to output probabilities or physical access for measurements such as power consumption.

Because ML (both training and inference) requires substantial computation, there is a strong focus on performance. Further, because of the hardware requirements of ML training and inference (such as in the case of an AI model which backs an end user-accessible web interface), there is an increasing need to host ML workloads in the cloud. However, this move onto (often shared) hardware introduces a number of security concerns. To mitigate the risk, substantial work has been done using hardware-based Trusted Execution Environments (TEEs) such as Intel Software Guard Extensions (SGX) or AMD Secure Encrypted Virtualization (SEV) to protect confidential ML workloads [1]–[5]. These TEEs are presumed to facilitate the secure and efficient outsourcing of critical ML computations to third-party hardware, while simultaneously preserving the confidentiality of model parameters that embody key Intellectual Property (IP).

A core component of NNs are their activation functions, chosen by developers (alongside other architectural decisions such as the number of layers or the learning rate) to introduce non-linearity. These functions are generally computed once for each neuron based on the sum of the multiplication of all incoming inputs by their weights (plus a bias term). Often, these functions are provided by a ML framework such as Tensorflow or PyTorch, which include routines for training, running inference, and deploying models. It is important to note that these framework-provided activation functions usually rely on basic mathematical functions such as `std::max()` or `exp()`, which are often provided by the system's standard libraries (e.g., `glibc`) and optimised for speed, not side-channel resistance. TEEs like Intel SGX often supply their own (restricted) standard library (e.g., SGX's `tlibc`), which provides implementations known to work within the special enclaved environment. Thus, ML frameworks inherit some security properties from underlying standard library maths functions.

Research into Side-Channel Analysis (SCA) (both with physical access as well as remotely exploitable side channels like timing) has led to the "hardening" of certain security sensitive algorithms, especially in cryptographic libraries. However, less attention has been paid to the side-channel security of ML libraries, even though their parameters (weights and biases) are today often akin to cryptographic secrets in sensitivity. Especially in the context of TEE-protected ML workloads, the strengthened side-channel adversary model has not been sufficiently studied.

In this paper, we show how pervasive data-dependent memory access patterns in activation functions across many ML frameworks can lead to the deterministic recovery of hidden model parameters when deployed in a TEE. Particularly, we develop a novel methodology to recover partial weights and biases from the first (and deeper) layer(s) of a victim network when provided with an instruction-granular page-access trace.

As a practical case-study, we deploy three Tensorflow Microlite models inside SGX enclaves and extract deterministic, instruction-granular page access traces using the SGX-Step [6] framework. Our first case study demonstrates how we can use those traces to successful recovery all first layer weights and biases to more than 5 decimal places of accuracy with 55 invocations of the network, or with less than 1% average error with around 20 invocations after an initial calibration phase. Our end-to-end attack outperforms Tramèr, Zhang, Juels, *et al.* model stealing attacks [7], which require around 100 invocations for each parameter in simpler classification networks. We then use our second and third case studies to explore deeper layers and much larger networks, respectfully. Considering the wider ecosystem, we survey widely used ML libraries and find widespread use of secret-dependent access patterns in activation functions. Furthermore, we discuss to what extent our attack vector applies to other TEEs.

Contributions. Summarised, our contributions are:

- A novel methodology to recover weights and biases from partial memory-access side-channel traces.
- Three end-to-end case studies demonstrating concretely how, by exploiting our memory-trace extraction technique, we can perform accurate weight/bias recovery from Tensorflow Microlite networks running inside SGX enclaves in a few different configurations.
- A comprehensive survey of input-dependent accesses in common activation functions across popular ML and standard libraries.
- An open-source Tensorflow Microlite SGX benchmark for future work on attacks and mitigations.

Ethics and Open Science. All experiments were conducted on Proof-of-Concept (PoC) implementations on our own local machines. To ensure the reproducibility of our results, and to enable future research on side-channel attacks and defenses, we release all code and data as open-source at https://github.com/heavyimage/afch_paper.

Scope. To the best of our knowledge, this is the first study to thoroughly analyse side-channel implications of deploying ML workloads in TEEs, which are increasingly being suggested by industry and academia to preserve IP and end-user privacy. In this paper, we do not claim a fully-fledged, optimised and weaponised real-world attack; instead we focus on developing a *novel attack methodology* to steal partial model parameters via case-study-driven evaluation. While full recovery of real-world networks may be out of reach, we show that motivated adversaries can fully automatically extract thousands of hidden, supposedly secure parameters. Following the rich tradition in cryptography, where a long line of increasingly potent side-channel attacks have highlighted the need for constant-time programming practices, we hope our work will draw attention to the security and privacy trade-offs of using TEEs for private ML and underline the importance of side-channel-resistant coding practices.

II. Background and Related Work

Trusted Execution Environments. One of most widely studied TEEs is Intel's SGX, which, even though discontinued for client CPUs, is widely supported on Xeon Scalable server CPUs and now marketed for use cases such as secure ML deployments [2]. SGX-secured code is executed in a hardware-protected *enclave* (with its memory encrypted) such that even the `root` user cannot gain access. SGX also allows secrets to be securely loaded into the enclave remotely, and subsequently be (un)sealed for local storage. This allows data of the enclave (such as a pre-trained ML model) to be also protected while at rest. SGX does not provide guarantees against timing or memory-based side channels [8], rendering enclaves vulnerable to cache and page fault attacks [9], [10]. Moreover, due to SGX's privileged adversary model, such attacks can achieve high temporal resolution using tools like SGX-Step [6], which exploit timer interrupts to single-step enclave execution, effectively interleaving enclave execution with attacker-controlled code at an instruction-level granularity.

Other TEEs of note include ARM's TrustZone which brings the concept of TEE to mobile devices [11]. Intel's Trust Domain Extensions (TDX) [12] and AMD's SEV [13] are TEEs that provide hardware-backed isolation guarantees at the level of entire Virtual Machines (VMs).

Model Stealing Attacks. The recovery of parameters from a trained neural network via "model stealing" is well documented in the literature. Such practical recovery attacks, first demonstrated by Tramèr, Zhang, Juels, *et al.*, enabled the duplication (stealing) of neural networks and other models [7]. In these attacks, the original network is used as an oracle to train a duplicate network with similar hyperparameters. The attack works by choosing inputs via a search of the parameter space of the network (using one of several different heuristics) and joining that input with the corresponding output probabilities which are returned. These data pairs are used as training examples for the new network. “Solving” a network in this way results in the calculation of hidden parameters that are derived from but not necessarily precise to the original source network. Furthermore, they require a large number of queries, around 100 per parameter, as well as access to the output probabilities to essentially brute-force the space of the network. While these attacks may reconstruct the overall structure and general features of the network, it remains an open question how effective they are at reproducing the specifics of the original network, e.g., when attempting to transfer adversarial examples from duplicate to original. An alternative approach by Canales-Martínez, Chávez-Saab, Hambitzer, *et al.* (and references therein) treat the recovery of parameters as cryptanalysis, showing that the *sign* information for each neuron in a 1.2M parameter CIFAR10 network can be recovered in around 30 min using a 40 GB A100 Graphics Processing Unit (GPU) [14].

Side-Channel Attacks on ML Implementations. An alternative approach to recover model parameters is to use a software-based (timing or cache) side channel. Most research in this direction has focused on the recovery of hyperparameters, such as the network’s architecture, but *not* the precise weights: Yan, Fletcher, and Torrellas demonstrate a technique for recovering the hyperparameters of a victim network such as the number of layers and their activation functions using a Flush+Reload cache attack [15]. Other works in this direction have used similar cache attacks [16], execution time [17], looked at embedded devices [18], and considered equivalent approaches on GPUs [19]–[21]. Other uses for software-based SCA include the recovery of network inputs from floating point timing [22], membership inference [23], and the generation of adversarial examples [24], [25].

Because the targeted leakages are much smaller and hence harder to exploit, less attention has been paid to full model recovery, *i.e.*, weights and biases, through software-based side channels: Alder, Van Bulck, Oswald, *et al.* show how an attacker-controlled configuration of the x86 floating point unit can be used to recover weights from a toy NN (using a custom implementation) running in an SGX enclave [26].

Gongye, Fei, and Wahl show that, if an adversary obtains the precise timing of each network layer, in certain cases they can use timing leakages from floating point to recover the full model [27]. In particular, the authors rely on different execution timing on certain x86 CPUs when processing “subnormal” numbers. Gongye, Fei, and Wahl do not specify whether their attack applies to real-world ML libraries, and assume that the necessary precise timing measurements “can be achieved by either analysing the cache access pattern [...] or through visual inspection of power traces”. In contrast to their work, in this paper, we consider the input-dependent memory access patterns of activation functions in widely used ML libraries, showing an end-to-end attack that uses SGX-Step for reliable side-channel observations.

In contrast, side-channel attacks that recover the full model with physical access to the hardware and measurements of electro-magnetic (EM) emanation or power consumption have been further developed [28]–[31]. However, as these attacks require measurements on or in close proximity to the targeted chips, they are of less relevance to datacenter applications, where physical access to the underlying servers is typically closely guarded. Hence, in this paper, we focus on software-based side-channel attacks to recover weights/biases without physical access and without access to the output probabilities.

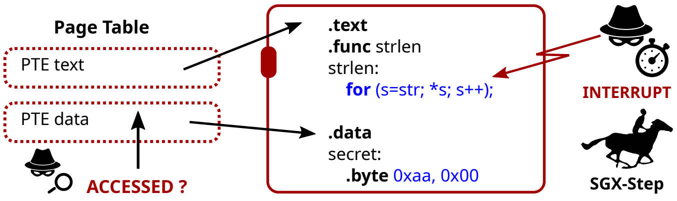

Fig. 1. Victim enclaves exhibiting tight memory-access patterns can be precisely interrupted at instruction-level granularity using SGX-Step allowing to extract deterministic page-access count traces.

Controlled Channel Attacks. For SGX, Intel explicitly places the burden of preventing secret-dependent memory accesses for security-critical code on the developer [32]. In practice, this means that enclave developers should avoid data-dependent patterns of the form `array[x]` or `if(x) func()`, where `x` depends on a secret. It is important to note that in practice, such secret-dependent access patterns may only become evident at the machine code level, as optimising compilers might e.g., opt to compile “branchless” code into code with conditional branches, and vice versa.

Apart from well-known microarchitectural cache leakage [33], which is notoriously noisy, a number of SGX-specific side channels were discovered that exploit the privileged attacker’s control over the untrusted Operating System (OS). One of the first ones being “controlled channels” as introduced by Xu, Cui, and Peinado [10]. This method exploits the fact that page faults within an SGX enclave are handled by the untrusted OS. A privileged adversary can, hence, temporarily unmap a page, observe whether a page fault occurs, and from this fully *deterministically* infer enclave memory access patterns at a 4 KiB, page-level spatial granularity. While such page-fault attacks have been proven particularly powerful, e.g., to extract text and images [10] or full cryptographic keys [34], their relatively coarse-grained spatial resolution may limit exploitability. Consider the example code snippet provided in Figure 1, where a tight `strlen` loop is executed that fits entirely within a single code and data page. As subsequent accesses to the same page are cached in the processor’s Translation Lookaside Buffer (TLB), and the enclave needs both the code and data page to make forward progress, page-fault adversaries only observe the first access to a page and are not able to distinguish successive `strlen` loop iterations. To overcome this limitation, Van Bulck, Weichbrodt, Kapitza, *et al.* showed that privileged adversaries may also monitor page-table attributes [35] using the ‘accessed’ and ‘dirty’ bits to count accesses to a certain page.

The open-source SGX-Step framework [6], [36] allows for precise single-stepping of production enclaves using privileged x86 APIC timer interrupts, such that page access patterns can be *deterministically* monitored for every enclave instruction, giving precise insights into the operation of a victim enclave. By conservatively under-estimating the APIC timer interval, SGX-Step results in either zero or single-steps, but *always* avoids multi-steps [36, Table 5]. Subsequent works [37]–[41] showed that the accessed (A) bit in the enclave code page *deterministically* distinguishes single-steps (enclave instruction retired; A=1) from zero-steps (A=0). This ability to perfectly and deterministically single-step production enclaves using SGX-Step prompted Intel to develop the opt-in AEX-Notify hardware-software mitigation (cf. Section VII). To date, instruction-granular page-access traces extracted with SGX-Step have been repeatedly abused to reliably exploit enclave interface vulnerabilities [40] or to reconstruct cryptographic key material [38], [39]. In the following, we show that such traces are sufficient to recover weights from real-world NN libraries running inside a production SGX enclave.

III. System and Adversary Model

Attacker Capabilities. We adhere to Intel's standard SGX threat model, where the adversary has full `root` access on the victim machine, as is e.g., the case when deploying an ML model on external cloud infrastructure. The attacker is a software-only adversary without local physical access, for instance they may gain root access remotely over the network. Such privileged software attackers can send arbitrary inputs through the enclave software interface and can further interrupt the enclave by manipulating page tables and interrupts (cf. Section II). For the latter, we leverage the widely-used SGX-Step [6] framework to precisely single-step a target enclave and obtain instruction-granular page access traces. Notably, while attackers may debug enclaves using a copy of the victim code to facilitate attack development, we conduct all final attacks with SGX-Step on target enclaves running in SGX production mode, i.e., without access to debug features.

While our work focuses on the Intel SGX architecture, the underlying insights and weight-recovery techniques are applicable to other TEEs. Notably, the required side-channel primitives offered by SGX-Step, *i.e.*, precise single-stepping and page accesses, have been demonstrated on alternative VM-based TEEs like AMD SEV [42] and Intel TDX [43], [44].

Target Model. We assume a pre-trained ML network. We do not consider the training process, which is orthogonal to our attack (but we believe it to be an interesting avenue for future research). We focus on Feedforward Neural Networks (FNNs) because they effectively illustrate the problem we exploit and are easily deployable in SGX enclaves (e.g., without dynamic linking or requiring `libc`) using Tensorflow Microlite. Lastly, we assume the network architecture (at least the number of neurons per layer and activation functions) is known, e.g., it is an open-source or well-known architecture or a transfer-learning model, or can be recovered via a cache attack [15].

Target Enclave. We assume that the network has been securely loaded into a production-mode SGX enclave to perform "secure" inference. The network's weights and biases are confidential (sealed), while the enclave source code is known. The latter is a common assumption in SGX attacks [6], [10], [33]–[35] and the default case in the Intel SGX architecture and SDK. Non-standard confidential-code deployments only add "security through obscurity" by requiring an additional, out-of-scope reversing phase [15], [45]. Since the attacker never has access to a self-contained binary that contains the hidden parameters, an attack such as the one proposed by Liu, Yuan, Wang, *et al.* is not possible [46]. Inputs are passed into the enclave and the output is returned outside the enclave through the standard `ecall` interface. Based on the security guarantees of SGX, an ML developer would assume that their network is protected against duplication, as inference is handled entirely within an enclave and weights are never stored outside.

Activation Functions. Finally, our attack requires that the victim network use a vulnerable activation function which contains at least two *input-dependent branches with unique step counts* such as the time they take to return. To give an example, Listing 1 shows some different code paths of the implementation of `expf()` in SGX's `tlibc` [47]. Observing the memory access/execution patterns, e.g., with SGX-Step, allows an attacker to learn which return case an input results in, and thus obtain information on the input to `exp()`.

For this reason, we mostly consider activation functions that include a call or calls to a functions in the `exp()` family, e.g., `sigmoid()`, `tanh()`, and `softmax()`. Notably, such activation functions are widely used in transformer architectures (even though we do not consider attacks on transformers in this paper), as with `sigmoid()` in Deepseek V3 [48], `GELU()` in BERT [49] and `SwiGLU()` [50] in Llama [51]. `softmax()` was specified in the initial transformer paper [52] and is still used in many of these implementations. Note that while several variants of `exp()` implementations exist, they commonly feature the same structure, where multiple error cases are handled with early return statements.

```c
if(hx >= 0x42b17218) {       /* if |x|>=88.721... */
  if(hx>0x7f800000) return x+x; /* NaN */
  if(hx==0x7f800000) return (xsb==0)? x:0.0;
  if(x > o_threshold) return ...; // overflow
  if(x < u_threshold) return ...; // underflow
} if(hx > 0x3eb17218) {
  if(hx < 0x3F851592) { /* ... */ } else { /* ... */ }
} else if(hx < 0x31800000)  { /* when |x|<2**-28 */
  if(huge+x>one) return ...; // inexact
}
```

Listing 1. Excerpt from `expf()` in `tlibc`.

Attacking an activation function containing a call to `exp()` is similar to attacking `exp()` directly. One must be careful to note the sign of the input (as in the case of the $-x$ in `sigmoid()`'s exp$(-x)$) and search in the correct direction. In the case of functions like `tanh()`, which can include multiple discrete calls to `exp()`, the code to process the memory access traces cannot assume a one-to-one relationship between a call to `exp()` and a single neuron's activation. Another important consideration is the expressive power of the function, which we discuss in more detail in Section IV-E.

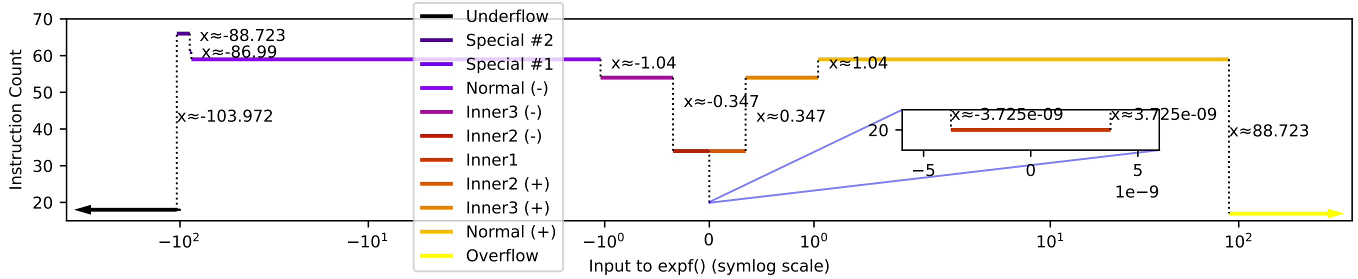

Fig. 2. Visualization of `expf()` CPU instruction count regions by input. The Underflow and Overflow regions have different instruction counts: 18 and 17 respectively. Note also the symmetrical regions of Inner1, inner2, and Inner3.

We note NNs may access the maths functions that underpin their activations from a few different sources. Whether defined by an ML framework itself or imported externally from low level standard libraries such as `glibc`'s `libm` or even dedicated high performance maths libraries like `Eigen` (which may be vectorised to exploit hardware acceleration features), the code may be vulnerable to our attack. We explore this with our second case study in Section V-B and will more generally consider how the source of these functions can affect their vulnerability to our attack in Section VI.

Finally, despite our focus on the `exp()` family of functions, any function exhibiting measurable input-dependent variation in instruction counts (or memory access patterns) is likely vulnerable. As long as the attacker can force a vulnerable function to process inputs that fall into multiple distinguishable classes, it is possible to search for the boundary points between these cases, as we discuss in detail in Section IV. In particular, `relu()` does not rely on a call to `exp()`, but on conditional logic or a call to `std::max()`. We summarise common activation functions in Table I and specifically discuss the vulnerability of `relu()` in greater detail in Section VI.

IV. Methodology

First, we describe the overall concept for our attack (Section IV-A) against a NN and then consider (Section IV-B) different ways ML frameworks may implement activation functions. Next (Section IV-C), we show how to perform our attack practically using SGX-Step to generate per-neuron traces. We then explain in more detail how to attack the first layer (Section IV-D) and the subsequent layers (Section IV-E). Finally (Section IV-F) we estimate the performance of our attack.

A. Attack Concept

The starting point for our attack is the above observation (cf. Section III) that, unlike in cryptographic libraries, the maths functions that underpin many ML activation functions have not been hardened to ensure they execute in constant time and without input-dependent branches and memory access patterns. A classic example in the cache timing literature is a Look-Up Tables (LUTs) that may be accessed in deterministic patterns based on the function inputs. We believe this work has not been undertaken in the context of ML due to performance concerns or because security is seen as less of a focus.

By examining the underlying functions statically, dynamically, or by reading the code (if available), we can deduce the possible branches through the function and the timing as well as memory access patterns therein. Also of interest are cases where certain input values are handled differently (via multiple `return` statements). A special case of those are *early returns*, where certain inputs cause the function to return before the full output is computed.

TABLE I
Survey of Potentially Vulnerable Activation Functions.

Activation Function	Output Range	Contains **`exp()`**	Contains **`max()`**
`sigmoid()`	$(0, 1)$	✓	✗
`tanh()`	$(-1, 1)$	✓	✗
`softplus()`	$(0, \infty)$	✓	✗
`ELU()`	$(-\alpha, \infty)$	✓	✗
`SELU()`	$(-\lambda * \alpha, \infty)$	✓	✗
`relu()`	$(0, \infty)$	✗	✓

Given that such code patterns take different numbers of instructions (and exhibit different memory access patterns) depending on the input, more generally there is a correlation between different code paths and the execution trace. Though these differences might be subtle (e.g., a single instruction), they are measurable given a suitable side channel. Thus, if an attacker can determine which branch a function has taken, they learn something about the input or argument to that function.

As stated, one function with these properties is `expf()`. Figure 2 shows the 11 different return cases of this function based on input and their associated CPU instruction counts (collected using `gdb`). We focus here on the `expf()` implementation from SGX's `tlibc`, but we note that we found other optimised implementations all using a similar algorithm and thus exhibiting similar input-dependent memory access patterns. For ease of discussion we have labeled these 11 cases using the names shown in Figure 2. For example, `expf(4)` results in case Normal $(+)$, whereas `expf(110)` results in Overflow. We define a *threshold* (shown in Figure 2 as dotted black lines) as an input value that is on the precise border of two timing classes. For example, the value 88.723 is between Normal $(+)$ and Overflow.

We can also use knowledge of these regions and thresholds in the other direction: for example, if we know that an execution of `expf()` takes 17 instructions (in the Overflow region), we know the input to that function call must be greater than 88.723. We note that the cases for Overflow and Underflow are both early returns and as such their instruction counts are much lower than the other 'normal' returns which actually perform some sort of unique computation.

We can use these threshold values to leak information about the precise value of a partially controlled input. If we consider the scenario where an input to `expf()` is being multiplied by a hidden constant c and we have access to the instruction count of the execution, we can reveal c by finding (via e.g., a binary search) a threshold value between two instruction count classes and then dividing that value by our input. For example, if we find that the input 3.2164 multiplied by c leads to threshold value 88.723, c must be ${}^{88.723}/{}_{3.2164} \approx 27.5846$. We therefore define 3.2164 as a *convergence point*: an input to the system that causes a threshold value and thus a leak of the precise input to the function.

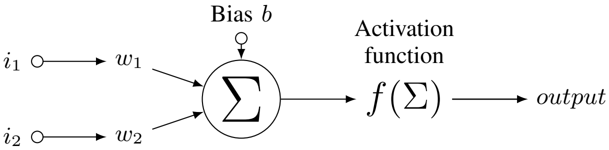

Fig. 3. Visualization of a single 2-input neuron. We use Σ in this paper to represent the sum of all $i_i * w_i$ and b passed into an activation function.

We now extend the above into an attack against a toy ML network: consider a simple NN with only a single neuron, as shown in Figure 3. The activation function is simply `expf()`. The goal is to recover w_1, w_2, and b. We are free to choose values for i_1 and i_2. We first note that the input to a neuron's activation function is the sum of the products of various user-controlled inputs and their hidden weights, plus a hidden bias term (Σ in Figure 3); this is similar to the case described previously except we have two hidden constants multiplied by two different inputs and an additional term, the bias, that is never multiplied with our inputs.

If there was no bias term, we could simply use a variant of the approach described above and iterate through each input (setting all other inputs to zero, allowing us to focus on one weight at a time) and recovering each parameter independently. The addition of the bias term adds another unknown which means we can only find through constraints on the value of the hidden weights and bias:

$$i_1 * w_1 + 0 * w_2 + b = 88.723 \Rightarrow w_1 = \frac{88.723 - b}{i_1}$$

To solve for all three unknowns, we find enough sets of values (one for each input to the neuron) to generate a system of equations to solve for as many unknowns as we have (number of weights plus bias term), for example:

$$\begin{aligned} i_1 * w_1 + \cancel{0 * w_2} + b &= threshold_1 \\ \cancel{0 * w_1} + i_2 * w_2 + b &= threshold_2 \\ i_3 * w_1 + i_4 * w_2 + b &= threshold_3 \end{aligned}$$

Here, we define $(i_1, 0)$, $(0, i_2)$, and (i_3, i_4) as *convergence point sets*, because each set of chosen inputs forces the activation function to a threshold value. Once we have enough equations (one for each unknown), we solve using linear algebra for the unknown weights and biases. Note that the thresholds do not have to be different, though the more thresholds are hit, the more robust the solution.

Though it is possible to find convergence point sets like $(i_1, 0)$ and $(0, i_2)$ above where all but one input is locked to zero and the remaining input is used to search for a threshold point, this is not required. In fact, by locking all but one input to some small magnitude random numbers (e.g., i_3 in the third equation) and searching on the single unlocked parameter (i_4), an arbitrary number of convergence point sets can be found. Finally, if we replace `exp()` above with a different function (such as `sigmoid()`) that also leaks threshold values, the only change is how to conduct the binary search; the recovery procedure is otherwise identical. Building on this concept, we will show how it is possible to recover the hidden parameters of more complex networks to a high degree of accuracy.

B. ML Framework Functions

Thus far we have largely assumed a low level mathematical definition for these functions. However, as discussed in Section III, a Tensorflow neuron with a `tanh()` activation might not directly return the output of a call to the underlying maths library's implementation of `tanh()`; there might be wrapping or processing of the result at the framework level. This is of interest as we can also exploit input-dependent behaviour at this higher framework or application level. An example of this is Tensorflow lite's `sigmoid()` function (called `Logistic()` in the source), shown in Appendix A. We can see that on top of any side-channel leaks due to calls to `std:exp()`, a higher-level set of memory access patterns and instruction count cases are introduced by a conditional routing to different functions (or simply a return of 0) based on the input. In this way, this `sigmoid()` function can actually be attacked at two levels which we will describe in Section V-B.

C. Capturing Traces with SGX-Step

To practically execute our attack, we used SGX-Step to collect the CPU instruction counts necessary to understand the state of our victim networks and apply the steps described above. This required that our networks run inside an SGX enclave. Given the constraints of SGX's standard library replacement `tlibc`, we decided to employ Tensorflow Microlite, a software package for running (converted) Tensorflow lite models on microcontrollers, instead of "full" Tensorflow (which relies on a variety of system calls not easily provided by SGX). Our SGX enclave thus included a statically compiled Tensorflow Microlite library to perform inference. We converted our tensorflow models to Tensorflow lite and then further exported them as to a Tensorflow Microlite byte array which can be compiled directly into a C program. These byte arrays are added to the enclave code which allows us to perform "secure" inference inside SGX.

We note that for practical deployments of SGX-protected ML, using such a reduced variant of Tensorflow would ease deployment, reducing code and thus Trusted Computing Base (TCB) size, while maintaining compatibility with Tensorflow-trained models. The underlying implementation [47] of `exp()` used by our network during inference comes from SGX's `libc` alternative, `tlibc`.

We first profiled the execution of Tensorflow Microlite on an attacker-controlled debug enclave, so that we could understand execution flow by directly observing call traces via the interactive `sgx-gdb` debugger. An example can be seen in Appendix B. This provides an intuition for how Tensorflow Microlite performs inference and which functions and pages are accessed.

We then developed an attack application, which starts and interacts with the victim enclave in production mode, i.e., without access to debug features. We use this to record page accesses which we later interpret into activation function states. Using SGX-Step, we log all page accesses within the production enclave (after a certain trigger page was accessed) and interpret the results as part of a post-processing step. Unlike `sgx-gdb`, SGX-Step only offers a relatively coarse-grained spatial resolution of 4 KiB memory pages, which prevents production enclave attackers from directly identifying the specific symbol (e.g., function or data) accessed by the enclave code. However, SGX-Step's precise instruction-level temporal resolution enables the annotation of coarse-grained page-access traces with the exact number of instructions executed on each page. As illustrated in Figure 4, this capability allows to accurately reconstruct the inference execution structure, even without knowledge of the lower 12-bit page offset of the secret enclave instruction pointer (ERIP). By comparing the dotted red line which depicts the true ERIP values available to `sgx-gdb` and the black line which shows the reduced page-level granularity, one can see how little is lost. It is worth nothing that each page can contain multiple functions and so it is difficult to assign a precise "meaning" to each page without context. The instructions executed on one page could pertain to two very different activities at different points in execution. However, the combination of coarse-grained page-access patterns along with the exact amount of instructions executed per page, annotated in black text, clearly suffices to identify individual function calls or data accesses. In practice, this means that even if multiple symbols are co-located on the same 4 KiB page, we can still interpret the logs. We also know precisely the length (or length range) of different runs at different points during the execution, *i.e.*, we can also tell if a trace is invalid and immediately reject it.

By studying inference traces from the perspective of both `sgx-gdb` and SGX-Step (raw traces or visualizations like those shown in Figure 4), clear patterns emerge. For example, the function `ExpEval()` is located on a different 4 KiB page than `expf()` (each `0x1000` bytes is a new page) and we can plainly interpret the movement between them as the higher level neuron code invoking the lower level `expf()` maths function 16 times (once for each neuron in the first layer of the model this plot was constructed from). The 18 instruction count long runs at the extreme left and right on page `0x2c000` seem to correspond to the start and end of a layer, which makes parsing each layer trivial. We note that the step counts provided by `sgx-gdb` and SGX-Step may vary slightly (but deterministically) due to the documented effects of macro-op fusion [38].

The final part of our tooling is a Python script and library which further abstracts and manages invocations of the enclave program. Because each neuron is accessed in order and each return occurs in sequence, it should be possible to monitor each neuron's step count independently. In order to orchestrate the full attack, we use this script to interpret the log output by SGX-Step into a per-neuron state. Even though we receive the state for every neuron in each log, we only focus on one at a time. It takes about ten seconds to run each input through our SGX-Step setup and parse the trace. An example output of the running attack is shown in Listing 4 in Appendix C.

D. Neuron-Centric Attack on First Layer

Here we discuss how to convert the exploit discussed in Section IV-A into a full attack on the first layer of a production NN. As before, the goal of this phase is to find a number of convergence point sets equal to the number of inputs plus one in order to have a system of equations we can solve to recover the weights and bias.

The first layer may be a special case if (without loss of generality) there is no filtering or normalization which preprocesses the input values. If this is the case, we can choose any values we want to feed into the network (we cannot immediately inject values deeper into the network). If there is filtering or normalization, however, we treat the first layer like all subsequent layers.

If we can inject arbitrary values into the first layer, we can conduct a chosen-input, neuron-centric attack pass by iterating over every neuron in the first layer and recovering their hidden parameters in 2 phases: calibration and binary search.

1) The Calibration Phase: Because the attack is a binary search to find hidden parameters of unknown value, can achieve dramatic speedups if we can find a reasonable upper bound to start the search from, rather than stepping down from the maximum possible float value. It is also useful to know the signs of the weights connected to our neurons. We can recover both pieces of information through the calibration step.

We prepare a test input to the network that is all small, non-zero values except for the first value which is a very large value. Next, we perform inference on this test input and consult the trace to see if the target neuron overflowed or underflowed. If it has, we know both that this input is already large enough to cause an overflow/underflow and the sign of the weight between the first input and this neuron. If the neuron has not overflowed or underflowed, (owing to a comparatively lower magnitude weight) we simply multiply the large value in our test input by 10 and try again. We continue in this way until we find the sign and magnitude required to over/underflow this neuron from each input to the network. Note that we do not

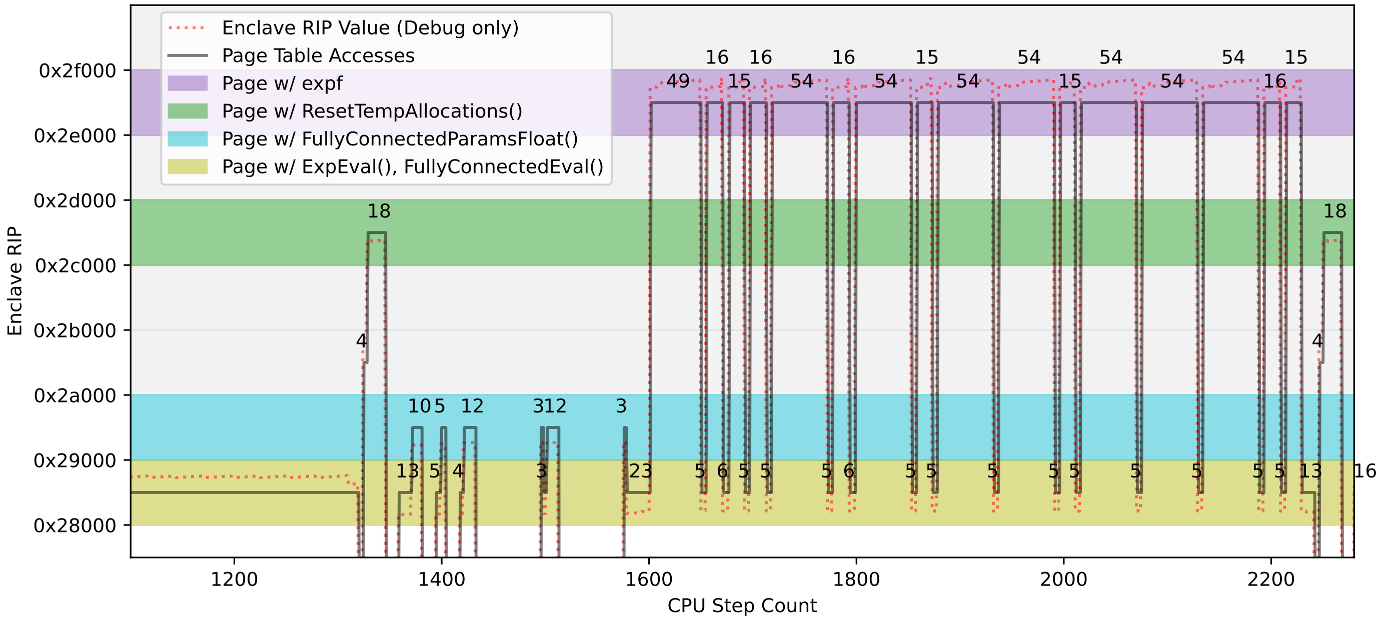

Fig. 4. Inset of an SGX-Step instruction-granular page access trace showing the execution of the first layer of a NN with 16 neurons. Note that the intra-page instruction pointer (RIP) values (recovered by putting an enclave in debug mode) are displayed here only for reference purposes and are not used in our attack. We include them to show that knowing the current page provides enough information to interpret traces without precise RIP values.

actually binary-search towards a threshold between two cases here; we merely attempt to reach an overflow/underflow case. It is not necessary to find precise threshold points to learn the rough magnitude and sign of the inputs.

Recovering the signs for each weight is useful since some of the cases (most of the “normal” ones) are symmetrical about zero, meaning we cannot learn the sign of the input only based on the state. Recovering the sign before we begin the binary search means we always know exactly where we are in the state space. Searching “up” in this calibration step gives us a sensible starting point to search down and thus reduces the number of invocations needed in the next phase.

2) The Binary Search Phase: Armed with the information collected in the calibration phase, we continue the attack against the same neuron as in phase 1 by finding the desired number of convergence point sets. For as many convergence sets as we need, we start by generating an input set that is made of n locked random values (one for each input) except that the i'th input is instead set to the large positive value recovered during calibration. We feed the input into the network and note the return case by interpreting the CPU instruction counts trace for the neuron. Even if some of the weights have different signs, the overall Σ should be dominated by the large input multiplied by its weight. If the neuron has overflowed, our input has caused the neuron to enter the state above the threshold between normal (+) and overflow and should search down. If the neuron has instead underflowed, we must be in the boundary between underflow and normal (-). If the activation's input is negated before being passed into `exp()` (as in `sigmoid()`), the direction to search should be reversed. We continue the binary search on that i'th input, searching back up if we fall below an underflow/overflow case in terms of magnitude. Eventually the search ends either because the desired search depth was reached or because we reached the limit of precision for the underlying floating point maths.

Using this algorithm, we can generate as many novel convergence sets as we require since the unchanging members of the input set are all randomised between inputs. It is worth noting that the system of equations does not produce usable results if the values on the right hand side are all the same threshold, so targeting multiple thresholds is required. One quick way to find more thresholds for the system of equations is to flip the signs of the inputs to ensure both the underflow-to-normal and overflow-to-normal thresholds are included in the solution set. Note however that it is not required to target the thresholds between underflow/normal and normal/overflow; these are just the easiest to find since we can force an under/overflow and then work our way up/down until we find the normal region. There are other thresholds within `exp()` (as shown in Figure 2) that can be exploited if they can be reached so long as we know the sign of the weight and thus in which precise state we are in.

Once we have the requisite number of equations, we can solve the system of equations as described in Section IV-A to recover the weights/bias for this first neuron. We then perform the exact same two-part algorithm on each remaining neuron until we solve the entire layer. The output from our recovery tool as it recovers one neuron is depicted in Appendix C.

E. Extending the Attack to Deeper Layers

In order to extend the attack into subsequent layers, we must solve two key problems. First, how can we mitigate or 'unwrap' already-solved early layers to insert arbitrary inputs beyond them, deeper into the network? And second, is it always possible to generate those values?

Unwrapping Solved Layers. Our attack methodology for the first layer relies on being able to precisely craft a set of "target" in order to search towards threshold points. In order to be able to do this for deeper layers, we first have to "unwrap" all the recovered layers between the input layer and the layer we target. In Appendix D, we show the simple linear algebra required to rearrange the normal operation of a neuron to isolate the input as opposed to the target.

In these equations, act and $actinv$ are known since we know the architecture of the network and W and b are known because they belong to a solved layer. Therefore, given these recovered parameters, a desired 'target' set of values, and the appropriate inverse activation function, we can calculate the input we must feed into the layer to produce a desired target as the output of that layer. Note this assumes the activation function is invertible without a substantial loss of accuracy; inverting these functions might lead to an inability to pass certain values forward, which we discuss below.

Expressive Power. Even if we can algebraically calculate how to pass arbitrary values through a solved layer, subsequent layers are harder to attack because we are limited by the "expressive power" (see Table I) of the neurons before the target layer in terms of both magnitude and sign. There are two consequences to this.

The first is some values may be impossible to generate in subsequent layers. For example, since the output of `sigmoid()` is always positive, baring negative biases, it would be impossible to send negative values into the next layer. Similarly, `relu()` can never produce a negative output. The second is that certain thresholds such as those between `expf()`'s overflow/underflow and normal cases have a high magnitude ($\approx 88.723 - 103.972$) relative to generally smaller magnitude weights and biases. `sigmoid()` has a very small output range $(0-1)$ making it impossible to produce a large enough Σ in the next layer to 'reach' one of these two thresholds unless either the network is very dense with small-magnitude positively weighted neurons or has some positive and large magnitude (> 1) weights connected to it. This suggests denser networks are more expressive and therefore more vulnerable. In both of these cases, we rely on certain network properties. Though this means that certain networks might not be vulnerable to this method, security guarantees should never rest on architectural decisions.

Attack Outline. Nevertheless, extending the attack into deeper layers is possible with certain caveats. There is no need to focus only on the high magnitude thresholds we previously discussed between the underflow/overflow cases and the normal one; there are six in the range from ≈ -1.04–1.04 (see Figure 2). By searching anywhere within this region (e.g., near the subnormal case as in [27]) we can, as before, build convergence sets for our inputs and solve for the hidden parameters. Because we might pass through a low expressiveness activation function like `sigmoid()` (output range from 0–1), which is likely being further reduced by multiplication with a weight with magnitude <1, we might not cross a single threshold point (the presence of the bias as a component of Σ means that we might be shifted away from zero far enough that we cannot reach both sides of the subnormal case).

As such, we instead start with a grid search to scan (at some resolution up to the attacker which trades off speed and the chances of finding a narrow instruction count class) for multiple instruction count classes. As with the binary search, we start by creating an input set that is made up of n locked random values (one for each input) except that the (randomly chosen) i'th input is nominated as a dynamic value, initially set to a high magnitude positive number. We scan the dynamic value from its initial value down to a high magnitude negative number, we can check to see if encounter multiple return cases. If we are able to reach multiple classes, we can then perform a binary search between those classes to isolate the threshold and build a convergence point set, as before.

F. Estimating Attack Performance

We note that the number of executions to perform our attack is dependent on the architecture of the network (and therefore the number of parameters to recover) but also some factors the attacker can control such as the depth of the binary search. Naively, the total number Q of queries to recover the first layer can be computed as: $Q = P \cdot N \cdot S$, where P is the number of hidden parameters (in the first layer, e.g., one weight per input plus one for the bias) to solve for, N is the number of neurons in the first layer, and S is the maximum number of steps in the binary search, minimised by the calibration process described in Section IV-D1.

The P value correlates to the minimum number of equations we require to solve a system of equations: one for each unknown. We note that we can compensate for noise or other errors by adding extra equations to produce better solutions at the cost of more queries. In testing, finding 3 additional convergence sets helped reduce the maximum error by 28% at depth 25 compared to an attack where precisely P convergence sets were generated (this is visualised in Appendix K). It might be possible with future study to detect particularly weak systems of equations analytically and dynamically add extra convergence sets to improve the results.

V. Experimental Evaluation

Here we introduce and share the results of our attack against three PoC networks trained on different datasets in "full" Tensorflow (in Python) and then converted to Tensorflow Microlite using the process described in Section IV-C.

A. $M_{insurance}$: A Regression Model using `expf()`

Model. We first focus on a regression model to predict home insurance costs, which was trained on the "Medical

Cost Personal Dataset" [53]. The network has three hidden layers. The first hidden layer contains 100 neurons that use the `Exponential()` activation function which is just a direct call to `expf()`. There are 11 input nodes, meaning that there are in total 1200 parameters (1100 weights and 100 biases) that we target. After the first layer, there is a second hidden layer of 10 neurons which uses a `relu()` based activation function. A final layer contains a single `relu()` neuron.

Recovery. For this first network, we only focus on the first hidden layer. Despite that, we note that:

1) These choices give us the clearest method to discuss and explain the underlying vulnerability (secret-dependent control flow).
2) More complex networks or types of networks do not inherently offer greater security unless the underlying problem is addressed, even if demonstrating the attack might be more technically challenging.
3) The attack is agnostic to the accuracy of the network.

Results. Our attack fully recovered all the weights and biases for the neurons in the first layer of a Tensorflow Microlite network running inside an SGX enclave. After calibration, the full attack took 55 binary searches of the network per parameter to correctly recover each of the 1200 parameter in the first layer down to or beyond 3 decimal places of accuracy. This is in contrast to Tramèr, Zhang, Juels, *et al.*'s budget of 100 queries per parameter.

During the attack, each first-layer neurons' parameters were recovered at around 99% accuracy (when compared to the ground truth values) by <36% of the way through the full search (around search depth 20). The average error is <1% at this point. The error continues to slowly improve until the search hits the limits of a 32-bit floating point number at around depth 55. These findings are further summarised in Appendix E, which shows how the achieved average and maximum error in the recovered weights and biases varies with the number of queries per neuron in $\mathrm{M}_{\text{insurance}}$.

The maximum error rates were higher in our full attack on $\mathrm{M}_{\text{insurance}}$ than we had seen in smaller scale testing. We speculate this is due to numerical precision issues when solving certain neurons, particularly ones with very small weights or perhaps a combination of very low and very high magnitude weights. During the binary search there may be an ideal magnitude for the non-dynamic input values based on the magnitude of the dynamic input. We consider this an area for future study and discuss a workaround below. As this network was substantially larger than those used in small scale testing, we developed two ideas to help accelerate the attack for the other case studies.

Max Search Depth. We note that the attack can be sped-up by only binary searching to a certain depth of iterations. As can be seen in Appendix E, we already achieve an $< 1\%$ average error rate by depth 20. Based on the desired level of accuracy, an attacker can trade off between time and fidelity to the original network and choose to only search to some depth, rather than letting the binary search finish. We note that achieving sub 1% precision may not be as necessary as it seems; the widespread adoption of quantisation has shown that models can still be quite performant despite the precision loss incurred by the lossy conversion of hidden parameters to 8 bit integer representations. We also note that our sub 1% precision attack has complexity below that of Tramèr, Zhang, Juels, *et al.*, which also only targets classification networks and not continuous output networks like this one.

Improved Method of Calibration. As described in Section IV-D1, the calibration phase saves us from wasting many search steps down from some large safe constant to the smallest values that will overflow/underflow the weights. In the previous equation, S is therefore optimised by starting the search from the smallest possible point that still overflows/underflows our target. By automating this calibration process for each neuron, we ensure a search with the fewest steps possible.

Though we first implemented calibration as a per-neuron process (as previously described) we realised when trying to attack this model that this step could be performed across an entire layer of neurons simultaneously. We can leverage the fact that our SGX-Step traces for a given input provide an instruction count measurement for all neurons simultaneously to perform a combined or input-centric calibration. We first create an input array for the network that is all zeros (one for each input). Next, we iterate through each index in the input, replacing the 0 with increasing powers of some constant, e.g., 10 and passing the array into the network. We then record when an input array has underflowed/overflowed all the neurons in the layer. This maximal value of the power of that constant, D, when multiplied by $P-1$ (the number of inputs) allows us to calculate C, the number of executions required to calibrate the whole network: $C = (P-1) \cdot D$.

Note that C is independent of N and S. As Q would likely be dominated by N in a real network, this calibration phase should account for only a small fraction of the total queries in the full attack. By adding C to Q we get a good approximation for the number of executions required to recover the first layer and calibrate faster than via the original method.

B. M_{mult}: A Multi-Layer Model using `sigmoid()`

Next, we show how to attack a more realistic activation function and recover partial information from deeper layers.

Model. We now focus on a network trained to multiply two numbers with the following architecture: there are two inputs, followed by the first hidden layer which contains 4 `sigmoid()` neurons. Then there is a second hidden layer containing 8 `sigmoid()` neurons followed finally by the final layer with a single `relu()` neuron.

Recovery. We follow the steps described in Section IV-D and Section IV-C to recover the first layer. The only difference is that we exploit a different CPU instruction count leakage; instead of targeting `tlibc`'s exponential function, we now attack the Tensorflow lite framework-level `sigmoid()` function as described in Section IV-B. This shows how our attack is adaptable to different and more realistic activation functions, as well as side channel leakages elsewhere in the ML inference pipeline. We then use the procedure described in Section IV-E to attack the second layer.

Results. This recovery is summarised in Appendix F and is comparable to the results of our attack against $\mathrm{M}_{\text{insurance}}$. We are also able to recover partial (signless) convergence sets for the neurons in the second layer. For example, we can recover 35 sets for the first neuron in the second layer. Given its 4 inputs and single bias term (5 total unknowns) these are many more convergence point sets than we would need for a solution. However, because we do not have sign information, we are unable to conclusively solve for the neurons' parameters.

C. M_{MNIST}: MNIST with `sigmoid()`

Model. Lastly we discuss a more complex model which was trained on the MNIST dataset [54]. Our MNIST case study is much more complex than the others before it featuring $28x28 = 784$ inputs and a first layer with 128 neurons. Adding the biases, this network has $100,480$ hidden parameters in the first layer making it two orders of magnitude larger than $\mathrm{M}_{\text{insurance}}$. For this study, we focus on only the first layer.

Recovery. Here we combine parts of the recovery strategies of the first two case studies: a larger network which we attack using the framework-level `sigmoid()` vulnerability described in Section IV-B. As in the first model, we only attempt recovery of the first layer.

Results. Because of the greater number of hidden parameters for this model, recovery using the methods described above would take millions of iterations and while feasible, not entirely practical. In Section V-A we discussed how the value Q could be reduced by reducing the search depth, S. Another way to reduce Q is to try to minimise the $P \cdot N = 785 \cdot 128 = 100480$ term. This is possible if we approach the binary search a different way, inspired by the input-centric calibration described previously.

Rather than a neuron-centric regime (focusing on each neuron one at a time and ignoring the effect of the changing inputs on the other neurons in the same layer), we instead record the state of all the neurons in the layer simultaneously as they respond to changes to each input of the network in sequence. We note that in this model's first layer, the weights are normally distributed meaning that there are clusters that can be easily searched together. This allows us to exploit the structure of the distribution of the hidden parameters, group weights with similar magnitudes, and combine searches allowing better than $P \cdot N$ performance. This also allows us to abandon searches in regions without any neurons. See the output of this process over an entire input in Appendix G and a visual in Appendix L. Note that this approach also integrates the input-centric calibration discussed above, which is completed for this input in 18 executions of the network. Also note that every iteration before depth 18 (where each neuron is effectively in its own search space) is a savings against the more naive approach described above.

The output in Appendix G recurses to a minimum gap size of only 0.1, so it does not have the same precision as the previous attack code example from Appendix C. Note that the output of the input-centric attack to depth 0.1 (1050 executions) is roughly analogous in recovery precision to a depth 20 neuron-centric attack ($128*20 = 2500$ executions) meaning that we recover the same value using 42% of the executions. Extrapolating the full attack to maximum depth, we project that the neuron-centric approach would take $2,460,800$ executions (without calibration) and this input-centric approach would take $1,180,765$ executions with calibration, or 48%.

The Tramèr, Zhang, Juels, *et al.* attack requires a budget of 100 times the number of parameters in the network. If we only consider the first layer, this network would require a budget of $(28*28+1)*128*100 = 10,048,000$ queries (there are 28*28 weights between each input and each neuron, and we add one extra parameter for the bias term). Using the values above, we project a full depth scan of this layer in $1,180,765$ queries, which is 11% of the Tramèr, Zhang, Juels, *et al.* budget.

While we cannot report the accuracy of the full recovery, spot checks against ground truth values were very promising and could be made more (or less) precise by tuning the min gap parameter to stop the search when desired.

VI. Ecosystem Analysis

We examined implementations of common activation functions in popular ML frameworks to assess if they exhibit input-dependent patterns. For this, we used a combination of static (Ghidra) and dynamic (gdb) analysis to explore the call stacks of these frameworks until an underlying maths function was reached. Our findings are summarised in Table II.

There are several dimensions to our survey: first, we found that in all cases we tested, the low-level maths functions that underpin activation functions are not included in the ML frameworks themselves. Instead, they rely either on additional high performance maths libraries, e.g., `sleef` in the case of PyTorch, `Eigen` and `DNNL` in the case of Tensorflow and `XNN` in the case of Tensorflow Lite or call into standard libraries like `tlibc` (in the case of SGX) or `glibc` on a Linux system. In some cases, these frameworks do both for different functions, adding to the complexity of the analysis.

The (in)security of an activation function can stem from the underlying library and implementation of these functions. For example, a call to `sigmoid()` that ends up in `glibc`'s highly input-dependent `exp()` in `libm.so` exposes the caller's ML framework to vulnerability. Similarly, a call to the same function in a different framework that ends up in a highly optimised maths library like `Eigen` may be harder to exploit. However, as we note in Section IV-B and Section V-B, higher level framework code can also be the culprit of exploitable memory access patterns. The operational context may also play a roll in vulnerability; e.g., an embedded edge device may not have the hardware functionality for high performance or even floating point maths routines.

The reason optimised maths libraries may be harder to exploit is their use of vectorised instructions (e.g., AVX on Intel or NEON on ARM) which discourage branching (since that can affect performance). For example, we observed that in PyTorch, `softmax()` was realised through calls into the

TABLE II
SURVEY OF VULNERABILITY ACROSS VARIOUS ML AND STANDARD LIBRARIES AND FOR DIFFERENT OPTIMISATION LEVELS.

ML Framework	**RELU**	**Exponential**	**Sigmoid**	**Softmax**
TFLiteMicro (SGX/tlibc, `-O0`)	✗	✗	✗	✗
TFLiteMicro (SGX/tlibc, `-Os`)	✓	✗	✗	✗
TFLiteMicro (`glibc`, `-Os`)	✓	✗	✗	✗
TFLite (`glibc`)	✓	✗	✓	✓
TensorFlow CPU (`glibc`)	✓	✓	✓	✓
PyTorch (`glibc`)	✓	n.a.	✗	✓*

Legend: ✓: Secure ✗: Insecure due to data-dependent access pattern
*: Only secure if `cpuid` indicates AVX support

`sleef` library, which at default optimisations compiled into AVX assembly code without memory access leakage. We note however that `sleef` incorporates a *dispatch* mechanism that calls into different implementations depending on the outcome of `cpuid`. Notably, if this library was used in an SGX enclave, where `cpuid` is typically implemented as an *untrusted* `ocall`, the adversary could spoof the `cpuid` result and force usage of the insecure non-AVX version. Curiously, for `sigmoid()`, PyTorch resorts to the standard library implementation of `exp()`, rendering the implementation insecure independent of `cpuid`. Tensorflow uses `Eigen` for all the functions we tested except for `Exponential()`, which calls directly into `glibc`. Finally it is worth restating that even though vectorised code is non-branching, it is not guaranteed to execute in constant time [27].

Finally, Table II shows how the optimisation level has a direct impact on the security of `relu()` for Tensorflow Microlite. While the compiler default of `-Os` happens to be free of memory access leakage, this is more by accident than by design, which is not desirable in the case of libraries that handle sensitive data. Analogously, the security benefits of vectorised code against our attack is not a proactive security-related decision but a side effect of pursuing performance.

Vulnerability of* `relu()`*. In addition to `expf()`, we also considered `relu()` and its implementation in Tensorflow Microlite, which relies on the `std::max()` function. When running inside of an enclave, the `std::max()` library call is provided by an implementation in SGXs `tlibc` library which can be seen in Appendix H. With the default settings (`gcc 11.4.0 -Os` with function marked `inline`), the above code was compiled into the instructions shown in Appendix I.

When compiling with `-O0`, however, the function was assembled as shown in Appendix J. This is significant because when the compiler emits `jmp`-class instructions, it is possible (using `sgx-gdb`) to detect the single step difference and therefore discern between the cases where $input \leq 0$ and $input > 0$. This is a threshold point between two instruction count classes about 0.0, similar to the ones discussed previously in `expf()`. Though the difference might only be a single CPU instruction (whether a jump was taken or not) this difference is detectable.

Appendix I, on the hand, has no step count differences due to its use of the `CMOVA` instruction which combines a branch and a move into a single instruction. Whatever the outcome of the conditional, `CMOVA` always takes the same number of steps, removing the timing leak.

Within our case study, `relu()` within Tensorflow Microlite was not vulnerable, but it was not secure by design either. This is worrisome because in contrast to vetted cryptographic code, ML libraries currently *do not* exhibit input-independent memory access and execution behaviour, and further the eventual security properties of high-level ML code may depend on compiler optimisations and other non-explicit behaviour as with `relu()` in some configurations. Without any specific safeguards taken, ML libraries are not intrinsically secured against the vulnerability discussed in this paper.

We note that on x86, many floating point instructions can raise exceptions or otherwise exhibit operand-dependent timing as mentioned by Gongye, Fei, and Wahl and (in a different context) Kohlbrenner and Shacham, which might result in timing vulnerabilities even if branchless code is emitted [27], [55]. We leave the exact measurement and exploitation of such timing side channels in real-world settings as an area for future research.

Vulnerability of Framework Code. As discussed in Section IV-B, and demonstrated practically in Section IV-E, exploitable memory access pattern vulnerabilities can exist outside mathematical library code in ML frameworks themselves. This further supports our case that these frameworks are not suitable for enclave execution under a SCA advisory. Beyond the security properties of the libraries they use, security-conscious ML framework developers need to also consider their own code and if they are introducing exploitable timing side channels. This is especially true for more complex activation functions like `softmax()` which are not mathematical functions likely to be found in standard maths libraries.

VII. DISCUSSION AND MITIGATIONS

We show that input-dependent memory accesses and branching are prevalent in ML implementations, and that the necessity for secure programming practices is not widely understood in the ML community. Our PoCs shows that an attacker with a high-resolution controlled or side channel is able to exploit data-dependent memory accesses to recover model details, whereas prior software SCA typically only recovered hyperparameters such as model architecture [15]. Though there are certain limitations, $\mathrm{M}_{\mathrm{insurance}}$ and $\mathrm{M}_{\mathrm{mult}}$ show precise weight recovery can be performed with at maximum 55 queries per weight, and at over 99% accuracy at less than half that number (plus calibration). Search depth can be parametrically adjusted to account for the desired accuracy.

Ultimately, while out of scope for this paper, we believe that if an attacker can obtain precise-enough timing measurements, similar attacks may be possible outside of a TEE context, e.g., for virtualised cloud deployments. Cache attacks with high temporal resolution, e.g., Prime+Scope [56], could be enough to recover traces similar to SGX-Step, though attacks would likely require modifications to deal with increased noise.

Regardless of whether ML is being performed in enclaves, we argue that hence, ML library developers should pay attention to the full implementation attack surface including library and framework code. Even if e.g., `relu()` implementations were often secure in the surveyed libraries, this largely relied on certain compiler optimisations and was not 'by design'. Similar to cryptographic libraries, ML libraries should consider hardening their codebase against software side channels such as data-dependent memory access patterns. Security and privacy-sensitive algorithms, to which ML now belongs, must be robust by design, regardless of context, platform, or (mis)use of library interfaces by a programmer.

We briefly explored quantised networks as an extension to our work. With fewer bits representing each hidden parameter, we reasoned binary searches might encounter discrete boundaries in fewer iterations. We converted our MNIST model using Tensorflow's full integer quantisation to a Tensorflow lite model with 8bit integer representations of its weights and biases and captured traces using our SGX-Step setup (an inset of which is visualised in Appendix M). Aside from the additional (de)quantisation phases during the network's execution (not pictured), the patterns of page accesses and timings were very different to previous traces, owning to the different functions / processes used to perform inference on fixed point values. Though we did not further investigate the access patterns for this paper, we noticed evidence of non-constant time behaviour in the evaluation of neighbouring neurons suggesting non-constant time effects.

Countermeasures. As with the work done in cryptography libraries, there are established steps to avoid data-dependent branching and memory accesses, e.g., by relying on bitwise logical operations, constant-time conditional moves, and other related techniques. For x86, as discussed for the `sleef` library, using vectorised SIMD code can also mitigate the issue, though needs to be thoroughly vetted for other side channels like floating point timing [27], [55].

At present, akin to a "chosen plaintext" attack for cryptographic algorithms, our attacks entails sending arbitrarily large or small floating point numbers into the network. Any input normalisation might mitigate the fast calibration and binary search approaches described in this paper, only enabling partial recovery through the grid search technique. Similarly, as the attack requires a large number of queries to the network, rate-limiting or detection of "suspicious" usage would render exploitation harder in practice. However, again, ML security should in our view not rest on developers having to normalise inputs or limit performance, so these measures are supplementary to a securely developed library.

Finally, researchers have explored defenses at the level of the TEE itself to frustrate (but not fully eliminate) side-channel leakage exploitation. Initial approaches [57]–[59] focused on detecting suspicious interrupt rates as a side-effect of an ongoing controlled-channel attack, which may be error-prone and suffer from false positives. Alternatively, custom oblivious RAM solutions [60]–[62] may probabilistically hide secret-dependent enclave memory accesses, however this is at the cost of prohibitive performance overheads. Most prominently, AEX-Notify [41] is a recent Intel SGX hardware-software extension that aims to eliminate SGX-Step's single-stepping capabilities by prefetching selected application pages. Notably, AEX-Notify is an *opt-in* feature only available in recent CPUs. Enclaves that did opt-in would preclude the ability to gather precise instruction counts through single-stepping. However, AEX-Notify explicitly does *not* prevent general information leakage through page faults and, hence, cannot fully mitigate our attacks. While specialised, compiler-assisted defenses [34], [63] have been proposed to mitigate page-fault leakage on off-the-shelf SGX platforms, these approaches incur prohibitive performance overheads, and their applicability to general-purpose ML workloads remains unexplored.

Limitations. We require the victim to run in a co-located TEE in order to use our controlled channel, which is a different requirement than previous model stealing attacks, that require chosen inputs and the output distribution of the inference pass.

Despite the recovery of the first layer of neurons fundamentally undermining SGX's promise to secure the entire computation, our attack is limited in recovering parameters from deeper layers, depending on the model architecture. This is mostly due to the (limited) expressive power of some common activation functions as noted in Table I and Section IV-E. The grid search is less efficient than the binary search used for the first layers, and depending on the magnitude of the weights, small search steps might be needed to discover thresholds. An adaptive approach could deepen the sweep region until a threshold is hit; however, small weights likely contribute little to the network and may be replaced by a static value to produce an equivalent (non-identical) network.

Finally, we focused on input-dependent memory access patterns and instruction counts in this paper, but did not take input-dependent latency of single instructions into account. For example, floating point multiplication is not necessarily guaranteed to execute in constant time [27]. This could render vectorised code like the implementation from `sleef` vulnerable to a modified version of our attack: while different inputs might have identical instruction counts (as reported by SGX-Step), the actual runtime might be different and be observable with other side channels like interrupt latency [37].

VIII. Conclusion

The recent surge in ML applications necessitates an in-depth understanding of new attack vectors and defenses. In this respect, confidential computing architectures, and in particular Intel SGX, have gained significant traction in recent years and hold the potential to securely outsource critical ML computations to pervasive untrusted remote cloud platforms. However, we show that SGX-protected ML implementations exhibit subtle side-channel vulnerabilities that allow the extraction of precise model parameters in our PoC attacks. In the wider perspective, our work may generalise to other TEEs and highlights the issue of input-dependent memory accesses that are prevalent in today's ML implementations, suggesting that security-critical ML applications should consider adopting coding practices studied in the cryptographic community.

ACKNOWLEDGMENTS

The PhD of Jesse Spielman is partially funded by a donation from Intel. This research was further partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/X03738X/1 and EP/R012598/1. This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Program Flanders.

REFERENCES

[1] M. Russinovich, "Confidential computing: Elevating cloud security and privacy," *Commun. ACM*, vol. 67, no. 1, 52–53, Dec. 2023.

[2] Intel, *Reference Architecture for Privacy Preserving Machine Learning with Intel® SGX and TensorFlow*, en, 2024.

[3] U. Kumar and E. Sakata, *Protecting Sensitive Data and AI Models with Confidential Computing*, https://developer.nvidia.com/blog/protecting-sensitive-data-and-ai-models-with-confidential-computing/, 2023.

[4] Q. Li, Y. Xie, T. Du, *et al.*, "Coreguard: Safeguarding foundational capabilities of llms against model stealing in edge deployment," *arXiv preprint arXiv:2410.13903*, 2024.

[5] R. R. Kethireddy, "Secure model distribution and deployment for llms," *Journal of Recent Trends in Computer Science and Engineering (JRTCSE)*, vol. 12, no. 4, pp. 1–14, 2024.

[6] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *2nd Workshop on System Software for Trusted Execution (SysTEX)*, Oct. 2017, 4:1–4:6.

[7] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing Machine Learning Models via Prediction APIs," *arXiv:1609.02943 [cs, stat]*, Oct. 2016, arXiv: 1609.02943 version: 2.

[8] Intel Corporation, *Intel software guard extensions (intel sgx) developer guide*, version 2.26, May 1, 2025.

[9] A. Nilsson, P. N. Bideh, and J. Brorsson, *A Survey of Published Attacks on Intel SGX*, en, arXiv:2006.13598 [cs], Jun. 2020.

[10] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," en, in *2015 IEEE Symposium on Security and Privacy*, San Jose, CA: IEEE, May 2015, pp. 640–656, ISBN: 978-1-4673-6949-7.

[11] ARM, *TrustZone for Cortex-A*, 2024.

[12] Intel, *Trust Domain Extensions (Intel TDX)*, 2024.

[13] AMD, *AMD Secure Encrypted Virtualization (SEV)*, 2024.

[14] I. A. Canales-Martínez, J. Chávez-Saab, A. Hambitzer, F. Rodríguez-Henríquez, N. Satpute, and A. Shamir, "Polynomial time cryptanalytic extraction of neural network models," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2024, pp. 3–33.

[15] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache Telepathy: Leveraging Shared Resource Attacks to Learn {DNN} Architectures," en, 2020, pp. 2003–2020, ISBN: 978-1-939133-17-5.

[16] S. Hong, M. Davinroy, Y. Kaya, *et al.*, *Security analysis of deep neural networks operating in the presence of cache side-channel attacks*, 2020. arXiv: 1810.03487 [cs.CR].

[17] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, *Stealing neural networks via timing side channels*, 2019. arXiv: 1812.11720 [cs.CR].

[18] Y.-S. Won, S. Chatterjee, D. Jap, S. Bhasin, and A. Basu, "Time to leak: Cross-device timing attack on edge deep learning accelerator," in *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, 2021, pp. 1–4. DOI: 10.1109/ICEIC51217.2021.9369754.

[19] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered Insecure: GPU Side Channel Attacks are Practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, 2139–2153, ISBN: 9781450356930. DOI: 10.1145/3243734.3243831.

[20] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 125–137. DOI: 10.1109/DSN48063.2020.00031.

[21] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Side Channel Attacks on GPUs," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 4, pp. 1950–1961, 2021. DOI: 10.1109/TDSC.2019.2944624.

[22] G. Dong, P. Wang, P. Chen, R. Gu, and H. Hu, "Floating-point multiplication timing attack on deep neural network," in *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, 2019, pp. 155–161. DOI: 10.1109/SmartIoT.2019.00032.

[23] R. S. Ali, B. Z. H. Zhao, H. J. Asghar, T. Nguyen, I. D. Wood, and D. Kaafar, *Unintended memorization and timing attacks in named entity recognition models*, 2022. arXiv: 2211.02245 [cs.CR].

[24] T. Nakai, D. Suzuki, and T. Fujino, "Timing black-box attacks: Crafting adversarial examples through timing leaks against dnns on embedded devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 3, 149–175, Jul. 2021. DOI: 10.46586/tches.v2021.i3.149-175.

[25] Y. Dan, T. Shibahara, and J. Takahashi, "Timing attack on random forests: Experimental evaluation and detailed analysis," *Journal of Information Processing*, vol. 29, pp. 757–768, 2021. DOI: 10.2197/ipsjjip.29.757.

[26] F. Alder, J. Van Bulck, D. Oswald, and F. Piessens, "Faulty Point Unit: ABI Poisoning Attacks on Intel

SGX," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20, Austin, USA: Association for Computing Machinery, 2020, 415–427, ISBN: 9781450388580. DOI: 10.1145/3427228.3427270.

[27] C. Gongye, Y. Fei, and T. Wahl, "Reverse-engineering deep neural networks using floating-point timing side-channels," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20, Virtual Event, USA: IEEE Press, 2020, ISBN: 9781450367257.

[28] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465773.

[29] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 515–532, ISBN: 978-1-939133-06-9.

[30] H. Yu, H. Ma, K. Yang, Y. Zhao, and Y. Jin, "DeepEM: Deep Neural Networks Model Recovery through EM Side-Channel Information Leakage," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 209–218. DOI: 10.1109/HOST45689.2020.9300274.

[31] P. Horvath, L. Chmielewski, L. Weissbart, L. Batina, and Y. Yarom, *BarraCUDA: GPUs do Leak DNN Weights*, 2024. arXiv: 2312.07783 [cs.CR].

[32] W. Wang, G. Chen, X. Pan, *et al.*, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, 2421–2434, ISBN: 9781450349468.

[33] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC: USENIX Association, Aug. 2017.

[34] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016, pp. 317–328.

[35] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th USENIX Security Symposium*, Aug. 2017, pp. 1041–1056.

[36] J. Van Bulck and F. Piessens, "SGX-Step: An open-source framework for precise dissection and practical exploitation of Intel SGX enclaves," in *ACSAC 2023 Cybersecurity Artifacts Competition and Impact Award Finalist Short Paper*, Dec. 2023.

[37] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, 178–195, ISBN: 9781450356930. DOI: 10 . 1145 / 3243734 . 3243822.

[38] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled instruction-level attacks on enclaves," in *29th USENIX Security Symposium*, Aug. 2020, pp. 469–486.

[39] A. C. Aldaya and B. B. Brumley, "When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 196–221, 2020.

[40] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes," en, in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, London United Kingdom: ACM, Nov. 2019, pp. 1741–1758, ISBN: 978-1-4503-6747-9.

[41] S. Constable, J. Van Bulck, X. Cheng, *et al.*, "AEX-Notify: Thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves," in *32nd USENIX Security Symposium*, Aug. 2023, pp. 4051–4068.

[42] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "SEV-Step A Single-Stepping Framework for AMD-SEV," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 1, 180–206, Dec. 2023. DOI: 10.46586/tches.v2024.i1.180-206.

[43] L. Wilke, F. Sieck, and T. Eisenbarth, "TDXdown: Single-stepping and instruction counting attacks against intel TDX," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14–18, 2024*, 2024. DOI: 10.1145/3658644.3690230.

[44] P. Shome, *Closing the Intel TDX page fault side channel, or, the case for TDExit-Notify*, https://collective.flashbots.net/t/closing-the-intel-tdx-page-fault-side-channel-or-the-case-for-tdexit-notify/3775/1, Aug. 2024.

[45] I. Puddu, M. Schneider, D. Lain, S. Boschetto, and S. Čapkun, "On (the lack of) code confidentiality in trusted execution environments," in *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2024, pp. 4125–4142.

[46] Z. Liu, Y. Yuan, S. Wang, X. Xie, and L. Ma, "Decompiling x86 deep neural network executables," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7357–7374.

[47] Sun Microsystems, *Tlibc's expf implementation*, https://github.com/intel/linux-sgx/blob/80a662/sdk/tlibc/math/e_expf.c, 1993.

[48] A. Liu, B. Feng, B. Xue, *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[49] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[50] N. Shazeer, *Glu variants improve transformer*, 2020. arXiv: 2002.05202 [cs.LG].

[51] H. Touvron, T. Lavril, G. Izacard, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[52] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[53] M. Choi, *Medical cost personal datasets*, 2018.

[54] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[55] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 69–81, ISBN: 978-1-931971-40-9.

[56] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 2906–2920, ISBN: 9781450384544. DOI: 10.1145/3460120.3484816.

[57] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *24th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2017.

[58] S. ul Hassan, I. Gridin, I. M. Delgado-Lozano, *et al.*, "Déjà vu: Side-channel analysis of mozilla's nss," *arXiv preprint arXiv:2008.06004*, 2020.

[59] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX Annual Technical Conference (ATC)*, 2018, pp. 227–240.

[60] S. Aga and S. Narayanasamy, "Invisipage: Oblivious demand paging for secure enclaves," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 372–384.

[61] P. Zhang, C. Song, H. Yin, D. Zou, E. Shi, and H. Jin, "Klotski: Efficient obfuscated execution against controlled-channel attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1263–1276.

[62] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "Dr. SGX: automated and adjustable side-channel protection for SGX using data location randomization," in *35th Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 788–800.

[63] D. Vanoverloop, A. Sanchez, F. Toffalini, F. Piessens, M. Payer, and J. Van Bulck, "TLBlur: Compiler-assisted automated hardening against controlled channels on off-the-shelf Intel SGX platforms," in *34th USENIX Security Symposium (USENIX Security 25)*, Aug. 2025.

APPENDIX A
IMPLEMENTATION OF SIGMOID() (CALLED LOGISTIC()) IN TENSORFLOW LITE

```
inline void Logistic(const RuntimeShape& input_shape,
    const float* input_data,
                     const RuntimeShape& output_shape,
    float* output_data) {
  const float cutoff_upper = 16.619047164916992188f;
  const float cutoff_lower = -9.f;

  const int flat_size = MatchingFlatSize(input_shape,
    output_shape);

  // [comments removed]

  for (int i = 0; i < flat_size; i++) {
    float val = input_data[i];
    float result;
    if (val > cutoff_upper) {
      result = 1.0f;
    } else if (val < cutoff_lower) {
      result = std::exp(val);
    } else {
      result = 1.f / (1.f + std::exp(-val));
    }
    output_data[i] = result;
  }
}
```

Listing 2. Implementation of `Logistic()` in Tensorflow lite

APPENDIX B
EXPF() CALL STACK FOR TENSORFLOW MICROLITE NEURON

```
0x7ffff762b460 in expf()
0x7ffff7625e9c in tflite::(anonymous namespace)::ExpEval
    ()
0x7ffff762318b in tflite::MicroInterpreterGraph::
    InvokeSubgraph()
0x7ffff760f04e in performInference()
0x7ffff7610577 in entry_point()
0x7ffff7611633 in sgx_entry_point()
0x7ffff76130a2 in do_ecall()
0x7ffff762c5a5 in enter_enclave()
0x7ffff762c7d5 in enclave_entry()
0x7ffff7f95309 in __morestack()
0x7ffff7f986fb in CEnclave::ecall()
0x7ffff7f9a852 in _sgx_ecall()
0x55555555bac9 in entry_point()
0x555555557a24 in main()
```

Listing 3. Simplified call stack for `expf()`-based neuron in Tensorflow Microlite.

APPENDIX C
NEURON-CENTRIC ATTACK OUTPUT

```
[ Neuron 2 ]
stats:
  952.99 seconds since start
  1439 executions so far
  4 Incomplete logs
  0 non-deterministic steps
  6 neuron_check failures
Function = exp
Search strategy: seeded binary search
  Calibrating:
    count: 14
    recovered signs: + - - - + - - - - + +
    recovered maxvals: 1000 1000 1000 1000 1000 10000
 1000 1000 10000 10000 1000
  Finding convergence points for equation 1
    depth 0: Overflow
    depth 10: Normal
    depth 20: Overflow
    depth 30: Normal
    depth 40: Overflow
    depth 50: Overflow
  Finding convergence points for equation 2
    <SNIP>
  Finding convergence points for equation 12
    depth 0: Overflow
    depth 10: Normal
    depth 20: Normal
    depth 30: Normal
    depth 40: Overflow
    depth 50: Overflow
deepest solution: [
  0.14825567 -0.17573831 -0.28457861 -0.24754734
  0.09355622 -0.09359772 -0.2866397  -0.2083804
  -0.10068617  0.07542896  0.10219476 -0.04099777]
ground truth: [
  0.14825566 -0.17573832 -0.28457862 -0.24754737
  0.09355621 -0.09359772 -0.28663969 -0.2083804
  -0.10068618  0.07542896  0.10219476 -0.04099637]
abs percent error: [
  0.0%, 0.0%, 0.0%, 0.0%, 0.0%, 0.0%,
  0.0%, 0.0%, 0.0%, 0.0%, 0.0%, 0.0%]
Checkpoint saved!
```

Listing 4. Example output from our neuron-centric attack tool finding convergence points.

APPENDIX D
LAYER UNWRAPPING

$$\begin{aligned} act(input * W + b) &= target \\ input * W + b &= act^{-1}(target) \\ input * W &= act^{-1}(target) - b \\ input * \cancel{W * W^{-1}} &= (act^{-1}(target) - b) * W^{-1} \\ input &= (act^{-1}(target) - b) * W^{-1} \end{aligned}$$

Fig. 5. Note that it is also possible to do this without matrix inversion for greater numerical efficiency, e.g., using `numpy.linalg.lstsq(W, act_inv(target)-b)`.

APPENDIX E
$\mathrm{M}_{\mathrm{INSURANCE}}$ RESULTS

TABLE III
AVERAGE AND MAXIMUM ERROR FOR DIFFERENT SEARCH ITERATIONS OF $\mathrm{M}_{\mathrm{INSURANCE}}$

Iterations	**Average error**	**Max. error**
5	6.029863100281893	844.0875066255469
10	2.836460066697138	786.2672913954896
15	0.271652801976045	156.0382954397002
20	0.008040720460419	4.5014965290057
25	0.000536511600745	0.2550122973594
30	0.000389102878050	0.1506828257822
35	0.000382396333823	0.1465018434229
40	0.000382652272853	0.1466460219544
45	0.000382651560351	0.1466454696212
50	0.000382651519816	0.1466455095500
55	0.000422195967951	0.1466455084049

APPENDIX F
$\mathrm{M}_{\mathrm{MULT}}$ RESULTS

TABLE IV
AVERAGE AND MAXIMUM ERROR FOR DIFFERENT SEARCH ITERATIONS OF $\mathrm{M}_{\mathrm{MULT}}$

Iterations	**Average error**	**Max. error**
5	5.104921006404077	17.91686599947229
10	1.675645751550916	17.51883507815715
15	1.529534168402632	17.52095831363468
20	0.069626021904943	0.79887716427850
25	0.002053382484092	0.02355709795055
30	0.000041334563160	0.00045190244010
35	0.000033134719817	0.00036192843021
40	0.000032159960239	0.00035062330641
45	0.000032157624911	0.00035059400594
50	0.000032157548965	0.00035059308250
55	0.000032157569172	0.00035059331140

Appendix G
Input-Centric Attack Output

```
Equation 3 / 785 / Other Value: 0.017745 / Input idx: 2
Calibrating...
Calibrated in: 18 executions to 131072
iteration: 0 / gaps: 1 / spans: 131072.0
iteration: 1 / gaps: 2 / spans: 65536.0
iteration: 2 / gaps: 2 / spans: 32768.0
iteration: 3 / gaps: 2 / spans: 16384.0
iteration: 4 / gaps: 3 / spans: 8192.0
iteration: 5 / gaps: 5 / spans: 4096.0
iteration: 6 / gaps: 7 / spans: 2048.0
iteration: 7 / gaps: 9 / spans: 1024.0
iteration: 8 / gaps: 12 / spans: 512.0
iteration: 9 / gaps: 17 / spans: 256.0
iteration: 10 / gaps: 25 / spans: 128.0
iteration: 11 / gaps: 34 / spans: 64.0
iteration: 12 / gaps: 43 / spans: 32.0
iteration: 13 / gaps: 55 / spans: 16.0
iteration: 14 / gaps: 74 / spans: 8.0
iteration: 15 / gaps: 94 / spans: 4.0
iteration: 16 / gaps: 108 / spans: 2.0
iteration: 17 / gaps: 117 / spans: 1.0
iteration: 18 / gaps: 123 / spans: 0.5
iteration: 19 / gaps: 130 / spans: 0.25
iteration: 20 / gaps: 134 / spans: 0.125
Done with equation #3/785 in 1050 iterations!
Saving equation 3 to cache
```

Listing 5. Example output from our optimzied input-centric attack tool finding convergence points. Note how after iteration 18 each of the 128 neuron is in it's own gap.

Appendix H
TLIBC's std::max()) in Tensorflow Lite

```
template <class _T1>
struct __less<_T1, _T1>{
    _LIBCPP_INLINE_VISIBILITY
  _LIBCPP_CONSTEXPR_AFTER_CXX11
    bool operator()(const _T1& __x, const _T1& __y) const
    {return __x < __y;}
};

template <class _Tp, class _Compare>
inline _LIBCPP_INLINE_VISIBILITY
  _LIBCPP_CONSTEXPR_AFTER_CXX11
const _Tp&
max(const _Tp& __a, const _Tp& __b, _Compare __comp){
    return __comp(__a, __b) ? __b : __a;
}
```

Listing 6. std::max() source code from sgxsdk/include/libcxx/algorithm.

Appendix I
std::max() at -Os

```
MOVSS     XMM0,dword ptr [param_2]
UCOMISS   XMM0,dword ptr [param_1]
MOV       RAX,param_1
CMOVA     RAX,param_2
RET
```

Listing 7. Resulting assembly for std::max() at -Os with gcc 11.4.0.

Appendix J
std::max() at -O0

```
PUSH     RBP
MOV      RBP,RSP
MOV      qword ptr [RBP + local_10],param_1
MOV      qword ptr [RBP + local_18],param_2
MOV      RAX,qword ptr [RBP + local_10]
MOVSS    XMM1,dword ptr [RAX]
MOV      RAX,qword ptr [RBP + local_18]
MOVSS    XMM0,dword ptr [RAX]
COMISS   XMM0,XMM1
JBE      001019f0
MOV      RAX,qword ptr [RBP + local_18]
JMP      001019f4
MOV      RAX,qword ptr [RBP + local_10]
POP      RBP
RET
```

Listing 8. Resulting assembly for std::max() at -O0 with gcc 11.4.0.

Appendix K
Error Comparison with Extra Convergence Sets

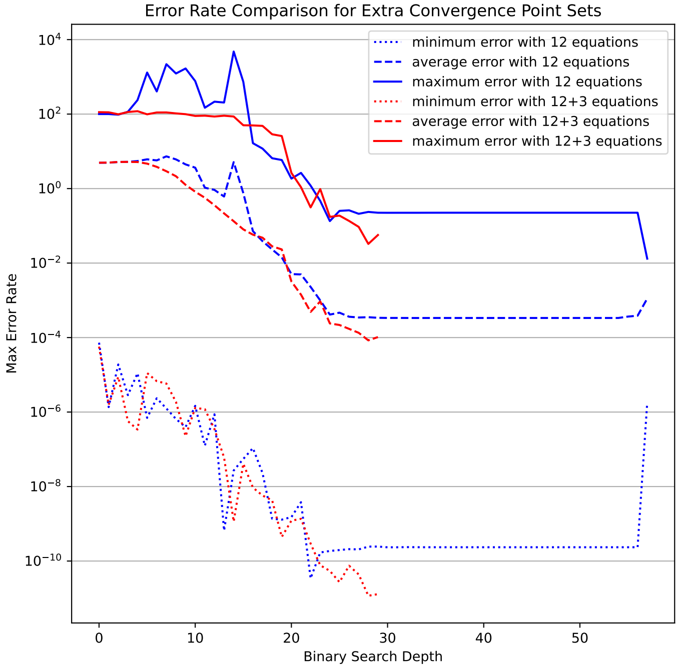

Fig. 6. Note the effect that extra convergence sets have to reduce variance and lessen the errors, especially after depth=25

Appendix L
Search Strategy Comparison

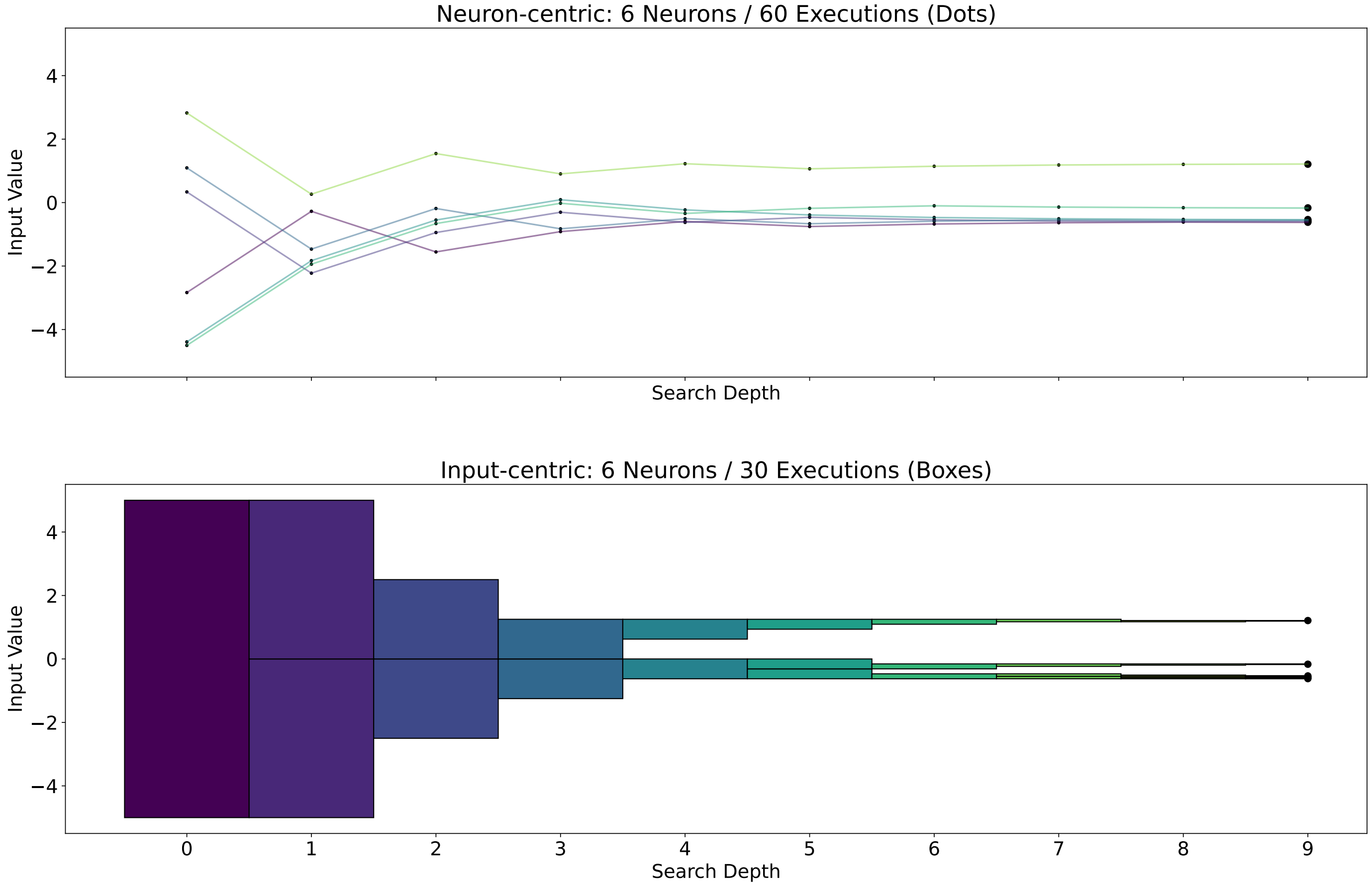

Fig. 7. A visualisation comparing the execution count of a neuron-centric approach (top) and input-centric approach (bottom) for a single input and 6 normally distributed synthetic 'neurons'. For simplicity, we ignore the calibration process. The neuron-centric approach is purely multiplicative (neurons times search depth) so for six neurons and a depth of 10 we cause 60 executions. Here, each colour represents one neuron and the lines represent the binary search process.

The input-centric approach, on the other hand, exploits the distribution of the neurons to 'save' executions by recursively bundling queries together for as long as possible until they have diverged into their own unique streams. Due to the clustering of these 6 neurons, we can gain the same amount of information in half the number of executions as the first approach. In this subplot, colour represents search depth.

Appendix M
Inset of trace from quantised MNIST network

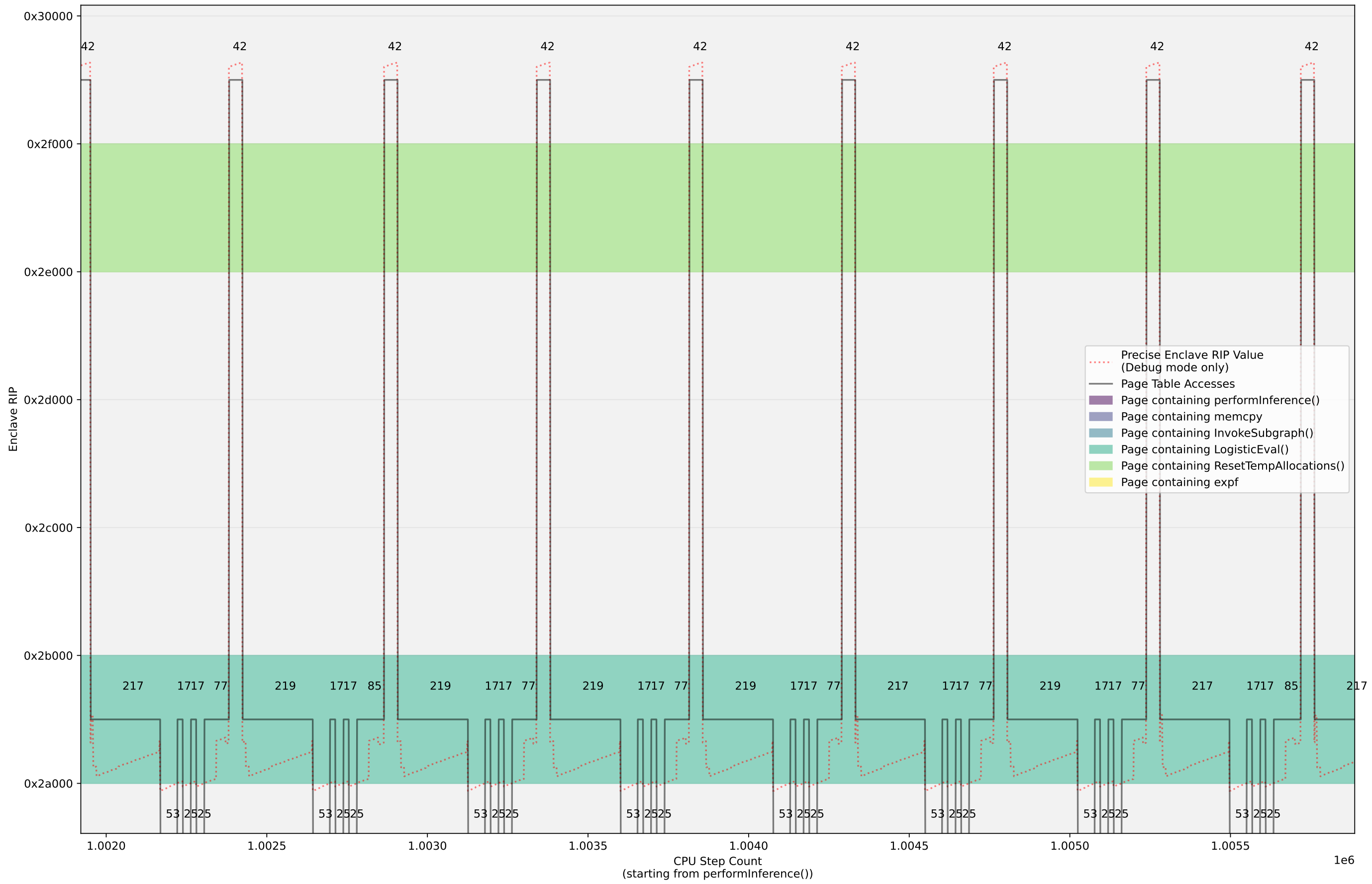

Fig. 8. This is an inset of the region of the trace that corresponds to the 128 neurons in the first layer. Note the largely but not entirely constant pattern of steps in the blue-green page at the bottom suggesting possible non-constant time behavior; the first and last values in each "valley" are either 217 or 219 and 77 or 85; everything else appears constant. This difference between activations is echoed in the subtly different RIP patterns.