UNIVERSITY OF CALIFORNIA, MERCED

Towards Robust Heterogeneous Computing in Diverse-Scale Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Mujahid Al Rafi

Committee in charge:

Professor Hyeran Jeon, Chair
Professor Shawn Newsam
Professor Meng Tang
Professor Fan Yao

2026

The dissertation of Mujahid Al Rafi is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

(Professor Shawn Newsam)

(Professor Meng Tang)

(Professor Fan Yao)

(Professor Hyeran Jeon, Chair)

University of California, Merced

2026

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

VITA

2018	Bachelor of Science, Bangladesh University of Engineering and Technology
2018–2021	Software Engineer, CodeCrafters International
2021–Present	Doctor of Philosophy, University of California, Merced
2021–Present	Graduate Student Researcher, University of California, Merced
2025–Present	PhD Intern, Pacific Northwest National Laboratory

ABSTRACT OF THE DISSERTATION

Towards Robust Heterogeneous Computing in Diverse-Scale Systems

by

Mujahid Al Rafi

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Merced, 2026

Professor Hyeran Jeon, Chair

Modern computing systems increasingly rely on heterogeneous architectures that integrate general-purpose processors with specialized accelerators such as GPUs and deep learning engines. To meet the performance and energy demands of modern workloads, these systems depend heavily on optimization techniques including speculative execution, transfer learning, hardware acceleration, and dynamic power management. While these mechanisms substantially improve efficiency, they also introduce new forms of shared state, resource contention, and externally observable behavior, thereby expanding the attack surface. As a result, security vulnerabilities in modern platforms often emerge not from a single component, but from interactions across multiple layers of the computing stack.

This dissertation investigates such cross-layer vulnerabilities in heterogeneous computing systems and develops new insights into how performance-oriented design choices can unintentionally expose sensitive information. It presents three case studies spanning machine learning systems, edge GPU platforms, and CPU microarchitecture.

First, this dissertation presents Decepticon, a model extraction attack on large-scale transfer-learned models. Decepticon exploits execution fingerprints inherited from shared pre-trained models to identify the underlying model used by a black-box victim and significantly reduce the effort required to reconstruct a high-fidelity clone. The results demonstrate that even large transformer-based models remain vulnerable when transfer learning and GPU execution characteristics are jointly exploited.

Second, this dissertation investigates the power and frequency behavior of edge GPU platforms and their security implications. Through detailed characterization of instruction-level and application-level telemetry on commercial NVIDIA Jetson devices, this work demonstrates that GPU and deep learning accelerator behavior can be leveraged to construct covert channels. It further evaluates mitigation strategies based on telemetry access restriction, resolution reduction, and perturbation, highlighting the trade-offs between observability and security.

Third, this dissertation introduces SCPC (Securing Cross-Process Collision-Based Transient Attacks), a low-overhead defense against cross-process collision-based transient execution attacks in modern CPUs. SCPC protects speculative execution structures by selectively virtualizing vulnerable predictor state while preserving much of the performance benefit of shared hardware resources. Evaluation shows that the mechanism significantly improves security with minimal performance overhead and can be generalized to other speculative structures.

Together, these contributions show that information leakage in modern heterogeneous systems is often a byproduct of the same mechanisms that enable high performance. This dissertation argues that securing future computing platforms requires a cross-layer perspective in which machine learning frameworks, accelerator behavior, and processor microarchitecture are analyzed jointly rather than in isolation. By exposing these vulnerabilities and exploring

corresponding defenses, this work advances the design of more secure and robust heterogeneous computing systems.

Chapter 1

Introduction

Modern computing systems are rapidly evolving toward heterogeneous architectures that combine general-purpose CPUs with specialized accelerators such as GPUs and deep learning engines. This evolution is driven by the increasing computational demands of emerging workloads, including artificial intelligence, autonomous systems, and large-scale data analytics. To meet these demands, system designers rely heavily on performance optimization techniques such as speculative execution, hardware acceleration, power management, and transfer learning.

While these optimizations are essential for achieving high performance and energy efficiency, they fundamentally reshape how hardware and software interact. In particular, they introduce new forms of shared state, resource contention, and observable side effects, which unintentionally create opportunities for information leakage. As a result, modern systems are increasingly vulnerable to cross-layer security threats that emerge not from a single component, but from the interaction between multiple layers of the computing stack.

Recent research has shown that microarchitectural side channels can leak sensitive information through shared hardware resources such as caches, predictors, and memory subsystems. At the same time, machine learning systems introduce new attack surfaces due to their unique execution patterns, large model sizes, and reliance on shared pre-trained models. Furthermore, edge computing platforms introduce additional security challenges because they operate in physically exposed and potentially untrusted environments while handling privacy-sensitive

data. Together, these trends indicate that security vulnerabilities in modern systems are no longer confined to a single layer, but instead emerge from the interaction between performance optimizations and shared resources across the entire computing stack.

This dissertation argues that performance-driven design decisions across heterogeneous computing systems introduce systematic information leakage channels, and that securing modern systems requires a cross-layer understanding of these vulnerabilities. To support this claim, this dissertation presents three works across different layers of the computing stack: machine learning systems, GPU-based edge accelerators, and CPU microarchitecture.

First, Chapter 2 of this dissertation presents Decepticon, the first model extraction attack targeting large-scale transfer-learned models such as Transformers. Unlike prior works that focus on CNN architectures or rely on fine-grained memory probing, Decepticon introduces a novel execution fingerprinting technique that uniquely identifies the pre-trained model underlying a black-box victim. By leveraging the observation that execution fingerprints are preserved across transfer learning, Decepticon reduces the search space of model extraction and enables practical large-scale weight recovery, which was previously considered infeasible.

Second, Chapter 3 of this dissertation demonstrates that power and frequency behaviors of edge GPU and Deep Learning Accelerator (DLA) platforms introduce previously unexplored covert channels. To the best of our knowledge, this is the first work to expose covert channel vulnerabilities in NVIDIA DLAs and the first to construct covert channels using frequency utilization statistics on mobile GPUs. Through systematic characterization of instruction-level and application-level power and frequency behaviors, this work demonstrates that passive monitoring of hardware telemetry can enable stealthy communication channels, highlighting previously unexplored security risks in edge AI systems.

Third, Chapter 4 of this dissertation addresses security vulnerabilities in modern CPUs caused by speculative execution. It introduces SCPC (Securing Cross-Process Collision-Based Transient Attacks), a defense mechanism against cross-process transient execution attacks. Unlike prior approaches that rely on static partitioning or complete isolation, SCPC introduces

selective virtualization of speculative state, enabling secure sharing of hardware resources without sacrificing performance benefits. This design achieves strong security guarantees while maintaining low overhead, addressing a key limitation of existing mitigation techniques.

Finally, Chapter 5 outlines ongoing and future research directions that extend the contributions of this dissertation. In particular, I am now investigating the security implications of emerging disaggregated memory systems. Our preliminary results suggest that CXL-attached disaggregated memory can amplify transient and cache timing attacks.

Together, these works demonstrate that information leakage in modern computing systems is fundamentally a byproduct of performance optimization and resource sharing, and that these vulnerabilities manifest across multiple layers, from machine learning frameworks to hardware accelerators and processor microarchitecture. This dissertation therefore advocates for a cross-layer approach to secure heterogeneous computing systems, where security mechanisms are designed with awareness of performance optimizations and shared hardware behaviors.

Chapter 2

Decepticon: Attacking Secrets of Transformers

2.1 Introduction

Machine learning (ML) models are widely used for almost all computing solutions, including smart homes, autopilot programs of self-driving vehicles, and so on. In many applications that handle critical data for the users' safety or privacy, any perturbed predictions lead to tragic consequences. Such an ML prediction perturbation is one of the major targets of cyber attacks, which is enabled by stealing important parameters of the target ML model. Recently, a few studies demonstrated that it is feasible to steal ML model topology and hyperparameters through various side channels (e.g., performance counters, cache access timing, etc.) [74, 118, 161]. Most existing attacks target convolutional neural network (CNN) models. While the model topology and hyperparameters provide important hints about the victim model, knowing the weight values shifts the security surface to another level by enabling more advanced model extractions, including constructing local (clone) models with extremely high fidelity (i.e., compromising model privacy) and empowering adversarial inputs attacks (i.e., tampering model integrity).

Stealing model weights is extremely challenging for two main reasons. First, modern ML workloads are very different from traditional security-sensitive applications (e.g., crypto programs) in that there is no explicit secret dependent control/data flows, making it almost

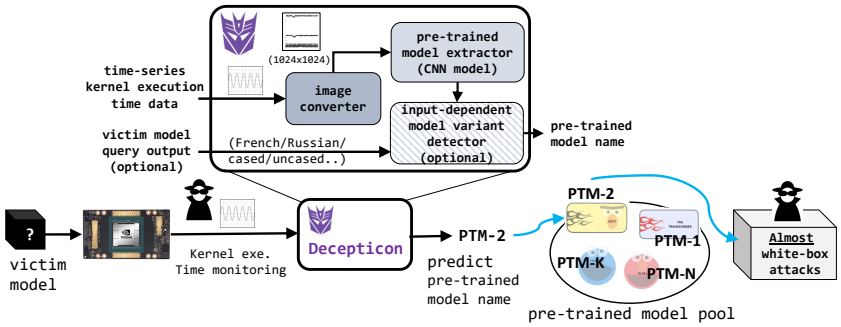


Figure 2.1. Decepticon Architecture

impractical to accurately reveal model weights using a conventional side channel (e.g., via caches). Second, even if stealing can be done in real systems, state-of-the-art ML models are very large with tremendous amounts of weights. Any such attack will require model stealing at an extremely large scale. Such challenge is further exacerbated in today's gigantic transformer models that come with billions/trillions of weights (e.g., BERT [61], GPT [124, 51], Llama 2 [145]). As a result, pure side channel-based model stealing in such use cases is unlikely to be practical. For instance, very recent work in [126] repurposed row-hammer as a side channel to exfiltrate bit-level information for model weights. However, to reveal part of the weight, thousands of rounds of row-hammer are needed. Such a method may not scale to today's large-scale models. Then, can we conclude that large-scale (billion- to trillion-parameter) models are secure enough?

In this work, we present the first investigation of model extraction attacks on large-scale models. We first show an in-depth characterization of large models (240 models downloaded from popular model zoos [25, 13, 7, 18, 19]) and propose a novel two-level model extraction attack, namely *Decepticon*. Unlike existing approaches that run rigorous memory probings directly on every single bit of individual weights, we show that an indirect (two-level) approach makes large-scale model extraction practical. The two-level approach especially leverages the unique characteristics of transfer learning that most large models use.

Transfer learning enables large-scale model development with significantly reduced training time. With transfer learning, individual developers and research teams can develop

their own models by fine-tuning a pre-trained model with task-specific datasets. The pre-trained models are either publicly shared through popular model zoos [13, 16, 7] or privately shared within individual companies or institutions. In any scale, pre-trained models are purposely shared by multi to many ML developers. For example, some BERT-base and GPT-2 pre-trained models on Huggingface model repositories have been downloaded more than 10 million and 20 million times, respectively [8, 11]. Security and privacy-sensitive domains such as medical and defense applications also share their pre-trained models with limited accesses [128, 115, 6, 152]. We focus on this unique training process of large-scale models.

What if an adversary grabs an access to a pre-trained model? Can he/she derive the secrets of a black-box fine-tuned model with the pre-trained model? According to our experiments, pre-trained models provide ample secrets of the fine-tuned model including very similar weight values. But, **can we identify the pre-trained model?** Our exhaustive investigation shows that individual ML models have unique execution fingerprints. Unlike existing studies have assumed [74], the fingerprints are substantially different across models even when the models use the same architecture, dataset, and task. More importantly, the fingerprint is inherited from a pre-trained model to its fine-tuned models. Thus, the fingerprint can be leveraged to identify the pre-trained model.

With these observations, *Decepticon* first identifies the pre-trained model with the fingerprint of the blackbox model's execution on off-the-shelf GPUs. Then, it extracts the entire model weights based on the pre-trained model's weight values. Decepticon uses two design methods to increase the attack success rate with minimal extraction effort; 1) selective weight extraction that exploits the diverse impact of individual weights and layers towards the final prediction output and 2) model fingerprint classification with CNNs that is inherently noise tolerant. The clone model that is created by the extracted model's secrets is used for an adversarial attack to perturb the victim model's prediction. We also show that Decepticon is applicable for any ML models that use transfer learning by using a CNN model example (Section 2.7.7). Figure 2.1 shows the end-to-end attack scenario of Decepticon.

Our contributions are:

1. To our best knowledge, this is the first study that demonstrates model extraction attack (including entire weight extraction) for large-scale models (e.g., Transformers). The proposed attack is applicable for any ML models that use transfer learning.
2. We provide in-depth characterization of transfer-learned models.
3. We apply a noise-tolerant image classification algorithm for model architecture extraction and introduce a selective weight extraction.
4. We demonstrate the impact of weight extraction through a clone model construction and an adversarial attack evaluation.

2.2 Background

2.2.1 Model Extraction Attacks

Model extraction attack is a security concern where an attacker aims to recover the secrets of ML models and reconstruct a copy of a victim model. The architecture and parameters of a victim model are the main targets of these attacks. The architecture of a model includes the number, type, and dimension of individual layers of the target ML model. The parameters include hyper-parameters, weights, and biases. Model extraction attack is a threat to the intellectual property of the ML solution providers. Moreover, the detailed knowledge about the victim model helps the attacker craft adversarial examples to fool the victim model or even extract sensitive training data. Several studies demonstrated various model extraction attacks [74, 172, 109, 161, 118, 146, 162, 131, 126, 167]. These studies leveraged various leakage vectors through EM side-channels, PCI-e bus snooping and message trapping, cache timings, and memory probing.

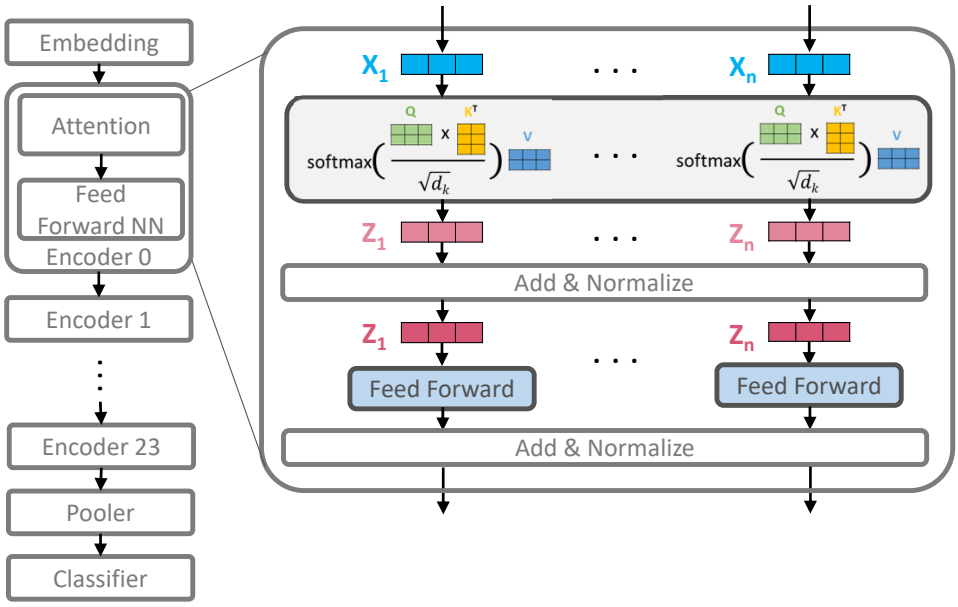


Figure 2.2. BERT-large Architecture

2.2.2 Transformer Models

Though any models that use transfer learning can be a target of Decepticon, we explain our attack scenario mainly with Transformer models. Transformer models use transfer learning as the default training method. Transformers have proven their effectiveness in various domains including natural language processing and computer vision [149]. Transformers are efficient to be adapted to several diverse tasks such as question answering, sentiment analysis, etc. with a task-specific last layer. A Transformer model runs multiple rounds of attention computations that check relations across all tokens of the given input in parallel. Transformer models consist of either *encoder* or *decoder* (or both). Each *encoder* is comprised of two layers: self-attention and feed-forward, as illustrated in Fig. 2.2. In a self-attention layer, input tokens are multiplied with three weight matrixes; Key, Query, and Value and generate K, Q, and V vectors. These vectors for each token are used for understanding the importance of the token in the input context. Decoders are similar to encoders, except the masked self-attention. Popular Transformer models are BERT[61], RoBERTa[105], ALBERT[92], and GPT-2[124]. These reference models use a fixed number of encoders/decoders such as 12 (BERT-base) and 24 (BERT-large).

2.3 Threat Model

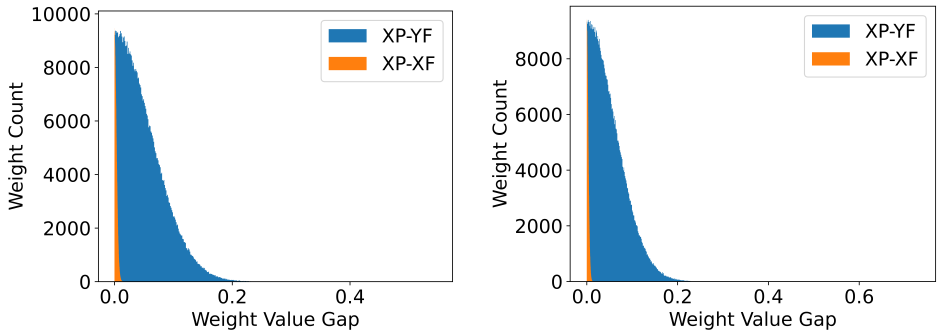
The goal of our threat model is to steal the secrets of a black-box large model. The secrets are used for creating a clone model, which can be used for an adversarial attack. The victim model is assumed to be developed through transfer learning. The attacker is assumed to have no information about the victim model, while he/she can 1) collect architectural hints such as GPU kernel execution time and memory addresses and 2) query the victim model and check the prediction outputs. Note that various studies demonstrated that kernel execution time and memory addresses can be monitored through EM-side channels and bus probing on the interconnects between CPU and GPU [53, 74, 118, 161]. We leverage the existing side-channels but exploit the unique/novel characteristics of transfer-learned models for a novel model extraction attack. We assume that the attacker has a pool of candidate pre-trained models. The models can be either collected from public repositories or from internal spy routes if the victim model is designed with private pre-trained models. But, the attacker does not know which one was used for fine-tuning the victim model.

2.4 Transformer Model Vulnerabilities

We tested 70 pre-trained and 170 fine-tuned models downloaded from various model repositories on an NVIDIA GeForce RTX 3050 (Ampere) GPU with CUDA v11.8. We used Python v3.8, PyTorch v1.11.0, and TensorFlow v2.8.0 for evaluations. More details can be found from Section 2.7.

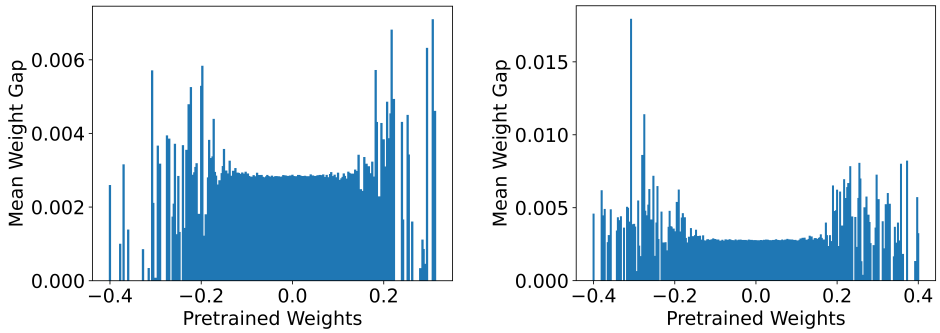
2.4.1 Pre-trained vs. Fine-tuned Models

Amount of weight updates during fine-tuning: To understand the relation between a pre-trained model and its fine-tuned model, we checked the significance of weight updates. We measured the absolute weight value gap for each pair of a pre-trained model and a fine-tuned model that use the same model architecture; weights on the same location of the two models were



(a) Attention Self Key **(b)** Attention Self Query

Figure 2.3. Weight Value Gap Distribution



(a) Attention Self Key **(b)** Attention Self Query

Figure 2.4. Impact of Weight Value to Amount of Updates

compared. Fig. 2.3 shows example results where the graphs with (XP-XF) are the comparisons between a pre-trained model and its fine-tuned model and those with (XP-YF) are between a pre-trained model and a fine-tuned model that used different pre-trained model. Note that there was not a pair of pre-trained and fine-tuned models designed for the same task.

In all cases, the distribution follows long tail. For (XP-XF) cases, the majority of weights encounter less than ± 0.01 value distance (thus looks like a sharp vertical line in the graph). Almost 50% weights show less than ± 0.002 distance. Any pair of pre-trained and its downstream models in our tested models showed similarly small weight gap. The weight value ranges of the tested models were at least 1.74 up to over 26.3.

On the other hand, (XP-YF) cases show at least $20\times$ higher gap. While most weights

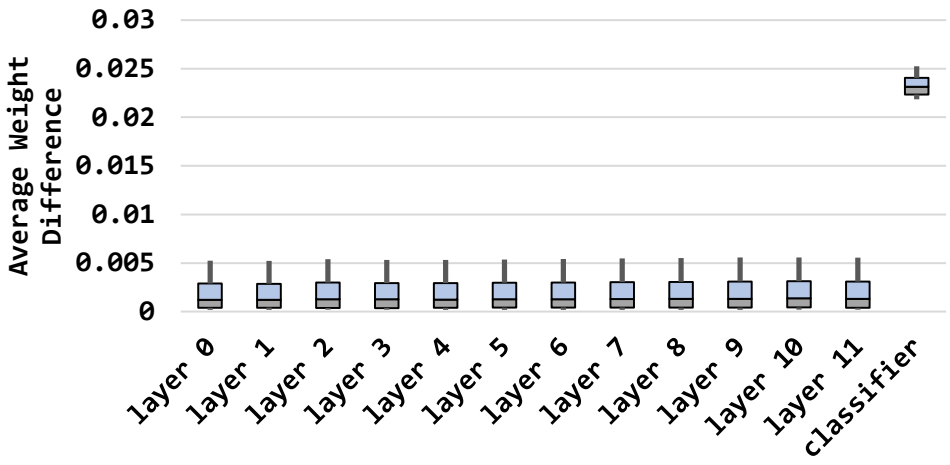


Figure 2.5. Avg. Weight Differences of 9 BERT-base Models Trained for Different Tasks

encounter less than ± 0.2 distance, the difference stretches up to over ± 0.6. Only less than 3% weights are within ± 0.002 distance.

Impact of weight values on updates: To understand the impact of weight values for the update amount, we measured the average value gap per pre-trained model weight value range. Fig. 2.4 shows example results of the models of (XP-XF) cases in Fig. 2.3. X-axis is the pre-trained weight value and Y-axis is the update amount during fine-tuning. Most of the layers show U-shape graph, which means that the weights further from zero encounter over $3\times$ higher weight changes than those closer to zero. Relating to the weight value gap result, the outliers such as outermost 10% weights (outside the ± 0.25 boundary in Fig. 2.4) are the sources of the long tail region in Fig. 2.3.

Impact of downstream tasks on updates: As a pre-trained model can be fine-tuned for different tasks, we also evaluated if we can still observe weight similarities if different task models share the same pre-trained model. We downloaded a BERT-base pre-trained model, fine-tuned it for nine tasks by using Huggingface GLUE benchmark [12], and compared the weight values among the nine models. As shown in Fig. 2.5, all layers except for the task-dependent last layer show almost zero distances (< 0.002 on average). This shows that the main updates are made in the last layer during the fine-tuning and the remaining layers might be very similar across fine-tuned models.

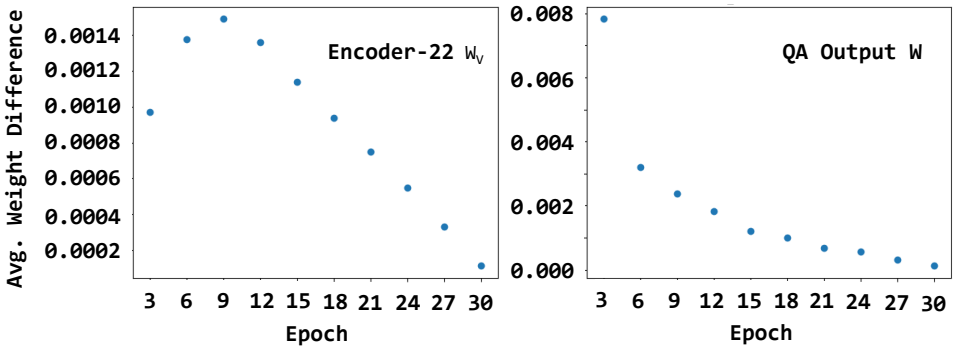


Figure 2.6. Avg. Weight Updates during Fine-tuning

Impact of learning rates and epochs in fine-tuning: Fine-tuning typically uses small learning rates (and weight decay) and a limited number of epochs (up to three epochs typically), which may lead to little change in weight values during fine-tuning. To understand the impact of the short epochs, we tested the weight changes along longer fine-tuning epochs. Fig. 2.6 shows the average weight value changes of a layer (encoder 22) and the task-specific last layer of a BERT-large pre-trained model during 30 epochs of fine-tuning. Until epoch 9, the average weight value gap between epochs increases up to 0.0015 and then drops linearly to below 0.0002 at epoch 30. In the meantime, the output layer's weight value gap saturates exponentially as plotted in the second graph, which means that the fine-tuning converges well. This result shows that the weight gap may be increased with longer epochs but it is not significant (quickly saturated).

The theoretical reason of using small learning rates and epochs in fine-tuning is to prevent *catastrophic forgetting* [111, 87], which is a known phenomenon for neural networks that forget what they have previously learned given a new task or new data distribution. Catastrophic forgetting is one of the risks in the context of foundation models [50], particularly when the models are adapted to a new task with small data (e.g., few-shot learning). Therefore, the weight value similarity is not sourced from premature fine-tuning but is an unavoidable side-effect of transfer learning.

Contribution of layers towards model prediction: Inspired by the small weight value change during fine-tuning, we examined if a pre-trained model's weights can be reused

Table 2.1. Downstream Task Accuracy with Frozen Layers

Number of Frozen Layers	Accuracy (%)
0	80.1325
1	80.2271
2	79.1485
3	76.5279
4	71.9111
5	68.543
6	64.579

without hurting fine-tuning prediction output. For a BERT-base model [9] fine-tuned for question answering task, we tested downstream task accuracy by replacing the first a few layers with the pre-trained model's weights. Table 2.1 shows the results when replacing up to sixth layers. When freezing the first 2-3 layers, the fine-tuned model experiences only 1-3% accuracy drop.

Summary: Here are our observations.

Observation 1. Fine-tuned models have meaningfully close weight values to their baseline pre-trained models, which is at least 20× closer than with other pre-trained models.

Observation 2. Few outlier weights change by over 3× more and are the sources of the long tail weight gap distribution.

Observation 3. Fine-tuned models show very similar weights for all layers except for the last layer if they use the same pre-trained model even when fine-tuned for different tasks.

Observation 4. The impact of layers towards the downstream task's accuracy is different. The first layers may reuse pre-trained model's weights without hurting accuracy.

Decepticon leverages these for extracting secrets of a blackbox victim model from its pre-trained model.

2.4.2 Model Execution Fingerprints

To extract the weights of a victim model by leveraging the weight value similarity, it is required to locate the pre-trained model from the victim black-box model. To identify the pre-trained model, there should be distinguishable model signatures. We tested if models have unique signatures.

Model signature in architecture hints: We compared kernel execution times of the

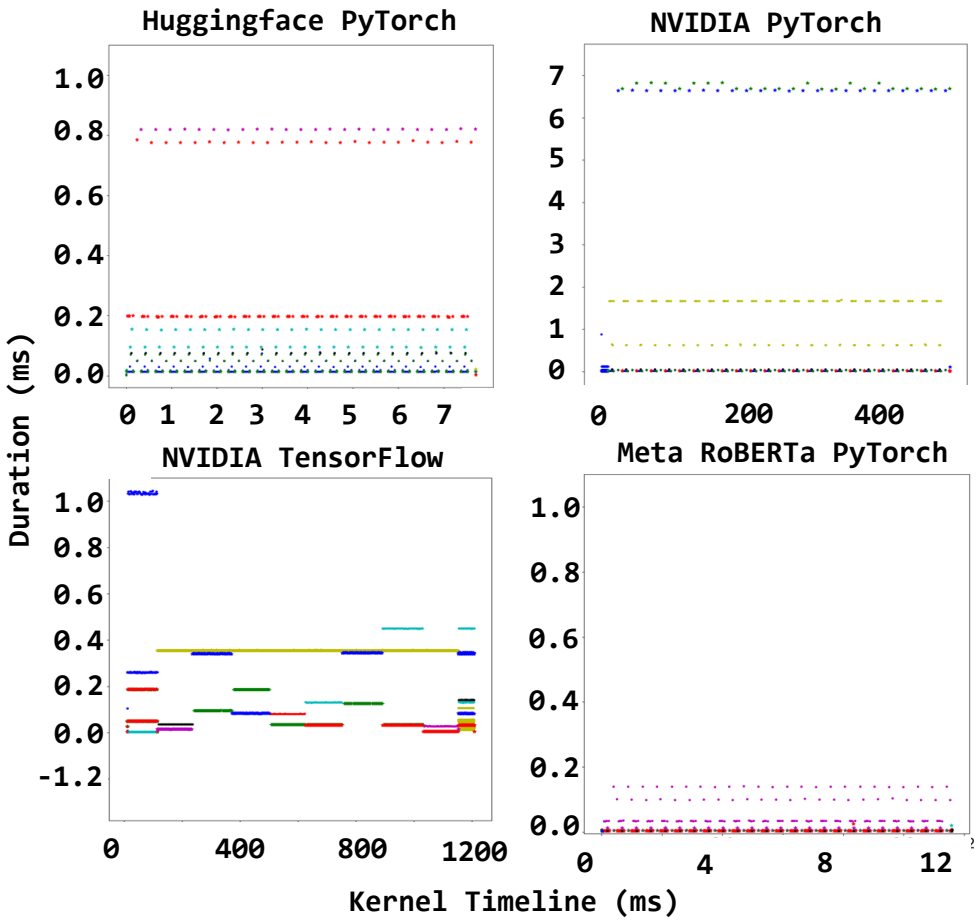


Figure 2.7. Diversity in Time-series Kernel Execution Times

models having the same architecture. Fig. 2.7 shows the example results of different versions of BERT-large models measured on the same GPU with the same inputs. Each dot indicates the execution time of a kernel and the same-colored dots are the multiple invocations of the same kernel. Each BERT-large model is fine-tuned and released by different repository/developer as stated. Note that RoBERTa uses the same architecture with BERT. Interestingly, we couldn't find common patterns among them. On the other hand, when we tested the models released by the same repository/developer, the statistics showed high consistency even when they were fine-tuned for different tasks, as shown in Fig. 2.8.

Impact of algorithms and software interfaces: We observed that such model fingerprint is highly influenced by the algorithms and software interfaces (e.g., framework), which

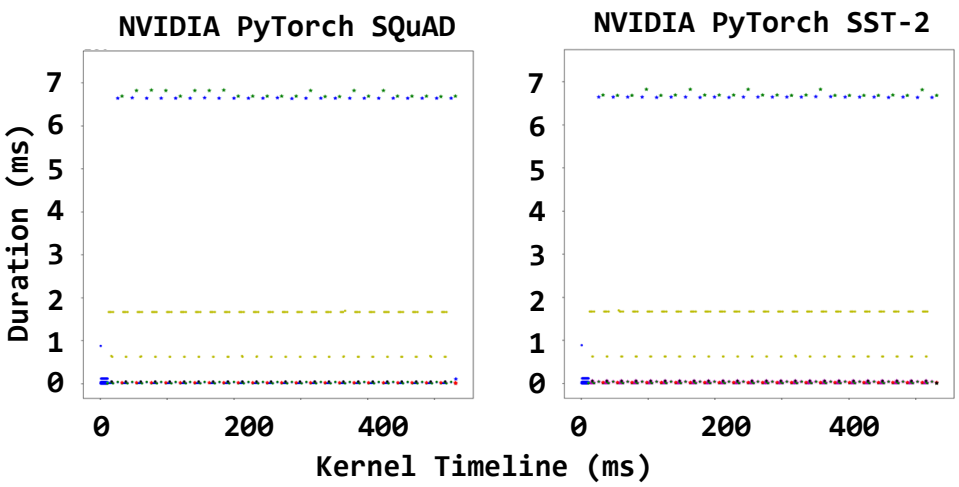


Figure 2.8. Consistency in Time-series Kernel Execution Times

Table 2.2. Impact of Model Fingerprint for CNN Layer Detection Accuracy

Models	Error Rate (LER)	Kernel Seq. Length	# of Unique Kernels
DeepSniffer Original Results [74]	0.091	222	16
DeepSniffer Pytorch Model [2]	0.567	256	16
Nvidia PyTorch Model [23]	2.628	1235	38
Google Tensorflow Model [29]	6.274	3399	50
Amazon Mxnet Model [17]	6.768	2652	59

lead to different GPU kernel selections. Fig. 2.9 shows an example list of kernels executed by BERT-large models of different sources. Though the same model architecture is executed, only a handful of kernels out of hundreds are commonly used across the models. We found that the framework is one of the main contributors that make the differences. TensorFlow models run up to 8× more kernel executions and use almost 40× more unique kernels than PyTorch models. Also, TensorFlow models tend to use their GPU backends, while PyTorch models use more GPU library functions. Developer-specific kernel preferences were also observed. Though different frameworks were used, NVIDIA models were commonly optimized to leverage their Tensor Core by running functions using half-precision data types. On the other hand, Meta models tend to run many short kernels such as reduction operations and hence the statistics had crowded kernel executions on the bottom of the graph as shown in Fig. 2.7. Similarly, we observed unique signatures from most of the 70 pre-trained models used in our evaluation (Section 2.7). Fine-tuned models showed very similar signatures with their pre-trained models.

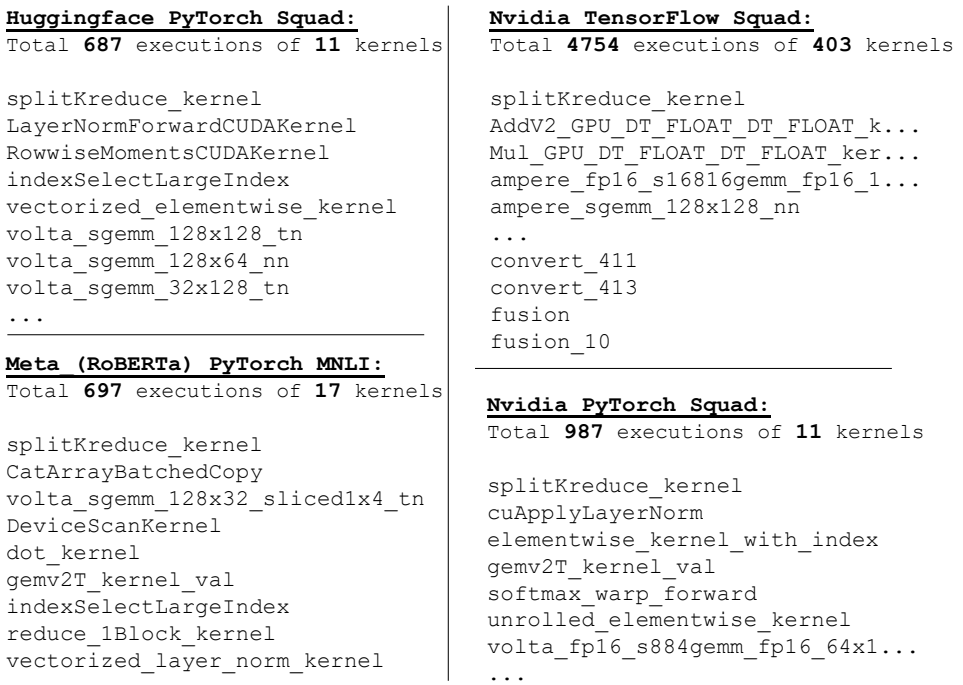


Figure 2.9. Kernels Executed by BERT-large Models

Existing model extraction attacks do not work due to fingerprint: We also observed the model fingerprints from non-Transformer models but existing model extraction attack studies ignored the impact. When we fed a state-of-the-art CNN extraction framework [74] with statistics of CNN models that were developed by different sources, the layer prediction accuracy dropped significantly, as shown in Table 2.2. LER means how many layer sequences are incorrectly predicted per layer [74]. Thus, the prediction results with LER over 1 are not countable. We instead leverage the fingerprint for locating pre-trained model.

Model signature in query outputs: Though most of the models are recognizable with the unique model fingerprints, we found that there are cases that can't be distinguished through architectural hints. For example, in large language models (e.g., Transformers), 1) models that are trained with different languages (CamemBERT [110] and RuBERT [168] are French and Russian versions of BERT), 2) models that are trained with different training datasets (RoBERTa is trained with more datasets than BERT), and 3) models that are trained differently (e.g., cased vs. uncased) may not be distinguished if they are released by the same source

and use the same architecture. For these cases, we found that query outputs can be used as a secondary fingerprint. For example, the dataset differences in RoBERTa and BERT can be checked with their vocabulary file (vocab.txt or vocab.json). When BERT was tested with queries including {`debugging, capitalize, cloves, indignation, hijab, selfies, misogynist, acupuncture`}, the predictions were incorrect while RoBERTa performed well.

Summary: Here are our observations.

Observation 1. Models have unique execution fingerprint even when using the same model architecture. The fingerprint is inherited from a pre-trained model to its fine-tuned models.

Observation 2. Query outputs can be used as a secondary model fingerprint if architecture hint is not distinguishable.

Decepticon leverages these for finding the pre-trained model of the victim model.

2.5 Decepticon Design

We propose *Decepticon*, a new model extraction attack that exploits weight similarity and model fingerprints.

2.5.1 Decepticon Architecture

Fig. 2.1 shows the architecture of Decepticon. Decepticon uses one *architectural hint* (time-series kernel execution time) and *model query outputs* as leakage vectors. The *pre-trained model extractor* receives the inputs in 2-dimensional image formats and finds the most matching pre-trained model out of the candidate models with an image recognition algorithm. For some models that show very similar model fingerprint, query outputs are used to improve the prediction accuracy in the *input-dependent model variant detector*. The output of Decepticon is the *pre-trained model name*. Once the pre-trained model is revealed, the attacker can try a variety of gray (or almost white) box attacks such as weight extraction and adversarial attacks.

2.5.2 Architectural Hints

Out of various side-channels (Section 4.2), we observed that time-series GPU kernel execution time reflect model fingerprints well. A model runs hundreds to thousands of kernels. Thus, individual kernel execution time does not reflect the model identity. Instead, we use a time-series kernel execution time, which shows the invocation timing and duration of all kernels during the model inference time. Thus, the attacker collects ($T_{invocation}$, $T_{termination}$) of all kernels, where T is timestamp.

2.5.3 Query Outputs

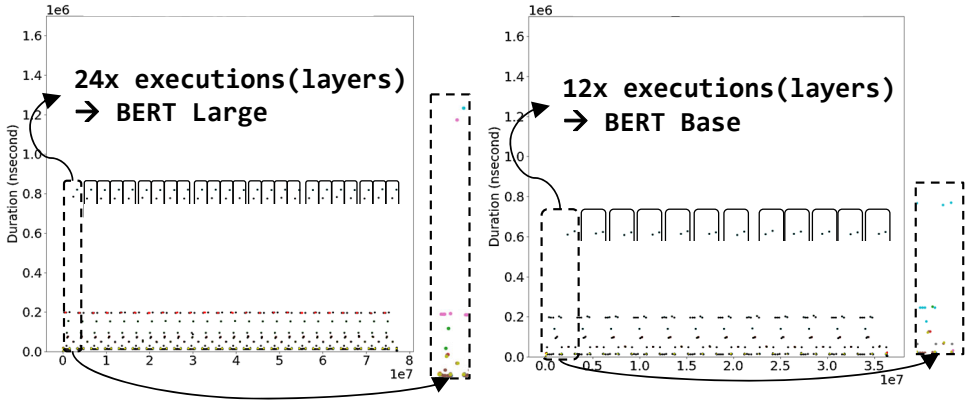
As noted in Section 2.4.2, if there are multiple models having similar architecture hints, the attacker uses a special set of query inputs as the secondary fingerprint. By targeting large language models (e.g., BERT-uncased/cased, CamemBERT, RuBERT, RoBERTa, etc.), the set is compiled with several queries in different languages, special vocabularies that each candidate is uniquely trained with (by using the vocabulary file released with the pre-trained model or through exhaustive testing), and special words that have different meanings in upper/lower characters (e.g., Apple as a company name vs. apple as a fruit name). As attacker knows the differences of candidate pre-trained models, he/she can compile a query list with such knowledge. Note that the attacker is only aware of the pre-trained models in his/her model pool, not the downstream training datasets or architecture of the victim fine-tuned model.

2.5.4 Pre-trained Model Extractor

Model Fingerprint Recognition

The shapes of the model fingerprint are different for different types of ML models (e.g., CNN and Transformers use totally different architecture). Pre-trained model extractor uses a proper pattern recognition algorithm for the target model architecture. We describe Transformer model extraction.

Transformer models run identically-shaped encoders or decoders repeatedly (Section 4.2).



(a) BERT-large (b) BERT-base

Figure 2.10. Layer Boundary Identification

Thus, architecture hints also show repetitive patterns. Therefore, a group of repeated measurements can be considered as one layer. However, the detection of the repeating groups is challenging because the number of layers, the volume of per-layer computations, and the intra-layer structure are pretty diverse in different Transformer models. Thus, the pre-trained model extractor for Transformer is designed to recognize diverse patterns of repetition from time-series kernel execution data. Fig. 2.10 shows example time-series kernel execution graphs where the group of kernels in the box is repeatedly executed. Though the shape and size of the group in the two models are very different, it is clearly seen that the repeating count matches the number of layers of each model (BERT-base has 12 group executions and BERT-large has 24). We observed similar patterns from most of the tested Transformers, while the shape of each group is pretty diverse due to model signatures.

Along with the number of layers, the size of layers is also an important parameter that determines different Transformer architectures. For example, DeBERTa-xsmall [73] uses 12 layers with 384 hidden states. GPT-2 [124] also uses 12 decoders but includes 768 hidden states. Due to the different number of hidden states, the layer size of the two models is notably different. The model extractor recognizes the layer size through the peak execution time in each kernel group. In Fig. 2.10, BERT-base's peak kernel execution time is around 0.6*ms* while BERT-large's is around 0.8*ms* because BERT-base uses 768 hidden states while BERT-large uses 1024 states.

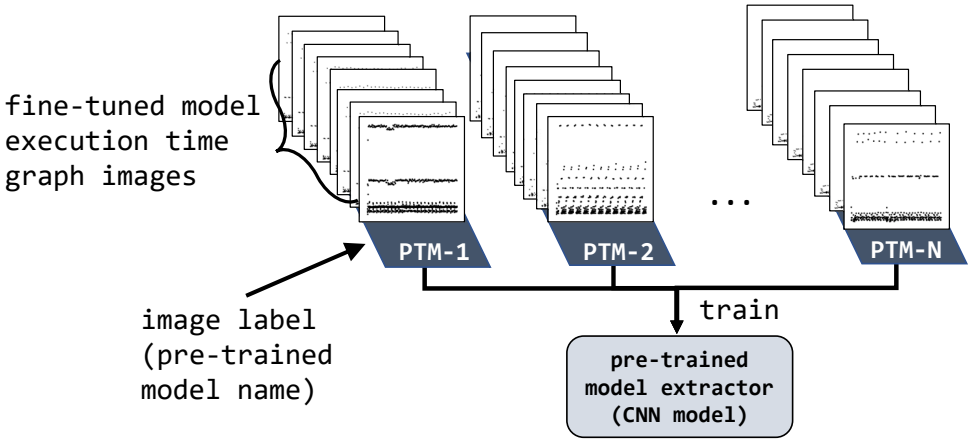


Figure 2.11. CNN Model Training

Model Extraction through Image Recognition

As the shape and count of kernel groups vary from one model to another, it is undesirable to manually detect the model fingerprint. Therefore, the pre-trained model extractor internally runs an automated detection method.

To automate the attack process, there are two challenges: the automation needs to 1) detect kernel groups having different size and shape and 2) process 2-dimensional information of time-series kernel execution data (($T_{invocation}$, $T_{termination}$) as noted in Section 2.5.2). We found that this problem can be reduced to a pattern recognition problem for 2-dimensional inputs, which is similar to *image recognition*. The repetitive kernel groups are the target patterns to detect and the time-series kernel execution graph can be considered as an input image. We chose to use a CNN model that can detect patterns from images efficiently.

Architecture Hint Data Conversion: To apply CNN on the kernel execution time data, we convert the data to 2-dimensional images. We first plot the time-series kernel execution graphs with the same x- and y-scales (square shape) and convert them into images. Then, all other information (x- and y-axis, numbers, and texts) except for the execution patterns are removed. The images are then converted to gray-scale to remove any color bias. Finally, the images are re-shaped to 1024x1024 equal-sized images to be fed to the same CNN model. The example images are shown in Fig. 2.11. We collected 1787 images from 70 pre-trained and 170 fine-tuned

models (Section 2.7). 80% of them were used for training and 20% for testing. As the goal of the detection is to recognize the pre-trained model, we labeled each graph image with each model's pre-trained model name. For example, a MobileBERT fine-tuned model's image was labeled as *google_mobilebert-uncased*.

CNN Model Training: The CNN model receives a kernel execution time image of a fine-tuned model and predicts the pre-trained model name. We explored various CNN architectures and finalized the CNN model with seven hidden layers: two convolution and pooling layers interleaving each other and followed by three fully connected layers - conv (in: 3, out: 6, kernel: 5×5), pool (kernel: 8×8, stride: 8), conv (in: 6, out: 16, kernel: 5×5), pool (kernel: 8×8, stride: 8), fc (in: 3600, out: 120), fc (in: 120, out: 84), fc (in: 84). We used PyTorch for training with learning rate and epochs as 0.001 and 10 respectively. The ground truth of each prediction is the pre-trained model name of the input image. If multiple pre-trained models are high-ranked in the CNN classification, we locate one through the query outputs (Section 2.5.3).

Handling Corner Cases

There are some models that have indistinguishable layer boundaries, mainly due to some optimizations (e.g., NVIDIA TensorFlow in Fig. 2.7 shows irregular execution pattern due to such optimizations (e.g., XLA [32])). Fig. 2.12 shows three example execution patterns of BERT-large models that a simple layer boundary detection can't be used. For these cases, layers can be detected in the regions excluding the gray-colored regions (marked as XLA region and output layer). XLA optimization runs compiler optimization operations in the middle of the inference and the executions for encoders are at the beginning and the end of the inference. In both encoder regions, we can find 24 repetitive kernel group executions. For these models, we do pre-processing to find the encoder regions and feed the CNN model with the image of the encoder regions.

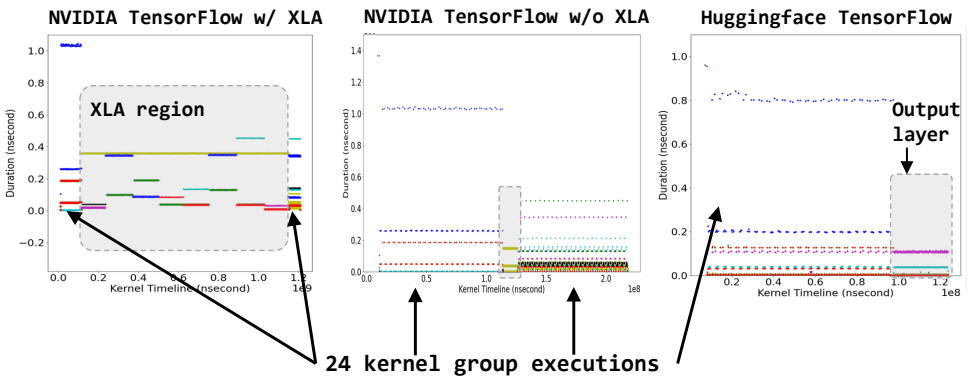


Figure 2.12. Irregular Execution Patterns

2.6 Gray/White-box Attacks with Decepticon

Decepticon enables various gray-box attacks. We explore two attack scenarios 1) stealing intellectual property by creating a clone of the victim model and 2) perturbing victim model's prediction through adversarial attack.

2.6.1 Stealing Proprietary Fine-tuned Model

Once the pre-trained model is recovered, we can use the weights of the pre-trained model for extracting the victim model weights. The weights of the pre-trained model and the fine-tuned model are very similar but not identical. According to our experiments (Fig. 2.17), the pre-trained model itself cannot be used for downstream tasks, unless the attacker has the fine-tuning datasets. In our threat model, we do not assume to have fine-tuning datasets because fine-tuning is normally done with multiple in-house training datasets. Thus, the attacker needs to further reduce the weight value gap to clone the victim model.

We leverage an existing side-channel (row-hammer-based bit value checking [126]) to recover the actual weight value (weight address is collected (Section 2.3)). Unlike the existing study that had to check every weight value bits [126], we use the recovered pre-trained model as a baseline and check only the essential bits that are likely to be updated during fine-tuning. We design a *selective weight extraction* that systemically finds these essential bits to check.

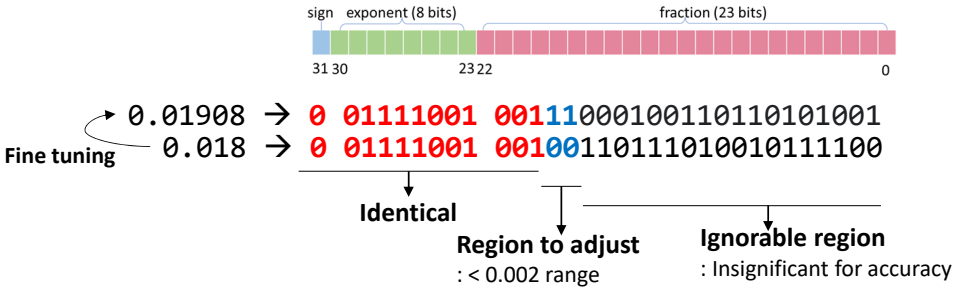


Figure 2.13. Selective Weight Extraction

Selective Weight Extraction

Selective weight extraction significantly reduces the scope of bit checking to those that cover the weight value gap between a pre-trained model and its fine-tuned model. We describe the process with float32 data while the algorithm can be applied to any data type. From our experiments, sign and exponent fields rarely change; an average of 99% weights keep their sign when fine-tuned. Only the fraction fields are updated mostly where only a handful of bits are corresponding to the small value gap (e.g., 0.002-0.01 in our earlier examples). Fig. 2.13 shows an example when weights use IEEE 754 format. Suppose that a weight in a pre-trained model is 0.018 and is fine-tuned to 0.01908. The fine-tuned weight value is black-box. Thus, the attacker should estimate the value, 0.01908, from the pre-trained model's weight value, 0.018. As illustrated, the sign, exponent, and first a few bits of the fraction field are identical. Only the two bits in blue color are responsible for the value gap, thus need checking. The remaining 18 bits in the fraction field do not need to be checked because those make very subtle differences (less than 0.001). From our experiments, the model prediction accuracy is almost the same (F1 score is dropped by less than 0.01) when discarding all weight values below 0.001.

The selective extraction identifies the bits of individual weights that *do not need to be checked* based on the pretrained model weight values through two steps; 1) excluding the weights less than 0.001 and 2) for the remaining weights, excluding all the bits except for a small number of bits that correspond to the weight value gap. In the second step, the bit locations to check may vary depending on the exponent field value. For example, in Fig. 2.13, the blue bits are checked

because they are corresponding to 0.00097 (=2^{-10}) and 0.00048 (=2^{-11}) respectively (together cover the estimated weight value gap (e.g., 0.002)) as the first bit value of fraction field is 2^{-7} (=$2^{121-127-1}$) according to IEEE 754. As weights are updated differently (U-shape distribution in Section 2.4.1), we check the essential bits for the estimated weight value gap based on the pre-trained weight value. We found that checking up to two bits per weight is sufficient for most cases. Algorithm 1 summarizes the process.

Algorithm 1: Selective Weight Extraction

```
target_weight ← W_i in fine-tuned model
base_weight ← W_i in pre-trained model
abs_base_weight ← abs(W_i) in pre-trained model
if abs_base_weight < 0.001 then
    clone model's W_i ← base_weight
end
else
    min ← base_weight − weight_dist
    max ← base_weight + weight_dist
    k ← most significant non-zero bit in fraction
    int_base ← 2^(exp−127) of abs_base_weight
    while up to 2 iterations do
        fr_base ← 2^(exp−127−k) of abs_base_weight
        if min ≤ (int_base + fr_base) ≤ max then
            check k_th bit of target_weight.
            clone model's k_th bit of W_i ← k_th bit of
              target_weight
        end
        k ← k + 1
    end
end
```

Leveraging the lower impact of first a few layers towards the downstream task accuracy (See Table 2.1), we begin the extraction from later layers and gradually recover earlier layers while checking the prediction accuracy of the clone model. The extraction stops when the clone model's accuracy becomes similar to the victim model.

For the task-dependent last layer, the pre-trained model does not have the baseline weight values because the layer is newly added while fine-tuning. Thus, we use row-hammer for all bit values. As the last layer has much fewer weights than the other layers, our selective extraction still reduces significant bit checking efforts. For all Transformer model sizes (from tiny to large), the last layer only contributes from 0.0005% to 0.009% of the total weight count (Section 2.7).

2.6.2 Adversarial Attack

The cloned model enables various white-box attacks, such as an adversarial attack. The adversarial attack finds adversarial inputs that perturb the prediction accuracy of the victim model. Suppose a victim model M has an output Y for a given input X (i.e., $Y = M(X)$). The attacker wants to find α that makes $Z = M(X+\alpha)$ where $Y \neq Z$. To demonstrate an adversarial attack, we generated adversarial inputs by using our clone model, tested the victim model with the adversarial inputs, and checked how many of the adversarial inputs led to incorrect predictions. We compared the effectiveness of our clone model with eight other substitute models. The substitute models (M') were fine-tuned from randomly selected pre-trained models with the victim model's (M) prediction records (i.e., $Y = M'(X)$ *where* $Y = M(X)$).

2.7 Evaluation

2.7.1 Methodology

We examined 70 pre-trained models and 170 fine-tuned models downloaded from various model repositories [13, 12, 18, 19, 9, 10, 16, 25]. The models use diverse Transformer architectures and sizes such as tiny, mini, distill, medium, base, large, xlarge, and xxlarge of BERT, GPT-2, RoBERTa, MobileBERT [139], CamemBERT [110], ALBERT [92], DeBERTa [73], XLNet [164], BART [96], T5 [125], and SpanBERT [80]. We selected these models based on their download counts (at least 100K downloads per month). The models are trained for various tasks but there is no single pair of pre-trained and fine-tuned models that are trained for the

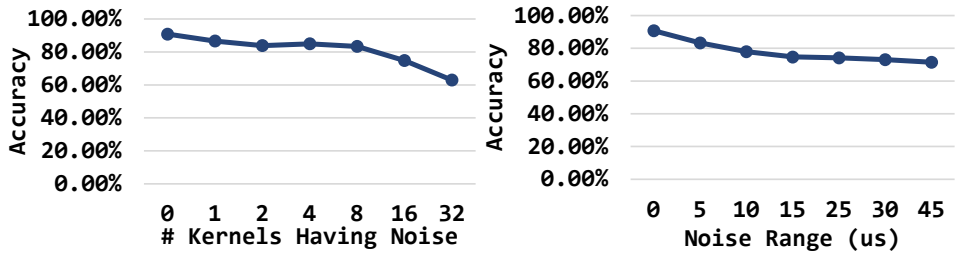


Figure 2.14. Extraction Accuracy: (left) impact of kernel count having noise (right) impact of noise lengths

same task. A ResNet-18 model was also downloaded from PyTorch vision repository [25] and fine-tuned to examine the generalization of our approach to non-Transformer models. We tested these models on an NVIDIA GeForce RTX 3050 (Ampere) GPU with CUDA v11.8.

2.7.2 Model Extraction Accuracy

We tested the model extraction accuracy with 350 data (of 1787 total training data). We evaluated the prediction accuracy by adding two types of noise to the kernel execution time measurements: 1) varying the scope of kernels impacted by noise and 2) varying significance of noise for individual kernel. Based on the typical kernel duration, we set the noise in the first test as $20\mu s$. We randomly selected 1 to 64 kernels from each input image and adjusted their execution times to $original\ execution\ time \pm 20\mu s$. For the second test, we selected 16 kernels from each validation image and changed their execution times to $original\ execution\ time \pm K\mu s$, where K is varied from 5 to $45\mu s$. Then, the adjusted time-series kernel execution images were fed to the CNN model for prediction. Fig. 2.14 shows the averaged accuracy results. The accuracy is 90.78% without noise and dropped slowly for both types of noises thanks to inherently error-tolerant CNN [97].

2.7.3 Cloned Model Accuracy

We tested the cloning accuracy by comparing the inference outputs of the victim model and the cloned model. Fig. 2.15 shows the results on a BERT-large victim model and the extracted clone model on 10570 SQuAD inputs. The accuracy and F1 score of the clone model had only

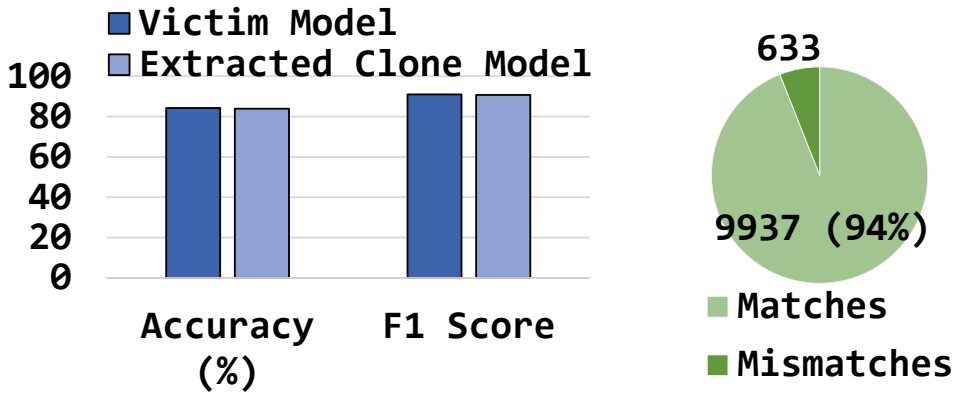


Figure 2.15. (left) Prediction Accuracy of Victim BERT model and Extracted model, (right) Fraction of Matched Predictions

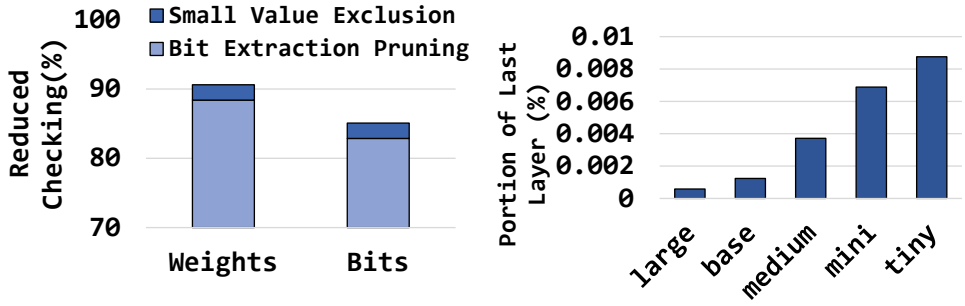


Figure 2.16. (left) Reduced Weight Value Checking w/o Errors, (right) Fraction of Last Layer in Total Model Weight Count

0.2% difference from the victim model. 94% of the cloned model's predictions matched with victim model.

2.7.4 Weight Extraction Efficiency

To understand the efficiency of selective weight extraction, we examined the weight values of a randomly selected fine-tuned model and its pre-trained model and counted the total number of bits that needed to be checked. If the actual weight value gap was larger than expected amount or the sign bit was changed, we considered that the selective weight extraction led to incorrect extraction. Fig. 2.16 shows the breakdown of the reduced weight checking. The *Weights* bar shows the total number of weights correctly pruned. The *Bits* bar shows the total number of bits correctly excluded from checking. 90% weights and 85% bits of weights were correctly

Figure 2.17. Cloning Accuracy with Fine-tuning Data

excluded from checking. The last layer hammering did not incur much overhead because the last layer contributes only up to 0.009% of the entire model weight count even in the smallest Transformer model, as plotted in Fig. 2.16.

2.7.5 Weight Extraction Necessity

To check if weight extraction is necessary, we also measured cloning accuracy by assuming that an attacker has access to limited amount of fine-tuning dataset. Fig. 2.17 shows the results of fine-tuning a pre-trained BERT-base with different amounts of downstream data. The attacker needs at least 40% fine-tuning data to copy the model with less than 5% accuracy drop, which is unrealistic.

2.7.6 Adversarial Attack Effectiveness

To compare the effectiveness of adversarial attack, we developed eight other substitute models besides our extracted clone model. The eight models were developed by downloading random pre-trained models and fine-tuning them with the victim model's prediction records that were collected from 18K inferences. Fig. 2.18 shows the success rate of the nine models when 10K sample inputs were tested to locate adversarial inputs. The adversarial inputs identified by

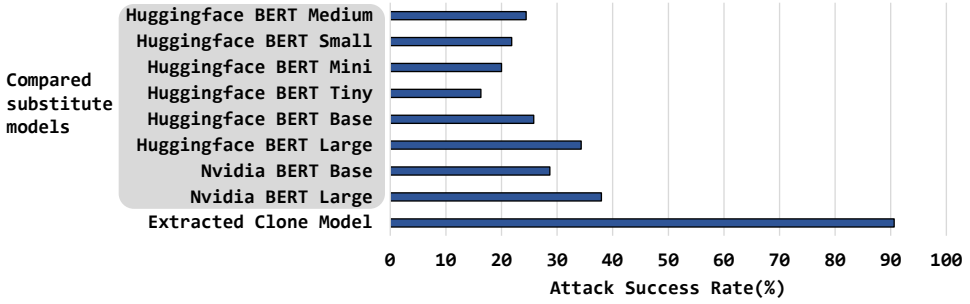


Figure 2.18. Effectiveness on Adversarial Attack

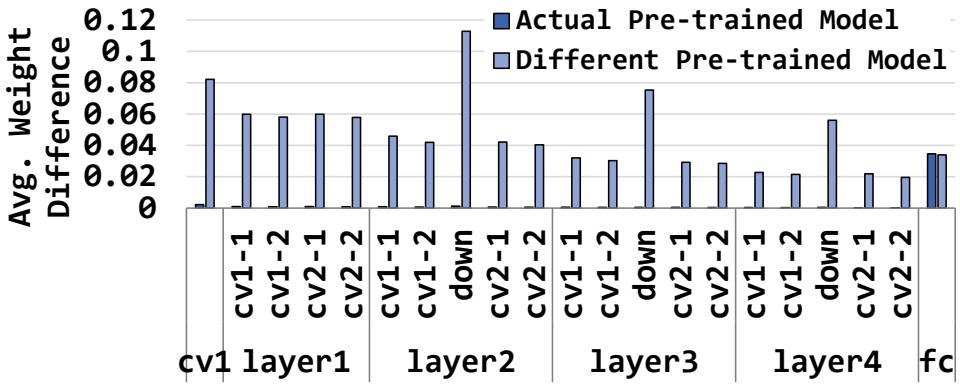


Figure 2.19. Weight Similarity in a CNN Model (ResNet-18)

our clone model showed a superior attack success rate (90.62%) than the other models (up to 38% accuracy).

2.7.7 Attack Generalization

To check if weight value similarity is a universal symptom of transfer learning, we measured weight similarity of a CNN model, ResNet-18, between a fine-tuned model and its pre-trained model as plotted in Fig. 2.19. We fine-tuned a pre-trained model with Hymenoptera dataset. For a comparison, we trained another ResNet-18 model with the same dataset from scratch (without using transfer learning) and compared the weight values with the fine-tuned model. The darker blue bars are the weight difference between the fine-tuned model and its pre-trained model. The lighter bars are the difference from the model trained from scratch. The fine-tuned model had almost zero weight difference from its pre-trained model in almost all layers, while it showed at least 20× higher difference from the other model, even when they

were trained with the same dataset, which is consistent with our observations from Transformer models.

2.8 Related Work

Several studies showed ML model extraction attacks. [74, 118] proposed a framework to extract a black-box CNN model architecture by using performance counter and bus/memory probing on GPUs. [172] proposed a technique to clone CNN model weights by monitoring unencrypted PCI-e bus traffic. [161] extended the cache-timing attack for model extraction on CNNs. [126] leveraged row-hammer attack [85] to recover weight values of victim models that use int8 weights. **All these studies targeted small CNN models, while we propose a new model extraction attack for billion-parameter models. Decepticon is the first approach using two-level attack model.**

Some studies [89, 158, 108] demonstrated security concerns in Transformer models by cloning a BERT model with prediction records. **These studies did not extract model architecture and Decepticon showed a superior cloning accuracy than these approaches as can be seen in the adversarial attack result (Fig. 2.18).**

Chapter 3

Uncovering Power and Frequency Behaviors and Their Security Implications in Edge GPU Platforms

3.1 Introduction

With the increasing demand for computing power in robots, personal assistance solutions, and medical devices, mobile platforms are widely adopting accelerators such as GPUs and deep learning accelerators (DLAs). Though these accelerators enable real-time processing for complex applications with their massive parallelism, their power and thermal behaviors, and the security implications, have not been evaluated in detail. While there have been extensive studies about power-efficient edge computing [91, 68, 90], most of them have focused on optimizing the application algorithms and utilizing the given compute resources better.

In this work, we uncover the hidden power and frequency behaviors of GPUs and DLAs on multiple edge GPU platforms on multiple levels, from individual instructions, data widths and values, and to application. With these observations, we demonstrate, for the first time, that edge GPUs and DLAs are vulnerable to covert channel attacks. By exploiting the unique power and frequency behaviors of GPUs and DLAs on edge platforms, we develop three novel covert channels that provide 31$\times$ more communication bandwidth than state-of-the-art covert channels [112].

Why do we need a separate security study on the edge platforms? Almost all prior works on GPU covert channels [117, 64, 65, 40, 112] limited their scope to server GPUs. Though some researchers started exploring edge GPU security [120], those leveraged SoC platform components, such as the refresh management units of low-power DRAMs (e.g., LPDDR), which are not uniquely equipped in edge GPU platforms. To the best of our knowledge, this is the first study that reports covert channels that leverage power and frequency behaviors of edge GPUs and DLAs. It is noteworthy that edge GPUs could be more vulnerable to security attacks, because they are deployed closer to the end users, who have direct access to the source of confidential or private data, thus more lucrative to attackers. Edge GPUs and DLAs, like in NVIDIA Jetson platforms, are deployed in autonomous vehicles, industrial automation, surveillance, IoT devices, and medical applications — all of which handle sensitive data, e.g., surveillance feeds, medical imaging data, financial transactions etc. while operating in untrusted environments [142, 138, 58, 5, 26, 31, 134]. Unlike server GPUs that operate in controlled data centers, edge GPUs are deployed in public, remote, or exposed locations, making them vulnerable to physical tampering, cyber threats, and unauthorized access. Furthermore, edge GPUs and DLAs offer unique opportunities for the attackers, which are not present in server systems. For example, server GPU covert channels can be susceptible to noise from other benign applications. However, edge DLAs and some lower-end edge GPUs allow executing one kernel at a time. Thus, they offer a relatively noiseless channel for covert communication between two attacker processes. Given these unexplored security threats, we need more security studies tailored for edge systems.

The contributions of this paper are as follows.

- To the best of our knowledge, this is the first study that provides in-depth power, energy, and frequency characterization for edge GPUs and DLAs on multiple levels, from instructions, SoC components, and to the application.
- We design micro-kernels to characterize power and frequency behaviors in PTX instructions, which can be used for testing different edge GPUs. Our results show surprisingly

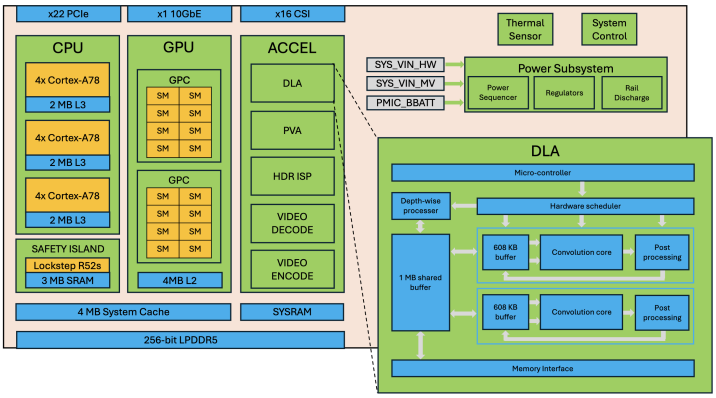


Figure 3.1. Jetson AGX Orin Architecture

different power and energy statistics among the tested platforms.

- With the characterization results, we design novel power- and frequency-based covert channels in edge GPUs and DLAs. The GPU frequency-based covert channels achieve 31× more bandwidth than the state-of-the-art frequency-based covert channel in server-scale GPUs [112]. Our DLA-based covert channel achieves 4× more bandwidth than the covert channels that use other special-purpose engines, such as video encoder and decoder [112]. To the best of our knowledge, this is the first study that demonstrates covert channel communication using NVIDIA DLAs. Our proposed covert channels are robust to conventional mitigations [159]. Unlike existing covert channel attacks that leverage resource contention, our proposed attacks allow the spy process to passively read system statistics to decode data. Such a passive nature makes our attack harder to detect because the trojan and spy processes look just like benign applications without causing intentional resource contentions.

- We suggest several mitigation methods, such as limiting the accessibility, resolution, and accuracy of the power and frequency monitoring tool, and evaluate their effectiveness.

3.2 Background & Related Work

3.2.1 GPU System on Chip

Figure 3.1 shows the baseline SoC architecture of one of our target embedded GPU platforms, NVIDIA Jetson Orin. NVIDIA Jetson platforms use high-end mobile GPU SoCs designed to support robots and edge AI solutions. The GPU SoC includes integrated GPUs (iGPUs), ARM-based CPUs, and various specialized compute units such as a DLA, video encoder, and decoder. The CPU and GPU share the same physical memory. To support battery-powered systems, the platforms integrate various low-power components, such as Low-Power Double Data Rate (LPDDR) memory devices for their main memory and power management integrated circuits (PMIC), voltage regulators, and a power tree for their power management.

3.2.2 Deep Learning Accelerator (DLA)

Figure 3.1 also illustrates NVIDIA DLA [20], which is a fixed-function accelerator designed for convolutional neural network (CNN) inference workloads. It enables a hardware-software co-design where the DLA software compiles the target CNN model into a DLA loadable binary and the DLA hardware executes that binary. DLA is equipped with specialized execution units for several neural network layers, such as convolution, fully connected, activation, pooling, and batch normalization. Currently it only supports INT8 or FP16 precision. Depending on the NVIDIA Jetson board architecture, each SoC can have up to two DLA cores.

3.2.3 Power Characterizations for GPUs

GPU power efficiency has been extensively studied [76, 153, 147]. To better understand power consumption, some studies also proposed power stressmarks for CPUs (e.g. MAMPO [70] for the multicore systems and SYMPO [69] for single-core CPU systems) and for GPUs (e.g., Guser [137]). These studies design a software tool to maximize peak power dissipation through concurrent executions of various combinations of instructions. These stressmarks can test the

power vulnerabilities of the target processors, rather than providing detailed power characterizations. Thus, these studies mostly focused on instruction-level power variations for CPUs and discrete GPUs. Various architecture simulators (e.g., GPUWattch [95] and Accel-Wattch [82]) have been used for evaluating the GPU power consumption. However, these tools support only a handful of discrete GPUs. There have been several studies that tackle the power efficiency for the embedded GPUs [93, 133]. However, these studies did not show detailed power characteristics such as power factors of individual instructions and SoC components. Without a clear understanding of the power consumption, they passively monitor the power sensors while running benchmarks on the embedded GPUs.

On the other hand, in this paper, we delve into detailed power characterizations of GPUs and DLAs on edge GPU platforms. We design micro-benchmarks and show diverse power impacts of instructions, data lengths, and data contents. With these detailed characterizations, we evaluate the security vulnerability of edge computing.

3.2.4 Covert Channels on GPUs

When two entities communicate with each other through a channel not intended for regular communication, this stealthy communication is called a covert channel. Typically, this involves two malicious processes where only one of them has access to some sensitive information and the legitimate communication channels are protected in a way that the sensitive information cannot be transmitted between them. To bypass the system's security policy, the malicious processes rely on some system features not intended for communication, such as hardware or software side effects and they cooperate to establish a stealthy channel to transmit the sensitive data. Covert channels have recently been extensively studied for GPUs. Naghibijouybari et al. [117] demonstrated covert channel attacks based on various shared resources including L1 and L2 caches, special function units, and memory. Nayak et al. [119] and Ahn et al. [40] developed covert channels based on GPU translation lookaside buffer (TLB) and on-chip network respectively. Leaky buddies [64] demonstrated cross-component covert channel attack

in integrated CPU-GPU systems where the trojan (sender) and spy (receiver) are located in GPU and CPU. They utilized the last-level cache (LLC) shared between CPU and GPU to build their cross-component covert channel. Spy in the GPU-box [65] built a cross-GPU covert channel in multi-GPU systems utilizing the shared remote L2 cache. Veiled Pathways [112] explored covert channels based on GPU uncore components like GPU DRAM, PCI-e, and special purpose engines like NVENC, NVDEC, and NVJPEG. Thermal and electromagnetic emanation have been used to build covert channels in edge NVIDIA GPUs [71, 72]. While these studies leveraged most of the core components in GPU platforms, none of them utilized the power and frequency of edge GPUs and DLAs for covert channels.

3.3 Methodology

Among several GPU SoC platforms, we investigate NVIDIA Jetson platforms because of their popularity and strong support by NVIDIA [45, 99, 78]. We tested four platforms, TX2, Xavier, Orin Nano, and AGX Orin to understand the impact of different CUDA and SoC architectures on power consumption. We intentionally limited the scope of the study to the CUDA architecture to delve deeper into the characterizations of the CUDA instruction set architecture (ISA) rather than providing shallow investigations over multiple ISAs. We used `tegrastats` [24] to measure power, energy, and frequency of GPUs and SoC components. To reduce any interference among tests, we ensured that the platforms returned to their idle state before each experiment. The idle state can be recognized via resource utilization, power consumption, frequency, and temperature. Out of them, we observed that temperature takes the longest time to become idle (i.e., the platform can still be hot while frequency, power, and utilization are stabilized). Thus, we enforce that the platform's temperature becomes the idle state (i.e., below 42°C). Specifically, the platform was left idle for at least 10 minutes between tests to cool down. We used the platform's temperature sensors for verification.

Table 3.1. Evaluated GPU SoC Platforms

Category	TX2	AGX Orin	Orin Nano	AGX Xavier
Architecture	Pascal	Ampere	Ampere	Volta
CUDA cores	256	2048	1024	512
Tensor cores	0	64	32	64
DLA cores	0	2	0	2
Memory	8 GB	64 GB	8 GB	32 GB
GPU max frequency	1122 MHz	1300 MHz	918 MHz	670 MHz
DLA max frequency	N/A	1369 MHz	N/A	115 MHz
Power Mode	MAXP_CORE_ARM	MODE_30W	MODE_25W	MODE_15W
CUDA Driver Version	10.2	11.4	12.6	11.4
OS	Ubuntu 18.04	Ubuntu 20.04	Ubuntu 22.04	Ubuntu 20.04

3.3.1 GPU Benchmarks

Micro-Benchmarks: To understand the power impact of individual instructions, we built micro-benchmarks where we run different PTX instructions [22] in a kernel with a loop of 1 million iterations for TX2 and 100 million iterations for Orin. The Orin required more iterations per micro-benchmark in order to saturate the power measurements, mainly due to much faster execution time. Note that we used PTX instructions to test the same micro-kernels for both platforms. The PTX code is compiled and run with the platform-specific SASS instructions on the two platforms. Figure 3.2 shows an example kernel. To reduce potentially unnecessary instructions, we implemented the micro-kernels directly with PTX instructions, instead of CUDA C. Once the input is set, the target instruction is executed for the given number of iterations. To offset the influence from all the other parameters except for the target instructions, we used fixed-size kernels that use 640 thread blocks of 1024 threads each. We configured the kernel size that the execution time is measurable (i.e., not too short) while reflecting execution variations of individual instructions (i.e., not too long). We used the highest compiler optimization level (-O3) to reduce the interference from activities in the pipeline (e.g., control branches) and also to minimize unnecessary instructions. We used different input values depending on the instruction type and operation. For example, we used a positive float, a negative float, and an unsigned integer value as input data for `sqrt.f32` (floating-point square root), `abs.f32` (floating-point absolute), and `add.u32` (unsigned integer addition), respectively.

```
__global__ void ADDCS64Kernel(int iterations, void *value)
{
    asm volatile (
        ".reg .s64 %r17, %r18;\n\t"
        "mov.s64 %r17, 0x7FFFFFFFFFFFFFFF;\n\t"
        "mov.s64 %r18, 0x7FFFFFFFFFFFFFFF;\n\t"
    );
    for (int i = 0; i < iterations; ++i)
        asm volatile("addc.s64 %r17, %r17, %r18;\n\t");      // target instruction

    long* l_val = (long*)value;
    asm volatile("mov.s64 %0,%%r17;\n\t" : "=l"(*l_val) :);
}
```

Figure 3.2. `addc.s64` Kernel

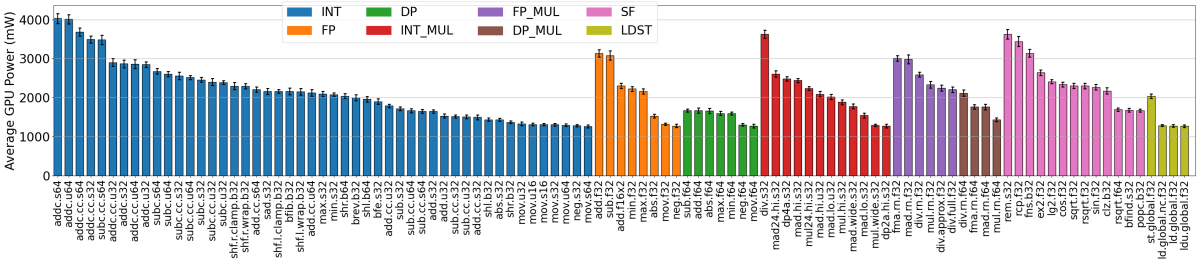
Instruction Classes: We classified instructions based on the execution units used for the instructions. A widely used GPU power simulator, AccelWattch [82] uses different power weights for each execution unit and operation. By following similar classification, we categorize instructions into INT, FP, DP, INT_MUL, FP_MUL, SF, DP_MUL, TENSOR, and LDST. The meanings of each class are described in Table 3.2.

Table 3.2. Instruction Classes Based on Execution Units

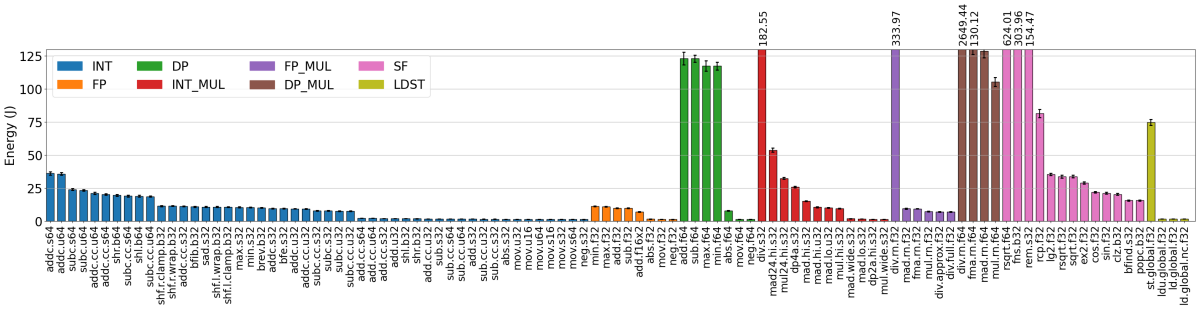
Class	Meaning
INT	Integer ALU instructions
FP	Single-precision float
DP	Double-precision float
INT_MUL	Integer Multiply instructions
FP_MUL	Single-precision Multiply
SF	Special function instructions
DP_MUL	Double-precision Multiply
TENSOR	Tensor Core instructions
LDST	Load and store instructions

3.3.2 DLA Benchmarks

To measure the architectural behaviors of DLA, we ran four DLA reference ONNX models provided by NVIDIA [21] as listed in Table 3.3. We used `trtexec` [30], a command-line tool provided by NVIDIA TensorRT, to run the inference of these models with INT8 precision. We used TensorRT version 8.5.2. These models have DLA-compatible layers like convolution, activation, pooling, batch normalization, and elementwise layers. They are heavily optimized for DLA inference, and all the layers of each model were merged into a single DLA-compatible



(a) Average Power



(b) Energy Consumption

Figure 3.3. Power and Energy of 1 Million Executions of Various Instructions on TX2

layer.

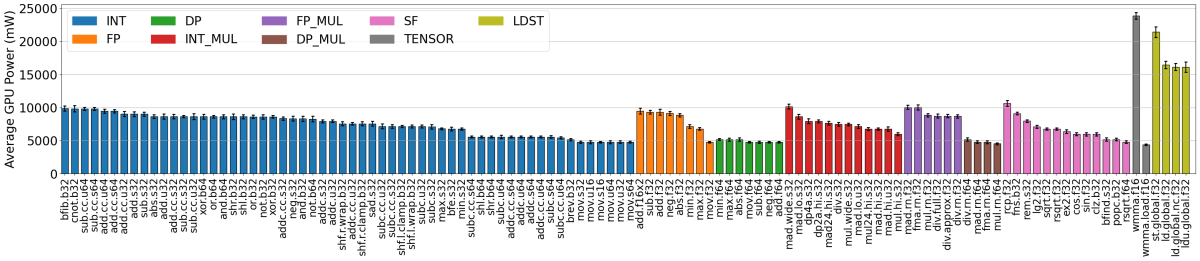
Table 3.3. DLA Reference Models

Model	Original Number of Layers	Layers after DLA Optimization
ResNet-50	121	1
SSD-ResNet-34	189	1
SSD-MobileNetV1	76	1
RetinaNet ResNet-34	125	1

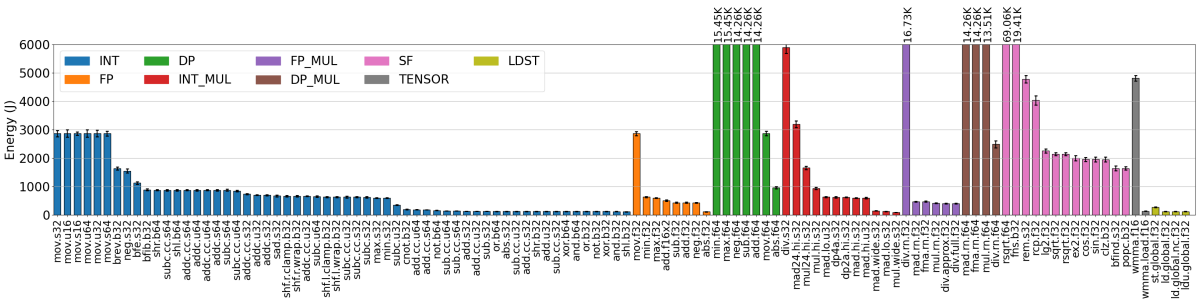
3.4 Observations

3.4.1 Instruction-Level Power and Energy Consumption

Figures 3.3 and 3.4 show the average power and energy dissipation of 113 and 115 PTX instructions on TX2 and AGX Orin, respectively. Orin includes an additional two Tensor Core instructions. Due to architectural similarities, Xavier's results were similar to TX2 and Orin Nano's statistics follow similar trend as AGX Orin. Thus, we show only the results of two represenative platforms (TX2 and AGX Orin) in this section. We include all four platforms in



(a) Average Power



(b) Energy Consumption

Figure 3.4. Power and Energy of 100 Million Executions of Various Instructions on AGX Orin

our security evaluation in Section 3.5. Each graph shows the sorted list in terms of each of the two metrics in each instruction class. For example, `addc.s64` and `mov.s64` in Figure 3.3a are the most and least power-hungry INT instructions on TX2, respectively. Each instruction is tested five times and the mean values with ±2 standard deviations are plotted. For instructions whose mean energy exceeds the axis limit, bars are truncated and only the visible portion of the error bar is shown; the annotated labels indicate the true mean values.

Overall, all two metrics show dramatic differences among instructions. We can see an almost linear power consumption trend for the sorted instruction list even within the same instruction class. For example, `addc.s64` consumes over 2× more power than `mov.s64`. This is because `addc.s64` is an extended precision instruction, which consumes more energy as it accesses an implicitly specified condition code register having a carry flag bit. In Figure 3.3a, the 15 most power-hungry instructions are all extended precision instructions. The special registers are accessed only by extended precision instructions. Some 32-bit instructions consume more power than 64-bit instructions. This is mainly due to two reasons. First, the compute intensity of

single-precision is much higher as TX2 has 128 single-precision units while having only four double-precision units. Second, the complexity of some single-precision instructions is higher. Instructions like `rcp.f32` and `rem.s32` use multiple compute units, such as multiply and special function units, which require extra power dissipation.

When comparing the two metrics, we also observe that power consumption does not have a high correlation with energy consumption, especially for SF instructions, as Figure 3.3b has a different sorted order with Figure 3.3a. This is because power-hungry instructions do not necessarily have a longer execution time. For example, `rsqrt.f64` is not the most power-consuming instruction within the SF instruction class, but because of its longer duration, it consumes the most energy among all SF instructions.

Impact of Architecture: Figure 3.4 shows the same evaluation results on AGX Orin. Note that, though we used the same tool (`tegrastats`) for both architectures, AGX Orin does not support GPU-only power measurement [24]. The `tegrastats` reports a combined power consumption of GPU and SoC. Therefore, the results in Figure 3.4a includes SoC power. When we compare the relative power intensity across instructions, there is no single class of instructions that shows the same order as TX2's results, even when we used the same copy of micro-kernels for the two platforms. This strongly shows that different power factors should be considered for different architectures.

AGX Orin consumes significantly higher power for TENSOR and LDST instructions. We believe this is because of the compute intensity of TENSOR core instructions and the $6\times$ larger memory capacity of Orin platform than TX2. Overall, AGX Orin consumes $3\times$ higher average power than TX2's. Besides the SoC power included in the measurements, the increased complexity of the architecture contributes to the high power consumption. AGX Orin is incorporated with $4\times$ more CUDA cores, $8\times$ more memory, and also embeds Tensor Cores, which are not included in the Pascal GPU used by the TX2. Unless using fine-grained component-level power management, all compute and memory resources dissipate at least leakage power for individual instruction execution, even when some components are not used

for the target instructions. While Jetson platforms support various power management, they do not gate the clock or power of individual architecture components based on their idleness. Given the increasing size and complexity of edge computing, effective power management should also be incorporated; otherwise, we will need huge battery capacity.

Impact of Instruction Modifiers It is also worth noting that, for the instructions supporting optional modifiers, the modifiers influence power and energy consumption significantly. For example, `div.rn.f32` (round modifier) consumes significantly more energy than `div.approx.f32` (approximate modifier) and `div.full.f32` (full modifier) (47× and 40× in TX2 and Orin respectively). Both `div.approx.f32` and `div.full.f32` compute fast and approximate divisions, but `div.rn.f32` computes IEEE 754 compliant rounding which is more complex and time-consuming.

Observation 1. Instructions consume notably different power and energy even on the same platform and within the same instruction classes.

Observation 2. The power and energy trends are different in different GPU architecture generations. This means that the impact of instructions for power and energy should be measured per platform.

Observation 3. Due to the resource size and lack of power management, emerging GPUs consume significantly higher power. As battery lifetime is one of the most important aspects in embedded systems, GPUs should be equipped with a fine-grained power management method, e.g., power gating inactive logics.

3.4.2 Impact of Data Width

To better understand the impact of data width on power consumption, we compare the average power consumption of instructions which have both 32-bit and 64-bit versions. Figure 3.5 shows the results. Interestingly, almost all the floating-point instructions and some

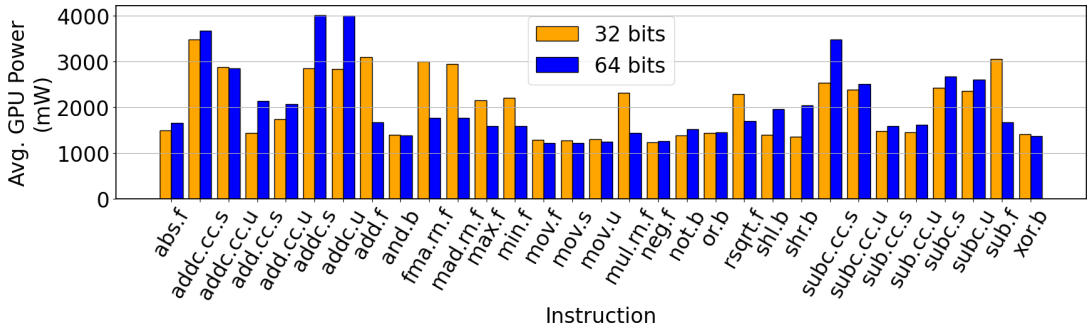


Figure 3.5. Effects of Data Width on Instruction Power (TX2)

integer instructions show higher average GPU power with the 32-bit version. Some instructions show almost 1.5× higher power consumption with the 32-bit version, such as `add.f`, `fma.rn.f`, `mad.rn.f`, `mul.rn.f`, and `sub.f`. This is because of significantly higher 32-bit compute resources than 64-bit units; there are 128 single-precision units and only 4 double-precision units in each SM on TX2. 32× more single-precision units lead to denser computations in a short period of time. Thus, if peak power consumption is a more critical concern (e.g., power supply limitation) than performance, a hardware power management logic may consider scheduling the single-precision instructions sparsely, rather than consecutively. However, the overall energy consumption of single-precision instructions is several orders of magnitude lower than double-precision instructions as shown in the FP and FP_MUL, and DP and DP_MUL instruction groups of Figure 3.3b. This is because of the inherently higher complexity of double-precision computations as well as longer execution times with fewer execution units. For the other instructions, 64-bit versions show up to 5% higher power consumption than 32-bit version instructions.

3.4.3 Impact of Data Value

Even for individual instruction execution, depending on the input data, the gate-level activities can be different. To understand the influence of data on the instruction power consumption, we measured the time-series power consumption of four instructions, as shown in Figure 3.6. Based on individual instructions' operations, we chose extreme data inputs that

will lead to the least or the most gate-level activities. For example, `brev.b32` reverses the bit order (e.g., 10000... will be reversed to ...00001). Thus, reversing alternate bit patterns (e.g., 101010... and 11110000...) results in much more bit flips than bit patterns like ...0001000. Similarly, for `xor.b32` instruction, XORing a value with itself can consume different power based on the value. For example, XORing bit pattern 101010... with itself causes more Hamming distance in the result than XORing bit pattern ...00001 with itself. More Hamming distance leads to more transistor-switching and thus increases dynamic power consumption. In general, if the Hamming weight of the input is high (more 1s than 0s), more power can be consumed because there are more active transistors. This effect of Hamming weight can be observed for all the instructions in Figure 3.6 where low Hamming weight data values consistently consume lower power. Surprisingly, some instructions (e.g., `brev.b32` and `xor.b32`) consume almost 1000mW more power only by changing the data value. Also, the impact of data value is different in different instructions. `add.s32` exhibits only subtle differences for different values.

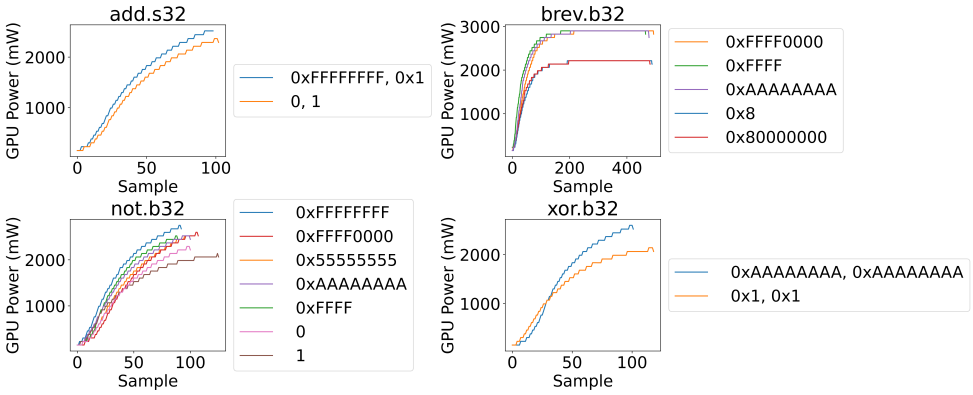


Figure 3.6. Effects of Data Value on Instruction Power (TX2)

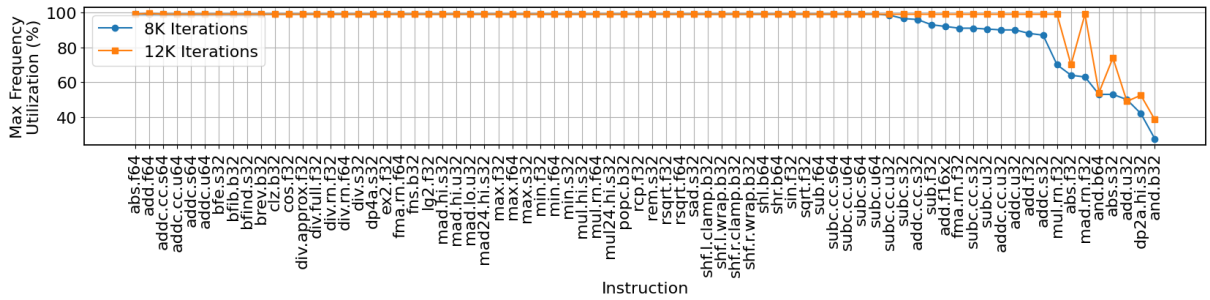


Figure 3.7. Maximum Frequency Utilization of Various Instructions (TX2)

Observation 4. Counterintuitively, 32-bit instructions consume significantly higher power than 64-bit equivalent instructions for some instruction classes because of the larger volume of 32-bit execution units than 64-bit units. However, due to the higher complexity and longer execution time, 64-bit instructions consume much more energy than 32-bit equivalent instructions.

Observation 5. Instructions consume notably more power for certain data values, especially those that have higher Hamming weight. The impact of data value on the instruction power consumption varies on different instructions; some instructions consume almost 1000mW more power by simply changing the input value.

3.4.4 Instruction-Level Frequency Utilization

To measure the impact of instructions on GPU frequency utilization, we collected frequency logs of PTX instruction-level micro-benchmarks described in 3.3.1 with various compute loads, 8K, 12K, and 16K iterations, in the TX2 platform. We used 8K as the minimum iteration because if we set the iterations lower than 8K, most of the micro-benchmarks finish too fast to capture any frequency data. Figure 3.7 shows the maximum frequency utilization of various PTX instructions. We omit 16K results because all instructions reach 100% frequency utilization. For the majority of instructions, both 8K and 12K iterations consume almost 100% frequency utilization for both cases. However, some INT, INT_MUL, FP, and FP_MUL instructions utilize lower

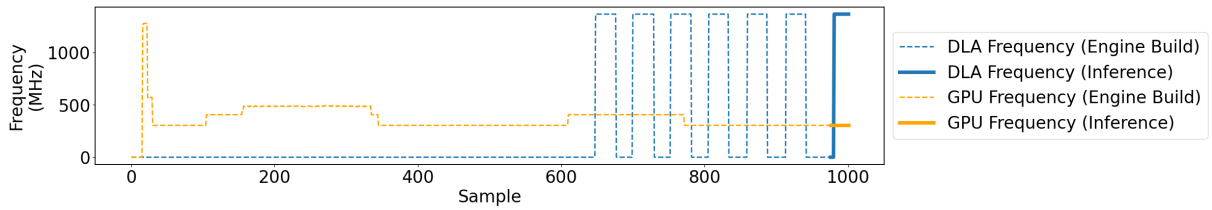


Figure 3.8. Time-series Frequency of DLA and GPU core from ResNet-50 (Orin)

frequencies, especially at 8K iterations. Instructions like abs.f32, abs.s32, and.b32, and.b64, add.u32, and dp2a.hi.s32 utilize lower frequency even with 12K iterations. This is an interesting statistic because we ran a sufficiently high number of compute loads in each micro-benchmark: 640 thread blocks of 1024 threads each on a GPU having 2 SMs with 128 CUDA cores per SM. This indicates that instructions induce diverse compute intensity even with the same degree of parallelism, and the GPU's frequency scaling mechanism increases the frequency level gradually, depending on the compute intensity, not by the application's parallelism degree.

Observation 6. Instructions bring about diverse compute intensity even for the same degree of parallelism in the application level. GPU frequency is scaled based on the compute intensity, which leads to certain instructions to exhibit lower frequency utilization even with massive parallel micro-kernel execution.

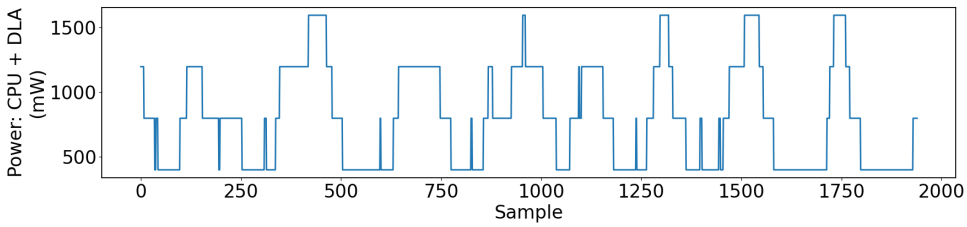
3.4.5 Behaviors of DLAs

An AGX Orin has two DLAs for CNN inference. As a specialized ASIC accelerator for CNNs, it is impossible to design instruction-level micro-kernels for DLAs. Therefore, to characterize the architectural behaviors of DLAs, we ran four DLA-compatible CNN models, listed in Table 3.3. We used `tegrastats` for measurements. `trtexec` is used to build the TensorRT engine to run inference on these models. Figure 3.8 shows the time-series frequency of both GPU and DLA while executing ResNet-50.

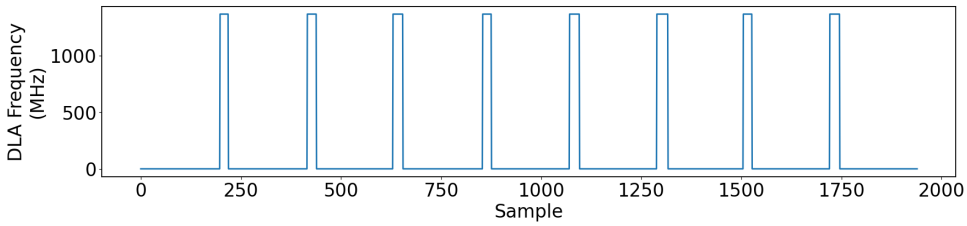
The model execution can be broken down into two phases: the TensorRT engine build phase and the inference phase. During the engine build phase, the model is offloaded to DLA

several times to select the optimum tactics and tensor formats. Tactics selection involves identifying the fastest low-level implementation of each layer from various library functions such as cuBLAS, cuBLASLt, cuDNN, etc. These dry-runs of testing memory mappings and low-level algorithms cause the six spikes in the DLA frequency plot of Figure 3.8. Note that, it is not necessary to build the engine every time to run inference on a DLA-compatible model. The engine can be saved after it is built, and can be loaded later for inference. Once the optimal format is selected, the inference is quick; the one DLA frequency spike highlighted with bold blue color in the figure indicates the inference phase length.

One interesting characteristic of DLA execution is that the DLA frequency is almost binary; either the highest (1369 MHz) or the lowest (0 MHz), unlike GPU frequency, which is gradually increased or decreased based on the compute load. The other three models also showed a similar frequency trend. We believe this could be because GPU is equipped with dynamic frequency scaling, while DLA operates as a passive co-processor that is turned on upon kernel invocation. It is also noteworthy that DLAs do not support multiprogramming (i.e., only one kernel can execute at a time). Therefore, DLA frequency measurement can indicate the DLA utilization by the target application with almost zero noise.



(a) DLA Power



(b) DLA Frequency

Figure 3.9. Power and Frequency of DLA Core from 8 Convolution Workloads on AGX Orin

To understand the finer-grained behaviors of DLA, we ran one convolution layer on one DLA core. We ran the same layer multiple times while posing a 5-second sleep period between two consecutive executions. The convolution layer has 3 input channels, 3 output channels, and a 1x1 kernel size. Figure 3.9 shows the power and frequency measurements. The built-in sensors in the AGX Orin report the frequency of DLA cores separately. But, the power consumption results include both CPU and DLA cores. To indicate the contribution of DLA cores towards the power consumption in Figure 3.9a, we show the frequency measurements in Figure 3.9b that are plotted with the identical time scale on the x-axis. As observed in the CNN evaluation, Figure 3.9b shows sharp rises and drops of DLA core frequency, which clearly indicate the DLA activities. We can see that whenever DLA is activated, the total power consumption increases rapidly by around 250 mW in Figure 3.9a. However, it is challenging to isolate DLA power consumption due to the noise from CPU activities.

Observation 7. Unlike GPU frequency, which scales with the compute loads, DLA frequency shows sharp rises and drops. This indicates that DLA doesn't have frequency scaling. As DLA does not allow multi-programming, DLA activity is easily detectable without any noise from frequency statistics, but not from power statistics.

3.5 Security Vulnerabilities of Edge GPUs

In the earlier section, we characterized the power, energy, and frequency behaviors of GPUs while executing various instructions in different data widths and values, as well as DLAs while running full and a layer of CNN models. One notable common behaviors that we were able to observe from the measurements was that the statistics vary notably depending on instructions. Such asymmetric behaviors are the main sources of various hardware-based security attacks. Especially, covert channel attacks leverage asymmetric characteristics of architectural components, such as access latencies and power consumptions, to encode data and transmit sensitive information to spy applications. In this section, we will show how our observations can

be leveraged to design various covert channel attacks against mobile GPU platforms.

3.5.1 Challenges in Developing Covert Channels on Edge GPUs

Covert channels have recently been extensively studied for GPUs [117, 64, 65, 40, 112]. Almost all prior works on GPU covert channels have limited their scope to consumer and server GPUs. Few studies [71, 72] have incorporated edge GPUs in their covert channel attacks. One of the key challenges in constructing covert channels in NVIDIA edge GPUs is the lack of Multi-Process Service (MPS) [36] support. MPS allows concurrent kernel execution from multiple processes. Most of the prior works on GPU covert channel exploited MPS to collocate and content for resources in the same GPU for covert communication between two different processes [119, 117, 40, 112, 120]. However, MPS feature is not supported in most NVIDIA edge GPUs [33]; NVIDIA started MPS support for Volta embedded GPUs very recently while their other edge GPU architectures still lack this support [35]. As a result, prior works on contention-based GPU covert channels are not readily applicable to edge GPUs. Another challenge is the lack of NVIDIA management library (NVML) [37] support for edge GPUs [38]. The prior attacks that leveraged NVML library functions [112] cannot be implemented in edge GPUs. We propose covert channels that neither require MPS nor special management libraries, which enables wider deployment on any NVIDIA Jetson platforms. We evaluate our attack models on all four platforms listed in Table 3.1.

3.5.2 Threat Model

Figure 3.10 depicts the covert channel threat model. We assume two processes where a trojan process's goal is to send data to a spy process through an indirect channel not intended for communication. Both of these processes are user-level processes without any root privileges. The trojan and spy processes run on the same integrated CPU-GPU edge device, such as a Jetson board. The trojan (sender) process is executed on the GPU or DLA core, and the spy (receiver) is executed on the CPU. The trojan and spy do not explicitly share any memory objects for direct

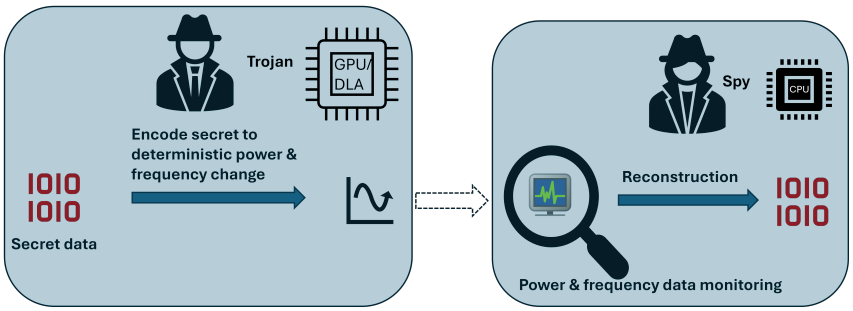


Figure 3.10. Edge Device Covert Channel Attack

communication, and they are not permitted to communicate with each other. We assume the attacker has no physical access to the device; remote access is sufficient for implementing the attack. Command line utilities like `tegrastats` [24] or `jtop` [15] are assumed to be available in the system to query system statistics. Finally, we assume there is no noise from a third process concurrently running on the GPU when the trojan tries to covertly send data.

Multi-tenancy in Edge Platforms: We assume two processes to run on the same edge device in our threat model. This is a realistic scenario because such a multi-tenancy does not necessarily mean the concurrent execution by multiple users; rather, the coexistence of multiple trust domains (e.g., applications and software modules) is sufficient. Several real-world applications consist of multiple modules that are developed by third party entities, which could carry trojan and spy modules. For example, due to the cost and efforts required to develop deep learning models, many AI applications integrate with third-party models from model zoos like huggingface [14], ultralytics [79], Jetson Zoo etc. Plugin ecosystems are also common in edge video/AI stacks on Jetson platforms, where the plugin could embed a malicious process. The widely used video analytics libraries, DeepStream and GStreamer, explicitly support the addition of custom/third-party plugins [1]. Such concurrent executions of third-party models, plugins, and services from different vendors on an edge device easily make mutually untrusted software components share the same GPU resources, even when the platform is not shared by multiple users. As a result, even when a device is operated within a single administrative domain (i.e., user), co-resident applications or modules may belong to different trust boundaries. In

such settings, an adversarial module can intentionally modulate GPU workloads to transmit information through shared hardware resources, while a colluding receiver observes the resulting hardware-level side effects.

Reducing Noise with Dynamic Parallelism: One common hurdle of covert channels is the presence of noise from a different process other than the trojan. To mitigate this noise, we implement our covert channels using the dynamic parallelism feature in CUDA [4], where the trojan GPU kernel internally forks child processes that run the micro-kernels that actually transmit information. Without using dynamic parallelism (i.e., a CPU process calls each micro-kernel directly), the GPU would yield to other waiting GPU kernels after each micro-kernel finishes, which makes the subsequent bit delivery interfere with other applications. Dynamic parallelism allows the trojan kernel to occupy GPU resources exclusively as long as the children micro-kernels finish the transmissions. Note that many edge GPUs, such as our target NVIDIA Jetson TX2 used for GPU power and frequency covert channels, do not support multi-process service (MPS). Therefore, during the trojan kernel's execution, no other process's kernels are allowed to execute on the same GPU. We use this dynamic parallelism-based noise reduction for both GPU power (Section 3.5.3) and frequency-based (Section 3.5.4) covert channels.

3.5.3 GPU Power-Based Covert Channel

Power-based covert channels are difficult to construct in NVIDIA GPUs as the built-in power sensors are inherently error-prone. A recent study [163] found the error rate in NVIDIA's built-in sensors to be 5% as only 25% of the runtime is sampled for power consumption. The coarse-grained sensory data of NVIDIA GPU platforms make it challenging to design accurate side or covert channels. In this work, we show that it is still possible to build power-based covert channels with such coarse-grained sensory data.

Algorithm 1. Trojan Gadget for GPU Power Covert Channel

```
Input: bit_pair_1 ... bit_pair_N
  function PowerTrojanFunction(bit_pair[ ])
    N ← length(bit_pair)
    for i ← 1 to N do
      if bit_pair_i = 00 then
        launch abs.f32 kernel
      else if bit_pair_i = 01 then
        launch sad.s32 kernel
      else if bit_pair_i = 10 then
        launch ex.f32 kernel
      else if bit_pair_i = 11 then
        launch addc.s64 kernel
      end if
      CudaDeviceSynchronize( )
      Sleep(1500ms)
    end for
  end function
```

Single-bit Transmission

With the coarse-grained power measurements, it is impractical to rely on fine-grained power fluctuations during individual instruction executions. Instead, we propose to leverage our Observation 1, which showed the diverse maximum power levels of different instructions. From our observation, we found that instructions inherently consume notably different amounts of power. If the attack gadget runs each instruction sufficiently many times, similar to our micro-kernels, to enable the built-in sensors to capture the maximum power level of the instruction, we can encode information by leveraging the different power levels. One simple example covert channel is to use just one instruction. Suppose that the trojan and the spy know what instruction is used in the covert channel gadget and how much power is dissipated when the gadget is executed.

Trojan Methodology: The trojan delivers binary information by either executing or not executing the gadget regularly (i.e., bit 1 is encoded by executing the gadget, bit 0 is encoded by not executing the gadget).

Spy Methodology: Spy will monitor the power sensor data and recognize the information by simply checking if the sensor data reaches the anticipated power level.

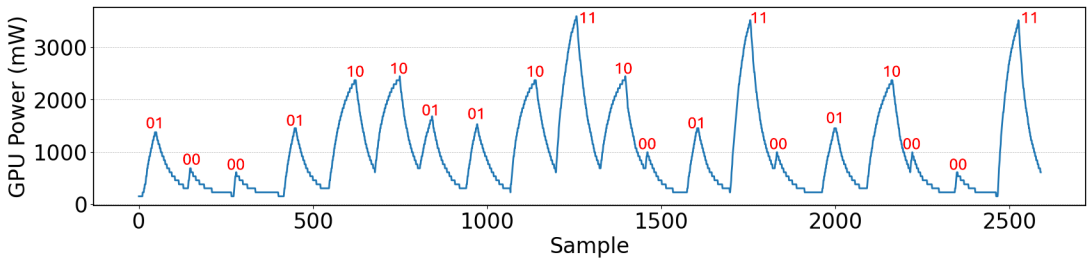


Figure 3.11. Sending Bits 01000001101001011011100001110001100000011 through GPU Power-Based Covert Channel

Multi-bit Transmission

By leveraging our observation that shows the notably different power levels across instructions, we further propose to deliver multi-bit information. To build the multi-bit power-based covert channel, we should choose a combination of instructions, which consume significantly different power from each other. As a prototype, we use four different instructions and run the micro-benchmark for each instruction as described in Section 3.3, to produce a stable power level for each instruction. Each micro-benchmark runs one instruction for 80K times. It is assumed that the trojan and the spy know the instructions that are used in the covert channel gadget and their power consumptions when the gadget is executed.

Trojan Methodology: Algorithm 1 shows our power-based covert channel. We use four PTX instructions `addc.s64`, `ex2.f32`, `sad.s32`, and `abs.f32` to transmit a 2-bit information 11, 10, 01, and 00, respectively. Note that this is just an example; any combination of instructions can be used as long as the instructions' power levels are distinguishable. After each 2-bit transmission, we use a sleep period of 1500 ms to reduce noise from the transmission by dropping the GPU power consumption.

Spy Methodology: The receiver (i.e., spy) passively keeps reading the GPU power sensor and interprets the different power peaks as different 2-bit information, as shown in Figure 3.11.

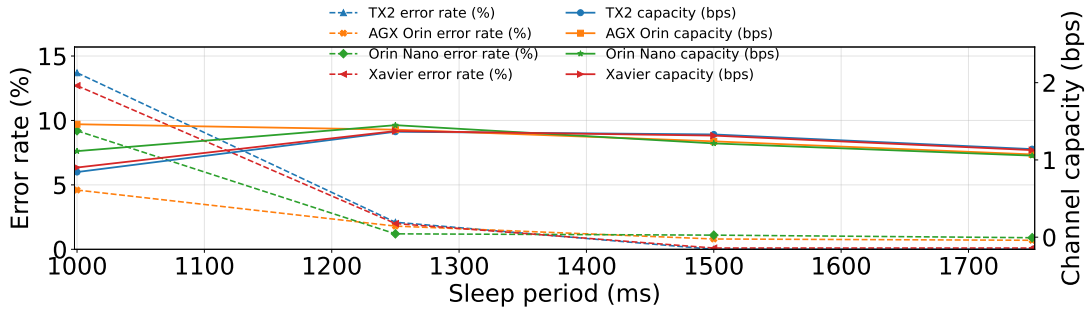


Figure 3.12. Effectiveness of GPU Power Covert Channel with Diverse Sleep Periods: Shorter sleep period between data transfer improves attack bandwidth but increases noise due to the coarse-grained power measurements of NVIDIA built-in sensors. The attacker can expect the best leakage performance by setting the sleep period to between 1250 ms and 1500 ms, which produces the highest leakage bandwidth (solid lines) at a low error rate (dotted lines).

In our experiment, the receiver monitors data transmission every 1 ms by using `tegrastats` with a sampling interval of 1 ms.

From this attack, we achieved a superior covert channel bandwidth, 1.3 bps, compared to earlier studies that showed 10 times lower bandwidth, 0.16 bps [60]. We observed that this covert channel induced 0% bit error rate when we transmitted 1K random bits in the Jetson TX2 board. The error rate was calculated using the Levenshtein edit distance [113] following standard methodology [120].

Error rates: The attack effectiveness is determined based on the leakage bandwidth and error rates, which are controlled by sleep lengths between transmissions. We evaluated the bandwidth and error rate across different platforms by testing the proposed attack gadget while changing the sleep period from 1000 ms to 1750 ms, as shown in Figure 3.12. The longer sleep period leads to a lower error rate due to less interference between signals. However, as the longer sleep period elongates the data transfer interval, the information leakage bandwidth also drops. Therefore, a sleep period should be set to produce a good leakage bandwidth at a sufficiently low error rate. We plot the error rate (dotted lines) with the effective channel bandwidth (solid lines) together in Figure 3.12. The effective channel bandwidth is calculated based on the binary symmetric channel model [121]. Suppose raw bandwidth is B and error rate is p, the channel

Algorithm 2. Trojan Gadget for Frequency Covert Channel

```
Input: bit_1 ... bit_N
  function FREQUENCYTROJANFUNCTION(bit[ ])
      N ← length(bit)
      for i ← 1 to N do
          if bit_i = 0 then
              launch abs.f32 kernel with 8000 iterations
          else
              launch abs.f32 kernel with 80000 iterations
          end if
          CUDADEVICESYNCHRONIZE( )
          SLEEP(100ms)
      end for
  end function
```

capacity C (effective bandwidth) is calculated as $C = B \cdot (1 + ((1-p) \cdot \log_2(1-p) + p \cdot \log_2(p)))$. In all four tested platforms, the error rate drops linearly when a longer sleep period is used; at around 1250 ms, the error rates become almost saturated. The effective channel capacity increases until the sleep period becomes 1250 ms due to the quickly dropping error rate. Then, the capacity drops slightly until 1500 ms, and then decreases more significantly afterwards due to the saturated error rate and slower data transmission. This means that after error rates are saturated, the additional sleep period only increases the latency to the transmission. Therefore, around 1250 - 1500 ms can be the best candidate sleep period. We used 1500 ms for the attack evaluation.

3.5.4 GPU Frequency-Based Covert Channel

While the GPU power-based covert channels have been considered impractical due to the coarse-grained sensor measurements, GPU frequency-based covert channels have been implemented by several studies. One limitation of frequency-based attacks is the low bandwidth due to the challenges of controlling the frequency especially on GPUs. GPU dynamic frequency scaling requires hundreds of milliseconds to transition from one frequency level to another, whereas CPU DVFS reacts within microseconds [150]. Hot Pixels [141] achieved only up to

0.1 bps from their GPU frequency-based covert channels on the Apple integrated GPUs and NVIDIA discrete GPUs. Veiled Pathways [112] achieved slightly higher bandwidth, 0.5 bps by using GPU on-device memory frequency, rather than GPU frequency for covert channels on NVIDIA discrete GPUs. However, as the GPU DRAM frequency stays high even after a kernel finishes, they had to use `cudaDeviceReset` CUDA API function [34] to force drop the DRAM frequency. The requirement of frequent usage of reset APIs makes the attack ineffective; as the APIs release all the allocated memories and reset the DRAM frequency, the attack is easily detected and mitigated by simply monitoring such bulky memory release and DRAM frequency reset.

To develop a high-bandwidth covert channel, we do not directly use the GPU frequency as the attack source. The `tegrastats` utility has a `GR3D_FREQ` statistic that reports both the current running frequency of the GPU engine and how much of this frequency is actually used at a given time. According to our Observation 6, some instructions, such as `abs.f32` and `mad.rn.f32`, consume notably different GPU frequency utilization, depending on the compute loads. Inspired by this observation, we propose to use the frequency utilization instead of the current running GPU frequency. As the frequency utilization is controllable by simply launching different amounts of executions of certain instructions that show diverse frequency utilization, using the frequency utilization enables a more proactive and fine-grained controllable attack model. Note that GPU frequency is relatively hard to control and has higher noise as the frequency is determined based on the target platform's dynamic voltage-frequency scaling algorithm in coarser-grained intervals. For example, while the GPU frequency stays high for quite some time even after a kernel finishes, the frequency utilization quickly returns to the idle state as soon as the computation finishes.

Trojan Methodology: Algorithm 2 shows the proposed trojan gadget that sends an N-bit information by leveraging the frequency utilization. Bit 0 and 1 are encoded by running a different number of `abs.f32`. Any other instructions can be used as long as the instruction shows notably different frequency utilization depending on the compute loads. After each bit

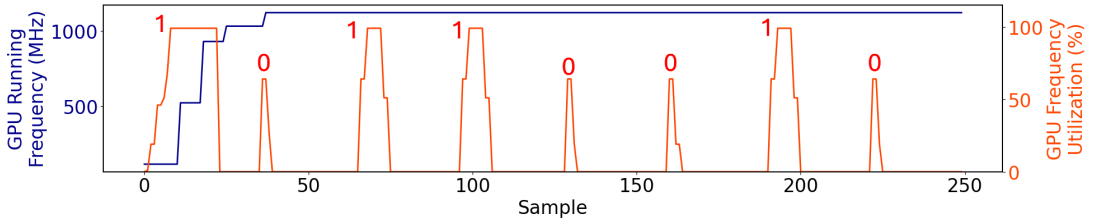


Figure 3.13. Sending Bits 10110010 through GPU Frequency-Based Covert Channel

Table 3.4. Comparison of GPU Frequency-Based Covert Channels

Study	Leakage Source	Tested Architecture	Bandwidth
Hot Pixels [141]	GPU engine frequency (Hz)	Apple iGPUS and NVIDIA dGPUs	0.1 bits/s
Veiled Pathways [112]	GPU DRAM frequency (Hz)	NVIDIA dGPUs	0.5 bits/s
Our proposed approach	GPU engine frequency utilization (%)	NVIDIA iGPUs	upto 15.65 bits/s

transmission, we used a sleep period of 100 ms to drop the frequency utilization to reduce noise.

Spy Methodology: Spy will monitor the GR3D_FREQ value of `tegrastats` and recognize the bit values. In our experiment, the receiver samples the frequency data every 1 ms by using `tegrastats` with a sampling interval of 1 ms. The spy sanitizes the collected frequency trace by clamping frequency utilization values below 20% to zero, which removes idle-state jitter and small frequency fluctuations. It then scans the time stamps and extracts a bit whenever the trace rises and then falls, recording the local maximum of that segment as the peak value for one transmitted bit. Finally, the spy applies a threshold-based decoding: if the peak is greater than 20% but less than 90%, the symbol is classified as bit 0; if the peak is above 90%, it is classified as bit 1. Note that `tegrastats` can sometimes miss some samples [39], but this didn't affect the receiver's ability to interpret the transmitted bits.

Calibration procedure: Before measuring channel performance, we calibrate the decoding thresholds using a short training transmission with known symbols, similar to earlier studies [77]. The trojan first sends a short preamble of a pre-determined number of repeated 0s and repeated 1s. The spy records the resulting frequency utilization peaks and determines the approximate peak ranges for each class.

Figure 3.13 depicts an example attack scenario when the trojan sends an 8-bit information

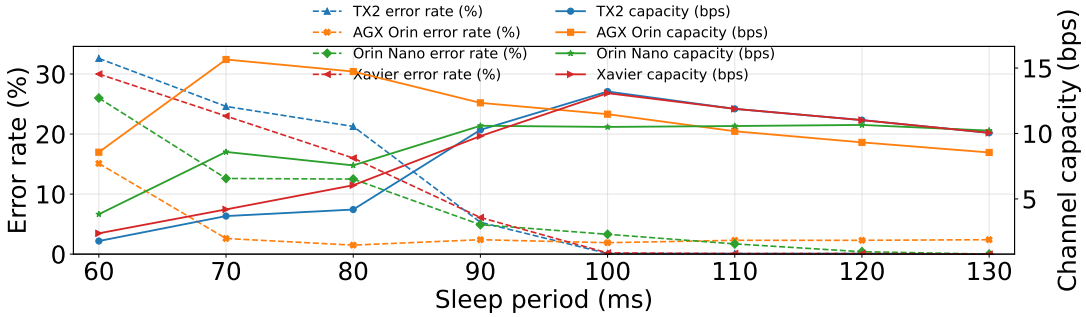


Figure 3.14. Effectiveness of GPU Frequency Utilization-Based Covert Channel with Diverse Sleep Periods: Shorter sleep period between data transfer improves attack bandwidth but increases noise. The attacker can expect the best leakage by setting the sleep period to around 100 ms for TX2, Orin Nano, and Xavier, and 70 ms for AGX Orin, which produces the highest leakage bandwidth (solid lines) at a low error rate (dotted lines).

(10110010) to the spy. The GPU running frequency (blue line) gradually reaches 1122 MHz and stays at this peak until after the attack gadget finishes its computation. So if we want to use the GPU running frequency as the leakage source, we have to either use a much longer sleep period or force the frequency to drop using reset APIs such as `cudaDeviceReset` explicitly, which is time-consuming and detectable. Even when the individual bit transfer can be distinguished with a longer sleep period, the running frequency level cannot clearly encode bits either 0s or 1s as the peak frequency is the same (1122 MHz) for both bit 0 and 1. The earlier studies, such as Veiled Pathways [112], manipulated the duration of the peak frequency to encode different bits, which makes it slow (i.e., low bandwidth) and error-prone. On the other hand, in our approach, GPU frequency utilization (orange line) reaches a peak of 99% while transmitting bit 1, and 64% for delivering bit 0. As the frequency utilization changes promptly, as shown in the figure, our attack model shows a much higher bandwidth. When we transmitted 1K random bits in the Jetson TX2 board, we achieved 13.36 bps with 0.1% bit error rate while for AGX Orin achieved upto 15.65 bps capacity. Table 3.4 shows a comparison of different frequency-based channels with our approach. Our approach shows 31× to 156× higher leakage bandwidth compared to two state-of-the-art solutions [141, 112].

Error rates: Similar to the power-based covert channel, we also evaluated error rates

and leakage bandwidth while changing sleep periods. The result is shown in Figure 3.14, where bandwidth and error rates are the average of 10 rounds of covert communication for each sleep period. Across all four platforms, increasing the sleep period reduces the error rate while the channel bandwidth increases until the error rate becomes saturated. However, the best sleep periods are slightly different. In AGX Orin, at around 70 ms, the channel capacity peaks (15.65 bps) and the error rate significantly stops decreasing further. Therefore, 70 ms is the best sleep period for the effective attack outcome. On the other hand, the other platforms, TX2, Orin Nano, and Xavier show the best attack bandwidth at around 100 ms. We believe such disparity is sourced from the advancement of SoC design. As AGX Orin is the latest Jetson platform among the four tested platforms, it provides a more stable dynamic frequency scaling, which enables significantly lower error rates at a shorter sleep period than older generations. It is noteworthy that our novel GPU frequency utilization-based covert channel requires an even shorter sleep period than our proposed GPU power-based covert channel, which requires 15× longer sleep period to enable stable attack bandwidth than frequency utilization-based covert channel.

3.5.5 DLA Frequency-Based Covert Channel

So far, we have demonstrated covert channels leveraging GPU statistics. We show that our observations can be used to effectively mitigate the limitations of existing attack models to enable high-bandwidth and error-resilient covert channels. In this subsection, we would like to step forward by demonstrating a novel covert channel utilizing the integrated accelerators. We will show that the integrated accelerator, such as DLA in NVIDIA Jetson platforms, can also be a source of attack. To the best of our knowledge, this is the first covert channel that uses DLA.

Inspired by our Observation 7, we exploit DLA frequency to build a covert channel. As shown in Figure 3.9, DLA frequency peaks during the DLA execution and immediately drops upon completion of the kernel. This characteristic enables almost noiseless indication of computing activities on DLA. Other sensory data, such as DLA power statistics, can be noisy because the power sensors of Jetson boards do not provide DLA's power consumption separately

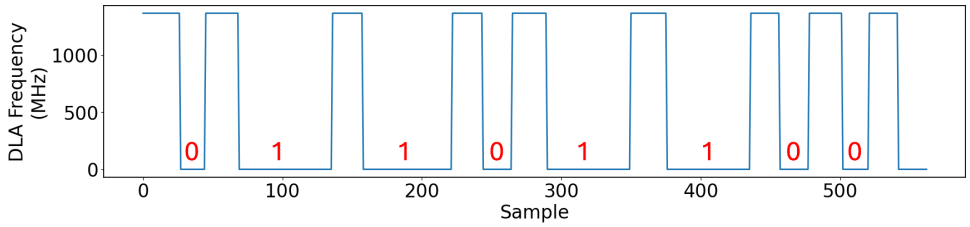


Figure 3.15. Sending Bits 01101100 through DLA Frequency-Based Covert Channel

but with the CPU power consumption.

To build a DLA frequency-based covert channel, we can encode the information either with the length of DLA's active cycles, where the frequency reaches the maximum value, or with that of DLA's idle cycles, where the frequency stays zero. Between them, we use idle cycles because the active cycles are harder to control, as we should identify the convolution layer sizes that derive notably different execution times. Instead of finding and using different convolution kernels, we simply place variable-length idle times between any two convolution executions to encode information.

Trojan Methodology: To control the DLA core frequency, the trojan process keeps launching a 1×1 convolution layer in the DLA core and adds a sleep period between two consecutive convolution workloads. The sleep period essentially encodes the secret data. In our prototype, the sleep period is set to 0.7 and 1 seconds to encode bits 0 and 1, respectively.

Spy Methodology: The receiver spy keeps checking the DLA frequency from the built-in sensor regularly (e.g., every 5 ms in our evaluation).

Figure 3.15 shows the DLA core frequency observed by the spy process when the trojan process tries to covertly send 01101100 as the message. The short period when DLA frequency is zero is interpreted as bit 0, and the longer period when DLA frequency is zero is interpreted as bit 1. The high DLA frequency period acts as a boundary between two bits. The bandwidth of this covert channel is 1.13 bps with 0% bit error rate when we transmitted 1K random bits in the Jetson AGX Orin board. We further improved the bandwidth to 2.27 bps by using both the DLA cores in parallel. We also tested the DLA frequency-based covert channel in an NVIDIA

Jetson AGX Xavier platform achieving an average bandwidth of 2.1 bps with 0% bit error rate. Our DLA frequency-based covert channel has 4× higher bandwidth than other special-purpose engine-based covert channels, such as NVENC-based covert channels proposed by [112].

3.6 Discussion

3.6.1 Limitations

Our attack models implement error-tolerant covert channels, achieving superior bandwidth compared to the state-of-the-art solutions. However, there are still some limitations.

Long Sleep Time for Noiseless Transmission

Our covert channel attacks require the power and frequency to return to an idle state after each round of transmission to minimize the interference among the transmissions. While the frequency relatively quickly returns to the idle state, the power takes much longer time to become idle. Therefore, in our experiment, we had to use a sleep period of 1500 ms after transmitting each bit pair through the power covert channel without compromising the bit error rate, as shown in Algorithm 1. Due to such a long idle time, the power-based covert channel shows relatively low bandwidth, 1.3 bps. We leave the task of building a faster power-based covert channel as future work.

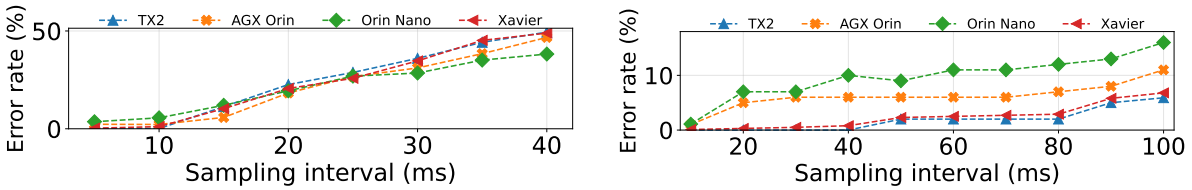
Inter-Kernel Interference

To avoid inter-kernel interference, we leverage dynamic parallelism, where a parent trojan kernel internally forks micro-kernels that deliver information, which prevents another application from being launched on the GPU. While this offers the trojan kernel a noiseless window to covertly send data, this window is still limited because, in the edge GPUs where MPS is not supported, the GPU is shared in a time-sliced manner due to GPU preemption feature [46]. To evaluate the effect of noise, we ran our power-based covert transmission of 1 Kb data while running a set of general-purpose GPU applications and machine learning

models. For the general-purpoase applications, we used 14 CUDA applications from Rodinia benchmark [55]. For the machine learning (ML) workloads, we continuously ran inference tasks with 6 YOLO detection models [129] (YOLO11 and YOLO26 families with nano, small, and medium variants [79]) on COCO dataset [100]. With the introduction of noise, the bit error rate was changed from 0% to 11.2% when general-purpose applications are used, and to 9.3% when ML models were used on TX2. We found that when multiple GPU workloads are scheduled, our trojan kernel was allowed to run for a period of $\sim$30 seconds before being preempted. This means that, to ensure an absolute noiseless covert communication, the entire communication should be split into multiple intermittent sessions where each session is finished within a time slice of 30 seconds.

Multi-bit Transmission Limitation

We demonstrated multi-bit leakage in Section 3.5.3. Though we showed 2-bit encoding as an example, our attack model can theoretically encode more bits by mapping multiple instructions to distinct power states. However, as more distinct power levels must be used in that case, the encoded data could be incorrectly decoded unless the power levels are sufficiently different. As more instructions are used for generating the power signals, the power gap between adjacent instruction-induced states becomes smaller, reducing the distinguishability of the transmitted symbols. Given the inherent noise by power and frequency sensors as well as the runtime dynamics that cause extra noise, it will be challenging to deliver longer bit values without significant errors. For instance, we tested 3-bit encoding by selecting eight instructions with distinct power consumption profiles. Although this configuration increases the raw transmission bandwidth to 1.9 bps, the higher bit error rate significantly reduces the effective channel capacity. After accounting for decoding errors, the maximum achievable capacity across all four evaluated platforms is limited to 1.25 bps. These results indicate that while multi-level modulation can improve covert channel transmission bandwidth, the effective information leakage will be limited due to a higher error rate.



(a) Frequency covert channel

(b) Power covert channel

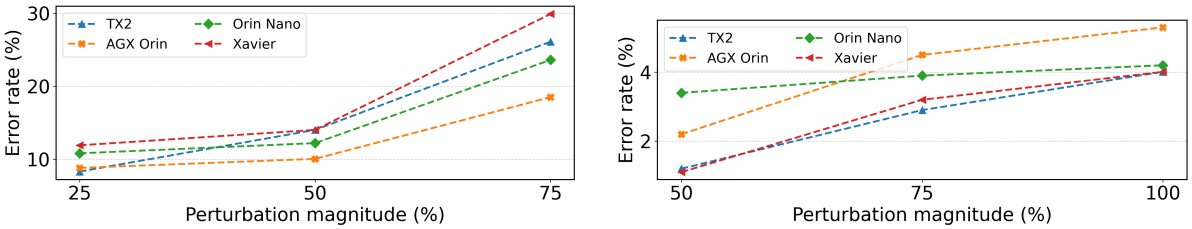
Figure 3.16. Impact of limiting the sampling rate of telemetry monitors.

3.6.2 Mitigations

Unlike other covert channels that induce shared resource contention and hence the communication is detectable through resource monitoring, our covert channels do not require resource contention between the trojan and spy processes. Therefore, the mitigations designed for contention-based covert channels like GPUGuard [159] will not work. Temporal partitioning-based mitigations [154] are also ineffective because we use dynamic parallelism to reserve the GPU resource as much as possible, and also the attack is short enough to be successfully tested on the edge platforms that already implement temporal resource sharing (30-second time quota as discussed in Section 3.6.1). Here, we discuss some mitigation strategies for the proposed security attacks.

1. Access Restriction: To prevent unprivileged attackers from building the power and frequency-based covert channels, the power and frequency monitors (e.g., tegrastats and jtop) can be restricted to be used by priviliged user. This might be effective, but could unnecessarily limit the useful usages of the sensory data.

2. Resolution Restriction: Limiting the sampling frequency of the power and frequency monitor can degrade the covert channel accuracy and bandwidth. `tegrastats` allows monitoring different system statistics like frequency, power, temperature, and memory usage etc. at regular intervals. The sampling interval can be specified in milliseconds (ms), and it can be as short as 1 ms. The default sampling interval is 1 ms, which is used for our other evaluations. We evaluated the effect of limiting the sampling rate on different covert channels. We ran our proposed attack



(a) Frequency covert channel

(b) Power covert channel

Figure 3.17. Impact of telemetry perturbation on covert-channel error rate.

gadget using the same trojan-spy methodology used in Figure 3.11 and 3.13. Figure 3.16a depicts that the frequency covert channel can retain its low error rate even when the sampling interval is restricted to 10 ms (0% bit error rate in case of TX2). However, once the sampling interval exceeds 10 ms, the error rate increases approximately linearly across all platforms. This occurs because longer sampling intervals begin to miss short-lived GPU frequency transitions that encode covert bits. At 40 ms, the frequency-based covert channel loses accuracy by almost half, leading to an almost random guess. Figure 3.16b presents the results for the power-based covert channel. Compared to frequency telemetry, power telemetry appears significantly more resilient to sampling-rate restriction. Even when the sampling interval is increased to 100 ms, the observed error rates remain relatively low (e.g. 5.9% for TX2, and 6.8% for Xavier). This robustness stems from the fact that our power-based covert channel uses longer intervals and lengths of data transmissions (i.e., sleep periods and the time persisting the encoded power value) than the frequency-based channel, which makes it possible to capture data correctly even with coarse samplings. The power-based attack uses a longer transmission window due to the longer transition time between power values than frequency, which limits the attack bandwidth. However, such a limitation makes the attack more robust. Overall, these results indicate that resolution restriction alone is insufficient to eliminate our proposed covert channels, particularly the power-based attack.

3. Telemetry Perturbation: Instead of completely restricting the telemetry monitoring utility to privileged users or reducing the monitoring resolution, calibrated perturbation (software

dither) can be added to telemetry only for unprivileged users. For a privileged user, the true telemetry can be reported. We evaluated a lightweight randomized telemetry perturbation mechanism that reduces the fidelity of observable GPU telemetry (i.e., power and frequency data in our attack model) without modifying the underlying hardware behavior. Specifically, we propose to add a new software module, a noise injection function, to the `tegrastats`. The power and frequency sensors pass the telemetry signals to the noise injection module and the module injects low-probability transient spikes, emulating natural background interference. For each sample, we select a small probability (p) of telemetry values and perturb the selected signals with a pre-determined noise magnitude (M). For example, we used 2% probability, which means every 100 signals, 2 signals are chosen to be perturbed. The noise magnitude is determined based on the scales of frequency and power values of signals. For example, we used 50% as the baseline frequency noise magnitude for all platforms, as frequency utilization scales up to 100% for all platforms. For power noise magnitude, we used 1000 mW for TX2 and Xavier, and 400 mW for AGX Orin and Orin Nano as the baseline noise magnitude, which is the power gap between any two signals. We evaluated this mitigation by using the same gadget and data that were used in the resolution restriction mitigation evaluation. As a sensitivity study, we varied the magnitude values by taking 25%, 50%, and 75% of the base magnitude for frequency attack mitigation and 50%, 75%, and 100% of the baseline magnitude for power attack mitigation. Figure 3.17 shows the results. With telemetry perturbation, the error rate increases significantly: for frequency-based channels, the error rate rises to as high as 14.01% and 29.9% with 50% and 75% perturbation, respectively. This demonstrates that a small number of randomized perturbations on the frequency data substantially degrade the attack accuracy. Unlike heavyweight approaches such as restricting access or reducing telemetry resolution, the proposed perturbation mechanism operates entirely in software and maintains realistic signal characteristics, making it a practical defense against passive observers relying on high-resolution GPU telemetry traces. However, this mitigation remains insufficient for power-based channels, as the observed error rates remain below 5.4% across the evaluated platforms, even at 100%

magnitude, as shown in Figure 3.17b.

4. MPS Support: Adding MPS support for edge GPUs can allow multiple kernels to run concurrently on the same GPU. This can induce noise from benign applications to the attacker-encoded power and frequency data.

Summary: While the proposed mitigations could make it challenging to design an accurate covert channel, they inversely limit the benign uses of those tools and sensory data significantly. Also, MPS cannot be an ultimate solution because contention-based covert channel attacks leverage MPS as the attack source. Thus, more efforts are needed to build practical mitigations for our proposed novel covert channels.

3.7 Conclusion

In this work, we show an in-depth characterization of power and frequency behaviors of individual instructions and applications on GPUs and DLAs on NVIDIA edge platforms. Based on the observations, we discover the security vulnerabilities of edge platforms; we demonstrate three novel covert channels, which are more noise-tolerant and have high bandwidth compared to existing attack models. We suggest various mitigations such as limiting the accessibility, resolution, and accuracy of the sensor data monitoring utility. However, none of them successfully alleviates all of our proposed covert channels; a more comprehensive approach will be needed.

Chapter 4

SCPC: Securing Cross-Process Collision-Based Transient Attacks

4.1 Introduction

Since Spectre and Meltdown have been introduced, transient execution attacks have been growing in numbers, targeting various microarchitecture components such as the branch predictor, branch target buffer, caches, etc. [88, 101, 54, 157, 127, 114, 130, 48]. The intuition behind all these attacks is the ability of an attacker to steer the speculative execution of a victim process by mis-training a prediction unit. Through separate covert or side channels, such as shared caches, attackers can identify the secret data.

Recently, a new out-of-place transient attack, namely Spectre-CTL, targeting speculative store bypass predictor (SSBP) has been discovered [102]. SSBP is incorporated in several generations of AMD CPUs to help run memory instructions out of order by predicting whether a load instruction can bypass an earlier store instruction before the dependency between the load and store pair is resolved. SSBP runs an eight-state finite state machine (FSM) to predict dependencies of each pair of load and store instructions. Once the attacker acquires an SSBP entry that aliases with a victim process, the attacker can not only steer the victim to misspeculatively bypass a dependent store instruction but also identify the secret data by checking the updated FSM state. This attack differs from Spectre-v4 (also known as Spectre-STL) in that the attacker and victim do not need to share the address space, making this attack more feasible in real

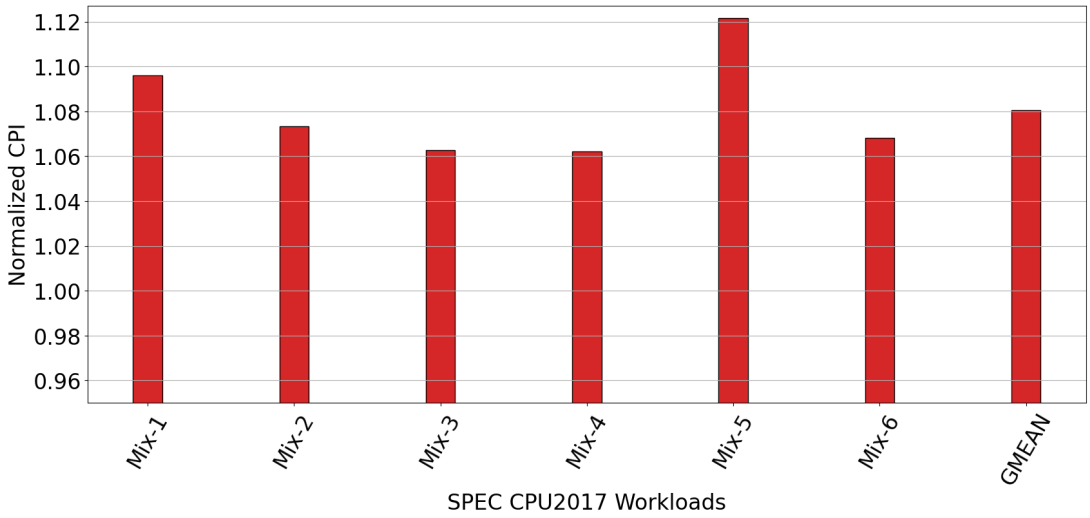


Figure 4.1. Normalized cycles per instruction (lower is better) when SSBP is disabled.

systems [102].

This *Spectre-CTL attack* garnered significant attention because of its high impact (applicable for multiple AMD processors) and its criticalness (unlike other transient execution attacks, it does not require an additional channel to deliver the secret; the SSBP's internal state change is sufficient to identify the secret value). Therefore, the existing mitigations on the secret delivery channels (e.g., shared caches) cannot completely prevent information leakage from the Spectre-CTL attack. It is also undesirable to disable SSBP for security purposes because SSBP contributes significantly to performance. Figure 4.1 shows the performance drop (average 8% and up to 12%) when SSBP is disabled (details in Section 4.5). Mitigations for Spectre-STL, such as using memory fence instructions, have similar (or worse) performance issues.

To tackle the root source of leakage, we can consider two approaches: 1) isolating SSBP for each process and 2) hiding SSBP state changes from other processes. Isolating SSBP entries could fundamentally obstruct any information sharing among processes. However, a proactive resource partitioning (regardless of static or dynamic partitioning) needs a clear understanding of resource utilization of the target applications [151, 143]; otherwise, sub-optimally partitioned resources will cause unnecessary resource contention, thereby leading to significant performance overhead for benign executions. Due to the limited resource capacity, it is especially challenging

to use partitioning for most small on-core components, such as the SSBP. Instead, if the state transition is hidden from other processes, the malicious process can only cause a load to bypass a store, but cannot identify the secret without an extra side or covert channel. Those channels can be alleviated by other conventional solutions [155, 106, 166, 86, 171, 75, 43, 156, 104, 123, 62].

In this work, we propose *SCPC*, the *first Spectre-CTL attack mitigation*. SCPC limits the visibility of SSBP state transitions among processes, without explicit resource partitioning. We propose to hide only certain state transitions that can be leveraged for secret leakage. In the Spectre-CTL attack, an attacker owns and trains an SSBP entry and catches the secret when a victim, which shares the same entry, updates the entry's FSM state upon a misspeculation. Thus, out of all possible transitions, we hide the one that is triggered by a sharer's misspeculation. Upon a misspeculation, SCPC spawns a new SSBP entry to protect the shared SSBP entry's state from being updated. This mechanism allows the corresponding state transition to occur in the new entry, while the original shared entry remains untouched. Once a process is linked to a new entry, it no longer uses the previously shared entry. Therefore, the victim's speculative execution results are not visible to the attacker.

As SCPC isolates SSBP selectively upon a certain condition, processes are technically allowed to benefit from the SSBP full capacity, unlike explicit resource partitioning. SSBP entries are shared by multiple processes, allowing benign information sharing (e.g., sharing predictions among load-store pairs that have similar dependency behaviors without separate training). Note that other transient execution attack mitigations, such as using complex or multiple hashing algorithms [81, 171] are orthogonal to SCPC. While complex hashing makes it challenging to find the aliased resources, SCPC blocks the secret leakage by isolating suspicious resource sharing.

To enable the victim process to use a new SSBP entry while its load address aliases with the attacker's SSBP entry, SCPC incorporates a small *virtual table*. Similar to a page table in a virtual memory system that maps a virtual page address to a physical page address, the virtual table maps a hashed load instruction address to an SSBP entry index. Unlike virtual memory, a

new mapping is created selectively under a certain condition. Therefore, we name it *selective virtualization (S-VLT)*. Each virtualized mapping is associated with a specific process. Therefore, the other sharers can continue using the previously shared SSBP entry.

S-VLT disables the sharing of the insecure FSM state transition among processes and effectively reduces resource contention of explicit resource sharing. However, S-VLT could still suffer from insecure resource utilization and resource starvation, if one process dominates all entries and evicts other processes' entries. To tackle this, SCPC incorporates *self-replacement* and *self-invalidation mechanisms*. The self-replacement makes each process replace its owned entries only, and the self-invalidation makes each process voluntarily release its own entries after a certain threshold. The released entries can be used by any process. Note that the conventional replacement policies such as least recently used (LRU) can also resolve the starvation problem, but it is known to be vulnerable because the eviction sequence can be another source of leakage.

Another potential downfall of S-VLT is the hardware and performance overhead introduced by the indirect SSBP access through the virtual table. To alleviate the overhead, we introduce a cost-effective S-VLT design. The cost-effective S-VLT promotes the virtual table as a secondary SSBP table by directly storing the SSBP FSM states instead of storing mapping information. By allowing parallel accesses to both SSBP and virtual table, the cost-effective S-VLT reduces both area and performance overhead.

While our primary target is the Spectre-CTL mitigation, SCPC is more generally applicable for other cross-process resource collision-based transient attacks. We demonstrate the generality of SCPC by adapting it to protect the branch history table (BHT). This means that, while the idea is generic, SCPC can still be selectively optimized for different predictor structures. For example, our mechanism can be used alongside a mitigation for branch target buffer protection [94], while our defense secures the SSBP. This generic but selectively tailorable idea helps SCPC achieve significantly better performance than generic mitigations [155, 166], which obstruct all potential channels across all predictors.

Our evaluation leverages 6 workload-mixtures, where each mixture includes 8 bench-

marks from the SPEC CPU 2017 benchmarks [52] to show that SCPC achieves superior performance and reduced attack surface compared to state-of-the-art defenses for branch predictors [171] and simultaneous multi threading (SMT) [143], when the proposed schemes are adapted to SSBP. We also show superior performance than the security oracle solutions, such as static partitioning and SSBP disabling, as well as a generic mitigation [166].

Our contributions are as follows:

- To the best of our knowledge, **this work proposes the first Spectre-CTL mitigation**, which can protect CPUs using SSBP-like memory disambiguation units.
- **Our proposed SCPC mitigation reduces the amount of information leakage and the risks of resource contention and starvation.** Our security evaluation demonstrates superior protection capability of SCPC with zero attack successes for an attack gadget designed following earlier study [171], compared to the state-of-the-art solution [171], dynamic resource partitioning [143], and flushing.
- **SCPC allows applications to benefit from the full capacity of the shared resources.** Unlike conventional isolation mechanisms, the proposed mitigation allows applications to continue sharing benign predictions, which leads to superior performance than resource partitioning-based mitigations.
- **SCPC's mechanism is more generally applicable to various cross-process collision-based transient attacks.** We demonstrate the generality with a BHT-based implementation in Section 4.6.
- Unlike most other mitigations, **SCPC results in almost zero performance overhead**, while notably reducing the attack surface.

4.2 Background & Motivation

4.2.1 Transient Execution Attacks

Meltdown [101] and Spectre [88] were the first attacks disclosed in the category of transient execution attacks that leverage delayed exception handler execution and branch predictor, respectively. Both attacks create speculative execution windows by mistraining prediction units, during which attackers can access secret data. Then the accessed secret can be retrieved using cache side-channel attacks, such as Flush+Reload [165].

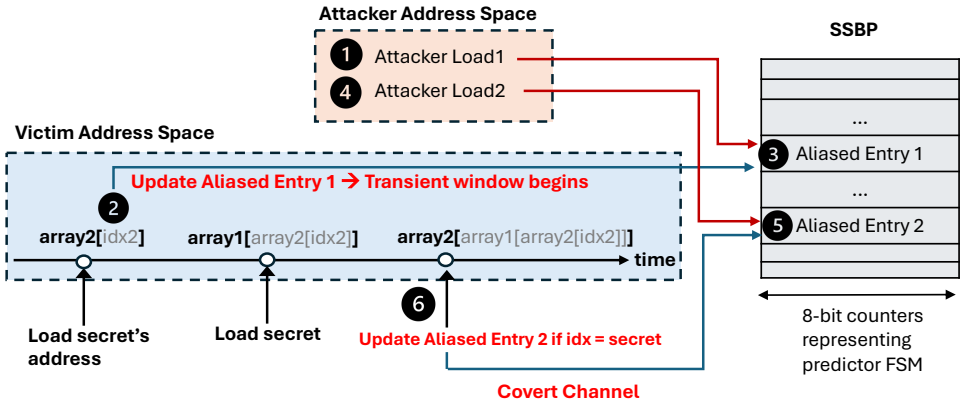
While original transient attacks leverage cache side-channels to extract the secret data, several studies demonstrate variants of transient attacks using various other microarchitecture components [102, 98, 66, 54, 57, 88, 169]. These studies demonstrate that any resources shared across processes can be a potential source of leakage. In this paper we focus primarily on defending against attacks that leverage memory disambiguation units.

4.2.2 Transient Execution Attacks leveraging the Memory Disambiguation Unit

Modern out-of-order processors use several optimizations to hide pipeline stalls. One such optimization is the memory disambiguation unit (MDU). The MDU helps run memory instructions out of order. Typically when a store instruction's address takes time to be calculated, any younger load instruction's address may be aliased with the awaiting store instruction's address. The MDU predicts whether a load instruction has data dependencies with any older store instruction. If a data dependency is predicted (i.e. store-load pair's addresses are the same), the younger load instruction waits for the store instruction to verify that their addresses actually match, in the conservative approach. A more aggressive strategy allows the load to issue before the dependency is confirmed, and the load simply uses the value directly from the store instead of reading it from the cache. If no data dependency is predicted, the younger load instruction is speculatively executed, bypassing the store instruction. If this prediction is wrong, the load and

```
void victim_function() // idx == idx2
{
    array2[idx] = 0;
    temp = array2[array1[array2[idx2]]]
}
```

(a) Spectre-CTL Gadget



(b) Spectre-CTL [102]

Figure 4.2. Spectre-CTL Attack Gadget and Execution Flow.

its dependent younger instructions are flushed. Typically the MDU is implemented as a table of counters which is indexed with the address of the load instruction. For example, AMD uses a combination of a Predictive Store Forwarding (PSF) and Speculative Store Bypass Predictor (SSBP) table [44, 102]. The PSF is used for a single load following a single store. The SSBP, which is indexed with the IPA (Instruction Physical Address) of the load, is used to compare against a number of preceding stores. Each SSBP entry uses a FSM to predict, and the states are represented with an 8-bit value. Upon a memory order violation (incorrect bypassing of a store), the state is updated to reflect the aliasing nature of the load instruction and subsequent loads simply fetch the value from the predicted aliasing store instruction (as described in the aggressive approach). When two aliasing mis-predictions are recorded, the predictor falls back to forcing subsequent load instructions to block and wait until the address of both instructions (load and store) can be confirmed to be the same (the conservative approach).

AMD Spectre-CTL: Recently, a study leveraged an SSBP as a side-channel source [102]. This attack exploits the resulting values fetched with a load instruction upon the SSBP state

transition after misspeculation. Figure 4.2 shows the attack gadget and the interaction with the SSBP, where the nested array accesses in the attack gadget (Figure 4.2a) are illustrated on the timeline in the blue box in Figure 4.2b.

Suppose that the target secret is in `array1[array2[idx2]]` (data loaded by the 2nd load in the timeline). To initiate a transient window within the victim gadget, the attacker first finds a store-load pair within its address space so that both the attacker-controlled load ❶ and the victim gadget's 1st load ❷ are mapped to the same SSBP entry ❸ (Aliased Entry 1 in Figure 4.2b). To achieve this collision, the attacker process must train its entry and wait for the victim load to access and potentially update the same entry. If the attacker later finds the same entry's prediction to be deviant from its own training, it can detect that the victim load is using the same entry. The attacker then poisons this entry by executing its store-load pair and when the victim gadget is executed again, the 1st load is incorrectly predicted as independent of the earlier store (the first line of the gadget) and bypasses the store operation.

Before this memory order violation is detected and the pipeline is flushed, the secret can already be encoded by the 2nd load in a side channel. In the original Spectre-STL, cache side channel is leveraged to leak the secret. One notable difference between Spectre-STL and the target Spectre-CTL is that Spectre-CTL can leak the secret directly from the SSBP without a cache side-channel. The attacker finds another load ❹ in its address space that uses the same SSBP entry ❺ (Aliased Entry 2) with the load instruction that uses the secret value as the index of `array2` (3rd load) ❻. If the secret data matches the idx, this 3rd load also has a dependency on the store instruction (first line of the gadget) and will encounter misspeculation, updating the SSBP state. Thus, by simply checking if SSBP Entry 2's state is changed, the attacker can recover the secret value. To trigger the vulnerable transient window and to encode the secret in the SSBP predictor state, the attacker and the victim have to take turns updating the same SSBP entries.

4.2.3 Transient Attack Mitigations

Several mitigations have been introduced for transient attacks [171, 157, 54, 144, 86, 103, 106, 155, 166, 81, 143]. First, to avoid speculative data access, various memory fence techniques have been introduced (e.g., using LFENCE instruction before data loads) such that the data is not loaded until speculative conditions are satisfied [75, 43, 122, 144]. Second, some methods limit cross-process resource sharing (e.g., flushing upon a context switch, logical or physical resource partitioning) such that the attacker process cannot access the victim's secrets [106, 41, 170, 160, 151, 143]. Third, some solutions apply randomization or encryption for shared resource indexing mechanisms to disassociate the address of an instruction from the accessed index, complicating the attacker's goal of finding an entry that they share with the victim [123, 140, 132, 67, 104, 81]. However, these solutions incur significant performance and area overhead due to restricted resource sharing and complex indexing.

Can existing general mitigation solutions work? Some studies presented more general solutions (e.g., NDA [155] and STT [166]) that protect against all possible transient-execution attacks, instead of mitigating individual attack variants. Rather than focusing on the target predictor's behavior, they focus on the data and control flows and block any channels that potentially leak information. These solutions could mitigate Spectre-CTL, like many other attack variants. However, such a high protection coverage comes at the cost of performance overhead, because they unnecessarily disallow many benign predictions. Our evaluation results also verify STT's uncomparably high performance overhead than the specialized mitigations (Section 4.5). It is noteworthy that industry is also moving towards mitigating individual attacks, rather than using a general solution, due to the performance concerns. On ARM processors, CSV2 [27] is used for Spectre v2 attacks, and user pointer sanitization is used for Spectre v1. On x86 processors, IBRS and IBPB are used for Spectre v2, while usercopy, swapgs barriers, and user pointer sanitization are used for Spectre V1 [28]. We aim at providing a practical solution for Spectre-CTL mitigation without hurting overall performance.

4.3 Threat Model

We assume the same threat model as the one assumed in the Spectre-CTL work [102]. The attacker and victim run in different processes on the same processor core. The attacker's goal is to extract sensitive information leveraging the shared SSBP [102]. The attacker leverages the prediction outcome of an aliased SSBP entry by executing a load instruction after the victim executes. The target core can support simultaneous multithreading (SMT). Since AMD isolates the SSBP per logical SMT core, the attacker and victim must run on the same logical core to ensure they observe the same SSBP state across context switches. The attacker is aware of the source code of the victim application, but does not have direct access to the victim's code nor its physical mappings. The attacker can execute unprivileged instructions that are necessary to perform the Spectre attack gadgets.

In this work we focus on the unique attack surface of Spectre-CTL, where the SSBP itself is a source of leakage without additional side channels. We consider the leakage through other side-channels, such as caches, to be out of scope. We assume that those side-channels can be mitigated with orthogonal solutions [83, 160, 155, 106, 166]. Attacks involving non-transient side-channels and intra-process attacks, such as Spectre-V1, Spectre-STL, and cross-privilege boundary attacks (e.g., user and kernel mode), are also out of scope.

4.4 SCPC: Securing SSBP from Cross-Process Collision-based Transient Attacks

4.4.1 Overview

In this work we propose a Secure Cross-Process Collision (SCPC) sharing structure to mitigate the Spectre-CTL attack. SCPC is inspired by the Copy on Write (COW) mechanism commonly used in Linux memory management [47]. In SCPC, SSBP entries are shared across processes by default. Only when a sharer process attempts to update the shared predictor state, the process's state is copied to a new SSBP entry. We leverage the observation that as long as

no state is updated in the SSBP table, then no information can be derived through the structure, since the distinct victim's execution flow is decoupled from the state encoded in the predictor table.

Unlike many other transient execution attack mitigations, we do not completely isolate SSBP because 1) resource partitioning requires a clear understanding or runtime monitoring of individual applications' resource utilization, which incurs performance overhead and complex code analysis, and 2) complete isolation limits the potential benefit of sharing the store bypass prediction results across load-store pairs having similar behaviors i.e., we can prevent similar predictor values from redundantly occupying the limited SSBP resource.

Instead, SCPC allows benign SSBP sharing across processes as much as possible by isolating processes only when a security-critical state transition is about to happen. As described in the previous section, SSBP uses an FSM for each load-store pair to predict their equivalence. When a prediction is incorrect, the FSM is updated, and the attacker can recognize that the victim aliases with the entry and has dependency in the load-store pair. The Spectre-CTL attack (Figure 4.2) leverages the transition that is caused by the victim's misspeculation. To hide the victim's transition from the attacker, when a state update is required, a new SSBP entry is created. The process that encounters the event that necessitates a state update uses the new entry from then on, leaving the shared entry unchanged.

To make processes use a dedicated entry only upon a certain condition, SCPC incorporates *selective virtualization (S-VLT* (Section 4.4.2). While S-VLT tackles the limitations of resource partitioning-based transient execution attack mitigations through selective isolation, if more processes are mapped with new dedicated entries, the risk of resource starvation increases. To reduce such concern, SCPC also uses *self-invalidation (Timer)* (Section 4.4.3). Together with S-VLT and Timer, SCPC reduces the SSBP attack surface with almost negligible performance and area overhead.

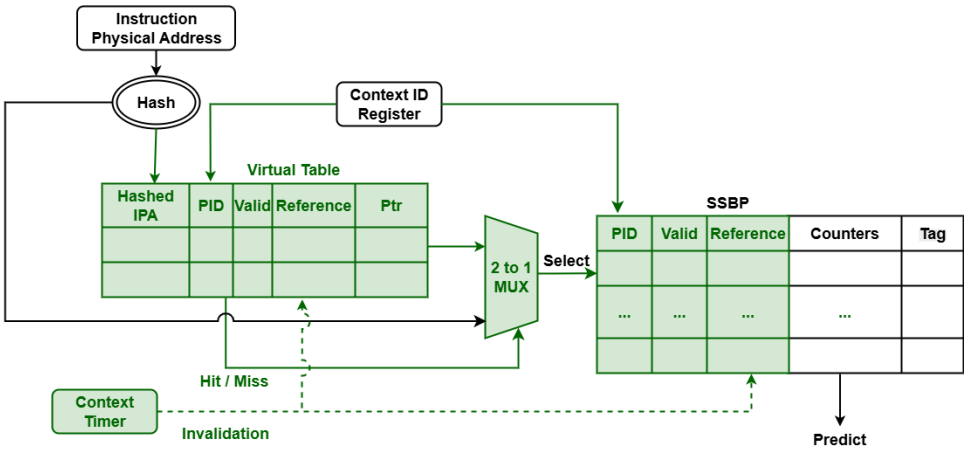


Figure 4.3. SSBP with SCPC: new components in green color.

4.4.2 Selective SSBP Virtualization (S-VLT)

Baseline S-VLT Design

Owner and Sharers: Figure 4.3 shows the SSBP when SCPC is applied. The changes made for SCPC are highlighted with green color. To distinguish the attacker's behavior from the victim's, we augment the SSBP table with two new fields: *PID* field, which marks the **owner** process ID and *Valid* field, which is a one-bit flag indicating if the entry is owned by any process. When a process occupies a new entry, that process becomes the owner of that entry and marks the PID and Valid fields with its process ID and 1, respectively. When another process's load aliases with an already owned entry, the process uses the prediction as a **read-only sharer**. Only the owner process has the right to update the predictor entry state (i.e., the counters). The aliased process can continue to use another process-owned entry for its prediction as long as it does not encounter an event that necessitates the updating of the counters. When the sharer process encounters an event that requires the updating of the predictor state, another entry is allocated in the SSBP where the aliased process becomes the new owner. The sharer's FSM transition upon misspeculation is written to the new entry. Therefore, the original shared entry remains the same (i.e., the attacker cannot recognize victim process's behavior).

Virtual Table: To make the sharer process use a new entry without changing the hash

function for indexing, we incorporate a *virtual table*, which redirects the index to another SSBP entry. As shown in Figure 4.3, each entry of the virtual table has the linked SSBP entry id (marked as *Ptr* in the figure). While the SSBP uses set-associative indexing, the virtual table uses fully-associative mapping such that the small virtual table can be better utilized. To lookup the virtual table with the original SSBP index (i.e., hashed load instruction address), the index value is stored in *Hashed IPA* field in the virtual table.

Virtual Table Indexing: An entry is created in the virtual table only for the process that needs an isolated entry. In other words, all the owner processes of the SSBP entries and the sharer processes that do not encounter misspeculation bypass the virtual table. The 2-to-1 MUX in the figure is used to choose either the index retrieved from the virtual table or the original hashed index value to find the proper SSBP entry. For example, when a load instruction is decoded, to check if it can bypass earlier stores, the virtual table is searched first with the hashed load instruction address. If the search hits and the PID field matches with the process's PID, the Ptr field of that entry is used to access the corresponding SSBP entry. If there is no matched entry in the virtual table, it means the process does not have an isolated entry so the SSBP is directly looked up. If the corresponding SSBP entry's Valid field is zero, the process becomes the owner of the entry and sets the Valid field. Otherwise, the process uses the SSBP entry as a sharer as long as it does not encounter a misspeculation.

Benefits of S-VLT: Unlike earlier resource partitioning methods [151, 143], S-VLT allows most of the benign data sharing, which limits performance overhead, while making victim's behavior invisible to the attacker. S-VLT has little area overhead because it can maintain a small virtual table as an isolated virtual entry is created only when a sharer process encounters a misspeculation. We configured the virtual table to be 12.5% of SSBP size as a baseline in the evaluation. More importantly, SSBP does not need to change its original hash-based indexing, unlike state-of-the-art solutions that use randomized and complex indexing for transient execution attack mitigation [171]. Although the fully-associative virtual table needs to be accessed before accessing the SSBP, this additional search does not cause significant performance degradation

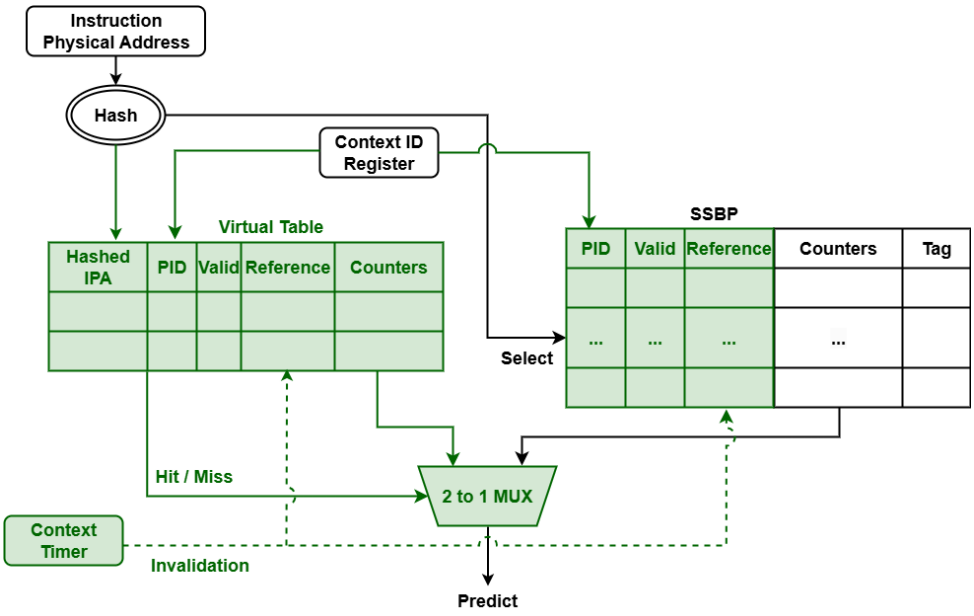


Figure 4.4. SSBP with optimized S-VLT.

because the memory dependence prediction can start at the decode stage, while the prediction outcome is not due until the load reaches the issue queue [84]. We further reduce the prediction latency with an optimized design, in which both tables are accessed in parallel.

Cost-effective S-VLT Design

The Ptr field in the virtual table needs to have logN bits when the SSBP has N entries. For bigger SSBP tables, the Ptr field is longer, and may incur additional area overhead. On the other hand, the predictor states (i.e., counters) of each SSBP entry use only 8 bits. If SSBP has more than 256 entries (which requires 8 bits in Ptr), the Ptr field is longer than the Counters field in the SSBP. Thus, instead of storing the pointers to the SSBP entry, in the optimized S-VLT design, we directly store the counters in the virtual table. As in the baseline S-VLT, an SSBP entry is shared by multiple processes. Only when a sharer process needs an isolated entry upon a misspeculation, an entry is created in the virtual table. Unlike the baseline S-VLT, the virtual table entry stores the counter values; SSBP does not map a new entry. From then on, the sharer process uses the virtual table entry for prediction.

As the virtual table is no longer a mapping table but an additional SSBP storage, we can look up the virtual table and the SSBP in parallel. Figure 4.4 shows the optimized architecture. To predict if a load should bypass outstanding stores, both the virtual table and SSBP are searched in parallel. If an entry is found in the virtual table, the counters in that entry are used for prediction. Otherwise, the SSBP entry is used for prediction. Though the virtual table uses fully-associative mapping, thanks to its small size and the parallel lookup, the optimized S-VLT does not incur any delay over the SSBP access latency, according to our CACTI measurement.

4.4.3 Timer for Secure Resource Replacement

Self-Replacement for Secure Resource Utilization

So far, we discussed how SSBP and virtual table entries are owned and shared across processes. To make the resource sharing secure, we also need to design a safe replacement policy. If a process can evict another process's entry to acquire a new entry, a new attack model could be designed that intentionally victimizes a certain process's entries. Thus, we allow a process to replace one of *its own entries* only (i.e., *self-replacement*). Among the entries of its own, each process replaces one of the most inactive entries. To do that, we design a *Timer*, which tracks the access activities.

Timer can be designed in various ways. One way would be to track the *Least Recently Used (LRU)* entry among *each process's owned entries*. Though it may be effective to release the most inactive entry, LRU is costly to implement because all entries owned by a process should update their timer values whenever any of the entries are accessed. Instead, we use a lightweight reference-bit-based activity checking, inspired by the operating system's page activity tracking method [63]. We add a one-bit *Reference* field per SSBP entry. Each process sets the reference bit of an entry that it *owns* whenever it accesses it. Then, when the process needs a new entry, it replaces one of its own entries that does not have the Reference bit set. The Reference bit is periodically reset. If all of its entries have Reference bit set, a process randomly selects one of its own entries for replacement.

If a process does not have any entry to replace (i.e., the process has not occupied any entries before) and there is no available entry, the process falls back to not use predictions and makes the load not to *bypass any earlier stores*. Note that this does not stall the process's execution. Only the timing of load execution is delayed until all prior store addresses are calculated.

Self-Invalidation for Resource Starvation Avoidance

Though our proposed self-replacement policy limits the cross-process interference by allowing each process to evict its own entries, when few processes occupy all entries, other processes cannot enjoy the performance benefits of SSBP prediction, and the delayed load executions could induce a timing side channel. To tackle this, we propose to make each process voluntarily release (i.e., *self-invalidate*) its own entries after a certain time period. The released entries can be reused by any process. As processes cannot invalidate other processes' entries, this approach reduces the risk of being leveraged by another attack model. To implement the self-invalidation, we leverage the Timer approach. In our Timer-based replacement policy, the Reference bits are periodically reset, and the processes reuse one of their own entries that have not been accessed since the reset time. By extending this mechanism, each process periodically checks its owned entries and *releases* the entries that have not been accessed since reset time, *even when they do not need a new entry*.

The checking interval is set in terms of SSBP accesses *per process*. Note that counting the total SSBP accesses across processes (i.e., a global Timer) might be insecure in self-invalidation, because a malicious process could access all its entries frequently and make the global Timer reach the checking interval quickly with the intention to invalidate the victim process's entries. To count the number of SSBP accesses and check if it reaches the checking interval, we add **a counter per process**. The counter can be implemented with one 32-bit register per process. Whenever an SSBP entry is accessed *by an owner process*, the counter is incremented. Then, the counter value is compared with the pre-determined checking interval threshold. If the values

match, we invalidate all the SSBP entries owned by the process that have the reference bit unset. Both the entry's Valid field and Counters field are reset to zero so that another process can claim this entry. The checking interval threshold is set to a bigger value than the number of SSBP entries so that the entries are not released before the predictors are fully trained.

Corner Case Handling: In case some processes terminate while leaving their owned entries unreleased, those entries cannot be released because the access counter can't reach the interval threshold. To resolve this problem, we can use two solutions: self-cleanup and periodic garbage collection. In the self-cleanup, when a process terminates, its owned SSBP entries are also released altogether. In the periodic garbage collection, SSBP entries are checked periodically (e.g., every several context switches) and the entries that are not owned by active processes are released. We use garbage collection in our evaluation.

Alternative Design: While the private counter-based self-invalidation reduces cross-process intervention, some malicious processes could still own all SSBP entries and never release them by quickly accessing all of them before the checking interval threshold. To prevent this, we can consider an alternative design at the cost of area overhead. Instead of recording the activity, we record the time when each entry is owned. Then, we periodically scan the SSBP and release the entries that have been allocated longer than a threshold period of time. By eliminating the activity factor from the release condition, attackers cannot manipulate the release of the entries. Furthermore, if we change the release interval periodically to a randomly chosen number, attackers cannot estimate the timing of release, and hence the risk of starvation can be significantly reduced. One drawback of this randomization approach is the area overhead. The private counter-based self-invalidation needs one bit for the Reference field. On the other hand, the alternative designs need to have a longer Reference field to store the time of ownership. For example, if clock cycle count is used, the cycle count can easily span several billion, so the field may need to be at least 32 bits or longer. We evaluate both solutions in our experiments.

4.4.4 Support for simultaneous multi-threading (SMT)

SMT is used by modern processors for better parallelism and resource utilization. With SMT, one physical CPU core can work like multiple logical cores. The logical cores share the hardware resources provided by the physical core. Due to this intra-core resource sharing, security concerns have been raised by several studies [49, 157, 42, 88, 148]. SMT makes the sharing of hardware resources easier to realize, increasing the attack surface of many attacks. SCPC is not vulnerable to cross-SMT attacks because the Spectre-CTL paper reverse engineered that AMD processors employ a separate SSBP per logical core [102]. Even when an SSBP is shared across logical cores, SCPC can still effectively mitigate Spectre-CTL as the selective SSBP entry isolation happens immediately when a process encounters a state transition, and the isolation is enforced with PID. Mitigations like flushing cannot protect against cross-SMT attacks because flushing only occurs at context switch boundaries.

4.4.5 Generality

Though we designed SCPC by considering the architecture of SSBP, the proposed mechanisms, the selective isolation and self-invalidation, can be more broadly applied to similar attacks. SCPC can be tailored to mitigate the transient execution attacks that leverage the cross-process shared resource aliasing to infer secrets from the updates of the aliased resource contents. We demonstrate and evaluate the effectiveness with BHT in Section 4.6.

4.4.6 Hardware Overhead

SCPC adds a virtual table and three new fields per entry in the SSBP table. The baseline 512-entry SSBP is 768 bytes as each entry stores an 8-bit counter and a 4-bit tag. The SSBP with S-VLT is 1152 bytes, where each of the 512 entries has a 12-bit PID, a 1-bit Valid field, and a 1-bit Reference field. The virtual table is 256 bytes, where each of the 64 entries has a 9-bit Hashed IPA field, a 12-bit PID, a 1-bit Valid field, a 1-bit reference field, and a 9-bit Ptr. Besides that, a 2-to-1 MUX and a 32-bit timer per process are added.

Table 4.1. gem5 Simulation Configuration

Parameter	Configuration
ISA	ARMv8
Frequency	3 GHz
Processor type	8-decode,8-issue,8-commit
ROB/LDQ/STQ	192/32/32 entries
L1 ICache	32KB, 4-way, 64B line
L1 DCache	48KB, 4-way, 64B line
L2 Cache	512KB, 16-way, 64B line
DRAM	2GB, Dual Channel DDR4 2400
SSBP	512 entries, 2-way set associative
Virtual table	64 entries, fully-associative
Kernel	linux-kernel-5.4.49

Table 4.2. 8-Workloads Mixtures: SSBP access intensity of each benchmark is in parentheses (h: high, m: medium, l: low).

ID	Benchmarks
Mix-1	mcf_r (h), parest_r (h), xz_r (l), bwaves_r (m), cactuBSSN_r (m), omnetpp_r (m), xalancbmk_r (l), imagick_r (l)
Mix-2	parest_r (h), leela_r (h), povray_r (h), bwaves_r (m), exchange2_r (m), nab_r (m), roms_r (l), xalancbmk_r (l)
Mix-3	povray_r (h), parest_r (h), leela_r (h), blender_r (m), deepsjeng_r (m), nab_r (m), lbm_r (l), roms_r (l)
Mix-4	mcf_r (h), blender_r (m), omnetpp_r (m), deepsjeng_r (m), nab_r (m), xz_r (l), xalancbmk_r (l), lbm_r (l)
Mix-5	mcf_r (h), parest_r (h), blender_r (m), omnetpp_r (m), cactuBSSN_r (m), exchange2_r (m), namd_r (l), xz_r (l)
Mix-6	leela_r (h), povray_r (h), exchange2_r (m), bwaves_r (m), nab_r (m), cactuBSSN_r (m) , xalancbmk_r (l), roms_r (l)

4.5 Evaluation

4.5.1 Methodology

We model the proposed SCPC using a full system simulator, gem5 [107]. Gem5 is configured as shown in Table 4.1. To understand the effectiveness of SCPC for cross-process collision-based attacks, we run eight applications concurrently for each evaluation. Applications from SPEC CPU2017 [52] are grouped based on the SSBP access intensity, as shown in Table 4.2. We fast-forward the first one billion instructions and collect simulation statistics for the next 2 billion instructions. For S-VLT, the virtual table has 12.5% of the total SSBP entries. We

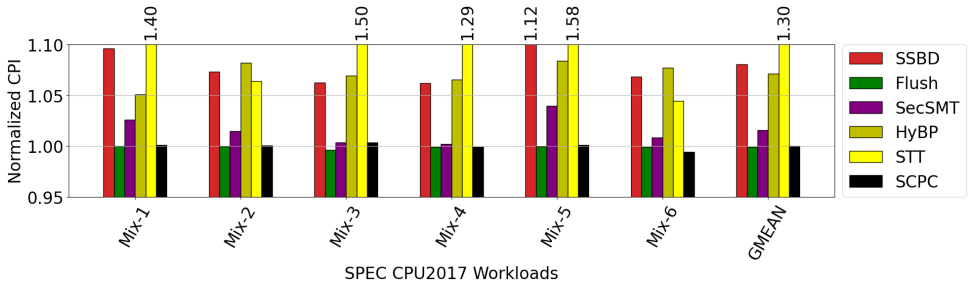


Figure 4.5. Overall performance comparison normalized by baseline (lower is better).

evaluated SCPC with optimized S-VLT. We measured the latency of accessing a 64-entry fully-associative virtual table and a 512-entry 2-way set-associative SSBP with CACTIv7.0 and found that both take 3-cycle access latency. Since the virtual table and the SSBP are accessed in parallel under the optimized S-VLT architecture, we did not add extra delay for accessing virtual table.

We compare SCPC against an insecure baseline and several existing mitigations:

1. **Flush:** SSBP is flushed to all 0's upon each context switch. SSBP with 0's means all loads are predicted to be independent (**bypassing** all stores) so this might increase the number of transient windows and reduce attacker's effort. But, the counter values are not propagated among processes.

2. **SecSMT [143]:** We adapted the SecSMT's adaptive partitioning for SSBP by partitioning the SSBP based on the runtime application demand.

3. **HyBP [171]:** HyBP protects branch predictor tables with a mixture of replication and index randomization. We adapted this defense for SSBP by using a randomized index keys table (codebook) with the same number of entries as the SSBP. The codebook is periodically re-encrypted with a 263-cycle overhead. Each load first uses the baseline SSBP index to get a random index key from the codebook and that key is XORed with the baseline index to get the encrypted index. The counters and tags are also encrypted with a content key.

4. **SSBD:** This design is an AMD-provided mitigation that completely disables the SSBP

structure to avoid store bypassing. Loads are serialized and must wait for all the older stores' data addresses to be resolved before issuing.

5. **STT [166]:** STT is a generic spectre mitigation that allows forward progress in a speculative window as long as any speculatively read data does not reach any transmitter instruction that encodes the data in a covert channel.

4.5.2 Performance overhead

We compare SCPC performance with the six mitigation solutions listed in Section 4.5.1 and we normalize values against the insecure baseline design. Figure 4.5 shows the normalized cycles per instruction (CPI) results when tested with a 512-entry SSBP with a 64-entry virtual table. Overall, most mitigation methods show non-trivial performance overhead, except Flush and SCPC. STT incurs the highest performance overhead of 30% because it generically tries to secure both branch and memory dependency speculation. But in absence of safe instructions in between the data load instruction and the transmitter instruction, STT in essence can incur similar performance overhead as fencing-based mitigations. The overhead of HyBP comes from the increased latency of choosing a random encrypted index for each load. SCPC also has a virtual table lookup overhead, but the latency of this table lookup is hidden as both the virtual table and the SSBP are looked up in parallel, and both lookups take 3 cycles. On the other hand, when SSBP is disabled, the SSBD design, the performance drops by 8% on average, and up to 12% in the worst case. This means that many independent load-store pairs are unnecessarily stalled without SSBP. Therefore, SSBP is an essential optimization component in the core design, which justifies our mitigation efforts.

4.5.3 Security Evaluation

To evaluate the effectiveness of different mitigations along with SCPC, we take inspiration from proof-of-concept (PoC)-based attack evaluation for branch predictor in [171] and adapt it for SSBP. In the PoC attack, the victim has 20 dependent load-store pairs and each of them

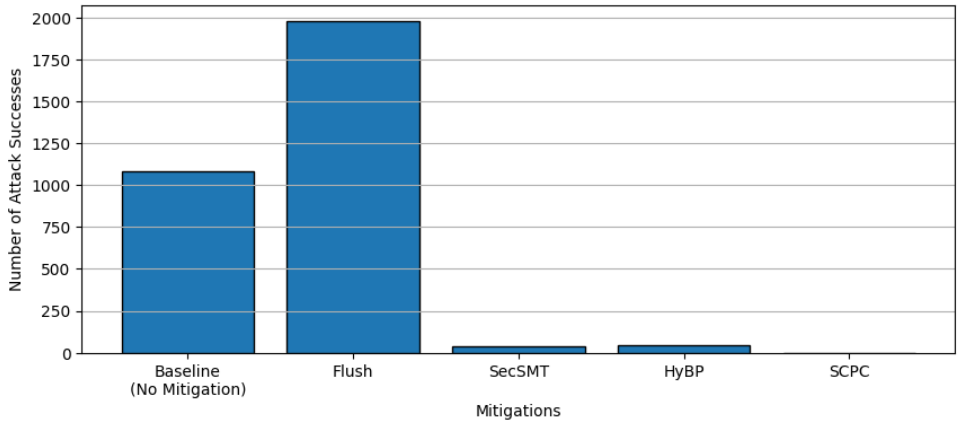


Figure 4.6. Effectiveness of mitigations: successful attacks out of 2K attempts triggered by the same PoC-based attack gadget.

are executed 4 times while the attacker runs many more load-store pairs that are independent to mistrain the SSBP to bypass stores. In each iteration of this attack, the attacker process tries to mistrain SSBP entries used by the victim process and induce transient windows in victim to leak secret data. We define a successful attack as a victim load incorrectly bypasses a store *more than once* per iteration due to attacker interference, by following the earlier study [171]. We assume once a successful malicious transient window is triggered, the loaded secret value can be encoded in any covert channel like cache. We run 100 iterations of this PoC attack with different mitigations and show the number of successful attacks in Figure 4.6. We excluded SSBD and STT from this evaluation because SSBD is a security oracle solution and STT is almost impractical due to performance overheads.

Without any mitigation, this PoC leads to 1081 attack successes out of a total of 2000 attack attempts on the insecure baseline. This is because some victim loads could not be mistrained by the attacker loads as no attacker load coincides with the victim load's SSBP entry. SCPC successfully defends against the attack because the loads of victim and attacker that point to the same SSBP entry are diverged after the first attack attempt. On the other hand, Flush suffers from even higher success rates than the baseline; 1979 successes. This is because, in Flush, as long as the attacker can trigger a context switch while the victim is running, it can successfully

reset all the SSBP entries to zero and eventually when the victim is switched in again, all the subsequent victim loads are incorrectly predicted to be independent. This technically eases the attacks because the attacker doesn't need to spend time to train the target SSBP entries to predict bypassing; he/she can simply yield to be context switched, then wait for the victim load to incorrectly bypass a store and encode the secret value to some general purpose covert channels like cache. Note that, no SSBP collision between victim and attacker is needed for Flushing as far as the attacker knows where the secret will be encoded in the covert channel. Flushing triggers the transient window needed for the attack without carefully crafted mistraining. SecSMT and HyBP are also effective, but some attacks still manage to evade them.

4.5.4 Sensitivity Study

SSBP size

We tested with 512- (default), 256-, and 128-entry SSBPs with 12.5% virtual table size with the same 6 workload-mixtures from Table 4.2. The average CPI of different mitigations normalized by the insecure baseline using each table size is shown in Figure 4.7. SCPC incurs almost 0% performance overhead even with smaller SSBPs, mainly because the SSBP entries are shared across processes, with a small number of isolated entries maintained in a separate virtual table. On the other hand, partitioning based SecSMT exhibits a notable performance drop when a smaller SSBP table is used, due to limited resources provided for each process. SSBD's absolute CPI is stable across different SSBP sizes, because SSBD always forces loads to stall.

Virtual table size

We applied SCPC to a 512-entry SSBP using a virtual table with 64, 32, and 16 entries. All three configurations showed almost zero performance overhead. We also observed no successful attack from all three configurations. From this result, we can conclude that even a small 16-entry virtual table can effectively reduce security surface without performance overhead.

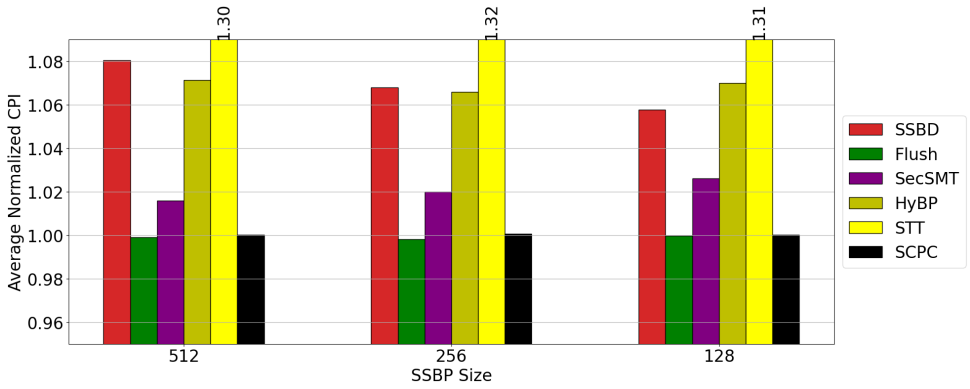


Figure 4.7. Normalized performance with different SSBP sizes.

Concurrency

We also tested how different defense mechanisms react to different concurrencies using different number of workloads running concurrently in the same core. Along with the 6 mixtures of 8 concurrent workloads from Table 4.2, we also tested 6 mixtures of 4 concurrent workloads and 12 mixtures of 2 concurrent workloads. We also tested with another security oracle baseline, static partitioning (*Partition*) to understand its efficiency. An SSBP is partitioned into equal sized chunks so that each workload can use a dedicated entries without any information sharing. Figure 4.8 shows that partitioning-based mitigations, partition and SecSMT, show notably higher overhead as concurrency increases. This proves that complete isolation through resource partitioning cannot be a scalable solution. Due to the increased contention, the potential security and performance gains can be easily diminished, regardless of static or dynamic methods. HyBP is also sensitive to the number of workloads. On the other hand, SCPC did not show a performance drop because of SCPC's benign sharing and selective isolation can alleviate resource contention. Both SSBD and STT are also not sensitive to different concurrencies, but their performance overhead is always worse than the other mitigations.

Timer alternate designs

We compared the two Timer designs, discussed in Section 4.4.3, while varying the release interval from 100K to 1M per-process SSBP accesses for private counter and the same number

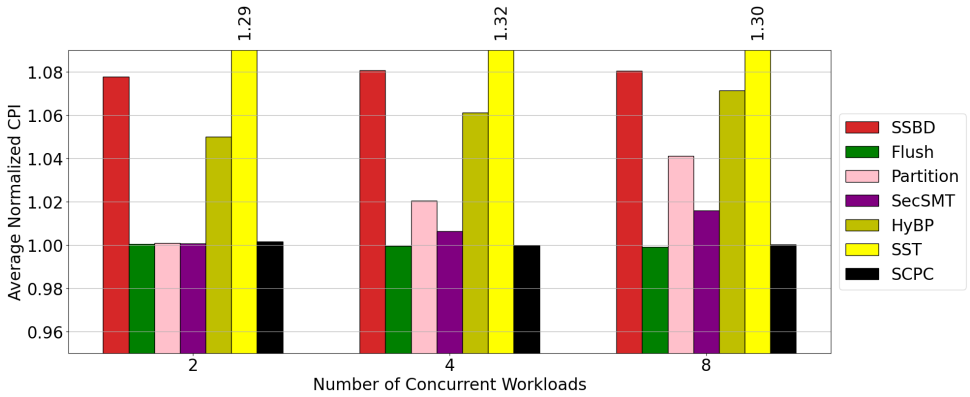


Figure 4.8. Normalized performance with increased concurrency (4 and 8 workloads per Mix).

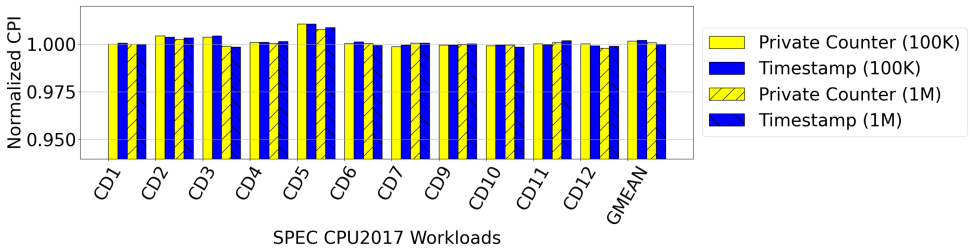


Figure 4.9. SCPC Performance with various Timer schemes.

of clock cycles for timestamp. Figure 4.9 shows that the lightweight private counter provides almost identical performance with the timestamp. We did not observe a significant performance impact of release interval either, which is due to the limited resource contention among the SPEC benchmark processes.

4.6 Broader Applicability of SCPC

In this section, we demonstrate the general applicability of SCPC besides SSBP with an example application of the S-VLT approach used on the branch history table (BHT).

4.6.1 Vulnerable Components of TAGE-SC-L Branch Predictor

The state-of-the-art branch predictor, is the TAGE-SC-L [136]. This branch predictor incorporates various components to be able to capture the most amount of information to make accurate predictions on a branch direction. There are a total of four components: the global

predictor, the loop predictor, the tagless predictor and the statistical corrector.

Of those, the components maintaining the branch information learned across processes thus vulnerable to the cross-process contention-based transient execution are the global predictor (GP) and the loop predictor (LP). These are the only predictors that use the branch instruction's address (PC) and history to generate both a tag and an index for their respective entries. In contrast, the statistical corrector collects both local and global information and makes predictions based on a tagless state—similar to a tagless predictor (which does not rely on tag aliasing to cause collisions). Flushing these structures on a context switch should serve as a sufficient mitigation without incurring more area overhead. Additionally, the statistical corrector rarely overrides the global predictor [135].

The GP is a skewed set-associative structure. It leverages the global history register, implemented as a shift register, to index into several tables of varied history lengths. The global history register keeps track of the branch direction of the last a few branches. The varied history lengths are subsets of this register. The GP also uses a subset of the branch instruction address to construct the index. The other part of the instruction's address is used to construct a tag. An access to the GP hits if the tag matches the requested tag. Each entry includes a counter in which the sign bit indicates the direction in which the branch should be predicted. The counter increases every time the prediction is correct, and decreases otherwise. If the overall prediction is incorrect, a new entry is allocated in a prediction table with a longer history than the one that provided the prediction. To do this, the controller searches for an entry with a zeroed useful counter in the tables with longer history lengths. The useful counter (u) tracks how much each entry contributes to prediction accuracy: it is incremented when the prediction is correct and decremented otherwise. The branch prediction controller periodically resets this counter to zero.

The loop predictor (LP) is a skewed set-associative table used to predict highly repetitive branch instructions (*i.e., fixed loops*) based on their local history. The LP table is indexed using both the instruction's address and its history, and it uses a portion of this information as a tag for entry identification.

4.6.2 SCPC for Securing TAGE-SC-L

Our objective is to secure the TAGE-SC-L branch predictor by creating a separation between contexts, in which one process cannot alter the branch prediction of another process. To accomplish this goal we propose to adapt the S-VLT idea to the GP and LP tables in the TAGE-SC-L predictor.

We start by describing the changes to the GP table. Leveraging the optimized implementation of SCPC on the SSBP, the virtual table is implemented as a small structure that is accessed in parallel to the main GP table and contains the same information as the original structure. The SCPC virtual table is a fully-associative structure where the tag is constructed using a hash of the full PC of the branch and a portion of the global history register. Each entry also has an added field that indicates the PID that first occupies the entry (shown in Figure 4.10). The PID field is also added to the main table, to identify the owning process.

In the original design, the GP is constructed of several tables that use varying lengths from the global history register as part of the index. To maintain the same accuracy as the original design, we also assume the same number of virtual tables as in the main table. The main difference in the virtual tables is the usage of the global history bits as part of the tags and not the indexing, given that the tables are fully associative. This design choice is to ensure that no information is lost from the original design.

The reclamation of entries of the main table remains unchanged. The reclamation process of the entries in the virtual table relies on the same process as the main table, which uses the u bit to identify entries that are no longer productive.

For LP, to be able to separate the updating of the information for each process, we add a new field to the LP table entries, the PID, and also add the virtual table (with the same modifications as for the GP). This PID information is used to determine if a process is allowed to update the information in the entry. If the PID is the same as the one in the entry, it is allowed to update the prediction metadata. Otherwise, a new entry is allocated in the virtual table. Just like

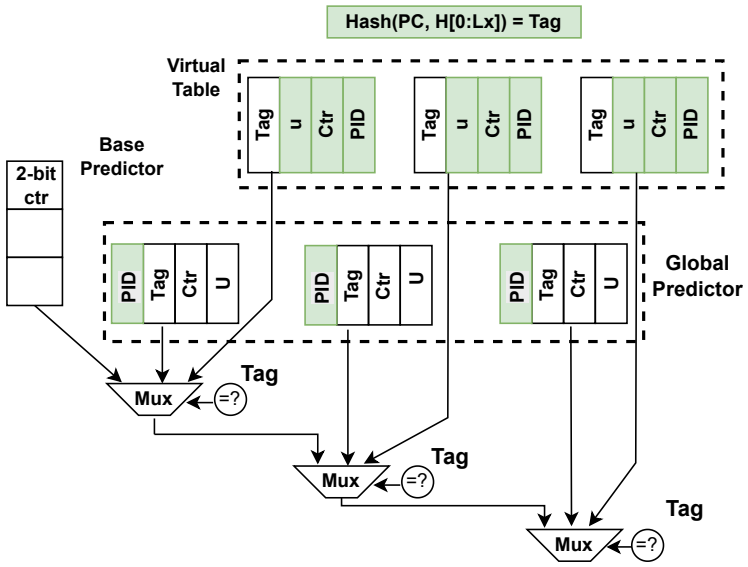


Figure 4.10. SCPC applied to the Global Predictor of the TAGE-SC-L branch predictor: new components in green color.

the GP, both tables (the main and virtual tables) are accessed in parallel.

Prediction: During branch prediction, the virtual address of an instruction and the corresponding history are used to access the GP and the LP. The GP (both main and virtual tables) uses the global history register, while the LP (both main and virtual tables) uses local history to find the entry that matches the target tag. This access is done in parallel for both predictors, the GP and LP. If a match is found in the virtual table, the entry's prediction takes precedence over the main tables. Similar to the original design, the entry with the longest history length has priority. If none of the tags match (neither in the virtual nor main tables), the prediction from the tagless table is used. Otherwise, whichever table has a matching tag is the one used to produce the prediction.

Updates: When updating a mispredicted branch, the corresponding entry that provides the prediction is only updated if the instruction's PID matches that of the entry. Otherwise, a new entry is allocated in the virtual table (for either the GP or the LP, according to the original design).

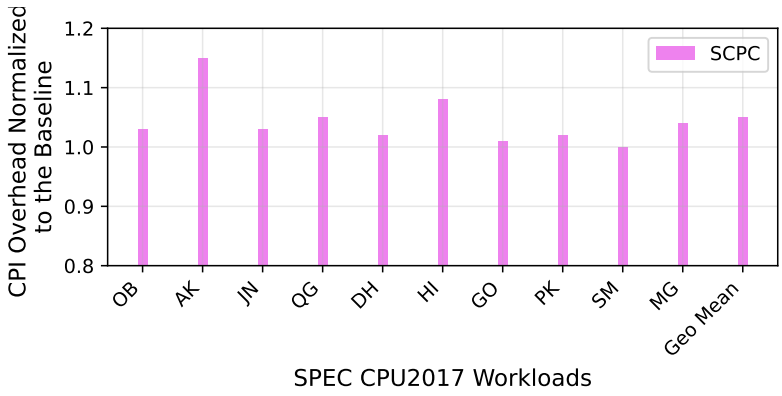


Figure 4.11. Performance comparison normalized to the insecure baseline for the branch predictor.

Insertion: When a branch instruction is mispredicted and none of the entries had a matching tag, or the entry that provided the prediction has a different PID, the controller inserts a new entry for the target instruction. Similar to the original design, to find a new entry, the branch predictor searches for the first entry that has a u counter set to zero. The controller first searches the entries in the GP virtual tables that have a longer history than the one that provided the original prediction. If no entries are found, then the controller searches in the main tables with the same heuristic on the history length. If the branch is determined to be a loop, the same process is used for the LP. First, the controller tries to allocate a new entry from the virtual table where an entry has a NULL age (same as in the original design). If no entry is found, then the main table is searched. If no entries are found, then no new allocations are made.

4.6.3 Performance Evaluation

We compare the performance of our SCPC branch predictor against an insecure baseline. For our experiments, we use the TAGE-SC-L predictor configured with a 64KB budget. The virtual table predictors are sized to be 12% of the combined size of the GP and LP tables, matching the overhead of the SSBP design. To determine whether our modifications introduce critical path delays in the pipeline, we used CACTIv7.0 [116]. We found that the virtual

table predictor, which interfaces with the GP tables, lies on the critical path with an access time of 0.7 ns and a cycle time of 1.2 ns. Holding all other design parameters constant (see Table 4.1), we model a 4-cycle penalty on all conditional branch instructions to account for the prediction latency. Figure 4.11 shows the CPI overhead of the SCPC BHT implementation, normalized to the baseline. On average, we observe a 4% performance overhead, which proves the efficiency of SCPC, given that branch predictor attack mitigations normally suffer from significant performance penalty (A state-of-the-art solution reported 0.5 to 21% IPC drop [171]). Further analysis reveals that the branch misprediction rate remains largely unchanged and that conditional branch instructions comprise only 9% of the total simulated instructions.

4.7 Related Work

ISA-based mitigations: Many defense mechanisms that address transient execution attacks have been proposed [157, 54]. Intel and AMD introduced new instructions that restrict the execution of branches under certain conditions (e.g., IBRS, IBPB) [75, 43, 3]. For the store bypassing attacks, the existing defenses fall under the pipeline detection and limitation of the transmission of transient data [106, 155, 166]. These mitigations are effective at hindering the cross-process attacks that transmit data through transient state but come with high hardware and performance costs. In contrast, our proposed defense offers lightweight selective isolation of shared structures without creating interference across processes while also maximizing the structure's utilization.

Randomized indexing-based mitigations : Several other pieces of work apply defense mechanisms into the microarchitecture by either randomizing the indexing function to access predictors or use the Address Space identifier (ASID) to separate entries across processes [56, 170, 171]. While these defenses result in low overheads and effective mitigations of the targeted attacks, they are unable to prevent attackers from leveraging cross process information leakage to steal sensitive data. Even with randomized indexing, there is still interference across context

switches as new updates modify existing entries. In contrast, our proposed defense mechanism strictly separates prediction histories across processes without enforcing strict static partitioning, and leveraging the benefits of adding a level of indirection to eliminate interference.

Indirect indexing-based mitigations : Several studies, such as NewCache [104] also use indirect mapping for security purposes. NewCache uses a CAM-based mapping table for indirect cache block indexing to mitigate cache timing side-channel attacks. NewCache effectively decouples the address of a block from its location in the cache. However, as CAM uses direct mapping for individual processes, each process can encounter more conflict misses, leading to a performance overhead. Also, NewCache uses the random replacement policy, where any entries, regardless of the ownership of the entry, can be selected for replacement. Therefore, cross-process interference cannot be completely avoided.

Compared to the aforementioned studies, SCPC introduces a novel selective isolation and self-invalidation to mitigate both information leakage and resource contention. SCPC mechanisms are also broadly applicable for any attacks that exploit shared prediction outcomes among processes. Unlike other proactive approaches using resource partitioning, SCPC allows benign sharing as much as possible, which can support high-performant mitigation strategies.

4.8 Discussion

This paper presents SCPC, the first Spectre-CTL mitigation. By exploiting the fact that the information leakage is triggered upon the updation of the shared SSBP state, SCPC incorporates a novel selective isolation (S-VLT). S-VLT allows benign information sharing while enforcing isolation when a process attempts to update other process' SSBP entry. For secure SSBP utilization, SCPC also proposes self-replacement and self-invalidation, which prevent malicious eviction and resource starvation. Our security and performance evaluations show that SCPC enforces secure computing with almost zero performance overhead.

Chapter 5

Ongoing and Future Works

This dissertation demonstrates that performance-oriented design choices in heterogeneous computing systems introduce new and often subtle security vulnerabilities across multiple layers, including machine learning frameworks, accelerator architectures, and processor microarchitecture. While the presented work provides concrete attacks and defenses in each of these domains, it also opens several important directions for future research. This chapter outlines promising future directions that build upon the insights of this dissertation and aim to advance the design of secure and robust heterogeneous systems.

Emerging memory technologies, particularly Compute Express Link (CXL)-based memory disaggregation [59], introduce new dimensions to the security landscape. By decoupling memory from compute and enabling access to remote memory with higher and more variable latency, these systems fundamentally alter microarchitectural behavior.

As part of ongoing work, I am currently investigating the security implications of memory disaggregation enabled by Compute Express Link (CXL) in collaboration with researchers at Pacific Northwest National Laboratory (PNNL). In this effort, we study how the increased latency and disaggregated nature of CXL-attached memory affect transient execution attacks. Our preliminary findings indicate that CXL-based memory can amplify transient execution windows, thereby increasing the effectiveness of speculative attacks compared to traditional DRAM-based systems. This observation suggests that emerging memory architectures may

unintentionally strengthen existing microarchitectural vulnerabilities. We are actively extending this investigation through both real-system measurements, and a paper describing these findings is currently under preparation for submission.

Additionally, new defense mechanisms must be designed to account for heterogeneous memory hierarchies. These may include latency-aware speculative execution controls, secure memory allocation strategies, and hardware-software co-design approaches that selectively place sensitive data in low-latency memory regions. Compiler-assisted data placement and prefetching strategies also present promising directions for mitigating the security risks introduced by memory disaggregation.

Chapter 6

Conclusion

Modern heterogeneous computing systems derive their performance and efficiency from aggressive optimization mechanisms such as transfer learning, speculative execution, hardware specialization, and dynamic power management. However, this dissertation shows that these same optimizations can also create systematic security vulnerabilities. Across machine learning systems, edge accelerators, and processor microarchitecture, shared resources and performance-driven behaviors expose observable signals that attackers can exploit to recover sensitive information or influence system behavior.

This dissertation presented three case studies that collectively support this thesis. First, it introduced Decepticon, a model extraction attack that exploits execution fingerprints inherited from shared pre-trained models. The results showed that transfer learning, while highly effective for model development, also creates structural similarities that can be used to identify the pre-trained model and enable high-fidelity reconstruction of a victim model. Second, this dissertation characterized power and frequency behaviors on edge GPU platforms and demonstrated that these telemetry signals can be used to build covert channels on commercial devices. These results reveal that low-level monitoring interfaces, though useful for performance tuning and debugging, can unintentionally expose a communication medium to adversarial software. Third, this dissertation proposed SCPC, a low-overhead defense against cross-process collision-based transient execution attacks. SCPC showed that it is possible to improve isolation in speculative

execution structures while still preserving most of the performance advantages of modern processor optimizations.

Taken together, these contributions highlight a common theme: security weaknesses in modern systems increasingly arise from cross-layer interactions rather than isolated flaws within a single component. Machine learning frameworks inherit vulnerabilities from their training methodology, accelerators leak information through power-management behavior, and processor optimizations expose transient side effects through shared prediction structures. These findings suggest that securing future heterogeneous systems requires security analysis and defense mechanisms that span the full stack, from software frameworks and runtime systems to hardware resource management and microarchitectural state.

This dissertation therefore advocates for a cross-layer approach to secure heterogeneous computing. Rather than treating performance and security as separate concerns, future system design must recognize that optimization decisions often reshape the attack surface. Building robust computing platforms will require hardware-software co-design strategies that explicitly account for information leakage, observability, and resource sharing.

Bibliography

[1] Custom gstreamer plugin.

[2] DeepSniffer ResNet Models. https://github.com/xinghu7788/DeepSniffer/.

[3] Documentation &x2013; Arm Developer — developer.arm.com. https://developer.arm.com/documentation/. [Accessed 08-01-2025].

[4] Dynamic parallelism in cuda.

[5] Edge processing camera based on nvidia jetson nano.

[6] Finding Startups with BERT.

[7] Google BERT TensorFlow. https://github.com/google-research/bert.

[8] Hugging Face BERT-base Pre-trained Model. https://huggingface.co/bert-base-uncased.

[9] Hugging Face BERT PyTorch. https://huggingface.co/docs/transformers/v4.17.0/en/model_doc/bert.

[10] Hugging Face BERT TensorFlow. https://huggingface.co/docs/transformers/model_doc/bert#transformers.TFBertForQuestionAnswering.

[11] Hugging Face Distilled GPT-2 Pre-trained Model. https://huggingface.co/distilgpt2.

[12] Hugging Face General Language Understanding Evaluation (GLUE) Benchmark. https://github.com/huggingface/transformers/tree/main/examples/pytorch/text-classification#glue-tasks.

[13] Hugging Face Models. https://huggingface.co/models.

[14] Huggingface.

[15] Jetson stats.

[16] Meta RoBERTa PyTorch. https://github.com/pytorch/fairseq/tree/main/examples/roberta.

[17] MXNet ResNet Models. https://github.com/apache/incubator-mxnet/blob/master/python/mxnet/gluon/model_zoo/vision/resnet.py.

[18] NVIDIA BERT PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT.

[19] NVIDIA BERT TensorFlow. https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow2/LanguageModeling/BERT.

[20] Nvidia deep learning accelerator.

[21] Nvidia dla reference models.

[22] Nvidia ptx isa.

[23] NVIDIA ResNet v1.5 for PyTorch. https://catalog.ngc.nvidia.com/orgs/nvidia/resources/resnet_50_v1_5_for_pytorch.

[24] Nvidia tegrastats utility.

[25] Pytorch Hub: Repository of Pre-trained models. https://pytorch.org/docs/stable/hub.html.

[26] Securing the edge.

[27] Spectre-bhb: Speculative target reuse attacks. https://documentation-service.arm.com/static/623c60d13b9f553dde8fd8e6?token=.

[28] Spectre side channels. https://www.kernel.org/doc/html/v6.1/admin-guide/hw-vuln/spectre.html#spectre-system-information.

[29] TensorFlow ResNet Models. https://tfhub.dev/google/imagenet/resnet_v2_50/classification/5.

[30] trtexec.

[31] United imaging healthcare enhanced the speed and quality of mr imaging.

[32] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.

[33] Mps for jetson (tegra) devices, 2022.

[34] Device management functions of the cuda runtime application programming interface, 2025.

[35] Mps limitations, 2025.

[36] Nvidia multi-process service, 2025.

[37] Nvml api, 2025.

[38] Nvml limitations, 2025.

[39] Tegrastats results (missed samples), 2025.

[40] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. Network-on-chip microarchitecture-based covert channel in gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 565–577, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Sam Ainsworth and Timothy M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 132–144, 2020.

[42] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE, 2019.

[43] AMD. Software Techniques for Managing Speculation on AMD Processors. Technical report, AMD Corporation, 2018. Retrieved May 2019 from https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.

[44] AMD. Security analysis of amd predictive store forwarding. In *White Paper*, 2023.

[45] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115. IEEE, 2017.

[46] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.

[47] Maurice J Bach. *The design of the UNIX operating system*. Prentice-Hall, Inc., 1986.

[48] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. Speculative interference attacks: Breaking invisible speculation schemes. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1046–1060, 2021.

[49] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 785–800, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, S. Buch, Dallas Card, Rodrigo Castellon, Niladri S. Chatterji, Annie S. Chen, Kathleen A. Creel, Jared Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea

Finn, Trevor Gale, Lauren E. Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas F. Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, O. Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir P. Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Benjamin Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, J. F. Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Robert Reich, Hongyu Ren, Frieda Rong, Yusuf H. Roohani, Camilo Ruiz, Jack Ryan, Christopher R'e, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishna Parasuram Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei A. Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. In *arXiv:2108.07258*, 2021.

[51] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[52] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[53] Robert Callan, Alenka Zajić, and Milos Prvulovic. A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events. In *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[54] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, pages 249–266, 2019.

[55] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[56] Congcong Chen, Chaoqun Shen, and Jiliang Zhang. Lightweight and secure branch predictors against spectre attacks. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 25–30. IEEE, 2022.

[57] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[58] Jianguo Chen, Kenli Li, Qingying Deng, Keqin Li, and Philip S. Yu. Distributed deep learning model for intelligent video surveillance systems with edge computing. *IEEE Transactions on Industrial Informatics*, pages 1–1, 2019.

[59] CXL Consortium. Compute express link (cxl) specification, version 3.0. Technical report, CXL Consortium, August 2022. Accessed: 2026-04-03.

[60] Gianluca De Lucia, Alessio Merlo, and Luca Caviglione. A gpu covert channel based on workload manipulation through the cupy library. 2025.

[61] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.

[62] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012.

[63] Richard Draves. Page replacement and reference bit emulation in mach. In *USENIX Mach Symposium*, pages 201–212. Citeseer, 1991.

[64] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: cross-component covert channels on integrated cpu-gpu systems. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 972–984. IEEE Press, 2021.

[65] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Spy in the gpu-box: Covert and side channel attacks on multi-gpu systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[66] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[67] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Trans. Archit. Code Optim.*, 13(1), March 2016.

[68] Abdoulayé Gamatie, Guillaume Devic, Gilles Sassatelli, Stefano Bernabovi, Philippe Naudin, and Michael Chapman. Towards energy-efficient heterogeneous multicore architectures for edge computing. *IEEE Access*, 7:49474–49491, 2019.

[69] Karthik Ganesan, Jungho Jo, W. Lloyd Bircher, Dimitris Kaseridis, Zhibin Yu, and Lizy K John. System-level max power (sympo) - a systematic approach for escalating system-level power consumption using synthetic benchmarks. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 19–28, 2010.

[70] Karthik Ganesan and Lizy K. John. Maximum multicore power (mampo) — an automatic multithreaded synthetic power virus generation framework for multicore systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[71] Jeferson González-Gómez, Kevin Cordero-Zuñiga, Lars Bauer, and Jörg Henkel. The first concept and real-world deployment of a gpu-based thermal covert channel: Attack and countermeasures. In *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2023.

[72] Chaojie Gu, Jiale Chen, Rui Tan, and Linshan Jiang. An electromagnetic covert channel based on neural network architecture. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 177–184, 2021.

[73] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. DeBERTa: Decoding-enhanced BERT with Disentangled Attention. *arXiv:2006.03654*, 2021.

[74] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints. In *Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[75] Intel. Speculative Execution Side Channel Mitigations. Technical report, Intel Corporation, 2018. Retrieved May 2019 from https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[76] Ali Jahanshahi, Hadi Zamani Sabzi, Chester Lau, and Daniel Wong. Gpu-nest: Characterizing energy efficiency of multi-gpu inference servers. *IEEE Computer Architecture Letters*, 19(2):139–142, 2020.

[77] Qisheng Jiang and Chundong Wang. Sync+sync: A covert channel built on fsync with storage. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, 2024.

[78] Jongmin Jo, Sucheol Jeong, and Pilsung Kang. Benchmarking gpu-accelerated edge devices. In *2020 IEEE international conference on big data and smart computing (BigComp)*, pages 117–120. IEEE, 2020.

[79] Glenn Jocher, Jing Qiu, and Ayush Chaurasia. Ultralytics YOLO, January 2023.

[80] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. SpanBERT: Improving Pre-training by Representing and Predicting Spans. *arXiv:1907.10529*, 2020.

[81] Arnabjyoti Kalita, Yilong Yang, Alenkruth Krishnan Murali, and Ashish Venkat. Ceviche: Capability-enhanced secure virtualization of caches. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 4082–4098. IEEE, 2025.

[82] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. Accelwattch: A power modeling framework for modern gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 738–753, 2021.

[83] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[84] Sebastian S Kim and Alberto Ros. Effective context-sensitive memory dependence prediction. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 515–527. IEEE, 2024.

[85] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[86] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[87] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming Catastrophic Forgetting in Neural Networks. In *Proceedings of the National Academy of Sciences*, 2017.

[88] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[89] Kalpesh Krishna, Gaurav Singh Tomar, Ankur P Parikh, Nicolas Papernot, and Mohit Iyyer. Thieves on Sesame Street! Model Extraction of BERT-based APIs. In *International Conference on Learning Representations*, 2020.

[90] Mohit Kumar, Xingzhou Zhang, Liangkai Liu, Yifan Wang, and Weisong Shi. Energy-efficient machine learning on the edges. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 912–921, 2020.

[91] Seyyidahmed Lahmer, Aria Khoshsirat, Michele Rossi, and Andrea Zanella. Energy consumption of neural networks on nvidia edge boards: an empirical model. In *2022 20th International Symposium on Modeling and Optimization in Mobile, Ad hoc, and Wireless Networks (WiOpt)*, pages 365–371, 2022.

[92] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *International Conference on Learning Representations*, 2020.

[93] Stefanos Laskaridis, Kleomenis Kateveas, Lorenzo Minto, and Hamed Haddadi. Melting point: Mobile evaluation of language transformers. *The 30th Annual International Conference On Mobile Computing And Networking (MobiCom)*, 2024.

[94] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. Securing branch predictors with two-level encryption. *ACM Trans. Archit. Code Optim.*, 17(3), August 2020.

[95] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. *ACM SIGARCH computer architecture news*, 41(3):487–498, 2013.

[96] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *arXiv:1910.13461*, 2019.

[97] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[98] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. Indirector:{High-Precision} branch target injection attacks exploiting the indirect branch predictor. In *USENIX Security Symposium (USENIX Security 24)*, pages 2137–2154, 2024.

[99] Yazhou Li and Yahong Rosa Zheng. Profiling nvidia jetson embedded gpu devices for autonomous machines. In *International Conference on Signal, Image Processing and Embedded Systems*, volume 10, 2020.

[100] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[101] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 973–990, USA, 2018. USENIX Association.

[102] Chang Liu, Dongsheng Wang, Yongqiang Lyu, Pengfei Qiu, Yu Jin, Zhuoyuan Lu, Yinqian Zhang, and Gang Qu. Uncovering and exploiting amd speculative memory access predictors for fun and profit. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 31–45, 2024.

[103] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[104] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro*, 36(5):8–16, 2016.

[105] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv:1907.11692*, 2019.

[106] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. Dolma: Securing speculation with the principle of transient non-observability. In *USENIX Security*, pages 1397–1414, 2021.

[107] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jer'onimo Castrill'on, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago M"uck, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and 'Eder F. Zulian. The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020.

[108] Lingjuan Lyu, Xuanli He, Fangzhao Wu, and Lichao Sun. Killing Two Birds with One Stone: Stealing Model and Inferring Attribute from BERT-based APIs. *abs/2105.10909*, 2021.

[109] Henrique Teles Maia, Chang Xiao, Dingzeyu Li, Eitan Grinspun, and Changxi Zheng. Can one hear the shape of a neural network?: Snooping the GPU via Magnetic Side Channel. In *31st USENIX Security Symposium*, 2022.

[110] Louis Martin, Benjamin Muller, Pedro Javier Ortiz Suárez, Yoann Dupont, Laurent Romary, Éric de la Clergerie, Djamé Seddah, and Benoît Sagot. CamemBERT: a tasty French language model. In *Proceedings of the Association for Computational Linguistics*, 2020.

[111] Michael McCloskey and Neal J. Cohen. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. In *Psychology of Learning and Motivation*, 1989.

[112] Yuanqing Miao, Yingtian Zhang, Dinghao Wu, Danfeng Zhang, Gang Tan, Rui Zhang, and Mahmut Taylan Kandemir. Veiled Pathways: Investigating Covert and Side Channels Within GPU Uncore . In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1169–1183, Los Alamitos, CA, USA, November 2024. IEEE Computer Society.

[113] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Levenshtein Distance: Information theory, Computer science, String (computer science), String metric, Damerau?Levenshtein distance, Spell checker, Hamming distance*. Alpha Press, 2009.

[114] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *USENIX Security*, pages 7179–7193, 2023.

[115] Glennn Moy, Slava Shekh, Martin Oxenham, Simon Ellis-Steinborner (Joint, Operations Analysis Division Defence Science, and Technology Group). Recent advances in artificial intelligence and their impact on defence. *DST-Group-TR-3716*, 2020.

[116] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

[117] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 354–366, New York, NY, USA, 2017. Association for Computing Machinery.

[118] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks are Practical. In *ACM Conference on Computer and Communications Security*, 2018.

[119] Ajay Nayak, Pratheek B., Vinod Ganapathy, and Arkaprava Basu. (mis)managed: A novel tlb-based covert channel on gpus. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 872–885, New York, NY, USA, 2021. Association for Computing Machinery.

[120] Ravan Nazaraliyev, Yicheng Zhang, Sankha Baran Dutta, Andres Marquez, Kevin Barker, and Nael Abu-Ghazaleh. Not so refreshing: Attacking gpus using rfm rowhammer mitigation. In *Proceedings of the 34th USENIX Conference on Security Symposium*, USA, 2025. USENIX Association.

[121] Sudheendra Raghav Neela, Jonas Juffinger, Lukas Maar, and Daniel Gruss. Eviction notice: Reviving and advancing page cache attacks. In *Network and Distributed System Security (NDSS) Symposium*, 2026.

[122] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass, 2018.

[123] Moinuddin K Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *International Symposium on Microarchitecture (MICRO)*, pages 775–787. IEEE, 2018.

[124] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI*, 1(8):9, 2019.

[125] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 2020.

[126] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories. *IEEE Symposium on Security and Privacy*, 2022.

[127] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. Pacman: attacking arm pointer authentication with speculative execution. In *International Symposium on Computer Architecture*, pages 685–698, 2022.

[128] Archana Tikayat Ray, Bjorn F. Cole, Olivia J. Pinon Fischer, Ryan T. White, and Dimitri N. Mavris. aeroBERT-Classifier: Classification of Aerospace Requirements Using BERT. In *Aerospace 2023, 10(3), 279*, 2023.

[129] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[130] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches. In *International Symposium on Computer Architecture (ISCA)*, pages 361–374. IEEE, 2021.

[131] Nicholas Roberts, Vinay Uday Prabhu, and Matthew McAteer. Model Weight Theft With Just Noise Inputs: The Curious Case of the Petulant Attacker. *abs/1912.08987*, 2019.

[132] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based cache attacks with a practical Fully-Associative design. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1379–1396. USENIX Association, August 2021.

[133] Souvika Sarkar, Mohammad Fakhruddin Babar, Md Mahadi Hassan, Monowar Hasan, and Shubhra Kanti Karmaker Santu. Processing natural language on embedded devices: How well do modern models perform? In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, pages 211–222, 2024.

[134] Vincent Schorp, Frédéric Giraud, Gianluca Pargätzi, Michael Wäspe, Lorenzo von Ritter-Zahony, Marcel Wegmann, John Garcia Henao, Dominique Cachin, Sebastiano Caprara, Philipp Fürnstahl, et al. A modular edge device network for surgery digitalization. *arXiv preprint arXiv:2503.14049*, 2025.

[135] André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127, 2011.

[136] André Seznec. Tage-sc-l branch predictors. In *JILP-Championship Branch Prediction*, 2014.

[137] Yalong Shan, Yongkui Yang, Xuehai Qian, and Zhibin Yu. Guser: A GPGPU Power Stressmark Generator. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[138] Rudra Pratap Singh, Harshit Srivastava, Hitesh Gautam, Rohan Shukla, and Rajendra Kumar Dwivedi. An intelligent video surveillance system using edge computing based deep learning model. In *2023 International Conference on Intelligent Data Communication Technologies and Internet of Things (IDCIoT)*, pages 439–444, 2023.

[139] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. MobileBERT: a compact task-agnostic BERT for resource-limited devices. In *Proceedings of the Association for Computational Linguistics*, 2020.

[140] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. Phantomcache: Obfuscating cache conflicts with localized randomization. In *NDSS*, 2020.

[141] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. Hot pixels: frequency, power, and temperature attacks on gpus and arm socs. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[142] Jie Tang, Shaoshan Liu, Liangkai Liu, Bo Yu, and Weisong Shi. Lopecs: A low-power edge computing system for real-time autonomous driving services. *IEEE Access*, 8:30467–30479, 2020.

[143] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. {SecSMT}: Securing {SMT} processors against {Contention-Based} covert channels. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3165–3182, 2022.

[144] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 395–410, New York, NY, USA, 2019. Association for Computing Machinery.

[145] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models. In *arXiv:2307.09288*, 2023.

[146] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security Symposium*, 2016.

[147] Khanin Udomchoksakul, Orathai Sangpetch, and Akkarit Sangpetch. Gpu performance tuning and power efficiency on the dgx a100 cluster. In *2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 170–177. IEEE, 2022.

[148] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[149] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arxiv:1706.03762*, 2017.

[150] Daniel Velicka, Ondrej Vysocky, and Lubomir Riha. Methodology for gpu frequency switching latency measurement. *arXiv preprint arXiv:2502.20075*, 2025.

[151] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. Brb: Mitigating branch predictor side-channels. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 466–477. IEEE, 2019.

[152] Tokuma Wachi. Business application of bert, a general-purpose natural-language-processing model. *Feature Articles: ICT Solutions Offered by NTT Group Companies*, 2021.

[153] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE, 2010.

[154] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236. IEEE, 2014.

[155] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *International Symposium on Microarchitecture*, pages 572–586, 2019.

[156] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. {ScatterCache}: thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019.

[157] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 54(3), may 2021.

[158] Qiongkai Xu, Xuanli He, Lingjuan Lyu, Lizhen Qu, and Gholamreza Haffari. Beyond Model Extraction: Imitation Attack for Black-Box NLP APIs. *abs/2108.13873*, 2021.

[159] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. Gpuguard: mitigating contention based side and covert channel attacks on gpus. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 497–509, New York, NY, USA, 2019. Association for Computing Machinery.

[160] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441, 2018.

[161] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security Symposium*, 2020.

[162] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security Symposium*, 2020.

[163] Zeyu Yang, Karel Adamek, and Wesley Armour. Accurate and convenient energy measurements for gpus: A detailed study of nvidia gpu's built-in power sensor. In *Proceedings*

of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '24. IEEE Press, 2024.

[164] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *arXiv:1906.08237*, 2020.

[165] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[166] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *International Symposium on Microarchitecture*, pages 954–968, 2019.

[167] Zhenrui Yue, Zhankui He, Huimin Zeng, and Julian McAuley. Black-Box Attacks on Sequential Recommenders via Data-Free Model Extraction. In *ACM Conference on Recommender Systems*, 2021.

[168] Mikhail Arkhipov Yuri Kuratov. Adaptation of Deep Bidirectional Multilingual Transformers for Russian Language. *arXiv:1905.07213*, 2019.

[169] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 667–682, 2020.

[170] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. A lightweight isolation mechanism for secure branch predictors. In *Design Automation Conference (DAC)*, pages 1267–1272. IEEE, 2021.

[171] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Xuehai Qian, Lixin Zhang, and Dan Meng. Hybp: Hybrid isolation-randomization secure branch predictor. In *HPCA*, pages 346–359, 2022.

[172] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes Attack: Steal DNN Models with Lossless Inference Accuracy. In *USENIX Security Symposium*, 2021.