Towards Hardware-Based Application Fingerprinting with Microarchitectural Signals for Zero Trust Environments

Tor J. Langehaug
Air Force Institute of Technology
tor.langehaug@afit.edu

Scott R. Graham
Air Force Institute of Technology
scott.graham@afit.edu

Abstract

The interactions between software and hardware are increasingly important to computer system security. This research collects sequences of microprocessor control signals to develop machine learning models that identify software tasks. The proposed approach considers software task identification in hardware as a general problem with attacks treated as a subset of software tasks. Two lines of effort are presented. First, a data collection approach is described to extract sequences of control signals labeled by task identity during real (i.e., non-simulated) system operation. Second, experimental design is used to select hardware and software configuration to train and evaluate machine learning models. The machine learning models significantly outperform a naïve classifier based on Euclidean distances from class means. Various configurations produce balanced accuracy scores between 26.08% and 96.89%.

Keywords: machine learning, microarchitectural data, constant monitoring, cache attack, zero trust

1. Introduction

The discovery of Spectre (Kocher et al., 2019) and Meltdown (Lipp et al., 2018) vulnerabilities reinforced the threat of hardware level information security risks inherent in contemporary modern processor designs. A key insight from speculative or transient execution attacks is that clever software, conceived with awareness of hardware design, can accomplish adversary goals that are virtually undetectable by contemporary malware detection techniques. In a “Questions and Answers” section of a website dedicated to the attacks (Graz University of Technology, 2018), the following question is posed: “Has Meltdown or Spectre been abused in the wild?” The posted response to this question is concerning: “We don’t know.”

This research proposes an incremental advance to improve upon “We don’t know” by embracing the ‘constant monitoring’ requirement commonly specified for zero trust environments. Transient execution attacks illuminate the operational challenges associated with eliminating side effects in shared computing environments. This research expands a hardware-based attack detection framework (Mao et al., 2022) to study the uniqueness of microarchitectural data for specific applications. Two machine learning techniques (K-Nearest Neighbor (KNN) and neural network classifiers) are used to demonstrate the possibility of confirming software behavioral identity using hardware gathered data. This research works toward a longer term goal of providing near real-time hardware monitoring capabilities to a range of devices from simple Internet-of-Things (IoT) to complex multi-core systems. As a first step, this research addresses an early stage of the problem using a reduced complexity open source five-stage in-order execution RISC-V microprocessor.

1.1. Motivation

Several hardware-based detection approaches for malware, cache side channel attacks, and transient execution attacks are discussed in Section 2. Hardware Performance Counters (HPCs) are found in many processors, whether embedded computers or high performance systems, as they are useful sensors for temperature monitoring, debugging, and related performance monitoring tasks. This availability led prior hardware detection research to rely on HPC data, which can be problematic for information security uses for several reasons, including: non-determinism

URI: https://hdl.handle.net/10125/103434
978-0-9981331-6-4
(CC BY-NC-ND 4.0)

(Weaver et al., 2013), the requirement for trusted code to set up and monitor, and information leakage (Uhsadel et al., 2008).

The research described in this paper desires to leverage microarchitectural data, similar to the data provided by HPCs, but monitored independently from the operating system during operational use. Hardware-based monitoring could assist in discovering transient execution attacks like Spectre and Meltdown.

1.2. Contributions

This research makes three primary contributions:

- Proposes a machine learning model training approach using a trusted system to develop models that are usable on untrusted systems.
- Validates a recently published framework (Mao et al., 2022) for microarchitectural data research.
- Informs future research directions through results obtained via experimental design and analysis.

1.3. Paper Organization

Section 2 provides a high-level overview of background material to facilitate interpreting the results while Section 3 discusses related works. Readers looking for a thorough review of related work should consult Das et al., 2019. Section 4 describes the threat model a final solution would be capable of defending against. Additionally, Section 4 describes the data collection, training, and performance evaluation methodology. Section 5 presents experimental results and analysis. Finally, Section 6 summarizes the main findings and Section 7 identifies areas for future work.

2. Background

This section introduces broad microprocessor design and machine learning concepts to provide context. These are large research domains. Consequently, a detailed discussion of microprocessor design and machine learning is outside the scope of this paper.

2.1. Microprocessor Design

A microprocessor is an integrated circuit providing key features of a Central Processing Unit (CPU) (e.g., arithmetic operations and register reads/writes, etc.) on a single chip (Shirriff, 2016). Microprocessors contain special purpose modules or units that perform specific functions such as integer or floating point arithmetic, hardware for encryption or decryption algorithms, address translation, and others.

Most commercially available general purpose microprocessors use a fixed architecture that is implemented through a series of fabrication steps that result in immutable circuitry. Microprocessors can also be implemented with field programmable gate arrays (FPGAs), which are composed of configurable logic blocks that can be reprogrammed to alter the system's functionality. Fabricated microprocessors significantly outperform FPGAs on general computing tasks. The complexity of high performance microprocessors from vendors like Intel and Advanced Micro Devices (AMD) are unlikely to be realized on FPGAs. Nevertheless, thanks to increasing open hardware design popularity there exists a variety of less-complex microprocessor implementations which can be used to study new designs for possible inclusion in future high performance designs.

The term microarchitectural data refers to any information produced by the microprocessor's internal logic. Examples of microarchitectural data include the instruction pointer, control signals transmitting information about system state (e.g., cache miss or branch mispredict), virtual addresses to translate to physical addresses, and hundreds more. HPCs are configurable registers to count signals correlated with performance problems in a CPU. HPCs are often user-accessible, making them undesirable for security.

Microprocessors implemented in an FPGA are often described as "softcore" processors. Contemporary tools, such as Chisel (Bachrach et al., 2012), Chipyard (Amid et al., 2020), and FireSim (Karandikar et al., 2018), make deployment of use case-specific hardware possible even with limited hardware knowledge. Softcore processors with an open design (i.e., access to Hardware Description Language (HDL) source) are viable for microarchitectural research because processors can be modified to collect data directly without exposing the presence of detection logic to the operating system.

The approach taken to provide hardware-based security is an extension of the MATANA framework (Mao et al., 2022), which provides a softcore processor System-on-a-chip (SoC) using a single RISC-V Rocket Core. MATANA also provides hardware detection of cache side channel and Return Oriented Programming (ROP) attacks. The MATANA SoC runs on a Xilinx ML605 evaluation board and boots a lightweight Linux operating system with Ethernet.

2.2. Machine Learning

This section provides a brief overview of three different machine learning classifiers. The first machine learning classifier is KNN. KNN classifiers identify the

K points in training data that are nearest to new data and predicts class membership based on the class with the greatest probability (James and Daniela Witten, 2017). KNN works well when the training data and number of features is relatively small. As the training data size and number of features grows, KNN becomes too computationally expensive for practical uses.

In contrast with KNNs, neural networks store weights associated with features. Neural networks tune a set of model weights through a stochastic search process. A benefit of neural networks is that only weights must be stored, not the training data. Long Short Term Memory (LSTM) networks are a recurrent network for predicting future values from a sequence of historical observations. LSTMs incorporate temporally aware "forget gates" that preserve information over long sequences (Goodfellow et al., 2016).

Convolutional Neural Networks (CNNs) use features observed in space or time. CNNs are commonly used in image classification tasks using two-dimensional spatial convolution where pixels are related to neighboring pixels in a grid. Temporal convolution operates in one dimension (the time domain). Temporal Convolution Networks (TCNs) are a related sequence processing network architecture bringing together ideas from convolutional and recurrent networks (Lea et al., 2016). TCNs are outside the scope of this research.

3. Related Works

Prior to MATANA, which was introduced in Section 2, WHISPER (Mushtaq et al., 2020) and Speculator (Mambretti et al., 2019) used HPCs to detect cache side channels and transient execution attacks. Proposals for detection systems built on HPCs range from threshold-based metrics to neural networks.

There is an open debate about the utility of HPCs for information security applications. Early research conducted by Weaver et al. (2013) identified overcount and non-determinism issues in HPCs, but did not relate these issues to information security. Das et al. (2019) observed that only 10% of information security papers were aware of non-determinism issues and often favored HPCs for information security use cases. In contrast, 45% of published performance analysis and high performance computing research–using HPCs for the intended purpose–found HPCs to be inadequate.

In FPGA research, public cloud infrastructure added AutoCounter functionality to FireSim to automatically insert performance counters into the intermediate register transfer language to precisely profile specific areas within the hardware design (Karandikar et al. (2020)). Lastly, proposals to accelerate machine learning inference tasks on FPGAs exist for KNN (Jamma et al., 2017), LSTM (Cao et al., 2019), and CNN (Bettoni et al., 2017) models.

4. Methodology

A primary contribution of this research is acquiring real data to develop machine learning models. The methodology section focuses on collecting training data and evaluating performance assuming an initial trusted hardware and software system are available. Subsection 4.1 describes the threat model targeted by this research in eventual deployment.

4.1. Threat Model

MATANA (Mao et al., 2022) uses a threat model requiring operating system collaboration. This choice is logical because the operating system maintains a global view of applications and the visibility can reduce negative consequences during response. Collaboration with the operating system enables hardware to benefit from the contextual information (e.g., process identification). This makes MATANA effective for detecting user misbehavior. However, adversaries subverting the operating system–a large attack surface that is difficult to secure–can evade detection.

The threat model targeted by this research assumes system defenders have a trusted supply chain for hardware and software such that a trusted system can be used to collect data under "normal" operating conditions. During training activities, defenders leverage the operating system's global view to preserve contextual information (e.g., labeled samples). However, once the system is deployed, the defender removes operating system accessible hardware interfaces so that all security-related decisions are performed in hardware. The data used to make security related decisions is inaccessible through software and therefore unavailable to adversaries, even if they control the operating system. This threat model supports a zero trust ecosystem where the operating system and user applications can be monitored to compare with baselines developed on a trusted system.

4.2. Data Collection Approach

The data is collected on the trusted system, requiring modifications to hardware and software. Figure 1 provides an overview of the hardware construction and software interactions. The hardware design modification connects existing signals into a structure that preserves a sequence of state changes over time. Existing control signals, performance events, or newly derived signals can be chosen during this step. This research used two performance event signals (branch misprediction and cache miss) and a derived jump instruction signal implemented in MATANA. The microarchitectural signal is preserved by left-shifting bits one position every clock cycle and concatenating the new input signal as the least significant bit. The output of each bit-preserving cell is mapped into a memory addressable register. The operating system writes control registers on context switch entry and exit to pause and resume microarchitectural signal propagation. Memory addressable registers are read and saved in a kernel buffer. A user program periodically requests that the operating system copy data from the kernel buffer into a user buffer that is eventually written to a file. This structure is extended for multiple signals containing multiple features per timestep.

The FPGA resources constrain the length of time to monitor. For the Xilinx ML605 evaluation kit, no more than 512 32-bit registers were used to collect data. The proposed software structure exists to collect training data and a different architectural composition would be required to perform inference.

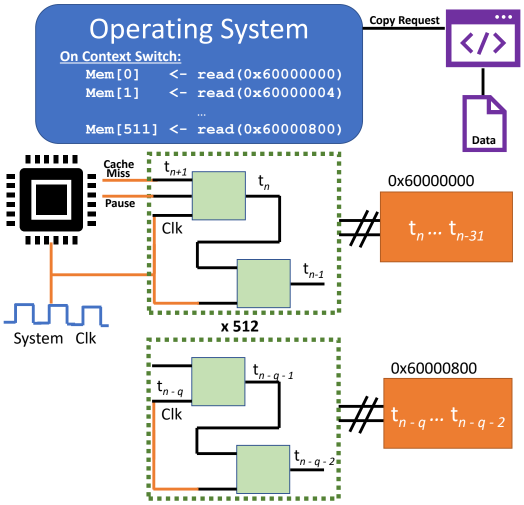

```
On Context Switch:
    Mem[0]   <- read(0x60000000)
    Mem[1]   <- read(0x60000004)
                …
    Mem[511] <- read(0x60000800)
```

Figure 1: Collecting Microarchitectural Data

The data collection scheme was validated with two different Xilinx ML605 evaluation kits, using the jump control signal. Three small RISC-V programs were written to perform 1) no jump operations, 2) exclusively jump and link operations (i.e., function calls) compiled without instructions to manipulate the frame pointer, and 3) a program that oscillates between not jumping and jumping. Data collected on both FPGAs was visually compared to confirm the expected behavior.

4.3. Experiment Design

The experimental design involves factors and levels associated with three distinct experiment layers, as summarized in Table 2. The factor and level combinations are reported alongside the results in Table 1. Although the experiment design was intended to use the same FPGA for all data collection, a second FPGA was added to reduce data collection time. Balanced classification accuracy, which accounts for class sizes, is the target variable of interest for the machine learning models described in Section 4.5.

The first experiment layer is the hardware (HW) layer, where signal connection modifications are required. Changing experiment settings in the hardware layer requires several hours of computation which constrained the number of factors considered. The microprocessor is defined using the Chisel hardware construction language (Bachrach et al., 2012). Changes to the design in Chisel must first be compiled to the target HDL and then synthesized with Xilinx tools to create a bitstream. The bitstream is then used to "program" the FPGA with the new design. The clock divisor (Clk Div) is a MATANA feature enabling detection logic to run slower than the CPU. When the clock divisor is `1`, data is logged as events occur in the CPU. When the clock divisor is greater than `1`, signals are `OR`'d together over the number of clock divisor cycles. In general, when the clock divisor is `1`, less frequent events such as cache misses appear sparsely in output data. In contrast, when the clock divisor is `16`, frequent events such as jumps saturate the data.

The second layer, data collection, is accomplished for all hardware layer configurations. When the "number of applications" is one, a single application is scheduled at a time, chosen from either the three "attack" applications, or the four "legitimate" applications, for a total of seven options. When the "number of applications" is three, one of three possible attacks is chosen alongside two different applications chosen from the set of four "legitimate" applications. In total, each hardware configuration requires 25 application runs:

$$\binom{7}{1} + \binom{4}{2} \times \binom{3}{1}$$

A subset of benchmark programs were chosen from the Coremark Pro RISC-V suite published with MATANA. The cache attack provided with MATANA successfully applies `prime+probe` to recover AES keys on an 64-set, 8-way cache. The integer and memory attacks follow the novel interrogator design described in Langehaug et al., 2021.

Table 1: Balanced Accuracy Scores for Naïve, KNN, CNN, and LSTM Classifiers

ID	Clk Div	Signals	Label	Attack	# Apps	Naïve	KNN	CNN	LSTM
0	1	cache, br, jal	binary	memory	3	0.6048	0.9689	0.9619	0.9516
1	1	cache, br, jal	multiple	integer	3	0.2080	0.5179	0.5553	0.5644
2	1	cache, br	multiple	integer	1	0.3491	0.8005	0.8781	0.8621
3	1	cache, jal	binary	cache	3	0.5939	0.6127	0.6293	0.6241
4	1	br, jal	multiple	memory	3	0.2666	0.4627	0.5213	0.5084
5	1	cache	multiple	cache	1	0.2727	0.8377	0.8688	0.8586
6	1	br	binary	memory	1	0.5660	0.8733	0.8775	0.8291
7	1	jal	binary	cache	1	0.6080	0.8264	0.8162	0.8054
8	16	cache, br, jal	multiple	cache	1	0.4837	0.9183	0.8971	0.8998
9	16	cache, br	binary	cache	3	0.5386	0.6579	0.6816	0.6806
10	16	cache, br	binary	memory	1	0.6162	0.9543	0.9473	0.9436
11	16	cache, jal	multiple	memory	1	0.4629	0.9152	0.9032	0.9027
12	16	br, jal	binary	integer	1	0.7609	0.9422	0.9308	0.8842
13	16	cache	binary	integer	3	0.5312	0.8576	0.8737	0.8761
14	16	br	multiple	cache	3	0.1763	0.2608	0.2800	0.2897
15	16	jal	multiple	integer	3	0.2049	0.3980	0.4346	0.4381

Table 2: Experimental Factors

Layer	Factor	Levels
HW	Clk Div	1, 16
HW	Signals	cache, br, jal
HW	# of Signals	1, 2, 3
Data	Attack	cache, integer, memory
Data	# Apps	1, 3
Data	Apps	linear algebra, parser neural net, radix
ML	Label Strategy	binary, multiple

The final layer, machine learning (ML), specifies whether to formulate the learning problem as a binary or a multiclassification problem. In binary classification, the model predicts attack or no attack for a single attack. In the multiclass problem, the model predicts one of many programs. Classification is only based upon data from instructions executed by a single task prior to the subsequent context switch.

Training machine learning models requires significant time. Therefore, the factors and levels were used to create a screening experiment design for 16 possible configurations with a goal of identifying main effects leading to reduced classification accuracy. Within a hardware configuration, the 25 data collection runs were randomized. Analysis of Variance (ANOVA) is applied to a linear model containing all factors to identify the significant effects on classification accuracy.

4.4. MATANA Validation

A complete validation of MATANA (Mao et al., 2022) is outside the scope of this work. However, the cache miss signal is validated as a feature to enable high detection rates for binary (attack or no attack) cache attack detection systems. To perform validation, an experiment with clock divisions of 1 and 16 was performed, observing the cache signal, and running one concurrent program. The results from this standalone classification experiment are used to support the MATANA claim that cache attacks are detected reliably with low false positive rates. Resulting classification accuracy that exceeds 90% should be considered as supporting evidence of its validity because the initial stages of an attack program often perform normal operations without correlation to an attack.

4.5. Machine Learning Process

This section describes training and evaluation of machine learning models. There are slight differences in training for KNN models versus neural network models. Both approaches assign 80% of the data as training/validation and 20% as test. For neural networks, 20% of the 80% of training/validation data was used for validation and removed from the training set.

The sequence of data produced by control signals, described as factors in Table 2, are used as features to train machine learning models. The data is structured as a time series and the experiment specification determines the number of features (i.e., channels) available in each timestep. For multiclassification, the name of the application or attack is used as the label. In binary classification the labels are “attack” or “no attack.” Due to the sequence length and trained model sizes, data preprocessing created shorter sequences that aggregated event counts for windows of time.

Preprocessing counted the number of times the signal was high per 256 timesteps using a 128-timestep stride. Neural networks could train over raw long sequences, but require significant storage for the calculated weights. For comparison, a neural network predicting the class for 16K timesteps required 256MB, whereas a neural network trained with the preprocessed rather than raw sequences, required only 100KB. This preprocessing step is roughly equivalent to multiple HPCs updating a vector of counts over time. Samples might contain only a few timesteps for the labeled class when the time between context switches is small.

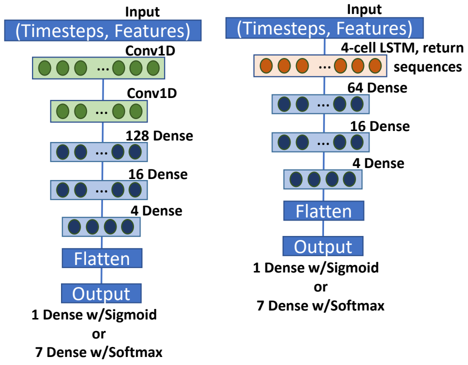
A naïve classifier was developed that calculated the mean value for each signal by class. To record test performance, a vector containing the mean for each signal in a sample is calculated. Class membership is determined based on the sample's Euclidean distance from the means found in the training data.

Each experiment configuration has a unique number of samples and distribution of classes. However there tends to be 40K-80K total samples per experiment with significant class imbalance. Some classes have as few as 2.5K samples while other classes have over 50K samples. To develop the KNN model, random *under-sampling* selects training data with class membership equivalent to the least occurring class. Test data is not under-sampled but left in the original class distribution. The balanced accuracy score accounts for class size in model performance.

The training and test data comes from the same data collection run conducted on the same FPGA. This experiment design decision was due to time constraints. An ideal experiment would compare results from independent runs on different physical devices. To consider the possible negative effects of training in this way, one model was evaluated with new data obtained from a different FPGA. The new data also included a benchmark program which was previously unseen during training. The results of this standalone experiment are discussed in Section 5.

To train the neural networks, random *over-sampling* was used to balance the training set. Over-sampling is appropriate for the neural network approach because the storage size of the weights is determined by the network structure and not the number of training samples. Validation and test data were not over-sampled.

Many parameters influence the predictive accuracy of a neural network. Given the large number of possible experiments, design choices were fixed. Many design choices could be optimized for individual scenarios. Models were trained for up to 500 epochs but early stopping halted training when validation loss did not improve. Binary classification models used binary crossentropy loss and multiclass classification used categorical crossentropy loss. The network structure was held mostly constant with input shapes changing to accommodate the shape for a hardware configuration (`batch size, timesteps, channels`). Due to FPGA constraints, as the number of channels grew, the number of timesteps was reduced.



(a) CNN Base Design (b) LSTM Base Design

Figure 2: Neural Network Structures

Figure 2 shows the basic CNN and LSTM architectures. The samples for a hardware configuration are equal width, so sequence padding is not required. Samples may include information from previously run applications. In Figure 2a the first convolution layer uses 4 filters with a kernel size of 8 while the second convolution layer uses 4 filters with a kernel size of 4. LSTMs predict sequences. However, in this research the machine learning task is sequence classification. The return sequences property shown in Figure 2b returns the hidden state output for each timestep (Brownlee, 2017). The hidden states are provided to dense layers to classify the entire sequence. The output layer uses the sigmoid activation for binary classification and softmax activation for multiclass problems.

5. Results and Analysis

This section presents experimental results to derive preliminary conclusions. These results guide future research toward specific data sources to implement constant monitoring via machine learning in microprocessors.

5.1. Data Collection Validation

The jump signal validates the functionality of the data collection and preprocessing steps. Figure 3 plots the received jump signal for four different software configurations. The rolling average is calculated for 256-timestep windows in the raw sequence to reduce visual jitter. As expected, Figures 3a-3c show the microarchitectural control signals preserving distinct software behavior. Noisy periods where a signal is inconsistently `0` or `1` could capture operating context switch code prior to the kernel driver pausing the collection hardware. Residual data from prior programs may also be present if the time between context switches is less than the capacity of the collection hardware. MATANA counts branches as a jump, so jitter near zero can be attributed to a loop condition guarding the sequence of non-jump instructions.

The jump behavior for a benchmark program shown in Figure 3d is observably distinct from the crafted jump program behaviors. The graph from the same benchmark program on a different FPGA exhibits greater jitter, indicating the potential for hardware-unique behavior, thus requiring training and evaluation with data from multiple systems.

5.2. MATANA Validation

Table 3 summarizes the cache attack detection results for a model using cache misses as the only feature. Given that not every inter-context switch contains malicious cache behavior, a direct comparison with MATANA (Mao et al., 2022) is not possible because the precise accuracy values are difficult to discern. Figure 4 compares confusion matrices for binary classification with data collected at CPU speed (Figure 4a) and 16x slower than the CPU (Figure 4b).

Table 3: Detecting Cache Attack with Cache Miss

Clk Div	KNN	CNN	LSTM
1	0.9350	0.9357	0.9191
16	0.9357	0.9227	0.8939

5.3. Machine Learning Experiments

Machine learning model performance results are summarized in Table 1. The inclusion of three different attacks makes directly interpreting the balanced classification accuracy difficult. The data from Table 1 were used to fit three different linear regression models (one for each classifier) to predict the response (balanced classification accuracy). ANOVA was used to identify significant effects with $\alpha = 0.05$. The process identified the same significant effects for all machine learning models. The significant effects were cache miss (β_1), multiclassification (β_2), and number of concurrent applications (β_3). A new linear model containing only the significant effects determined by ANOVA was then used. A Shapiro-Wilk test for normality (Shapiro and Wilk, 1965) confirmed residuals were reasonably normal. This result suggests a linear model is an adequate predictor for classifier performance at the defined factor levels.

Equation 1 predicts KNN classification accuracy ($\hat{y}$). The equation shows cache signal is related to an increase in model performance (expected behavior), while multiclassification and quantity of concurrent programs decreased model performance. The linear model achieved an R^2 value of 0.7370.

$$\hat{y} = 1.06 + 0.13\beta_1 - 0.21\beta_2 - 0.15\beta_3 \tag{1}$$

Interestingly, the attack program was not significant, suggesting that hardware monitoring could be broadly applied to detect various undesirable behaviors. These results might be of particular interest for IoT device security where a known and limited set of software behaviors should exist. The experiment screened for significant factors. Future studies may consider larger sets of applications, attacks, and device purposes.

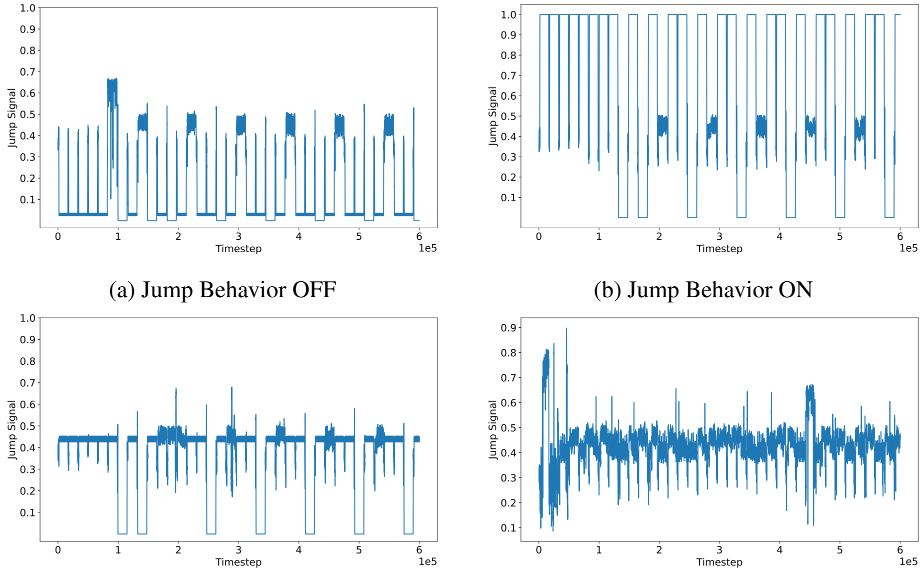
The design resulting in Table 1 does not cover all valuable experiment configurations. Figure 5 shows data from a new experiment where the sequence of cache misses is used to train a machine learning model that predicts the specific task identity (multiclass). Figure 5a uses kernel density estimation (Waskom, 2021) to visualize the distribution of events by class. The many overlapping density functions provides an indication of problem difficulty. Figure 5b is the confusion matrix for the CNN model's predictions on test data.

The original experiment design did not account for FPGA differences or previously unseen applications. The configuration from Experiment 8 was chosen to perform a standalone evaluation because it contained all three signals and the trained model reported performance above 90%. Table 4 reports the balanced classification accuracy of the trained model with data from a different FPGA and when the set of programs includes an unseen benchmark on a different FPGA.

Table 4: Performance on New Data from New FPGA

	KNN	CNN	LSTM
Same Programs	0.8562	0.8410	0.8280
Unseen Benchmark	0.8209	0.7901	0.8041

Truth-normalized confusion matrices were used to describe performance changes from the original experiment (Table 1) to "Same Programs," and "Unseen Benchmark." The confusion matrices (not shown) for the original and "Same Programs," was visually similar despite the number of correctly classified "other" programs being reduced by 40.8%. Basically, several



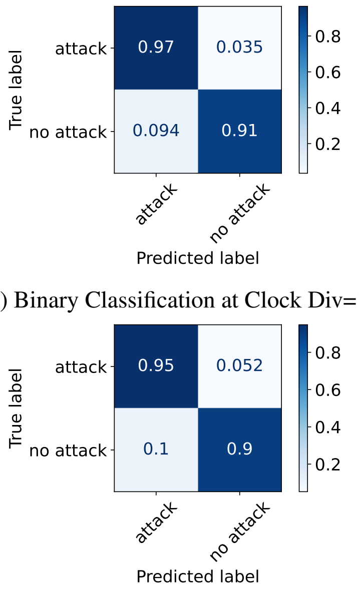
(a) Jump Behavior OFF

(b) Jump Behavior ON

(c) Jump Behavior ON/OFF

(d) Jump Behavior for Benchmark Program

Figure 3: Rolling Average for Jump Signal Over 256 Detection Timesteps at 16x Slower than CPU



(a) Binary Classification at Clock Div=1

(b) Binary Classification at Clock Div=16

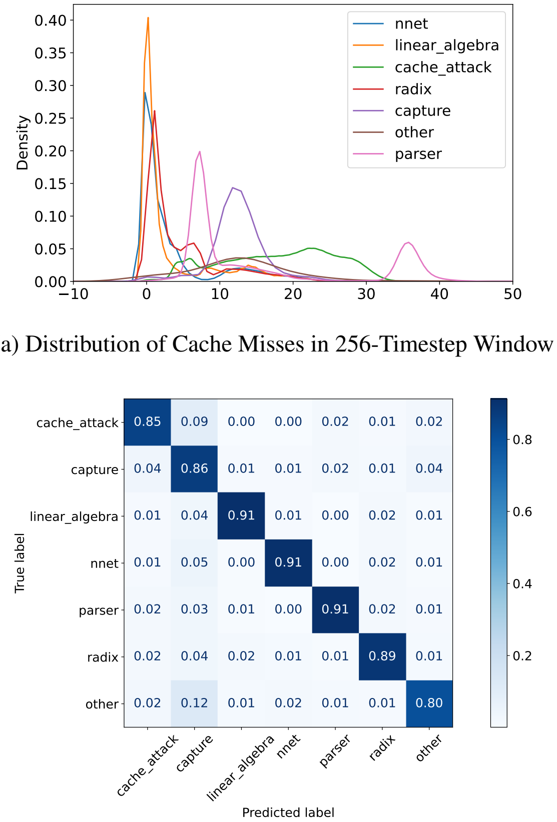
Figure 4: CNN Confusion Matrices for Attack Detection

“other” program predictions were distributed across all the classes. The “Unseen Benchmark” data was mis-classified by the model as “parser” rather than “other.” The “other” class in the original data did not include examples beyond shell scripts and routine kernel tasks, likely contributing to this behavior.

6. Conclusions

This research presented an approach to collect microarchitectural data on an in-order RISC-V softcore processor. The data was used to train and evaluate the performance of three different kinds of machine learning models with an aim to inform the development of constant monitoring, near real-time, operating system independent hardware detection mechanisms. This section summarizes the important findings while Section 7 outlines areas for future work.

This research demonstrated using microarchitectural data to identify running software and makes at least three contributions. First, the results validate the usefulness of the attack detection proposed in MATANA (Mao et al., 2022). Second, this research identified the cache miss signal, classification problem (binary or multiclass), and the number of concurrent programs as main effects influencing the balanced classification accuracy. Lastly, these results were obtained using only data from within a single operating system context switch. The accuracy achieved by some configurations suggests that high accuracy systems with low false-positive rates might be built when samples are linked across multiple context switches. Constant monitoring in hardware presents opportunities for zero trust ecosystems to reduce reliance on operating systems by independently verifying and proactively making decisions to preserve a computer system’s security.



(a) Distribution of Cache Misses in 256-Timestep Window

(b) Confusion Matrix for CNN on Cache Miss Events

Figure 5: Prediction with Cache Misses, ClockDiv=16

7. Future Work

This research lays a foundation for several areas which merit further exploration and development. The first area for future work is extending the analysis to processors with deeper pipelines, out-of-order execution, and multicore CPUs. Expanding MATANA or building similar frameworks for newer platforms would also be a worthwhile endeavor.

The second area of future work lies in the identification of context switches using hardware. A context switch is a meaningful transition in the operating system that cannot be relied upon if the end goal is to remove the inference dependency from the operating system. Examining operating system actions required to facilitate a context switch (e.g., saving user registers or invalidating the cache) could be used to design a hardware unit that recognizes them. Reliably identifying the occurrence of a context switch using microarchitectural data would allow hardware detection systems to keep statistics for the development of application profiles across many context switches. Moving the data logging operations to hardware (e.g., streaming data off the device over Ethernet) would support this area of work.

The third area of future work is instantiating trained machine learning models with inputs coming directly from microarchitectural data sources. This research area is rich with opportunity and many important questions. The following are examples of relevant questions: “Where should model weights (neural networks) or reference data (KNN) be stored and how large should it be?”, “What model structures are suitable for contemporary hardware?”, “From the time a malicious event occurs how quickly can hardware identify the malicious event?” or “How could adversaries defeat such detection schemes.” Future research in this area could lead to robust systems, significantly increasing the price adversaries must pay to succeed.

Acknowledgement

The views expressed in this paper are those of the authors, and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government. This document has been approved for public release; distribution unlimited, case #88ABW-2022-0578.

References

Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y. S., Asanovic, K., & Nikolic, B. (2020). Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, *40*(4), 10–21.

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., & Asanovi’c, K. (2012). Chisel: constructing hardware in a scala embedded language. *DAC Design automation conference 2012*, 1212–1221.

Bettoni, M., Urgese, G., Kobayashi, Y., Macii, E., & Acquaviva, A. (2017). A Convolutional Neural Network Fully Implemented on FPGA for Embedded Platforms. *2017 New Generation of CAS (NGCAS)*, 49–52.

Brownlee, J. (2017). *Difference Between Return Sequences and Return States for LSTMs in Keras*. Retrieved June 6, 2022, from https://machinelearningmastery.com/return-sequences-and-return-states-for-lstms-in-keras/

Cao, S., Zhang, C., Yao, Z., Xiao, W., Nie, L., Zhan, D., Liu, Y., Wu, M., & Zhang, L. (2019). Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity. *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 63–72.

Das, S., Werner, J., Antonakakis, M., Polychronakis, M., & Monrose, F. (2019). SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. *2019 IEEE Symposium on Security and Privacy (SP)*, 20–38.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press. Retrieved December 12, 2021, from http://www.deeplearningbook.org

Graz University of Technology. (2018). *Meltdown and Spectre*. Graz University of Technology.

James, G., & Daniela Witten, R. T., Trevor Hastie. (2017). *An Introduction to Statistical Learning: with Applications in R*. Springer, New York, New York.

Jamma, D., Ahmed, O., Areibi, S., & Grewal, G. (2017). Hardware accelerators for the K-nearest neighbor algorithm using high level synthesis. *2017 29th International Conference on Microelectronics (ICM)*, 1–4.

Karandikar, S., Mao, H., Kim, D., Biancolin, D., Amid, A., Lee, D., Pemberton, N., Amaro, E., Schmidt, C., Chopra, A., Huang, Q., Kovacs, K., Nikolic, B., Katz, R., Bachrach, J., & Asanovic, K. (2018). FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 29–42.

Karandikar, S., Ou, A., Amid, A., Mao, H., Katz, R., Nikolić, B., & Asanović, K. (2020). FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 715–731.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. (2019). Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, 1–19.

Langehaug, T., Borghetti, B., & Graham, S. (2021). Classifying Co-Resident Computer Programs Using Information Revealed by Resource Contention. *Digital Threats: Research and Practice*, *0*(ja).

Lea, C., Vidal, R., Reiter, A., & Hager, G. D. (2016). Temporal Convolutional Networks: A Unified Approach to Action Segmentation. *European Conference on Computer Vision*, 47–54.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., & Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. *27th USENIX Security Symposium (USENIX Security 18)*, 973–990.

Mambretti, A., Neugschwandtner, M., Sorniotti, A., Kirda, E., Robertson, W., & Kurmus, A. (2019). Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations. *Proceedings of the 35th Annual Computer Security Applications Conference*, 747–761.

Mao, Y., Migliore, V., & Nicomette, V. (2022). MATANA: A Reconfigurable Framework for Runtime Attack Detection Based on the Analysis of Microarchitectural Signals. *Applied Sciences*, *12*(3), 1452.

Mushtaq, M., Bricq, J., Bhatti, M. K., Akram, A., Lapotre, V., Gogniat, G., & Benoit, P. (2020). WHISPER: A Tool for Run-Time Detection of Side-Channel Attacks. *IEEE Access*, *8*, 83871–83900.

Shapiro, S. S., & Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, *52*(3/4), 591–611.

Shirriff, K. (2016). The Surprising Story of the First Microprocessors. Retrieved June 1, 2022, from https://spectrum.ieee.org/the-surprising-story-of-the-first-microprocessors

Uhsadel, L., Georges, A., & Verbauwhede, I. (2008). Exploiting Hardware Performance Counters. *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, 59–67.

Waskom, M. (2021). *Seaborn.kdeplot*. Retrieved August 30, 2022, from https://seaborn.pydata.org/generated/seaborn.kdeplot.html

Weaver, V. M., Terpstra, D., & Moore, S. (2013). Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 215–224.