Hardware-Based Detection of Spectre Attacks: A Machine Learning Approach

Yunjie Zhang and Yiorgos Makris
Electrical and Computer Engineering Department, The University of Texas at Dallas, Richardson, TX 75080, USA
E-mail:{yxz153430, gxm112130}@utdallas.edu

2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST) | 978-1-7281-8952-9/20/$31.00 ©2020 IEEE | DOI: 10.1109/ASIANHOST51057.2020.9358255

***Abstract*—The Spectre vulnerability, which has been found in a variety of processors, enables attackers to take advantage of speculative execution in modern computer architectures to access unauthorized memory content through temporal side channels. Herein, we propose a hardware-based defense mechanism that can detect Spectre attacks by utilizing the profile of a small fraction of malicious program execution. More specifically, we focus on malicious attempts to read data from theoretically inaccessible memory space. The corresponding instruction sequence is divided into consecutive *windows*, from which a performance counter-based tracker extracts descriptive features. These features are, then, processed by a trained machine learning model to analyze program behaviors and identify suspicious ones. Our experiment shows that Spectre attacks on thirteen vulnerable purpose-built victim code patterns can be detected by our system. Additionally, testing with benign benchmarks demonstrates that our framework is able to distinguish Spectre attacks from normal behavior.**

I. Introduction

As online applications and services have become an indispensable part of our society, our private data has also become the target of intense cyber-attacks. Malware is malicious software that is able to compromise security checking strategies and bypass defense mechanisms to launch Denial-of-Service (DoS) attacks or steal private data. Recently, the Spectre attack [1] demonstrated the ability to access private information without possessing appropriate privileges. Instead, it does so by leveraging a side-channel found in speculative execution which, essentially, constitutes a violation of the memory isolation property. Accordingly, methods for monitoring program execution and identifying behavior that is related to the launch of a potential Spectre attack are of great value. Generally speaking, defending against the Spectre attack can be achieved by inserting fence instructions before any speculative execution. However, this straightforward way which disables the branch predictor causes large performance overhead [2]. Thus, in most Spectre-defense solutions, the effective fence insertion strategy is applied only after Spectre behavior or vulnerabilities have been detected.

Current Spectre attack detection and defense mechanisms can be categorized into software-based methods and hardware-based methods. Binary analysis-based methods, such as *oo7* [3] and *spectector* [4], have been developed in the former category. These methods utilize binary analysis to protect potentially vulnerable parts of binaries and detect tampered speculative memory accesses. However, binaries have to be screened before being executed; thus, these methods cannot detect or take immediate action against an on-going Spectre intrusion.

On the other hand, since Spectre attacks which target speculative execution have to be launched on processors, hardware-based defense methods that block Spectre execution by taking advantage of its special cache-related behavior can also be devised. Yan et al. [5] proposed *InvisiSpec*, which detects data loads that might violate memory consistency and utilizes an extra buffer wherein it stores potentially unsafe loaded data. A similar strategy, which involves the use of a separate storage space, can be found in [6]. In another architecture design named DAWG [7], the cache side-channel is blocked by partitioning cache ways to limit data leakage and by providing strong isolation across protection domains. Despite their effectiveness, these hardware-based methods are designed to either exclusively or mainly defend against Spectre attacks. This limits their utility towards general security-related tasks, since they are not able to detect malicious behaviors other than Spectre without additional hardware modifications.

In order to address these shortcomings, we propose a hardware-based methodology that (i) performs real-time Spectre attack identification during its execution and does not require any markers or modifications in binaries, and (ii) utilizes a machine learning-based method to analyze program behavior, which can also be applied toward tasks such as workload forensics and process identification. To achieve these goals, instead of relying on memory hierarchy modifications, we focus on instructions of a program and divide its execution flow into separate concurrent small and large *windows*. More specifically, a small window contains short-term execution information, while a large window maintains long-term execution records. Descriptive features can be extracted from both types of windows and further analyzed with machine learning algorithms in order to detect an on-going Spectre attack.

To build a model that (i) combines short-term and long-term execution information, (ii) exhibits great generalization ability (i.e., performs well on previously unseen data), and (iii) learns from complex relationships among features, we follow the successful paradigm of applying the *Wide-&-Deep* [8] approach in recommendation systems. In this approach, wide linear models and deep neural networks are jointly used to make predictions. In our work, we modify the Wide-&-Deep model to fit our task, where short sequences of features extracted from successive small windows are handled by the 'Deep' part, while the feature vector of the large windows

978-1-7281-8952-9/20/$31.00 ©2020 IEEE

is processed by the 'Wide' part. Our work is evaluated on a modified version of $gem5$, an open-source, multi-architectural simulator, using Mibench as benign workloads (harmless programs). Experimental results gathered from applying attacks on variants of Spectre victim codes show an overall detection accuracy of 99.8% when the Wide-&-Deep model is deployed.

II. Related Work

Spectre, as a type of attack that exploits speculation, controls the branch predictor to speculatively execute code that will read data stored in illegal addresses before the jump condition is checked. The attack achieves this objective by forcing the branch predictor to make consistent correct predictions , so as to predispose the branch predictor toward making the same decision for the coming branch. Although illegal speculative executions will be blocked and control flow will be directed back to legal memory space right after an invalid cache access, by doing so the attack will have already successfully stored sensitive data in the cache. These data are stored as addresses, instead of their corresponding content; nevertheless, this provides a temporal side-channel for reading their value, as accessing these addresses whose value constitutes secret data is now much faster than un-cached addresses. A variety of defenses have been proposed to identify one or several of the Spectre steps introduced above. In general, these methods can be categorized into software-based and hardware-based.

A. Software-based Approaches

Microsoft's Visual C++ compiler offers a Spectre mitigation option, Qspectre [9], to enforce serial execution in binaries by inserting `lfence` instructions in detected victim locations. Retpoline [10] is also able to mitigate Spectre vulnerabilities in compiling stages. It replaces possibly unsafe indirect calls with a Retpoline sequence preventing the CPU from jumping to predicted targets. The method in [3] applies a binary analysis [11] which utilizes taint analysis, address analysis and speculation modeling to detect potentially vulnerable codes in binaries and insert fences to block speculative execution, which is the necessary condition of launching Spectre attacks. These methods, which analyze code or compiled binaries before being executed, are effective in defending against one or multiple Spectre variants; however, they are not able to detect the launch of a Spectre-attack after they have been applied.

B. Hardware-based Approaches

In contrast to software-based methods, hardware-based defense mechanisms that mitigate Spectre by interfering with critical stages of its attack can be applied *during* program execution. Among the hardware-based solutions that have been proposed, the implementations in [5] and [6] add extra buffers to the micro-architecture. In the former (InvisiSpec), instead of directly committing changes to cache hierarchy, potentially speculative loads are stored in a separate buffer which makes them invisible to other parts of the system. This temporary invisibility property of this data is enforced until it has been validated as a safe load. In the latter (SafeSpec), a similar strategy is followed, using a secure buffer to handle speculative state. If speculative instructions are identified as commitable, content stored in this buffer can be moved to normal structures. On the other hand, if a speculative execution fails, e.g., a branch predictor has given an incorrect prediction, the speculative side-effects are eliminated and no information leaks to normal structures. Methods that modify cache structure have also been considered. DAWG [7] provides a cache way partitioning scheme, in which the cache is divided into different security domains to prevent sensitive data leakage. However, these mechanisms are exclusively designed for mitigating Spectre attacks and require specialized micro-architectural changes; thus, they cannot be directly applied to other security tasks that are not related to speculative execution or cache access. Machine learning-based method [12] can also effectively detect Spectre by applying neural network in hardware performance counters analysis; nevertheless, it is not generalized for tasks other than Spectre detection.

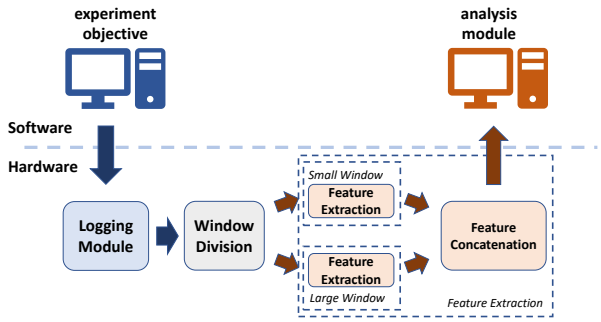

Figure 1: Overview of the proposed system architecture.

III. Methodology

An overview of the proposed system architecture can be found in Figure 1. A hardware-based logging module collects data directly from low-level hardware to avoid any software-based tampering. As mentioned in the previous section, Spectre attacks are launched through two stages, namely misleading the branch predictor and off-loading the secret data. The number of execution cycles required for each stage varies, depending on the platform being compromised as well as the amount of data to be off-loaded. This uncertainty makes it challenging to establish a fixed observation window wherein all descriptive information from both stages can be extracted. In fact, patterns of malicious behaviors extracted from a relatively short execution profile may be similar to those of benign behaviors. For example, frequent array boundary checks and mis-predicted speculations can also be found in mathematical calculation programs. This does not imply that the information extracted from short observation windows cannot be used for detecting Spectre attacks. It does, however, imply that such information may need to be placed in broader context, thereby necessitating combination with information extracted from a larger scale observation window. Therefore, our approach employs a window-division mechanism that splits the instruction flow into large and small windows.

Descriptive features are continuously extracted from both of these window types and analyzed by a machine learning model which is trained to perform Spectre behavior identification. For the purpose of training, the collected feature vectors for each window are sent to a separate secure software analysis module, where optimization of the model and selection of appropriate weights are performed. After training, the analysis module can be deployed directly on chip as a separate hardware module, thereby eliminating the need for offloading data across platforms. Once a malicious Spectre behavior is detected, fence instructions can be inserted before branch speculative execution in the current process. We discuss our algorithm in detail in the following sections.

A. Logging Mechanism

In order to reduce the amount of data to be transmitted and processed, we utilize a counter-based mechanism that summarizes the number of chosen events within a fixed number of clock cycles. As a single event (e.g., execution of a certain type of instruction) is counted independently of its context, this counter-based mechanism does not require analyzing reordered instruction sequences due to speculative execution. Previous successful application of counter-based methods in online malware detection and workload identification [13], [14] have revealed the effectiveness of analysis based on program signatures extracted from counters of architecture-level events. In order to generalize our work to other x86 machines, instead of tracking control signals that may differ among platforms, e.g., cache misses and system exceptions, we only extract features related to instructions at the fetch stage. Apart from zero-operand instructions, such as `nop` and `cflush`, most instructions can be divided into two parts, namely operators and operands. The former determine the operation type, e.g., addition, while the latter specify the data to be manipulated and their addressing types.

In practice, code snippets of Spectre attacks can be hidden in any normal program instead of being a complete process by themselves. Thus, it cannot be guaranteed that similar memory space or registers are allocated to the same snippets; instead, these can hide in different programs during compilation. In other words, the compiled Spectre attack function can have the same functionality with various addresses and/or register usage. Based on this premise, we focus exclusively on operators and discard operand information, which also helps in reducing the amount of logged data. A hash-based structure then converts operators into a corresponding index for the following step of feature extraction, which provides each operator with a unique identifier.

B. Window Division Mechanism

Our Spectre detection analysis is performed using features extracted at the granularity of a window. In our system, all basic small windows share the same *window size* which is the number of instructions fetched within a single window. The same principle is also followed for large windows. Our window extraction mechanism is geared toward collecting both long-term and short-term information. Features are extracted separately from each window type.

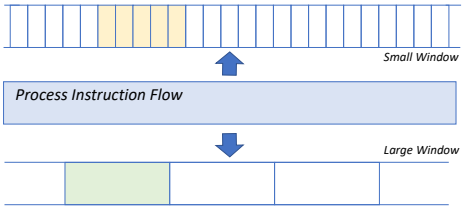

Figure 2: Instruction flow divided into two window types.

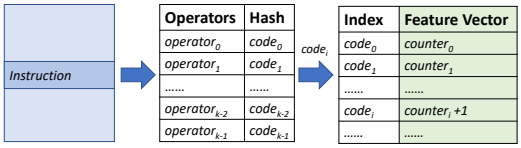

Figure 3: Feature vector generation for each window.

To avoid using extra buffers in our system, the construction of both small and large windows follows the First In, First Out (FIFO) rule that a new window will not be initialized until the previous one has been completely handled by the feature extraction module. An example is given in Figure 2, where there is no overlap between neighbouring windows. We note that large windows do not result in a larger feature vector size, as the same counter-based mechanism is applied to both types of windows. In each run, a random offset, as well as the starting point of the initial window and the feature extraction, are added at the beginning of each process to eliminate any possible bias introduced by the compiler.

C. Feature Extraction

As mentioned above, large windows and small windows contain execution information collected from different time scopes. This makes it feasible to extract features reflecting both long-term and short-term execution history. Herein, we apply two operator counters to instructions captured in both types of windows and we use operator frequencies summarized in these counters as our basic features. This counter-based mechanism has an advantage over raw instructions when the window size exceeds a threshold value. Given the total number of operator types N_{type}, the minimum number of bits required to represent an operator sequence is $ceil(\log_2 N_{type}) \times N_w$, where N_w is the size of a window and $ceil$ represents the ceiling operation. On the other hand, the bits needed to construct a vector of operator frequencies is $N_{type} \times ceil(\log_2 N_w)$. The threshold N_w can be determined experimentally, seeking to minimize the logging overhead of the counter-based mechanism. Another bottleneck of applying a smaller window size is the computational time, i.e., the time elapsed during data analysis, which should be less than the time for constructing a window to avoid the need for a buffer to cache unprocessed data.

Another advantage offered by the counter-based mechanism is that the feature vector can be constructed during execution and does not require utilization of a hardware structure to store previous instructions. As shown in Figure 3, a hash

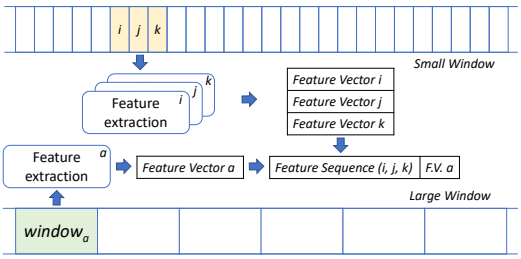

Figure 4: Feature concatenation strategy.

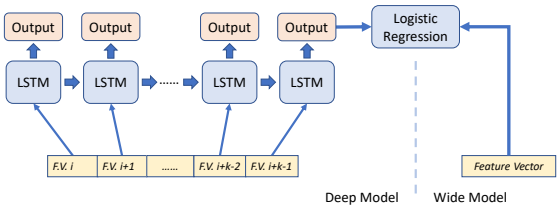

Figure 5: Structure of our model.

table is employed to decode each operator into a unique identifier, which will be used as the index of the operator counter. Contrary to previous window-like work [15], [16], where an instruction is not processed until a complete window is generated, here an instruction can be directly handled by the decoder and flushed immediately thereafter.

Additionally, we stack together multiple feature vectors of small windows to achieve a feature vector sequence, in order to record the dynamic pattern of execution among longer sequences of instructions. Figure 4 gives an example where, besides one set of counters capturing the fingerprint of each small window, another two sets of buffers are used to store the features of the last two small windows. Assuming that the length of feature sequences is K and that the feature vector of the i_{th} window is $F.V._i = < Op._0, Op._1, Op._2, ..., Op._{N_{type}-1} >$, a feature vector sequence $< F.V._i, F.V._{i-1}, F.V._{i-2}, ..., F.V._{i-k+1} >$ is generated for every K small windows. Also, another set of counters capture a fingerprint $F.V._i$ for each large window.

D. Analysis Module

As our Spectre detection method is performed at the granularity of a single window, a basic window identifier is required to perform binary classification using the features extracted from each window, with the positive class corresponding to Spectre-related behavior. The features extracted from the previous modules consist of two parts, namely *feature vector sequences* from small windows and *feature vectors* from large windows. Previous efforts [13], [14] employing simple machine-learning models, e.g., linear regression, Support Vector Machines, etc., have been shown to be quite effective in identifying malicious programs. Moreover, these straightforward types of machine learning algorithms have been able to overcome the problem of over-fitting caused by insufficient or unbalanced training datasets. In this context, this could happen, for example, because the occurrence frequency of malicious programs is much lower than that of normal programs, making it hard to collect large amounts of pertinent training data. However, these simple prediction models are unable to capture the more subtle and intricate relationships among short-term and long-term features, therefore necessitating more complex models. Similar prediction tasks can be found in research dealing with recommendation systems, such as friends recommendation in social networks and news-pushing systems. Using the nomenclature of these domains, our features are categorized into *wide* and *deep* types, based on scope and complexity. Therefore, in our work we use Google's *Wide-&-Deep* [8] scheme, which handles these feature types with different machine learning models.

To handle feature vector sequences collected from small windows, we utilize a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) [17], which has been widely applied in Natural Language Processing (NLP) tasks. As shown in Figure 5, the output vector of RNN, together with the dense feature vector collected from the previous large windows, are handled by a logistic regression model, the output of which indicates the likelihood of a Spectre attack. We note that the dimension of the RNN output vector impacts the overall performance, as vectors of larger dimensions carry more information than smaller-dimension ones, a point which we further explore in our experiments.

IV. Experimental Results

In this section, we evaluate experimentally our Spectre attack identification method and we explore optimal model configuration for this task. Our experiments are conducted on $gem5$, a multi-architecture simulator configured to work with branch prediction and out-of-order execution functionality. Our experimental results demonstrate ability of detecting Spectre attacks with high accuracy. Additionally, compared with previous work exclusively concentrating on Spectre attacks, our system can be extended to other security-related tasks.

A. Minimum Initial Window size

In our experiments, we use Spectre attacks with thirteen different vulnerable code patterns [1] as our target to protect from, alongside the MiBench testbench suite as benign processes. Applications in the Mibench suite are executed with several different arguments so as to eliminate any possible bias introduced by a fixed execution pattern. Since $gem5$ is an open-source project, it provides us with great flexibility to implement the logging module and the feature extraction module needed for our method. Here, we embed a tracer to track the instruction flow and a counter to record the total number of instructions that have been handled by the counter-based feature extractor. Each time an instruction is fetched from memory or cache, it raises a signal to the feature extraction module, instructing it to decode its operator and increment the corresponding counter. Data analysis and result evaluation are performed with TensorFlow on Python 3.6.

The x86 ISA is designed to support legacy instructions, which are no longer being used. Therefore, instead of establishing our hash-table through the x86 specification, we

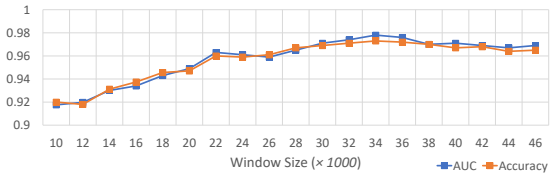

Figure 6: Accuracy and AUC of wide model.

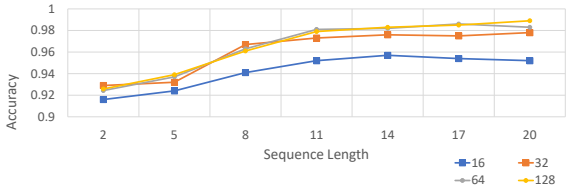

Figure 7: Accuracy and AUC of deep model.

focused only on the instructions that are commonly used by modern compilers. To identify these instructions, we first used a randomly-selected 10% of collected training samples to initialize the hash-table before using the operator counters. We collected their execution profile and identified a total of 326 types of operators that appeared during execution. A default operator type was also added to handle operators that have not been covered by the above selection process. Combining the equations discussed in Section III-C, we are able to estimate the minimum N_w by solving the inequality:

$$ceil(log_2 N_{type}) \times N_w < ceil(log_2 N_w) \times N_{type} \qquad (1)$$

where $ceil(\log_2 N_{type})$ is equal to 9. The least value of N_w, i.e., 512, is chosen as the first initial value candidate. On the other hand, the average run time of processing a set of feature vectors is 0.62*us* which sets another constrain on the window size. Assuming a clock frequency of 3.0 *GHz* and cycles per instruction (CPI) to be 1, the minimum number of instructions required to build a small window is estimated to be 1.8×10^3. Thus, to simplify the design of the counter, we adopt the closest power of two, i.e., 2048, as our initial window size, so that we leave adequate margin for data processing.

B. Identification Accuracy

There are several parameters in our system which need to be optimized, including the sizes of two window types, the output dimension of the RNN and the length of the sequential input, making it unrealistic to enumerate and experiment with all possible configuration options. Thus, we first seek the optimal size for the large window by exclusively evaluating the 'Wide' part of our model and then we seek the optimal set-up for the 'Deep' part.

In this initial experiment, we collected a dataset containing 32 Spectre and 68 Mibench execution profiles, where 70% of the samples are selected as the training set and the rest are used as the testing set. During dataset splitting, we enforced the constraint that execution profiles collected from the same program are not allowed to appear in both the training and the testing dataset, in order to avoid the controversial snooping situation where a program used for evaluation of the model has already been seen during the model training stage. Samples collected from Spectre and Mibench are labeled as positive and negative samples, respectively.

Our wide model uses a logistic regression classifier to process the large window features collected from Spectre attacks. In order to assess the robustness of our method against an unbalanced dataset, we use Area Under the ROC Curve (AUC) and accuracy as our evaluation metrics. These two metrics are summarized in Figure 6 for each window size. The results reveal that the overall Spectre identification accuracy starts to improve significantly once the window size exceeds 1.4×10^4, but stops improving after it reaches 3.8×10^4. It can also be noticed that the overall performance slightly drops when the window size is greater than 4.4×10^4. Therefore, the most effective value of 3.8×10^4 is adopted as our optimal size for large windows.

Finding the optimal configuration for the deep part of our model follows a different strategy. The output dimension of the RNN determines the bottleneck of the information that should be carried, i.e., a larger output dimension can represent longer sequences and/or larger input feature vectors. We seek the minimum output dimension and its corresponding sequence length for the deep part of our model by changing sequence lengths with various fixed output dimension setups. Our goal and expectation is to achieve similar or better performance than the logistic regression model used in the wide part. The sequence length here refers to the number of consecutive feature vectors which we rely on to make a prediction. The unified size of each small window is set to 2048, the minimum possible window size required for run-time processing of the data, as per our previous analysis. Since it is not realistic to go over all possible combinations of output dimension and sequence length, we manually selected 16, 32, 64 and 128 as candidate output dimensions. The testing results are shown in Figure 7. In all cases, except for output dimension of 16, prediction accuracy reaches higher than 97.2% when sequence length is set to 8. Thus, the second smallest dimension of 32 and a sequence length of 8 are adopted as the optimal configuration for the deep model.

Next, we combine the wide and deep model parameter choices from the above fine-tuning process to build a more effective and representative model that memorizes both long-term background in its wide parts and short-term changes in its deep parts. The output vector of the deep model and the input feature vector of the wide model jointly identify Spectre-related behaviors through a logistic regression output layer. We conducted multiple iterations of our experiment, each time randomly selecting 70% of the processes in the dataset as our training set, while keeping the remaining 30% as the testing set. Results from 5 random iterations with the same model structure are summarized in Table I. The table provides the accuracy metric (i.e., correctly predicted samples over all predictions), the false positive (FP) and the false negative (FN) rates. We remind that, in our case, Spectre samples are defined as the positive class. As may be observed, this

straightforward method of combining the wide and deep parts provides significant improvement over using separate models.

C. Model Generalization

Besides running experiments exclusively on detecting Spectre, we also applied our model in detecting other behaviors, in order to assess its capability to handle more general tasks. Using the same dataset collected from running MiBench and Spectre, we discarded malicious Spectre samples and sought to perform *process identification* on the rest of the samples. More specifically, in each experiment iteration, a randomly selected process is labeled as our detection target (i.e., positive samples) and the other processes are labeled as negative samples. A Wide-&-Deep model of similar structure to the one utilized for Spectre detection is, again, employed in this binary classification problem, to evaluate its generalization ability.

The accuracy of this approach for identifying each of the processes that we experimented with is summarized in Figure 8. As may be observed, our model exhibits excellent process identification performance, reaching an overall accuracy of 94.3%. We also investigated the source of the accuracy loss and concluded that it stems from our feature extraction stage, where we collect features from execution profiles of all privileges, without distinguishing system-level instructions from user-level ones. Execution of system calls, such as basic I/O function `printf`, have a similar execution pattern in different processes, thereby contributing to this drop in classification accuracy. Despite this drop, the high accuracy achieved confirms that, besides Spectre detection, our model can assist with other security-related tasks as well. Thus, it outperforms previous solutions which specifically target only Spectre and cannot be used for more general tasks.

V. Conclusion

We discussed the feasibility of performing hardware-based real-time detection and defense against Spectre attacks. Compared with previous hardware-based approaches that target exclusively Spectre attacks, our method can be generalized and utilized for other security-related tasks, such as process identification. Furthermore, unlike software-based methods, which perform detection before malicious code has been compiled or executed, the proposed method is a run-time solution and can be applied during code execution. Our system extracts features from the instruction flow directly through the hardware and subsequently constructs small and large windows. Features extracted from these windows are, then, analyzed further through trained Wide-&-Deep machine learning models, in order to identify the presence of any Spectre-related behaviors. We demonstrated our method on a modified version of a multi-architectural simulator, i.e., *gem5*, running malicious programs that contain the Spectre attack, as well as benign code from the Mibench benchmark suite. Overall, correct Spectre behavior identification accuracy of 99.8% is achieved when short-term and long-term counter-based features are jointly considered by our method.

Table I: Summary of accuracy and FP/FN rates of our model

Test #	average	test 1	test 2	test 3	test 4	test 5
Accuracy	99.76%	99.60%	99.81%	99.72%	99.86%	99.80%
FP Rate		0.43%	0.19%	0.18%	0.08%	0.15%
FN Rate		0.26%	0.18%	0.25%	0.11%	0.22%

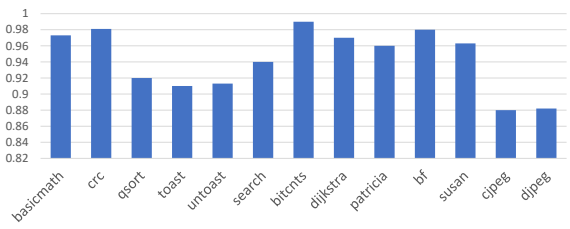

Figure 8: Accuracy of process identification.

References

[1] P. Kocher, "Spectre mitigations in microsoft's c/c++ compiler," https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.

[2] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*, 1981, pp. 135–148.

[3] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via binary analysis," *arXiv preprint arXiv:1807.05843*, 2018.

[4] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," *arXiv preprint arXiv:1812.08639*, 2018.

[5] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[6] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[7] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 974–987.

[8] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 7–10.

[9] A. Pardoe, "Spectre mitigations in msvc," https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc.

[10] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886.

[11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*, 2011, pp. 463–469.

[12] J. Depoix and P. Altmeyer, "Detecting spectre attacks by identifying cache side-channel attacks using machine learning," *Advanced Microkernel Operating Systems*, vol. 75, 2018.

[13] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *International Workshop on Recent Advances in Intrusion Detection*, 2014, pp. 109–129.

[14] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 559–570.

[15] Y. Zhang, L. Zhou, and Y. Makris, "Hardware-based real-time workload forensics via frame-level tlb profiling," in *37th IEEE VLSI Test Symposium (VTS)*, 2019, pp. 1–6.

[16] Y. Zhang, L. Zhou, and Y. Makris, "Hardware-based real-time workload forensics," *IEEE Design & Test*, vol. 37, no. 4, pp. 52–58, 2020.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.