Fast and Efficient Deployment of Security Defenses Via Context Sensitive Decoding

Mohammadkazem Taram, Dean Tullsen
Computer Science and Engineering
University of California, San Diego

Ashish Venkat
Computer Science
University of Virginia

Hossein Sayadi, Han Wang,
Sai Manoj PD, Houman Homayoun
Electrical Engineering
George Mason University

***Abstract*—Modern CPU cores feature translation of instructions into internal instructions, often called micro-ops, for simplified CPU design and improved instruction throughput. However, this translation is static in most known instances. This paper gives an overview of *context-sensitive decoding* (CSD), a technique that enables customization of the micro-op translation, based on the execution context or particular hardware triggers.**

This enables rapid deployment of security defenses, enabling changes to the instruction stream without the need for recompilation, translation, or interpretation of the original code. In addition, because the alternate decodings can be turned on and off as quickly as a single cycle, it enables the defense to be strategically deployed only on those instructions that require it, minimizing performance overhead.

In this work, CSD is paired with a novel machine-learning based attack detection mechanism, allowing the system to adapt the level of protection in the presence of suspected malicious code.

***Keywords*—security defense; microcode; side channel; attack detection**

I. Introduction

Modern computing systems are highly vulnerable to security attacks. In many applications, compromised security comes with an extremely high cost. However, it also true that in many cases failure to compute at cutting-edge speeds can also result in significant losses. As a result, we need systems that enable secure computation with minimal performance cost.

Context-sensitive decoding (CSD) exploits the fact that modern processors typically employ a translation layer, invisible to the user, that translates the user instruction-set architecture (ISA) to an internal ISA. By modifying this translation, we can instantly change the security features of any code, even code introduced by external users. In this project, we combine context sensitive decoding with malware and attack detection mechanisms [1], further enabling it to customize to a particular attack, or raise and lower the defense level according to the likelihood or severity of attack.

These solutions provide a security defense strategy that provides high coverage, with minimal architectural impact, greatly reduced performance impact, complete software compatibility (no software changes), and small power and area cost.

All code that runs on a processor, from user code to kernel, trusted or untrusted, even attack code itself, must run through the hardware decoder in the pipeline and is subject to whatever level of hardware defense we have currently enabled via CSD. Any system in which ALL software that runs cannot be proven secure can be made more secure through these techniques.

In this paper, we give an overview of Context Sensitive Decoding and briefly describe some use cases.

Because CSD can be easily turned on or off or defend against different types of attacks at different times, in this research we also propose novel machine-learning based malware detection techniques. This allows the system to adapt the level of defense, or the type of defense, based on the likelihood of particular types of attacks. This is accomplished through a machine-learning based classifier that observes performance counter data on the running code.

II. Architectural Approach

This architecture takes advantage of an underutilized feature of modern instruction set decoders in order to provide security guarantees, or quickly deploy new security features or defenses, for legacy code without recompilation or binary translation, and with minor changes to existing hardware. Most modern processors employ translated ISAs (instruction set architectures), as the Intel and AMD x86 processors and many ARM and SPARC processors typically feature translation from the native instruction set into internal micro-ops that enter the pipeline for execution [2], [3], [4]. These architectures enjoy the dual benefits of a versatile backward-compatible CISC front-end and a simple cost-effective RISC back-end. Moreover, the additional level of indirection enables seamless optimization of the internal micro-op ISA, under the covers, without any changes to the programmer interface. However, for those architectures the translation is static, changing once per generation. Instead, we propose that translation be dynamic, potentially changing frequently within the execution of a single program.

This allows native instructions to be decoded/translated into a different set of custom micro-ops based on their current execution context. This presents operating systems, runtime monitoring systems, antivirus programs, and other malware-detection hardware with the unique opportunity of triggering different custom secure translation modes with varying levels of protection, at microsecond or finer granularity, by simply configuring a set of model-specific registers (MSRs).

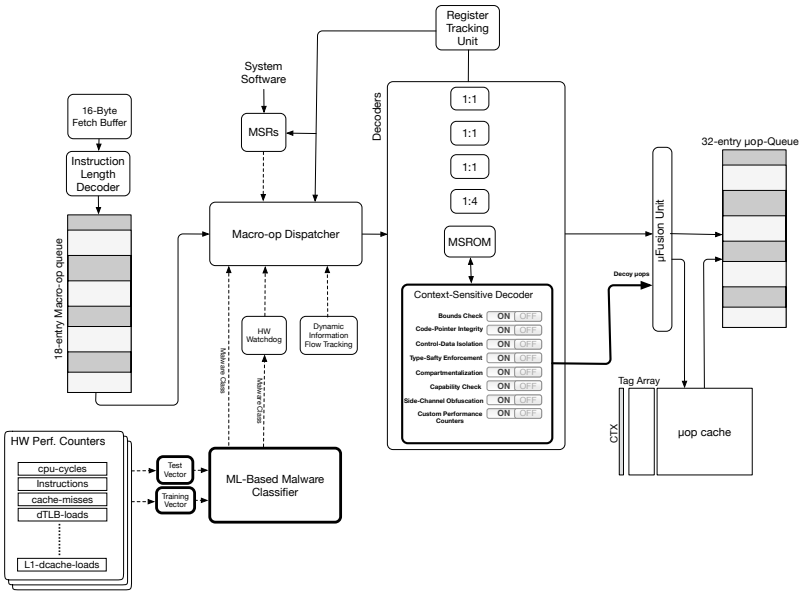


Fig. 1: Intel Front End with CSD Support.

This allows an insecure legacy executable to instantly become a secure executable without recompilation or binary translation, thereby simultaneously improving security without compromising software compatibility. Furthermore, the extra (security-enhancing) micro-ops injected into the instruction stream by custom decoders are unreadable from both user and kernel modes as they exist within the processor outside any addressable memory. As a result, they remain invulnerable to spyware, rootkits, and other rogue programs, even if those programs execute with the highest privileges.

To enable context-sensitive decoding in, for example, the Intel front-end, we plan to provision the legacy decode pipeline with one or more custom decoders that perform custom secure translations (see Figure 1) – notice that this is enabled without sacrificing the effectiveness of other pipeline front-end optimizations such as the micro-op cache, micro-op fusion, etc. The custom translations could feature micro-ops that perform bounds checking, code-pointer integrity checks, control-data isolation, type-safety enforcement, compartmentalization, capability checks, side-channel obfuscation, and implementation of custom performance counters that inform the learning-based malware detection model we employ. Furthermore, in order to conserve area and power, we plan to employ a simple static table-driven translation model for custom decoders, similar to the existing native x86 decoders. However, we note that it is possible to generate more sophisticated micro-op flows by relegating to the microcode ROM, if and when necessary.

To enable both flexibility and adaptability, we allow one or more custom secure translation modes to be triggered in the following different ways. First, they can be initiated by software, for example if the operating system or an antivirus program have identified certain new vulnerabilities/exposures – they trigger this logic by configuring a set of model-specific registers (MSRs). Second, dynamic information flow trackers and our hardware-based malware detectors can trigger secure translation modes that are specific to the identified malware class, and based on the ambient threat level. Finally, as an option to trade off performance for security, when performance is not critical, we plan to implement a hardware watchdog timer to periodically toggle between native and secure translation modes – exploiting the fact that our defense mechanisms stay in effect some time after initial triggering, and attack modes typically have some maximum attack frequency.

In the next section, we describe one of these possible triggers, our machine-learning based malware and attack classifier. Later in this paper, we will describe specific examples of context-sensitive decoding defenses.

III. Machine Learning-based Attack Detection

As seen in the previous sections, one of the triggers to enable the defense is through the machine learning (ML)-based attack detection. With advancements in technology, attacks can be crafted through applications or side-channel information such as timing, or data obtained through the covert channels. In this work, we deploy hardware-assisted ML-based techniques to detect attacks deployed by executing malicious applications (malware) on the processor or side-channel attacks such as Flush+Reload [5] and Spectre [6]. It is hardware-assisted in that we employ the available hardware related information such as microarchitectural events for detecting the exploits. The embedded hardware performance counters (HPCs) are utilized in this work along with ML classifiers for detecting the security exploits by the attackers as described below. HPCs are a limited set of registers in the microprocessor to capture the microarchitectural events such as instructions executed, cache-misses, hits of an executing application, etc.

A. Runtime Malware Detection Through HPCs

For traditional computing systems, several techniques have been explored for malware detection including dynamic bi-

nary instrumentation [7], anomaly detection [8], information flow tracking [9], and so on. There also exist traditional approaches such as semantic [10], [11] and signature-based [12], [13] solutions including off-the-shelf anti-viruses as well. However, most of these techniques are slow, and require heavy computational resources and memory [14], [15], making them infeasible to be adopted for runtime malware detection. Hardware-assisted malware detection emerged as one solution to these challenges.

One of the first works that introduced use of HPCs for malware detection is [15]. It uses ML models for malware detection. Similar works are explored in [16], [17], [18], [19], [20]. However, due to design constraints and complexity, the number of embedded HPCs are limited. As such, to perform runtime malware detection i.e., detect a security threat at runtime, it is non-trivial to effectively choose a subset of microarchitectural events that are equivalent to the full number of physically available HPCs. This calls for effective feature selection. As different ML classifiers achieve different performances, in this work, we study and evaluate different kinds of ML classifiers in the context of runtime malware detection that can be embedded in the utilized framework, shown in Figure 1.

Figure 2 depicts the overview of the deployed runtime malware detector that encompasses a systematic feature selection and classification. The feature selection step is performed offline, whereas the extraction and classification are performed online as given below.

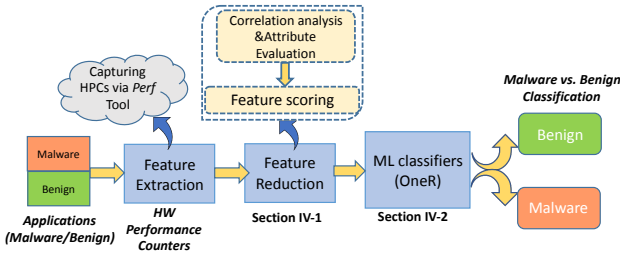


Fig. 2: Overview of the runtime malware detector

Feature Selection: The main objective of the detector is to perform attack detection during runtime. As such, the first step is to determine the set of features (microarchitectural events) that match the full suite of HPCs. To achieve this, we employ feature selection and reduction techniques. Thus, the chosen features will represent the most impacting events that determine the performance of attack detection. This feature selection process is performed offline, and during runtime (evaluation phase or when deployed in real-system) only those events are captured and fed to the ML classifier for attack detection.

For the utilized experimental setup, we collected diverse microarchitectural events (44 in our case) by iteratively executing the applications in a sandbox environment (described later). The maximum number of microarchitectural events captured during one execution of an application is limited to the number of physically available HPCs, hence applications need to be iteratively executed to capture all the possible events. For feature selection and reduction, we apply "Correlation Attribute Evaluation" to rank the most critical microarchitectural events (HPC events). The correlation attribute evaluation technique primarily determines the Pearson correlation coefficient as given below.

$$\rho(i) = \frac{cov(Z_i, C)}{\sqrt{var(Z_i)\ var(C)}} \quad i = 1, ..., 44. \tag{1}$$

Here, the Pearson coefficient is represented by ρ and Z_i denotes the i-th input feature (maximum of 44 in our case) and C represents the output class i.e., "Malware" or "Benign" in our case. The covariance between elements are represented by the function $Cov(.)$ and the variance by $Var(.)$. Thus, we determine the most critical features for malware detection with branch instructions being the pivotal event.

ML Classifier based Malware Detection: Once the non-trivial microarchitectural events are determined, those events are captured during the runtime to train the ML classifiers for classifying the benign applications from malicious applications.

1) Evaluation of Malware Detection: Applications (both malware and benign) are executed on an Intel Xeon X5550 processor with Ubuntu 14.04 OS having Linux 4.4 Kernel. We use the *Perf* tool for extracting the HPCs values during runtime. We evaluate on a diverse set of applications such as MiBench [21] and SPEC2006 [22], Linux programs, and browsers (to represent benign applications). For malware, we included Linux ELFs, python scripts, perl scripts, and bash scripts that represent four classes of malware: 452 Backdoor, 350 Rootkit, 650 Virus, and 1169 Trojan.

TABLE I: Evaluation of different ML classifiers when deployed in HMD using 4 HPCs

Classifier	Accuracy (%)	Area (%)	Power (mW)	Latency (@10ns)	F1-score
MLP	93.03	41.5	0.78	93	0.93
JRip	**91.08**	**0.2**	**0.28**	**1**	**0.92**
Logistic Reg.	92.21	19.9	0.55	58	0.92
SVM	81.55	4.1	0.42	13	0.82
J48	92.62	0.9	0.26	3	0.93
SGD	92.21	4.1	0.39	13	0.92

Table I presents the 10-fold validation of the malware detection performance and the silicon overhead incurred by the malware detector that employs 4 HPCs. As the software implementation of ML classifiers for malware detection is slow (in the range of tens of milliseconds) which is an order of magnitude higher than the latency needed to capture malware at runtime [14], hardware implementation is performed in this work. The deployed HMD's hardware footprint is evaluated on a Xilinx Virtex-7 FPGA. Based on the analysis, we deploy OneR classifier for malware detection and signaling the attack in this work.

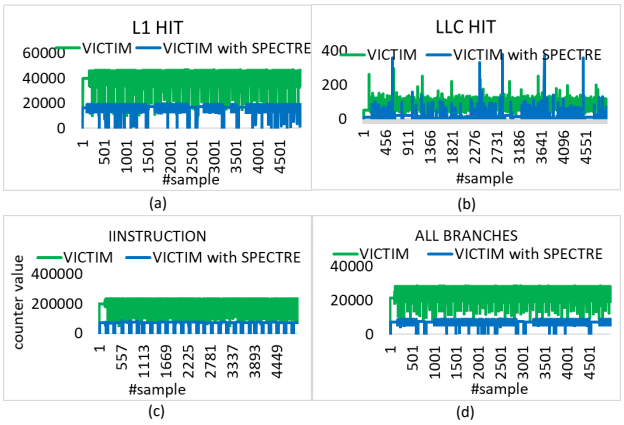


Fig. 3: HPC patterns under Spectre attack: (a) L1 hits; (b) LLC hits; (c) retired instructions; and (d) retired branch instructions

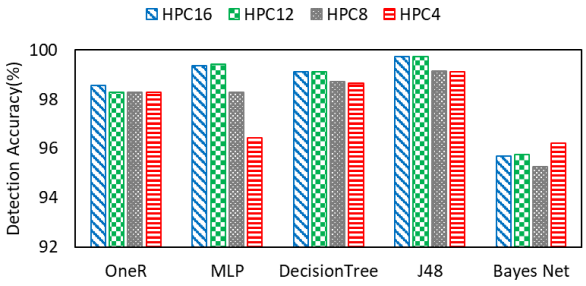


Fig. 4: Performance of side-channel attack detection with different ML classifiers and number of HPCs

B. HPCs for Side-Channel Attack Detection

Side-channel attacks are another class of security threats that can exploit the security vulnerabilities of the system. In this work, we intend to also utilize the previously described framework for side-channel attack detection. Some of the previous works such as [23], [24], [25], [26] propose utilizing the HPCs for attack detection. However, the analysis is limited and confined to a limited set of security vulnerabilities.

In this work, we capture the HPCs for a given set of applications executing on the system using the Intel-PCM tool. Simultaneously, we also run Flush+Reload [5], Prime+Probe [27], and Spectre [6] for evaluating the HPCs trends.

Figure 3 shows different HPC patterns under the Spectre attack. The Y-axis represents the HPC value and the sample in the X-axis represents the time-instant ($\times 50\mu s$) at which the HPC is captured. As seen, for a system under attack, the L1 hit count goes down and the LLC hit also reduces. This happens due to the fact that the Spectre attack also attacks the L1 cache and last-level cache (LLC), similar to Flush+Reload. In addition to the cache hits, the number of retired branch instructions also reduces. This is observed due to the fact that the employed Spectre attack mistrains the branch predictor leading to a significant reduction in the retired branches. With the increase in cache misses (or reduction in cache hits) and mistraining of the branch predictor, the instruction rate also reduces, as shown in Figure 3. Similar to malware detection, we evaluate the side-channel attack detection by feeding the ML classifier with the HPC events. However, instead of directly feeding the HPC values, we convert the time-series (HPC data captured in a given time-interval) into a subset of features (mean, maximum value and the sum) for classification.

Evaluation: We carried out the experiments on an Intel i5 processor with Ubuntu 16.04 OS. The side-channel attacks we use are Flush+Reload (with RSA and AES as victim programs) obtained from Mastik [28] and Xlate [29]. Similarly, Prime+Probe attack on RSA as the victim program is obtained from Mastik [28]. Spectre is executed based on [6] with multiple victim programs. We are able to detect the above mentioned attacks i.e., Spectre, Flush+Reload and Prime+Probe with an accuracy of more than 95%, as shown in Figure 4. As seen, with the reduction in the number of HPCs used, the detection performance also reduces with most of the classifiers.

IV. Specific Defense Case Studies

Context-sensitive decoding can provide both security and non-security advantages. Non-security applications include fast debugging (e.g., minimal overhead watchpoints), custom performance counters, and low-cost profiling requiring no manipulation of the binary. Below are some potential security applications.

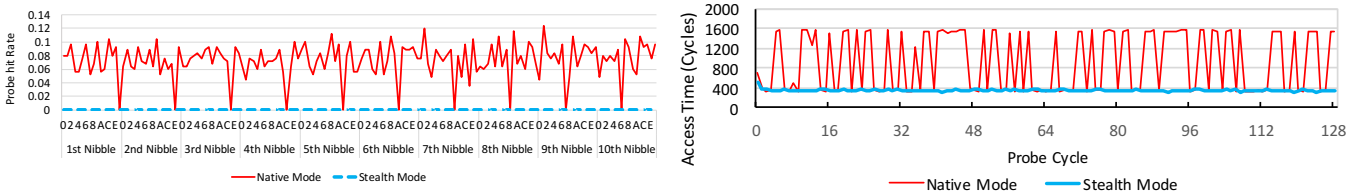
A. Cache Obfuscation

Many attack codes utilize side channels to observe data at some point. By identifying key secure data structures, we can replace individual loads with a loop that brings the entire structure in the cache, thus leaving no data-dependent footprint. We show this completely eliminates any signal left by known-vulnerable AES and RSA codes, with only a 5% performance cost [30]. This is far below existing software cache obfuscation schemes. In addition to data cache obfuscation, it can also be used on the instruction cache (by instrument key branches) that prevent control flow asymmetries from being exposed to the instruction cache.

To evaluate the effectiveness of our CSD-based cache obfuscation technique (stealth-mode translation), we subject AES and RSA running on our architecture to the FLUSH+RELOAD and PRIME+PROBE variants of cache-based side-channel attacks [31]. Figure 5a shows PRIME+PROBE attacker's perspective with and without our obfuscation technique. Furthermore, as shown in figure 5b our stealth-mode translation can completely obfuscate I-cache access pattern as a defense against FLUSH+RELOAD attack on the RSA algorithm [32]. These results are also shown in [30], and much more detail on the technique and the results can be seen there.

B. Dynamic Scheduling Control

Attacks such as Meltdown [33] and Spectre [34] exploit speculative execution to bypass security guarantees that only



(a) PRIME+PROBE on AES

(b) FLUSH+RELOAD on RSA

Fig. 5: Effect of the cache attacks on AES and RSA with stealth-mode translation enabled.

guard committed execution. The primary solutions involve recompilation with frequent fences (instructions which inhibit reordering of instructions across the fence). But those come at a heavy cost. With fine control of the instruction stream allowing us to strategically insert expensive fences only when necessary, dropping the overhead (vs. a more naive placement scheme) by a factor of 3 [35].

Other security applications of CSD include dynamic bounds checking, dynamic type checking, low-overhead dynamic information flow tracking, stack smashing detection and protection, code-pointer integrity, and branch predictor obfuscation.

V. Conclusion

In summary, then, context sensitive decoding has the potential to provide software-level protection (high flexibility, adapting to new attacks) at the hardware level (minimal overhead, unavailable to attackers) all with minimal changes to the pipeline – primarily just the decoder tables. In addition, by exploiting already existing mechanisms companies like Intel use to update the microcode, it provides fast deployment of new defenses without recompilation of the entire code base.

This paper also presents initial work on machine-learning based detector for malware and specific security attacks. They provide accuracy as high as 95% even when using only a small set of available performance counters. When implemented in hardware, the classifier has the potential to enable detection and adaptation to ongoing attacks at runtime.

By pairing these technologies, given context-sensitive decoding's ability to turn on and off various defenses, and adapt quickly to different potential threats, we can create a system that maximizes performance in the absence of threats, yet still reacts aggressively in the presence of possible attack.

VI. Acknowledgments

This paper represents research supported by DARPA under the agreement number HR0011-18-C-0020, titled "Mobilizing the Micro-Ops: Securing Processor Architectures via Context Sensitive Decoding".

References

[1] H. Sayadi, N. Patel, S. M. P D, A. Sasan, S. Rafatirad, and H. Homayoun, "Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification," in *Proceedings 2018 Design Automation Conference*, 2018.

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, Intel Corporation, August 2011.

[3] *ARM Cortex-A57 MPCore Processor Manual*, ARM Limited.

[4] *ARM Architecture Reference Manual*, ARM Limited.

[5] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium*, 2014.

[6] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan 2018.

[7] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *ACM Conference on Computer and Communications Security*, 2008.

[8] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "BotHunter: Detecting malware infection through ids-driven dialog correlation," in *USENIX Security Symposium*, 2007.

[9] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu, "Gotcha - sly malware!: Scorpion a metagraph2vec based malware detection system," in *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[10] M. Ozsoy, C. Donovick, I. Gorelik, N. B. Abu-Ghazaleh, and D. V. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *IEEE International Symposium on High Performance Computer Architecture*, 2015.

[11] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.

[12] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for signature-based protection from code reuse attacks," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[13] A. A. Elhadi, M. A. Maarof, and A. H. Osman, "Malware detection based on hybrid signature behaviour application programming interface call graph," *American Journal of Applied Sciences*, vol. 9, no. 3, p. 283, 2012.

[14] N. Patel, A. Sasan, and H. Homayoun, "Analyzing hardware based malware detectors," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017.

[15] 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 *International Symposium on Computer Architecture*, 2013.

[16] M. B. Bahador, M. Abadi, and A. Tajoddin, "HPCMalHunter: Behavioral malware detection using hardware performance counters and singular value decomposition," in *International Conference on Computer and Knowledge Engineering*, 2014.

[17] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *Asia Conference on Computer and Communications Security*, 2017.

[18] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, "Hardware performance counter-based malware identification and detection with adaptive compressive sensing," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, Mar 2016.

[19] X. Wang and R. Karri, "Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, Mar 2016.

[20] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *ACM Workshop on Scalable Trusted Computing*, 2011.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded

benchmark suite," in *Proceedings of the International Workshop on Workload Characterization*, 2001.

[22] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006.

[23] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *Research in Attacks, Intrusion and Defense*, 2016.

[24] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, "Cacheshield: Protecting legacy processes against cache attacks," *CoRR*, vol. abs/1709.01795, 2017.

[25] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, no. C, Dec 2016.

[26] TrendMicro, "Detecting attacks that exploit meltdown and spectre with performance counters," https://blog.trendmicro.com/trendlabs-security-intelligence/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters/.

[27] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[28] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," https://cs.adelaide.edu.au/ yval/Mastik/.

[29] VUSec, "Xlate," https://www.vusec.net/projects/xlate/.

[30] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[31] M.-C. W. K. Gruss, Daniel and S. Mangard, *Flush+Flush: A Fast and Stealthy Cache Attack*, 2016.

[32] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security*, 2014.

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

[34] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[35] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019.