Guided Processor Security Fuzzing Scheme Based on DQN

Yunkai Cui, Hanning Dong

Abstract— **In recent years, with the emergence of vulnerabilities such as "Spectre" and "Meltdown", efficient testing of processor security has become an important issue for research. Fuzz testing has also been applied in processor security testing in recent years. However, genetic algorithms exhibit a certain degree of blindness when performing fine-grained mutation, which can lead to the generation of test cases that trigger the same type of vulnerabilities, thereby failing to cover more types of vulnerabilities in a short period of time. To address the above issues, a guided processor security fuzzing scheme based on DQN reinforcement learning is proposed. This scheme constructs a reward function by associating the state values of test cases with the weights corresponding to the triggered vulnerability types. It employs reinforcement learning to guide the generation of test cases with targeted and directional mutations, thereby quickly triggering different types of vulnerabilities. Currently, there is no similar fuzz testing system in the industry. Experiments conducted on the ARMv8 platform demonstrate that compared to traditional approaches using genetic algorithms as feedback, this scheme significantly reduces the required time while covering an equivalent number of branch prediction vulnerabilities. The efficiency of vulnerability detection is improved by 1.29 times.**

Index Terms— **Processor vulnerability detection, fuzz testing, DQN reinforcement learning, ARMv8, branch prediction vulnerabilities.**

I. INTRODUCTION

With the rapid advancement of computer hardware technology, the microarchitecture of processors has become increasingly complex, posing numerous security challenges. Particularly in recent years, the discovery of vulnerabilities such as Spectre [1] and Meltdown [2] has revealed the severe security threats faced by modern processors. The existence of these vulnerabilities not only jeopardizes data security but also exposes shortcomings in existing processor designs and testing methods. Therefore, the development of more effective methods for exploring processor vulnerabilities and security testing has become an urgent need.

The latest research utilizing fuzz testing principles for processor vulnerability testing is exemplified by Moghimi et al., who developed a tool named *Transynther* [3] to scan MDS [4] vulnerabilities. Although *Transynther* is also a fuzz testing tool, its implementation is incomplete, lacking a feedback mechanism and being highly restrictive in processor monitoring methods. Additionally, the generation module of *Transynther* itself has a simplistic structure and can only test MDS-like vulnerabilities. However, where this tool can be informative is in its method of generating test cases through code block assembly.

Traditional fuzz testing systems employ genetic algorithms [5] for feedback in the test case generation module. Due to the fact that genetic algorithms evaluate the entire solution in each generation and perform selection and mutation at a higher level, while the test cases in this fuzz testing system are based on instruction blocks, the finest granularity of genetic algorithm mutation operations is at the level of instruction blocks. This leads to a certain degree of blindness in test case mutation, resulting in frequent triggering of the same type of vulnerabilities during actual testing and significantly reducing testing efficiency. According to Sutton and Barto's research [6], reinforcement learning, with its unique reward mechanism and policy gradient methods, can effectively guide fine-grained behavioral adjustments, making it significantly superior to traditional genetic algorithms when handling tasks with complex state spaces. This suggests that reinforcement learning is more suitable for making action selections at a very fine-grained level, where each action is chosen based on the current state and learned policy, making it more suitable for guiding fine-grained mutations at the level of instruction blocks.

Based on the advantages of reinforcement learning in handling fine-grained mutations, a high-performance guided processor security fuzzing scheme based on DQN(Deep Q-network) reinforcement learning [7] is proposed. Firstly, by defining the state values of test cases, they can accurately understand and represent the state of processor microarchitecture under attack. Secondly, utilizing the defined state values and weights corresponding to the triggered vulnerability types of test cases, a reward function is constructed. The DQN reinforcement learning model is updated based on the state and reward, and the mutation strategy is optimized according to the obtained model to generate targeted and guided test cases. The innovation of this research lies in defining the state values of test cases and constructing corresponding reward functions, applying DQN algorithm to fuzz testing, and optimizing the generation and mutation process of test cases. Currently, there is no similar fuzz testing system for branch prediction class processor vulnerabilities in the industry. Experiments conducted on the ARMv8 platform demonstrate that compared to traditional approaches using genetic algorithms as feedback, this scheme significantly reduces the required time while covering an equivalent number of branch prediction class vulnerabilities. This not only provides a new approach for processor security testing but also offers important references

Manuscript received May 24, 2024
Yunkai Cui, Computer Science, Beijing Information Science and Technology University, Beijing, China
Hanning Dong, Computer Science, Hebei Normal University for Nationalities, Chengde, China

for future more secure processor designs and security strategy formulation.

II. BACKGROUND

A. Processor Security Vulnerabilities

In recent years, the Spectre vulnerability, discovered, exploits flaws in the branch prediction mechanism. When a processor executes a branch instruction, it cannot immediately calculate the target address of the branch jump. The branch predictor predicts possible jump addresses and continues to fetch and execute instructions in the pipeline. When the processor detects a prediction error, it rolls back the processor state. However, if the predicted execution instructions modify the state of components such as the Cache, the processor does not roll back these components. Attackers can exploit this by using Cache side-channel attacks to steal sensitive information. The Spectre vulnerability can be classified into three types based on the different components of the branch prediction unit filled in the first step: Spectre v1 [1], Spectre v2 [1] and Spectre RSB [8]. In addition, some research has derived different attack methods based on the variants of these three Spectre vulnerabilities. For example, Zhang [9] et al. found that partial address bit matching can trigger Spectre v2 on Intel processors. SGXPectre [10], NetSpectre [11],and SmoTherSpectre [12] adaptively modify Spectre attacks for different application scenarios. The MDS vulnerabilities detected by Transynther are not entirely the same as Spectre and Meltdown. It can leak data from various buffers inside the processor, and depending on the source and method of data leakage, it has different variants. Fallout [13] leaks data from the Store Buffer, RIDL [13] leaks data from the Line Fill Buffer, and ZombieLoad [14] and Medusa [3] are variants of RIDL with the same basic principle but different attack methods. There are also attacks targeting other components. For example, Foreshadow [15] can attack Intel SGX, with a basic principle similar to Meltdown but requiring leaked data to be kept in the L1 Cache. TLBleed [16] exploits the feature of hyper-threading to share the TLB, allowing one thread to monitor the TLB usage of another thread, resulting in information leakage in specific scenarios. Mamjam [17] and PortSmash [18] exploit the sharing of components to degrade the performance of the target service.

B. DQN

When you submit your final version, after your paper has been accepted, prepare it in two-column format, including figures and tables. DQN combines the powerful representation learning capability of deep neural networks with the policy learning capability of Q-learning. Q-learning is a classic reinforcement learning algorithm designed to address problems in Markov decision processes by learning optimal action selection strategies. The algorithm learns the optimal policy by continuously trying actions in the environment and updating based on reward signals for each state-action pair. However, Q-learning faces challenges when dealing with problems with large state spaces, as it requires maintaining a Q-value table containing all state-action pairs, which may become impractical in practice.

To address this challenge, DQN has been introduced. By utilizing neural networks to approximate the Q-value function, DQN can handle large state spaces and learn directly from high-dimensional raw inputs. In recent years, it has been commonly combined with fuzz testing. By leveraging the learning capabilities of DQN and the automation features of fuzz testing, adaptive testing methods can be implemented, thereby enhancing testing coverage and efficiency.

III. BRANCH PREDICTION CLASS VULNERABILITY FUZZ TESTING SYSTEM

The Branch Prediction Vulnerability Fuzz Testing System, as depicted in Figure 1, consists of three main components: the test case generation module, the execution module, and the analysis module.

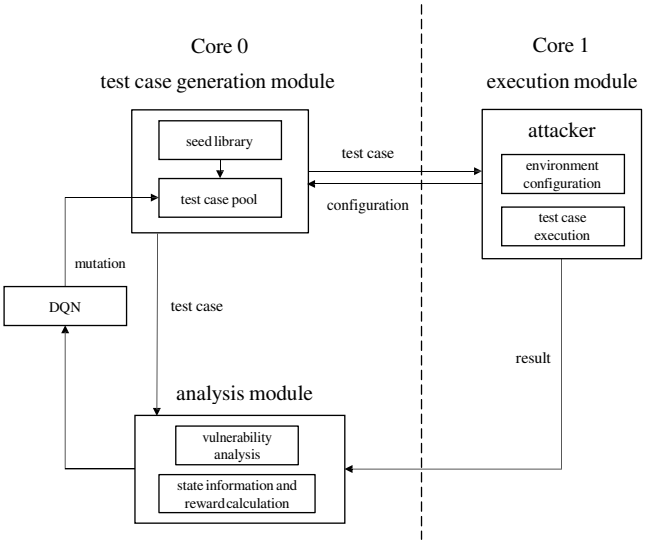

Figure 1: Overall architecture of the branch prediction vulnerability fuzz testing system

The test case generation module utilizes a DQN reinforcement learning model for feedback to generate new test cases through mutation. In the figure, the test case pool selects initial test cases from the seed library. In each round, new test cases are generated by mutating the previous ones under the guidance of the DQN. The execution module comprises two processes: the attacker process and the victim process. These processes execute independently to simulate attacks and assess vulnerability. The analysis module receives test cases, analyzes their features, and then receives the execution results to determine if the test cases trigger vulnerabilities and categorizes them accordingly. Based on the analysis results, reward information is calculated and fed back into the reinforcement learning model to guide further test case mutation. This testing system requires launching two cores to execute different modules, denoted as Core 0 and Core 1. Core 0 handles the test case generation and analysis modules, while Core 1 exclusively handles the execution module, aiming to mitigate interference from unrelated processes on the execution module.

IV. TEST CASE GENERATION MODULE BASED ON DQN

A. Test Case Structure

The test case generation module employs an "N+1 branch structure" pattern to generate a test case. The first N branch structures are utilized to populate the branch predictor microarchitecture, while the last one serves as the victim branch. The composition of each branch structure is illustrated in Figure 2.

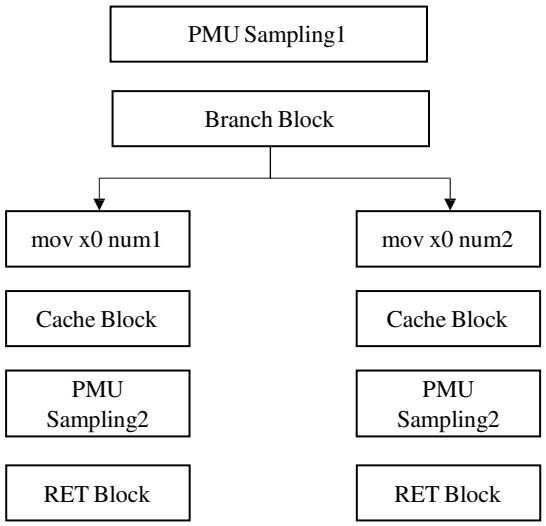

Figure 2: The composition of a single branch structure

A single branch structure consists of a branch block, two Cache channel blocks, two return blocks, and some auxiliary instruction segments. The auxiliary instruction block, specifically the PMU event sampling, is primarily used to monitor whether the branch block triggers branch-related events. By sampling with PMU [19] and detecting Cache side-channel, the system can assess the branch prediction behavior, thereby analyzing whether the processor is vulnerable to a particular type of vulnerability.

B. Mutation Action Set

The branch block includes four adjustable attributes: execution address, target address, branch direction, and branch instruction type. There are six types of execution addresses defined in this testing system, each corresponding to a mutation type based on a fixed virtual address. The purpose of setting different address mutation types is to potentially trigger cross-address branch prediction interference. There are 16 types of target addresses, with each type corresponding to a virtual address page. Branch direction has three types: jump, no jump (for conditional branch instructions), and unconditional jump (for unconditional instructions). Branch instruction type represents different types of branch instructions specified in the ARMv8-Aarch64 instruction set [20]. The instruction set provides seven types of conditional branch instructions, but *B.cond* instructions have different *nzcv* field values representing different branch conditions, resulting in a total of 14 conditional branch instructions. There are six unconditional branch instructions. Therefore, there are a total of 26 types of branch block instructions.

The output of the DQN consists of Q-Values corresponding to different actions, with the maximum value indicating the optimal action for the current mutation. To avoid getting stuck in local optima, the *ε-greedy* strategy is used. Considering the variable attributes of the branch block and the characteristics of known vulnerabilities, the following seven mutation methods are added to the action set:

1) Mutate the branch instruction type attribute in the train and victim branch structures to *B.cond*, and mutate the branch direction attribute to false.
2) Mutate the branch instruction type attribute in the train and victim branch structures to *RET*.
3) Mutate the branch execution address in the train branch structure to make it different from the victim branch structure.
4) Mutate the branch execution address in the victim branch structure to make it different from the train branch structure.
5) Mutate the branch direction attribute in the train branch structure to false.
6) Mutate the branch direction attribute in the train branch structure to true.
7) Random mutation with a probability of ε.

C. State Information Collection

In reinforcement learning, the state information is calculated by the analysis module. Below, we will describe the method for collecting state information in detail.

The analysis module pre-analyzes potential results based on known attack patterns of branch prediction vulnerabilities and possible branch prediction behaviors before obtaining execution results. It categorizes each result according to the reasons for its occurrence, as detailed in Table 1.

Throughout the execution process, the analysis module is responsible for collecting the test results (highest count of magic number values) and monitoring information (poisoning success rate and side-channel success rate) sent by the execution module through the pipeline. To accurately understand and represent the state of the processor microarchitecture under attack and balance the integrity of vulnerability information with the simplicity of dimensionality, the state value is constructed as shown in (1).

$$state = \frac{1}{\log_2 size + 1} \times \left(1 + train_{rate} + side_{rate}\right) \quad (1)$$

Equation (1) includes the poisoning success rate and side-channel success rate of the test case, reflecting the threat level to the processor microarchitecture and the quality of the test case. Additionally, the fitness function contains the *size* parameter, which reflects whether the test case triggers a large number of the same type of vulnerabilities. A higher state value indicates higher quality of the current test case and a greater likelihood of triggering different types of vulnerabilities.

D. Directed Reward Function

During DQN training, the reward values are primarily determined based on the state value of the test case and the type of vulnerability triggered. The first reward factor is calculated as the reciprocal of the difference between the current state value of the test case and the upper limit of the state value (denoted as state_max). Additionally, the second reward factor is determined by the weight ω associated with the type of vulnerability triggered by the test case. Different weights are assigned to various execution result types, as

detailed in Table 1. The principle for weight allocation is that execution result types with known attack patterns are assigned lower weights, while those with unknown attack patterns are assigned higher weights. Moreover, the higher the degree of unanalyzability, the higher the weight assigned.

Table 1: Weight Table for Categorizing Execution Results of Branch Prediction Vulnerability Fuzz Testing System

Vulnerability	Weight
v1	0.2
v2	0.2
rsb	0.2
v1_new	0.4
v2_new	0.4
no_prediction	0.3
neg	0.4
addr_match	0.6
pc+1	0.4
other	1
invalid	0.1

The actual reward value is calculated in the form of a joint reward, as shown in (2).

$$R = pr_1 + qr_2 \tag{2}$$

Here, p and q are multiplication factors, with their sum being r_1 and r_2 represent the difference between the test case's state value and the upper limit of the state value, and the weight of the test case triggering the vulnerability type, where r_1 and r_2 are shown in (3) and (4):

$$r_1 = \frac{1}{state_{\max} - tate} s \tag{3}$$

$$r_2 = \omega \tag{4}$$

The introduction of the joint reward R allows the reward to dynamically adjust during the system's operation, enabling faster discovery of different types of vulnerabilities.

The specific algorithm is illustrated in Figure 3. In the DQN module, two structurally identical but functionally independent neural networks are defined: the training network (TrainingNet) and the target network (TargetNet). Initially, the Agent receives state information and rewards from the fuzz testing system analysis module, guiding the mutation of test cases through interaction with the environment to train the network. To optimize the learning process, the model's parameters are periodically synchronized from the training network to the target network. The purpose of setting up the experience replay mechanism is to reduce the dependence between training samples and alleviate the potential instability of action value function estimation caused by this dependence. During actual training, the experience buffer provides a balanced mix of historical and new samples, breaking the time-series dependence of samples and increasing the utilization rate of samples, thereby accelerating the model's learning of new types of vulnerabilities.

The loss function of DQN is the square of the difference between the Q-values of the target network and the training network, expressed as (5):

$$loss_t = \left(targetQ_t - Q\left(s_t, a_t, \omega_t \right) \right)^2 \tag{5}$$

The target Q can be represented as (6):

$$targetQ_t = r_t + \gamma maxQ'\left(s_{t+1}, a_{t+1}, \omega_{t+1} \right) \tag{6}$$

Where ω represents the network parameters, which are learned using gradient descent as (7):

$$\omega_{t+1} = \omega_t + E[targetQ_t - Q(s_t, a_t, \omega_t)] \nabla Q(s_t, a_t, \omega_t) \tag{7}$$

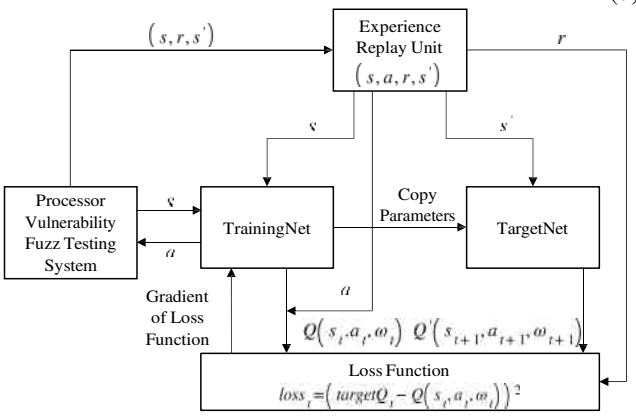

Figure 3: Processor Vulnerability Fuzz Testing System Based on DQN Algorithm

V. Experimental Results and Analysis

A. Experimental Environment Configuration

On the Hikey970 platform, branch prediction vulnerability fuzz testing systems with genetic algorithm and DQN as feedback mechanisms were respectively executed. The platform's operating system and processor configurations are outlined in Table 2. In terms of software configuration, only any version of the g++ compiler and Python 3 interpreter are required. Apart from the pteditor tool, this testing system does not rely on any third-party toolkits.

Table 2: Experimental Environment Configuration for Branch Prediction Vulnerability Fuzz Testing System

Device	OS	Kernel	CPU Architecture	L1 Data Cache	L2 Cache
Hikey970	Linux Ubuntu 18.04.5	4.9.78	ARM Cortex-A73	64KB	2048KB

B. Experimental Procedure

On the Hikey970 platform, branch prediction vulnerability fuzz testing systems were separately executed using genetic algorithm and DQN as feedback mechanisms. Prior to running the systems, configuration of the experimental platform is required. As the testing system relies on the pteditor tool for editing page table states and depends on the processor's PMU functionality, it is necessary to cross-compile the pteditor kernel driver module on the host machine and then install it on the target machine while setting

PMU-related registers to enable PMU functionality. The testing system requires configuration of four parameters: the length of the train sequence in test cases, i.e., the number of branch structures; the number of repetitions for executing a single test case; and the feedback mechanism.

In the experiment, the train sequence length was configured as 5, the number of repetitions for executing a single test case was set to 2000, and there were two feedback mechanisms: genetic algorithm and DQN. Due to the unreliable nature of the results decoded from the cache side channel, multiple executions were conducted, and the most reliable value was selected based on statistical results.

Table 3 presents the average generation time, average execution time, and average analysis time for each test case under the two different feedback mechanisms.

Table 3: Testing Time Statistics for Branch Prediction Vulnerability Fuzz Testing System

Feedback Mechanism	Average Generation Time	Average Execution Time	Average Analysis Time
Genetic Algorithm	2ms	409ms	4ms
DQN	2ms	405ms	4ms

The average testing time per test case under the two feedback mechanisms is essentially equal based on the data in the table. Therefore, the main factor affecting the overall testing efficiency is the number of executed test cases. This experiment will compare the performance of the two feedback mechanisms by measuring the number of test cases needed to achieve the same metric. In order to reflect both the quality and coverage of the mutated test cases, specific indicators are defined as follows:

1) The average state value of the 20 newly mutated test cases is greater than 1.2.
2) The mutated test cases cover the 11 possible types of vulnerabilities mentioned above.

The implementation of the DQN agent is based on a fully connected deep neural network model. The neural network model consists of three fully connected layers, each with 24 neurons, and ReLU activation functions are used to increase nonlinearity. The output layer size is equal to the size of the action space, and a linear activation function is used to predict the Q-values for each action. The loss function of the network is mean squared error (MSE), which evaluates the difference between the predicted Q-values and the target Q-values. The optimizer is Adam optimization algorithm with a learning rate of 0.001 to adapt to the parameter update process of the model. During training, the agent balances exploration and exploitation based on the ε-greedy strategy. Initially, the agent tends to randomly explore different actions; as learning progresses, the exploration rate gradually decreases, and the agent relies more on the learned Q-values for action selection. Additionally, the experience replay mechanism is employed, storing the agent's experiences in a fixed-size replay buffer, and then randomly sampling small batches of experiences for learning, which helps break the correlation between data and improve learning stability.

C. *Experimental Results*

The experimental results, as shown in Figure 4, indicate that the number of test cases required for the genetic algorithm-based model is 28,924, while for the DQN-based model, it is 22,364. The feedback mechanism based on DQN performs better, with an average speed improvement of 1.29 times compared to the genetic algorithm-based approach.

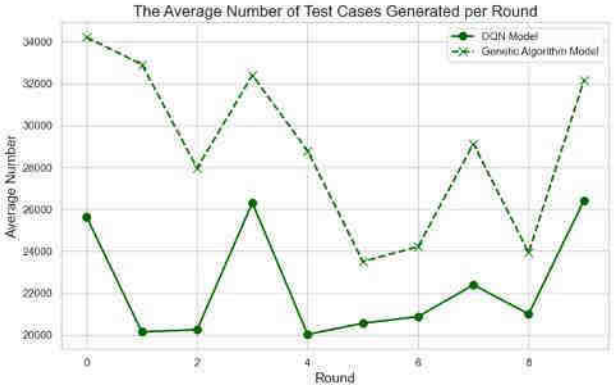

Figure 4: The number of test cases needed to be generated for 10 rounds under two feedback mechanisms

VI. Conclusion

This study proposes a guided processor security fuzzing testing method based on the DQN reinforcement learning algorithm. By designing a reward function and state values, dynamic adjustments to test cases are achieved to optimize the mutation strategy during the testing process. Currently, there is no similar fuzzing testing system targeting branch prediction processor vulnerabilities in the industry. Experimental results on the ARMv8 platform show that compared to traditional feedback mechanisms using genetic algorithms, this approach significantly reduces the required time while covering an equal number of branch prediction vulnerabilities. The efficiency of vulnerability detection is improved by 1.29 times. This achievement demonstrates the effectiveness of the DQN algorithm in guided fuzzing testing and its underlying principles, providing a new research direction for future processor vulnerability exploration and security testing.

References

[1] P. Kocher, J. Horn, A. Fogh, et al., “Spectre attacks: Exploiting speculative execution,” in 2019 IEEE Symposium on Security and Privacy (SP), IEEE, 2019, pp. 1-19.
[2] M. Lipp, M. Schwarz, D. Gruss, et al., “Meltdown: Reading kernel memory from user space,” in 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, 2018, pp. 973-990.
[3] D. Moghimi, M. Lipp, B. Sunar, et al., “Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis,” in 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 1427-1444.
[4] C. Canella, D. Genkin, L. Giner, et al., “Fallout: Leaking data on meltdown-resistant CPUs,” in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, 2019, pp. 769-784.
[5] D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.
[6] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, MIT Press, 2018.
[7] V. Mnih, K. Kavukcuoglu, D. Silver, et al., “Playing Atari with deep reinforcement learning,” arXiv preprint arXiv:1312.5602, 2013.
[8] E. M. Koruyeh, K. N. Khasawneh, C. Song, et al., “Spectre

returns! Speculation attacks using the return stack buffer," in 12th USENIX Workshop on Offensive Technologies, USENIX Association, 2018, pp. 3.

[9] T. Zhang, K. Koltermann, D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Association for Computing Machinery, 2020, pp. 667-682.

[10] G. Chen, S. Chen, Y. Xiao, et al., "Sgxpectre: Stealing Intel secrets from SGX enclaves via speculative execution," in 2019 IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, 2019, pp. 142-157.

[11] M. Schwarz, M. Schwarzl, M. Lipp, et al., "Netspectre: Read arbitrary memory over network," in European Symposium on Research in Computer Security, Springer, Cham, 2019, pp. 279-299.

[12] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, et al., "SMoTherSpectre: Exploiting speculative execution through port contention," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, 2019, pp. 785-800.

[13] S. S. Van, A. Milburn, S. Österlund, et al., "RIDL: Rogue in-flight data load," in 2019 IEEE Symposium on Security and Privacy (SP), IEEE, 2019, pp. 88-105.

[14] M. Schwarz, M. Lipp, D. Moghimi, et al., "ZombieLoad: Cross-privilege-boundary data sampling," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, 2019, pp. 753-768.

[15] B. J. Van, M. Minkin, O. Weisse, et al., "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, 2018, pp. 991-1008.

[16] B. Gras, K. Razavi, H. Bos, et al., "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, 2018, pp. 955-972.

[17] A. Moghimi, J. Wichelmann, T. Eisenbarth, et al., "Memjam: A false dependency attack against constant-time crypto implementations," International Journal of Parallel Programming, vol. 47, no. 4, pp. 538-570, 2019.

[18] A. C. Aldaya, B. B. Brumley, S. ul Hassan, et al., "Port contention for fun and profit," in 2019 IEEE Symposium on Security and Privacy (SP), IEEE, 2019, pp. 870-887.

[19] M. Spisak, "Hardware-Assisted Rootkits: Abusing Performance Counters on the ARM and x86 Architectures," in 10th USENIX Workshop on Offensive Technologies, USENIX Association, 2016, pp. 79-90.

[20] C. S. Components, "Technical Reference Manual," ARM DDI H, vol. 314, pp. 2004-2009, 2009.