AutoCAT: Reinforcement Learning for Automated Exploration of Cache-Timing Attacks

Mulong Luo*[1], Wenjie Xiong†‡[1], Geunbae Lee†, Yueying Li*, Xiaomeng Yang‡,
Amy Zhang‡, Yuandong Tian‡, Hsien-Hsin S. Lee§[2], and G. Edward Suh*‡

*Cornell University †Virginia Tech §Intel Corporation ‡Meta AI
{ml2558,yl3469,gs272}@cornell.edu, {wenjiex,geunbae}@vt.edu, linear@acm.org,
{wenjiex,yangxm,amyzhang,yuandong,edsuh}@meta.com

***Abstract*—The aggressive performance optimizations in modern microprocessors can result in security vulnerabilities. For example, timing-based attacks in processor caches can steal secret keys or break randomization. So far, finding cache-timing vulnerabilities is mostly performed by human experts, which is inefficient and laborious. There is a need for automatic tools that can explore vulnerabilities given that unreported vulnerabilities leave the systems at risk.**

In this paper, we propose *AutoCAT*, an automated exploration framework that finds cache timing-channel attack sequences using reinforcement learning (RL). Specifically, AutoCAT formulates the cache timing-channel attack as a guessing game between an attack program and a victim program holding a secret. This guessing game can thus be solved via modern deep RL techniques. AutoCAT can explore attacks in various cache configurations without knowing design details and under different attack and victim program configurations. AutoCAT can also find attacks to bypass certain detection and defense mechanisms. In particular, AutoCAT discovered *StealthyStreamline*, a new attack that is able to bypass performance counter-based detection and has up to a 71% higher information leakage rate than the state-of-the-art LRU-based attacks on real processors. AutoCAT is the first of its kind in using RL for crafting microarchitectural timing-channel attack sequences and can accelerate cache timing-channel exploration for secure microprocessor designs.

I. Introduction

As we use computers to handle increasingly sensitive data and tasks, security has become one of the major design considerations for modern computer systems. For example, from the hardware perspective, microarchitecture-level timing channels have emerged as a major security concern as they allow leaking of information covertly with a high bit-rate and bypassing the traditional software isolation mechanisms. The timing-channel attacks are also shown to be an even more serious problem when combined with speculative execution capabilities [30], [36].

Unfortunately, developing a system that is sufficiently secure and performant at the same time is quite challenging in large part because it is difficult to evaluate the security of a system design. By definition, a security vulnerability comes from an unknown bug or unintended use of a system feature, which is difficult to know or quantify at design time. While formal methods and cryptography can provide mathematical guarantees, it is difficult to scale the formal proofs to complex systems and security properties in practice. As a result, today's security evaluations and analyses largely rely on human reviews and empirical studies based on known attacks or randomized tests. However, the security evaluation based on known attack sequences manually discovered by humans makes it difficult to assess the security of a new microarchitecture design or a defense mechanism. Vulnerabilities in new microarchitectures are often left unnoticed for a long time, and defense mechanisms are often found vulnerable to new attack sequences even when they are similar to the known ones. For example, while caches existed in microprocessors for a long time, Bernstein's cache-timing attack was reported back in 2005 [6] followed by multiple variations such as evict+time (2006) [47], flush+reload (2014) [83], flush+flush (2016) [21], etc. New attacks in caches are still being reported recently, e.g., attacks in cache replacement states (2020) [8], [77], streamline (2021) [58], and attacks using cache dirty states (2022) [14].

This paper proposes to leverage reinforcement learning (RL) to automatically explore attack sequences for microarchitectural timing-channel vulnerabilities, and demonstrate the feasibility of this approach using cache timing channels as a concrete example. RL has been shown to achieve super-human performance in multiple competitive games (e.g., Go and Chess [63], DoTA 2 [5]) without starting from strategies based on human experience. In this paper, we show that microarchitecture-level timing-channel attack can also be formulated as a *guessing game* for an attack program and is a good fit for RL. In this case, an RL agent can learn by self-playing the game many times in a well-defined environment, which is provided by real hardware or efficient simulation infrastructures commonly used for architecture studies. The experiments show that our RL framework, named *AutoCAT*, can automatically adapt to a variety of cache designs and countermeasures, and find cache-timing attacks, including known attack sequences and ones that are more efficient than known attack sequences.

While the use of machine learning (ML) for system security has been explored in the past, the previous work largely focused on performing or detecting *known attack sequences* on known system designs. For example, ML models with supervised learning are used in side-channel attacks to recover secrets [32],

[1]Equal contributions.

[2]Work done while at Meta AI.

[38], [39], [74], [80], [84]. Similarly, ML models can be trained with known attack traces for intrusion detection [33]. This paper asks a different question: can an RL agent automatically 1) learn system designs without explicit specifications and 2) generate attack sequences that are not specified by humans? To be widely applicable, the RL agent should also be able to adapt to diverse new system designs without substantial changes to the RL environment. Our experimental results suggest that such autonomous explorations are indeed possible.

We believe that the RL-based approach has the potential to enable a more systematic and rigorous evaluation of system security. We envision the RL framework to be used for both 1) studying potential security vulnerabilities of a system design and 2) evaluating the robustness of a defense mechanism. For example, AutoCAT can be used to automatically generate cache-timing attack sequences for a diverse set of cache configurations, replacement policies, or real processors. AutoCAT also tries to find attack sequences with higher success rates and bandwidth, providing a way to quantitatively compare the effectiveness of attacks across different cache designs. AutoCAT's environment can be augmented with an explicit protection scheme such as an attack detector, and the RL agent can be asked to find an attack sequence that bypasses the defense. While AutoCAT cannot prove the security of a defense mechanism, testing a countermeasure with AutoCAT will provide a better measure of its robustness compared to only testing its effectiveness using the known attack sequences that it is designed for.

The following summarizes the main technical contributions and experimental findings of this paper:

- We present AutoCAT, the first framework to use RL to automatically explore cache-timing attacks. The framework can interface with a cache simulator or a real processor.
- We demonstrate that AutoCAT can find cache attack sequences for multiple cache configurations, replacement policies, and prefetchers. AutoCAT can also find attack sequences on multiple real processors with unknown replacement policies quickly, while manually applying known attack strategies to a new processor design requires significant reverse-engineering.
- We demonstrate that AutoCAT can bypass several cache-timing defense and detection schemes, such as the partition-locked (PL) cache [72], detection based on the victim program misses [4], [13], [31], [40], [86], detection based on autocorrelation [11], [79], and ML-based detectors [22].
- We present a novel cache-timing attack, named *Stealth-Streamline*, discovered by AutoCAT, which avoids detection based on the miss counts and has an up to 71% higher bit rate than existing LRU-based attacks on real machines.

The rest of the paper is organized as follows. Section II discusses background and motivation. Section III provides the high-level overview of our methodology. Section IV describes the design and implementation of AutoCAT. Section V presents the experimental results, including several case studies. Section VI discusses other aspects of AutoCAT. Section VII discusses related work, and Section VIII concludes the paper. The code is made publicly available at https://github.com/facebookresearch/AutoCAT.

II. Background and Motivation

A. Cache-Timing Attacks

The cache timing channel is a widely-studied vulnerability in modern microprocessors because of its practicality and high bit rate. Depending on the threat model, it can be used as a side channel (where an attack program and a victim program are non-cooperative) or a covert channel (where a sender and a receiver cooperate) on a shared cache to steal or send information. Without loss of generality, we assume a side channel scenario.

The cache-timing channels usually involve two parties, the attack program and the victim program. The victim program has a secret, and the memory operation of the victim program depends on the secret. The attack program's goal is to guess the secret without directly accessing the secret. The attack program needs to achieve this goal by making memory accesses and measuring the timing of the memory accesses or the victim program's external activities, e.g., using the timing/cycle measurement facilities such as `RDTSCP` in x86. The timing of memory access indicates whether a cache line is in the cache or not. For example, in a prime+probe attack, the victim program's memory access will evict the attack program's cache line from the cache, causing latency changes in the attack program's future memory accesses. Thus, the attack program can infer the victim program's memory operation from timing observations.

The recent cache timing-channel models [15], [73] divide the attacks into three essential components, including the attack program's actions, the victim program's actions, and the attack program's observations. The attack program's actions include normal memory accesses, cache line flushing, etc. The victim program's action is usually a secret-dependent memory operation such that it changes the state of the cache, e.g., accessing a secret address ($addr_{secret}$). For the attack program's observations, the attack program accesses certain cache lines and obtains the timing measurement to gain information about the secret ($addr_{secret}$). With correct combinations, the attack program can learn the secret, as demonstrated in known attacks such as prime+probe [37], flush+reload [83], evict+time [47], cache collision [7] attacks, and other recent cache timing-channel attacks. Table I summarizes the actions of the common cache timing-channel attack categories [15], [23]. Other cache-timing attacks also share the same set of actions and observations.

In this paper, we refer to a sequence of actions such as memory accesses, cache flushing, allowing a victim program's execution, and others in an attack on a specific system as an attack sequence. An attack category or strategy is a broader class of attack sequences that can be adapted to multiple systems, similar to the ones in Table I.

TABLE I: Actions/observations in known cache timing attacks.

Attack Category	Attacker actions	Victim actions	Observations
prime+probe [37]	access addrs	access an addr	attacker's latency
flush+reload [83]	flush addrs	access an addr	attacker's latency
evict+reload [47]	access addrs	access an addr	attacker's latency
evict+time [7]	access addrs	access addresses	victim's latency

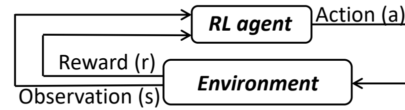

Fig. 1: RL formulation. RL agent does not need knowledge of the internals of the environment to learn a policy.

B. Challenges of Cache-Timing Attack

For security analysis and testing, given a hardware design, we need to analyze the vulnerabilities and generate attack sequences to exploit these vulnerabilities. Many attack sequences may belong to a previously known attack category, while there may also be new attack sequences. Even for known attacks, there is a need to adapt those to the given design to test the effectiveness and bit rate of the attack [71]. In general, a cache timing attack involves (1) reverse-engineering the behavior of target micro-architecture design, (2) designing attack sequences to expose information, and (3) improving the signal quality.

AutoCAT aims to address the challenges in the first two steps by focusing on reverse-engineering and attack sequence exploration when new or blackbox hardware is given. For other practical challenges in developing end-to-end attacks, such as building an accurate timer, reducing background noise, and synchronizing the victim program, we leverage the existing techniques in the literature [58], [70], [77], [80]. The following discusses some of the main challenges in the attack steps.

A cache contains many design options, such as a replacement policy [51], cache directory [80], cache prefetcher [12], [71], etc. The microarchitectural implementation details are often kept as proprietary information by chip designers. Cache operations can also be pseudo-random such as random replacement policies, meaning the cache behavior is not entirely predictable. Reverse engineering effort can be onerous and only reveals limited information [3], [70]. For example, it could take up to 100 hours to reverse-engineer a replacement policy [70].

Some of the microarchitecture states are not directly measurable by timing. The attack has to encode and decode the secret in such states, potentially resulting in complex and long attack sequences. For example, known attacks that use the cache replacement states and cache coherent states all require complex sequences of actions [8], [14], [77], [82], making it challenging to manually reason about an exploitable vulnerability and design an attack sequence.

To explore attacks in a given design, the above process needs to be repeated, which is laborious, and thus, tools helping attack explorations have been proposed. One method is to manually model the existing attacks in the cache [15], and then automatically generate attacks based on the model. However, so far, due to the complexity, such manual modeling methods [15], [23] are still limited to only the cache tag states, not including other cache states that could be vulnerable. Rigorous approaches can determine if certain security exploits exist in the design. However, formal methods [67], and information flow tracking [85] require whitebox modeling of the design, which in many cases is hard or impossible for a commercial microprocessor. In addition, lifting a new RTL-level design into a formal specification without manual effort is still challenging [25]. To deal with blackbox designs, fuzzing [19], [73] has been used to automatically generate attack sequences. However, to bound the size of the search space, fuzzing usually requires pre-defined attack sequences or predefined gadgets, which limit the attack strategies that can be explored. Ideally, we want fewer restrictions on the form of attack sequences and a tool that can explore (both known and unknown) attack sequences in blackbox designs.

C. Reinforcement Learning (RL)

RL aims to find a policy that generates an action sequence that maximizes long-term rewards. Figure 1 shows the high-level components and concepts in RL. First, there is an *RL agent*, which is controlled by a *policy*. There is an *environment*, which the RL agent interacts with. In each step, the RL agent takes an *action*, which feeds into the environment. The environment changes the state with the action, then exposes observations to the RL agent, and assigns *reward* values to the RL agent. The environment can optionally have a state where the environment state is reset. We call the sequence between two adjacent resets an *episode*. The goal of RL agent training is to generate a sequence of actions within an episode so that the sum of the rewards within one episode is maximized.

Recently, RL has been shown to learn human-level (or even super-human-level) policies in many game environments [41], [62] where there are well-defined rules and win/loss conditions. Those games are relatively lightweight and can be simulated easily. RL even discovers novel tactics, including novel openings in the game of Go [63], which humans have played for thousands of years. Similarly, cache-timing attacks can be formulated as a *guessing game*, where an attack program aims to guess a victim program's secret correctly while paying a small penalty (negative reward) for each cache access before making a guess.

The RL-based cache-timing attack discovery has several advantages over the alternatives. First, the RL formulation imposes few restrictions on the length of an attack, allowing flexible exploration of a broader set of attack sequences compared to the existing tool [73]. Second, RL is agnostic to the implementation of the environment as long as the interface including the action, reward, and observation is provided. Thus, RL can work with both existing simulators as well as commercial processors without finding a formal white-box model of their caches [67]. Third, the RL is parameterized with neural networks, which can generalize to unexplored states and make smart decisions [34], [43], compared to random test-case

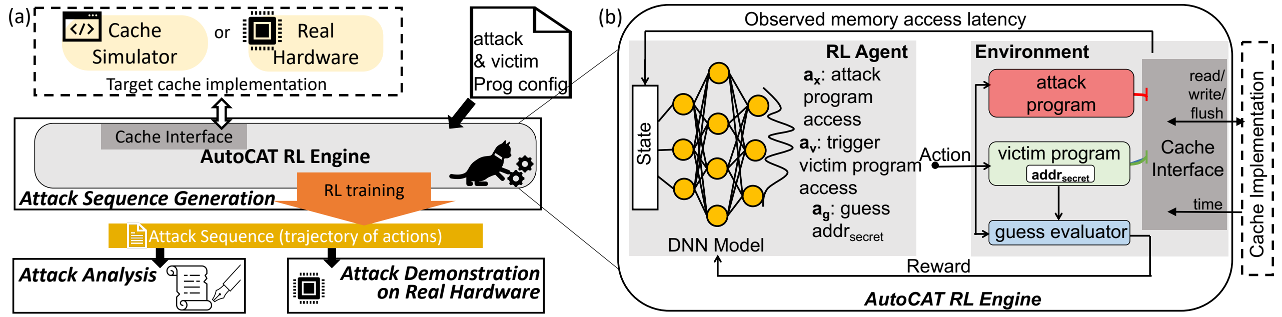

Fig. 2: (a) AutoCAT framework. (b) AutoCAT RL engine.

generations that need to explore each state independently. Thus, RL can be more efficient in searching for vulnerabilities.

III. AutoCAT Overview

A. Overview of AutoCAT Framework

Figure 2(a) shows the AutoCAT's overall framework. AutoCAT's RL engine takes a target cache implementation, the configuration of the attack program and the victim program, and the RL configuration to generate attack sequences. The cache implementation can be either a simulator (with a certain cache configuration) or a real hardware processor. The attack sequence is the sequence of memory operations, which can then be used in attack analysis by human experts to classify and identify potential new attacks. The attack sequence from the RL engine can also be used in an attack demonstration on real hardware. In this study, we manually analyzed the attack sequences to categorize them and applied a new attack sequence from AutoCAT to multiple Intel processors for real-world demonstration.

B. High-level RL Engine Formulation

Figure 2(b) shows the formulation of cache guessing game as an RL problem in the AutoCAT RL engine. In this setting, we let the RL agent explore possible attack strategies by controlling the attack program. For simplicity, AutoCAT currently allows the RL agent to also control when the victim program runs. The RL agent is controlled by a policy parameterized by a deep neural network (DNN) model. The environment interacts with a cache implementation, which handles the memory accesses of both the victim program and the attack program. The environment also contains $addr_{secret}$ which is the secret of the victim, and a guess evaluator to check if a guess is correct. Below we explain the RL formulation.

Episode: In each episode, the environment randomly generates the victim program's secret address ($addr_{secret}$). The agent can then explore different actions and get observations. When the agent decides to make a guess on $addr_{secret}$, the episode will terminate. Reward will be given based on the guessing result and the number of actions taken.

Rewards: As the goal is to let the agent learn how to guess $addr_{secret}$ correctly, we set the environment to return a positive reward if the agent's guess is correct, and a negative reward if the agent makes a wrong guess. To encourage the agent to optimize the attack strategy (i.e., minimize the number of steps), we give a small penalty for each step the agent takes.

Actions:

- a_X—access X where X is the memory address accessible by the attack program. As an attacker, the agent can access a cache line of address X, and observe a hit/miss.
- a_v—trigger the victim program. The RL agent can trigger the victim program's secret access; the victim program accesses $addr_{secret}$, which potentially changes the cache state.
- a_{gY}—guess that the value of $addr_{secret}$ is Y where Y is chosen from the addresses accessible by the victim program, and end the current episode if sending one secret, or change the value of $addr_{secret}$ if sending multiple secrets in one episode, as shown in Autocorrelation and ML-based detection in Section V-D.

Observations: When the agent takes a_X (access X) above, the cache implementation conducts the memory operation, i.e., looks up the address X in the cache, returns the latency of the access, and updates the cache state. For a_v (trigger victim program), the environment uses the $addr_{secret}$ of the current episode and lets the cache simulator access $addr_{secret}$. With this environment, the agent can explore attacks using a combination of actions.

IV. Design and Implementation

A. Cache Implementation and Configuration

As depicted in Figure 2, there are two choices for cache implementation: a cache simulator written in software or a real processor. The cache simulator allows quickly prototyping and implementing existing and new cache designs for vulnerability exploration. Exploring attacks on real hardware enables applying AutoCAT to real-world system designs even when their design details are not known.

Cache simulator: we embed an open-source cache simulator in Python [1] as the cache model in the AutoCAT framework. The cache configuration options are in Table II. We implement LRU, random [35], PLRU [65], and RRIP [26] replacement policies in the cache simulator. The cache simulator can be further extended to multi-level caches. For simplicity, we currently use physically-indexed physically-tagged (PIPT) caches and let the attack and victim programs directly use physical addresses for their accesses.

TABLE II: AutoCAT configuration parameters.

Option Type	Option Name	Definition	Type	Range
Cache configs in cache simulator	`num_blocks`	total number of blocks in the cache	integer	2,4,...
	`num_ways`	number of ways of the cache	integer	2,4,6,8,12,16
	`rep_policy`	replacement policy for all cache sets	str	"lru", "plru",...
Attack and victim program configurations	`attack_addr_s`	starting address of the attack program	integer	0,1,2,3,...
	`attack_addr_e`	end address of the attack program	integer	0,1,2,3 ,...
	`victim_addr_s`	starting address of the victim program	integer	0,1,2,3,...
	`victim_addr_e`	end address of the victim program	integer	0,1,2,3,...
	`flush_enable`	whether to enable flush instruction for the attack program	boolean	true, false
	`victim_no_access_enable`	whether the victim needs to make a memory access when triggered by the attack program	boolean	true,false
	`detection_enable`	whether the episode terminates when the attack detector signals a potential attack	boolean	true,false
RL config	`window_size`	size of the history window in the observation space	integer	1,2,3 ,...
	`correct_guess_reward`	reward when the attack program makes a guess and the guess is correct	float	$(0,\infty)$
	`wrong_guess_reward`	reward when the attack program makes a guess and the guess is wrong	float	$(-\infty,0]$
	`step_reward`	reward when the attack program make a memory access	float	$(-\infty,0]$
	`length_violation_reward`	reward value when the length of episode exceeds limit	float	$(-\infty,0]$
	`detection_reward`	reward when the attack detector signals a potential attack	float	$(-\infty,0]$

Real hardware: we leverage CacheQuery [70], an open-source tool to directly measure the cache access timing on Intel processors. CacheQuery automatically figures out the address mappings for L1, L2 and L3 caches and provides access to a specific cache level, without disabling prefetching, turbo boost, frequency scaling, etc., which is close to real world operating conditions. Currently, CacheQuery [70] supports timing measurements for access sequences to one cache set. Even though AutoCAT can interact with an arbitrary number of sets if the cache interface allows, our experiments focus on exploring attacks within small number of cache sets under different configurations, given that cache-timing attacks usually exploit the contention in each cache set independently.

B. Attack and Victim Program Configurations

Cache-timing attacks also depend on how the memory space is shared between the attack program and the victim program, and which cache operations are available. For example, if there is no shared memory between the victim program and the attack program, the flush+reload attack is not possible, but a prime+probe attack is still possible. We refer to these choices as the attack and victim program configurations. In our setting, for the victim program's access, the agent will not get the latency of that step (N.A. as the observation), since we assume that the victim program's access latency is not directly visible to the attack program. To allow exploring attacks with and without shared memory space between the attack program and the victim program, the address range of the victim program and the address range of the attack program are configurable in AutoCAT, as listed in Table II. These address ranges determine whether the attack program can touch the same addresses accessed by the victim program. In addition, cache-line flush instruction (`clflush` in x86) is not always available, *e.g.*, in JavaScript. We make `flush_enable` a configuration option in AutoCAT.

In many cache timing channels, rather than encoding the information with different addresses, whether the victim program has made access or not also leaks information, *e.g.*, prime+probe. To explore this scenario, we use the `victim_no_access_enable` option, and use $addr_{secret_e}$ to represent the victim program makes no access when triggered. When this is enabled, the victim program can access one of the addresses in the victim program's address space or make no access ($addr_{secret_e}$) with the same probability. To explore attacks under a detection scheme, we have the configuration option `detection_enable`, which terminates an episode when the sequence is recognized as an attack by the detector.

C. RL Engine and Configuration

RL Action Space. The RL agent can have the attack program take one of the three actions (i.e., a_X, a_v, a_{gY}), as discussed in Section III-B. If `flush_enable` is set, we extend the action a_X to also include cache line flush denoted by a_{fX}. Similarly, if `victim_no_access_enable`, we use a_{gE} to denote that the agent guesses that the victim program makes no access after triggered. When the attack program makes a guess, it can be either a_{gY} or a_{gE}. Overall, the RL agent can take one action for each step: 1) access/flush: a_X or a_{fX}; 2) trigger the victim program: a_v; 3) guess: a_{gY} or a_{gE}, depending on the attack/victim program configuration. We use one-hot encoding to represent the actions.

RL State Space. For its own memory access a_X, the attack program can directly observe the cache access latency in terms of a hit or a miss. To provide more information to let the RL agent learn efficiently, we encode a state incorporating the history of actions and observations, and compose the state space S as a Cartesian product of subspaces, as follows: $S = \Pi_{i=1}^{W}(S^i_{lat} \times S^i_{act} \times S^i_{step} \times S^i_{trig})$, where S^i_{lat}, S^i_{act}, S^i_{step}, S^i_{trig} are the subspaces representing the access latency, an action taken, the current step, and whether the victim program has already been triggered at step i. W is the window size that can be set using `window_size`. Empirically we set it to be 4-8 times `num_blocks`. The latency subspace S^i_{lat} is defined as $S^i_{lat} = \{s_{hit},\ s_{miss},\ s_{N.A.}\}$, representing hit/miss and N.A. states. The action subspace S^i_{act} is defined as $S^i_{act} = \{s_a | a \in \{a_X, a_{fX}, a_v, a_{gY}, a_{gE}\}\}$, representing the state in which action

TABLE III: Attack sequence found using AutoCAT on real hardware.

CPU	Cache level	#Ways	Rep. Pol.	Victim addr.	Attack addr.	Example attack sequence found by AutoCAT	Accuracy	Attack Category
Core i7-6700 (SkyLake)	L1	8	PLRU	0/E	0-15	$10 \to 14 \to 13 \to 12 \to 11 \to v \to 1 \to 3 \to 6 \to 0 \to g$	1.0	LRU
	L2	4	N.O.D.‡	0/E	0-8	$8 \to 7 \to 4 \to 2 \to v \to 0 \to g$	0.999	LRU*
	L3	4†	N.O.D.	0/E	0-8	$1 \to 7 \to 5 \to 6 \to v \to 0 \to g$	1.0	LRU*
Core i7-7700K (KabyLake)	L3	4†	N.O.D.	0/E	0-8	$2 \to 4 \to 7 \to 8 \to 4 \to 4 \to v \to 0 \to g$	1.0	LRU*
	L3	8†	N.O.D.	0/E	0-15	$15 \to 11 \to 6 \to 12 \to 14 \to 8 \to 9 \to 7 \to 12 \to 2 \to v \to 0 \to 1 \to g$	0.993	LRU*
Core i7-9700 (CoffeeLake)	L1	8	PLRU	0/E	0-15	$8 \to 12 \to 3 \to 6 \to 15 \to 7 \to 13 \to 4 \to 2 \to v \to 9 \to 0 \to g$	0.998	LRU*
	L2	4	N.O.D.	0/E	0-8	$3 \to 3 \to 3 \to 3 \to 8 \to 2 \to 6 \to 4 \to v \to v \to 0 \to g$	1.0	LRU*

† indicates way partition using Intel CAT. ‡ Not Officially Documented. * Attacks based on replacement states, but with different sequences due to the replacement policy. & We use non-distributed synchronous PPO [60] instead of asynchronous PPO for real hardware experiments due to less resource usage.

a is taken at step i. The step subspace S^i_{step} is defined as $S^i_{step} = \{s_{step1}, s_{step2}, ...\}$. The victim program triggering subspace S^i_{trig} is defined as $S^i_{trig} = \{s_t, s_{nt}\}$, representing whether the victim program has been triggered or not. The RL agent uses this information in order to make a guess after the secret-dependent memory access by the victim program is triggered.

For real hardware, having the agent interact with hardware for each action is slow and also makes the training more susceptible to system noise. To address this challenge, for training on real hardware, we execute all instructions in an episode together as a batch. The latency is masked until the agent make a guess, when all instructions within an episode are executed and latency of memory accesses are revealed.

RL Algorithm and DNN Model In this paper, We use proximal policy optimization (PPO) [60] since it typically performs similarly or better than other methods while being much simpler to tune [2]. Our preliminary study also indicates that PPO converges faster than Ape-X DQN [24] in our context. We use a Transformer [69] model (input feature dimensions 128, 1 layer encoder, 8-head, feed-forward network dimension 2048) as the backbone. Like BERT [17] model, we only use the Transformer Encoder to learn step-wise representation. We then use an average-pooling over steps to generate sequence embedding.

Reward Configuration. When the agent makes a guess a_{gY}/a_{gE}, depending on whether the guess is correct or not, our environment assigns the reward `correct_guess_reward` or `wrong_guess_reward`. For all the actions taken by the RL agent (*e.g.*, a_X, a_{fX}, a_v), the environment assigns a negative `step_reward` to encourage the agent to find short attack sequences. These rewards are listed in Table II. PPO is not very sensitive to the reward value combinations and we use the following rewards for the experiments: `correct_guess_reward` = 1, `wrong_guess_reward` = -1, and `step_reward` = -0.01. For real hardware experiments, we use `step_reward` = -0.005 to explore longer sequences. Once the sum of the reward within an episode is converged to a positive value, we use deterministic replay to extract the attack sequences. Tuning the reward values can increase convergence speed for some specific cases but in general the reward values work for all the cases. To discourage the RL agent from taking too many steps without making a guess, we have `length_violation_reward`, which is a large negative value when the length of an episode is longer than `window_size`. When we implement a detection scheme in the environment, there is also the `detection_reward` which is the reward (negative value) when the sequence is caught by the detector as a potential attack.

D. Attack Analysis and Demonstration

Once the attack sequence is generated using the RL engine, it can be further analyzed and demonstrated. First, for attack analysis, given a particular sequence, we want to know the type of the attack and whether it is a new type of attack. Currently, we rely on human inspections to classify attacks and identify new attacks. Automated classification and identification of the attack sequences is an orthogonal problem and we left it as future work. With the attack sequences generated by AutoCAT, we can demonstrate them in real hardware. We embed the attack sequence into an assembly template, which uses pointer chasing to perform measurement of timing of one access [76]. With the assembly code corresponding to the attack sequence, we can then measure the bit rate and the error rate of the attack on a real processor with realistic noise.

V. Evaluation and Case Studies

In AutoCAT, we use RLMeta [81] as the RL framework to train the RL agent. The PPO implementation in RLMeta is the asynchronous PPO similar to Sample Factory [52], with actors generating data and the learner learning asynchronously. Our DNN model is implemented in PyTorch [50]. The RL environment follows the OpenAI Gym [9] interface. We implement the cache simulator based on [1]. Except for real hardware experiments whose hardwares are specified in the table, the training process is performed on clusters with Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz CPUs and NVIDIA Tesla V100-SXM2-16GB GPUs.

A. Attacks on Real Hardware

AutoCAT can explore attack sequences on real hardware without explicitly knowing all the architectural details, including associativity, replacement policies, frequency scaling, hardware prefetching, and other undocumented features. In our experimental setup using CacheQuery [70], the attack program and the victim program run in a single process on the same core. We experimented with multiple cache levels from three different processors, all with the same attack and victim program configurations where the attack program needs

TABLE IV: The RL environment configurations tested, and the example attack sequences generated by AutoCAT.

No.	Cache config.			Attack&victim config.			Expected attacks	Example Attack found by AutoCAT	
	Type†	Ways used	Sets	Victim addr	Attack addr	Flush inst	Possible attacks‡	Attack sequence (p indicates prefetch)	Attack Category
1	DM	1	4	0-3	4-7	no	PP	$7 \to 4 \to 5 \to v \to 7 \to 5 \to 4 \to g$	PP
2	DM+PFnextline	1	4	0-3	4-7	no	PP	$6(p7) \to 7(p0) \to 5(p6) \to v \to 6(p7) \to 7(p0) \to 5(p6) \to g$	PP
3	DM	1	4	0-3	0-3	yes	FR	$f0 \to f3 \to f2 \to v \to 2 \to 3 \to 0 \to g$	FR
4	DM	1	4	0-3	0-7	no	ER, PP	$6 \to 5 \to 7 \to v \to 7 \to 6 \to 1 \to g$	ER and PP
5	FA	4	1	0/E	4-7	no	PP, LRU	$4 \to 5 \to 7 \to v \to 6 \to 4 \to g$	LRU
6	FA	4	1	0/E	0-3	yes	FR, LRU	$f0 \to v \to 0 \to g$	FR
7	FA	4	1	0/E	0-7	no	ER, PP, LRU	$5 \to 3 \to 2 \to 1 \to v \to 0 \to g$	LRU
8	FA	4	1	0-3	0-3	yes	FR, LRU	$f3 \to f2 \to f0 \to v \to 2 \to 3 \to 0 \to g$	FR
9	FA	4	1	0-3	0-7	yes	FR, LRU	$4 \to f0 \to 5 \to v \to 0 \to 2 \to 1 \to g$	FR
10	DM	1	8	0-7	0-7	yes	FR	$f4 \to f6 \to f5 \to f0 \to f2 \to f7 \to v \to 7 \to 6 \to 5 \to 2 \to 4 \to 0 \to f1 \to v \to 1 \to g$	FR
11	FA	8	1	0/E	0-7	yes	FR, LRU	$f0 \to v \to 0 \to g$	FR
12	FA	8	1	0/E	0-15	no	ER, PP, LRU	$1 \to 13 \to 14 \to 15 \to 5 \to 9 \to 11 \to 6 \to v \to 0 \to g$	ER
13	FA+PFnextline	8	1	0/E	0-15	no	ER, PP, LRU	$1\ (p2) \to 5\ (p6) \to 10\ (p11) \to 8\ (p9) \to 12\ (p13) \to 13\ (p14) \to 15\ (p0) \to 4\ (p5) \to v \to 0\ (p1) \to g$	ER
14	FA+PFstream	8	1	0/E	0-15	no	ER, PP, LRU	$11 \to 15 \to 7 \to 4 \to 6 \to 8(p10) \to 1 \to 13 \to v \to 0 \to g$	ER
15	SA	2	4	0-3	4-11	no	PP	$4 \to 7 \to 9 \to 8 \to 11 \to 5 \to v \to 9 \to 7 \to 4 \to g$	PP
16	2-level SA*	2*	4*	0-3	4-11	no	PP	$10 \to 11 \to 8 \to 7 \to v \to 11 \to 4 \to 8 \to 6 \to g$	PP
17	2-level SA*	2*	8*	0-7	8-23	no	PP	$12 \to 13 \to 15 \to 17 \to 10 \to 14 \to 21 \to 23 \to 16 \to 9 \to 18 \to v \to 15 \to 13 \to 10 \to 17 \to 20 \to 12 \to 8 \to 16 \to 22 \to 14 \to g$	PP

† FA: fully-associative, DM:direct-mapped, SA: set-associative, PFnextline: nextline prefetcher, PFstream: stream prefetcher. ‡ FR: flush+reload, ER: evict+reload, PP: prime+probe. 0/E means the victim program either accesses 0 or makes no access (i.e., **E**mpty) when triggered. In the attack sequence, the number in the attack sequence is from the attack program address range, v represents triggering victim program access, and g represents making a guess. (pn) means prefetching n which is done by the prefetcher automatically. fn means flushing address n. *2-core with 4-set-DM private L1 caches and a shared inclusive L2 cache where the victim program and the attack program each run on one core. The table shows the configuration of L2 cache.

to guess whether the victim program accesses the cache set or not. The configurations and the attack sequences found by AutoCAT are shown in Table III. The table also shows the accuracy for each attack sequence based on repeating the sequence 1,000 times on the same processor using CacheQuery. Due to noise in real processors, the accuracy is slightly less than 100%.

The results show that AutoCAT is able to find attack sequences on real processors without explicitly specifying the number of ways or reverse engineering the replacement policies, prefetchers, etc. Such information is usually needed by human experts to adapt known attacks to a new platform. For example, to demonstrate prime+probe, one needs to understand the replacement policy to prime and probe a cache set efficiently [71]. However, the replacement policies of recent processors are rarely documented publicly by the vendor and are difficult to precisely reverse engineer [70]. Even though it is possible to reverse engineer an unspecified replacement policy from a real-world processor, it takes a significant amount of time (as long as 100 hours) and then one still needs to develop attack sequences manually. AutoCAT can find effective attack sequences within *several hours* in our experiments.

B. Attacks on Diverse Cache/Attack Configurations

The flexibility of the cache simulator allows us to study diverse cache and attack configurations more easily. To evaluate how effective the RL agent can be across a broad range of environments, we tested AutoCAT under many different cache and attack/victim program configurations shown in Table IV. These environments use the (true) LRU replacement policy by default. Note that the attack/victim program configuration limits the feasible attacks in the environment. For example, if the environment does not allow the cache flush instruction, the flush+reload attack is not possible. If there is no shared address, flush+reload or evict+reload is not feasible. The expected attack category for each config are listed in Table IV.

The RL agent can successfully find working attack sequences for all configurations we tested. Table IV shows one example attack sequence for each configuration that was automatically found by the RL agent. The RL-generated attack sequences vary for different environment configurations and cover a range of known attack categories, including prime+probe, flush+reload, and evict+reload. We use the Transformer model in RLMeta framework for all these configurations.

In most cases, the RL agent generates attack sequences that are of the attack type expected for the configuration. Interestingly, the attack sequence found by the agent can be more efficient than the known attacks in the literature. For example, for configuration 1 in Table IV, a textbook prime+probe attack will result in the following attack[1]: $4 \to 5 \to 6 \to 7 \to v \to 4 \to 5 \to 6 \to 7 \to g$. Meanwhile, the attack sequence is given by AutoCAT is: $7 \to 4 \to 5 \to v \to 7 \to 5 \to 4 \to g$. The RL agent removes the unnecessary memory accesses in the textbook prime+probe attack, i.e., by attacking three cache sets, the

[1]For brevity, we only use the subscript i of an action a_i to represent the the action. For example, to represent accessing address 3, we directly use 3 to represent a_3.

TABLE V: RL training statistics and generated attacks for deterministic cache replacement policies.

Repl. alg.	Epochs to converge$	Episode length$	Attack sequence found by AutoCAT
LRU	26.0	7.0	$3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow v \rightarrow 0 \rightarrow g$
PLRU	15.67	7.0	$3 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow v \rightarrow 0 \rightarrow g$
RRIP	70.67	12.7	$2 \rightarrow 3 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow v \rightarrow 0 \rightarrow g$

$ Averaged over three training runs. One epoch is 3000 training steps.

TABLE VI: The RL-generated attacks on the random replacement policy.

Step reward	End accuracy	Episode length
-0.02	0.98	16.25
-0.01	0.98	18.85
-0.005	0.94	19.02

attack program can already learn which of the four possible addresses the victim program accessed. Most known attacks are written in for-loops for convenience, thus, it may contain more accesses than a solution RL found. For configurations 5 and 7 in Table IV, instead of a prime+probe attack, the RL agent found a shorter attack sequence leveraging the LRU state. However, the attack sequences found by AutoCAT are not always the shortest in length (e.g., configuration 10 in Table IV) and may contain unnecessary accesses. This is because the training converges to local optima (whose guess accuracy is high but the length is not the shortest) instead of a global optimum, which might be hard to find in complex configurations; nonetheless, they do capture the key mechanism that enables the attack for each configuration. In some cases, the sequence found by the agent is an interesting combination of different attacks, e.g., configuration 4 in Table IV results in an attack that is a combination of evict+reload and prime+probe, which could make attack detection and defense more difficult.

The cache configurations 2, 13, and 14 include either a next-line prefetcher [64] or a stream prefetcher [27]. The RL agent could find attack sequences even with prefetching.

While not shown in the table for brevity, we also studied caches with a fixed random address-to-set mapping where an address is mapped to a set using a fixed random permutation. AutoCAT could find successful attacks after training on the cache with a given random permutation.

C. Case Study 1: Attacking Replacement Policy

AutoCAT's cache simulator allows us to implement different replacement policies and address mappings in the same setting and the RL agent can adapt to them. We focus on replacement policy in this case study because recent attacks based on replacement states [8], [77] show long and complex sequences, and we want to demonstrate the effectiveness of AutoCAT. We use a 4-way cache (set) with four replacement policies: three deterministic (LRU, PLRU, and RRIP) and one non-deterministic (random). For a 4-way cache set, both LRU and RRIP keep 2-bit state information (ranging from 0-3) for each cache block, called *age* in LRU and *re-reference prediction value (RRPV)* in RRIP, which will be incremented correspondingly. The cache block with the largest state bits will be evicted upon a cache miss. In LRU, the most recently used cache block will be assigned age=0. In RRIP, a newly installed cache block will be assigned RRPV=2, and only upon a cache hit will it be promoted to RRPV=0. Pseudo-LRU implemented using a tree structure is a way of approximating LRU with less state information, whose behavior will be slightly different from LRU. The attack program's address space is configured to be from 0 to 4 (large enough to fill the 4-way cache set). The victim program is configured to either access address 0 or make no access depending on a one-bit secret. The configuration is similar to that of configuration 6 in Table IV.

As shown in Table V, the RL agent can successfully generate valid attack sequences for all three deterministic policies, with the RL training time ranging from about 20 min to 3 hours. RRIP needs longer training time and a longer attack sequence compared to PLRU and LRU. In the RRIP attack example, the $2 \rightarrow 3 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow 4 \rightarrow 2$ sequence is needed to ensure that RRPV is 0 for address 2, address 3 and address 4, and address 0 will be evicted before the victim program access. For the deterministic replacement policies, RL finds attacks that always make a correct guess, *i.e.*, there is no noise.

Unlike a deterministic replacement policy where the next state will be fully determined given the action and current state, the next state is difficult to predict in the (pseudo)-random replacement policy. Thus, an attack sequence that results in a correct guess may result in a wrong guess in another evaluation. The RL agent can also produce different actions depending on the current observation. In that sense, unlike a deterministic replacement policy, there is no single attack sequence that always works in the random replacement policy, whose eviction rate depends on the number and the sequence of memory accesses. Instead, we evaluate the attack accuracy of the RL agent over 100 evaluation runs. As shown in Table VI, the step reward determines the tradeoff between the attack length and the accuracy. The evict+reload strategy is similar to the prior attack on random replacement policy [35].

D. Case Study 2: Bypassing Defense and Detection Techniques

To protect against cache timing-channel attacks, a variety of detection and mitigation techniques have been proposed. AutoCAT's cache simulator can be used to test research prototypes of defense and detection mechanisms proposed in previous works that lack actual real processor implementations. This is especially useful when evaluating new protection or detection schemes and finding vulnerabilities. We implemented four cache timing-channel protection schemes in the cache simulator: 1) Partition-locked (PL) cache [72], 2) Autocorrelation-based detector that is similar to CC-hunter [11], 3) machine learning-based detector that is similar to Cyclone [22], and 4) microarchitecture statistics-based detection [4], [13], [31], [86], AutoCAT successfully finds attack sequences that can bypass these protection schemes.

Partition-Locked (PL) Cache. PL cache [72] provides special instructions to lock specific cache lines in a cache to prevent them from being evicted. The victim program can lock its own cache lines so that they cannot be evicted by the attack program. Further, the victim program's access to the locked

TABLE VII: Comparison of PLRU w/ and w/o PLCache.

Cache	Epochs to converge[$]	Final episode length[$]
PL Cache	37.67	8.1
Baseline	7.67	7.0

$ Averaged over three training runs. One epoch is 3000 training steps.

cache lines will not evict any of the attack program's cache lines. In [23], the formal analysis on a simplified cache model concludes that PL cache is secure when the attack program and the victim program do not share address space.

We implemented the PL cache with the lock/unlock interface in our cache model. To use the PL cache as a defense mechanism, we assume the victim program's cache line is pre-installed and locked in the cache. We use a 4-way cache, and the address range of the attack program is 1-5 and the victim program either accesses 0 or has no access depending on the secret value. The setting is considered to be secure in [23]. Table VII shows the training time (in # of epochs) and the attack sequence length for a cache with PLRU, with and without the PL cache. The training runs take about an hour. AutoCAT successfully found an attack that works even with the PL cache, represented by attack sequence $1 \to 3 \to 5 \to 2 \to v \to 4 \to 4 \to g$. In this attack, the victim program's cache line (address 0) always stays in the cache, and the victim program's behavior (whether the victim program makes access or not) does not evict any of the attack program's cache lines. However, the victim program's access affects the LRU state. When the attack program made subsequent accesses, it can tell whether the victim program accessed address 0 or not by observing if a new block (e.g., address 4) can be brought into the cache. This attack is reported in recent literature [77].

Autocorrelation-based Detection. Autocorrelation of cache events has also been proposed to detect the existence of cache-timing channels [11], [79] based on the observation of the common covert-channel sequences. In CC-Hunter [11], two types of conflict miss events (i.e., the victim program evicting the attack program's cache line, $V \to A$, encoded with "0", and the attack program evicting the victim program's cache line, $A \to V$, encoded with "1") are considered in the event train $\{X_i\}$ where $0 \leq i \leq n$, and n is the length. In a contention-based cache side-channel attack like prime+probe, these two events are interleaved periodically. We can check the autocorrelation C_p at lag p using the following equation [79]: $C_p = \frac{n\sum_{i=0}^{n-p}\left[(X_i-\bar{X})(X_{i+p}-\bar{X})\right]}{(n-p)\sum_{i=0}^{n}(X_i-\bar{X})^2}$. If there exists p where $1 \leq p \leq P$ (P is a predefined parameter) such that $C_p > C_{threshold}$ (e.g. 0.75), then it is an attack.

For example, in a 4-set direct-mapped cache where the victim and attack program's address space is 0-3 and 4-7, a "textbook" prime+probe attack would perform the following steps. First, the address space 4-7 is primed by the attack program, after that one victim program access is triggered, and then addresses 4-7 are probed. The event train and the corresponding autocorrelogram is shown in Figure 3. The maximum autocorrelation for $p \geq 1$ is 0.916, which is over the threshold.

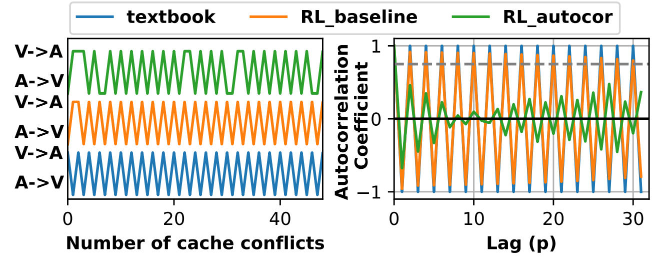

Fig. 3: (a) Event train (A→V: attack program's conflict misses with the victim program and V→A: victim program's conflict misses with the attack program.) (b) Autocorrelograms of the event trains. The dashed line shows the threshold for detecting an attack.

TABLE VIII: Bit rate, accuracy, and autocorrelation of attacks.

Attack	Bit rate (guess/step)*	Guess accuracy*	Avg max autocor*
Textbook	0.1625	1.0	0.973
RL_baseline[$]	0.229	0.989	0.933
RL_autocor[$]	0.216	0.997	0.519

$ Averaged over three training runs.
∗ Averaged over 1000 episodes. The last column shows the average of the maximum autocorrelation of each episode

AutoCAT can train a baseline attack agent where multiple guesses happen in one fixed-step (e.g., 160-step) episode and each guess corresponds to one secret. The more correct guesses it makes, the higher reward it will get. There is a negative reward when there is no guess action in the episode to encourage guesses. However, the AutoCAT generated baseline sequence (*RL_baseline*) can be also detected by this autocorrelation-based detector, as shown in the conflict miss event train and the autocorrelogram (Figure 3). The maximum autocorrelation for $p \geq 1$ is 0.916. Note that the event train and the corresponding autocorrelograms will be different for each sample. However, AutoCAT can learn to bypass the autocorrelation-based detector if the reward of the RL agent is augmented to avoid high autocorrelation. We use L_2-penalty of C_p to penalize high autocorrelation, which is defined as $R_{L_2} = a\sum_{p=1}^{P}\frac{C_p^2}{P}$ where a is a negative number and $P << n$ is the length of C_p used for autocorrelation-based detection. The sampled cache conflict miss event train and the autocorrelogram of the resulting agent (*RL_autocor*) are shown in Figure 3. The maximum autocorrelation beyond lag 0 is 0.477 in this example. The results indicate that the agent (*RL_autocor*) will be able to evade autocorrelagoram detection with high probability.

Table VIII compares the attack sequence from the textbook, *RL_baseline*, and *RL_autocor* in terms of average bit rate (number of guesses per step), average accuracy, and average maximum autocorrelation of each episode. All attacks achieve over 0.98 accuracy and the RL agents have a higher bit rate than the textbook attack. This is because the RL agents optimize the bit rate to gain higher rewards. We observe when a miss is already observed during a probe step, RL agents can guess the secret while the textbook attack still completes the remaining accesses. We also observe that the bit rate of *RL_autocor* is lower than *RL_baseline* because *RL_autocor* makes additional accesses to reduce autocorrelation.

TABLE IX: Comparison of bit rate, guess accuracy, and detection rate by the SVM.

attacker	bit rate (guess/step)	guess accuracy	detection rate
textbook	0.1625	1.0	0.997
RL_baseline $	0.228	0.998	0.715*
RL_SVM $	0.168	0.998	0.00333

$ averaged over three training runs. ranging from 0.338 to 0.909.

ML-based Detection. Machine learning classifiers can also be used for detecting cache-timing attacks. For example, in Cyclone [22], the frequency of cyclic access sequences by different security domains (e.g., $a \rightsquigarrow b \rightsquigarrow a$) for each cache line within each time interval is used as the input of an SVM classifier to detect cache timing channels efficiently. We use a 4-set direct-mapped cache as an example and implement domain tracking and cyclic access sequence counting for each cache line following [22]. We train an SVM-based detector using SPEC2017 benchmarks for benign memory access traces and the textbook prime+probe attack for malicious memory traces. The 5-fold validation accuracy of the SVM detector in correctly predicting the benign/malicious traces is 98.8%.

We then train the AutoCAT's RL agent (*RL_SVM*) with this detector. If the SVM detector correctly reports the existence of an attack, the RL agent gets a negative reward. We also train an *RL_baseline* agent without detection penalty. We show the result in Table IX. The textbook and *RL_baseline* attacks can be easily detected by the SVM detector, with the detection rate of 0.997 and 0.715, respectively. However, when the RL agent is trained with the SVM detection penalty, it can find attack sequences that can bypass the SVM detector, with the detection rate of 0.00333, at the cost of a reduced bit rate. This indicates ML-based detector trained on static traces can be bypassed by an RL-based attack and a novel training method for ML-based detector is needed.

μarch Statistics-based Detection. Most of the cache-timing attacks cause the victim program's process to incur more cache misses during the attack. Thus, detection schemes based on microarchitecture event counts/statistics have been proposed to leverage hardware performance counters (HPCs) to monitor the cache hit-rate of the victim program process and detect an attack at run-time [4], [13], [31], [86]. More recently, [40] observes that fine-grained statistics can achieve better detection accuracy. When an abnormally large number of cache misses are observed, the detector signals a potential attack.

To evaluate the statistics-based detection scheme in AutoCAT, we consider an attack is detected when the victim program's access triggers a cache miss. We terminate the episode and assign a negative reward if the victim program's access results in its cache miss during training. This configuration encourages the RL agent to avoid victim program's misses, and thus, avoid the miss-based detection.

Figure 4(b) shows the attack sequence generated by AutoCAT for a 4-way cache with the miss-based detection scheme. This is a new attack sequence, which is a novel combination of the two recent attacks in literature (shown in Figure 4(a)).

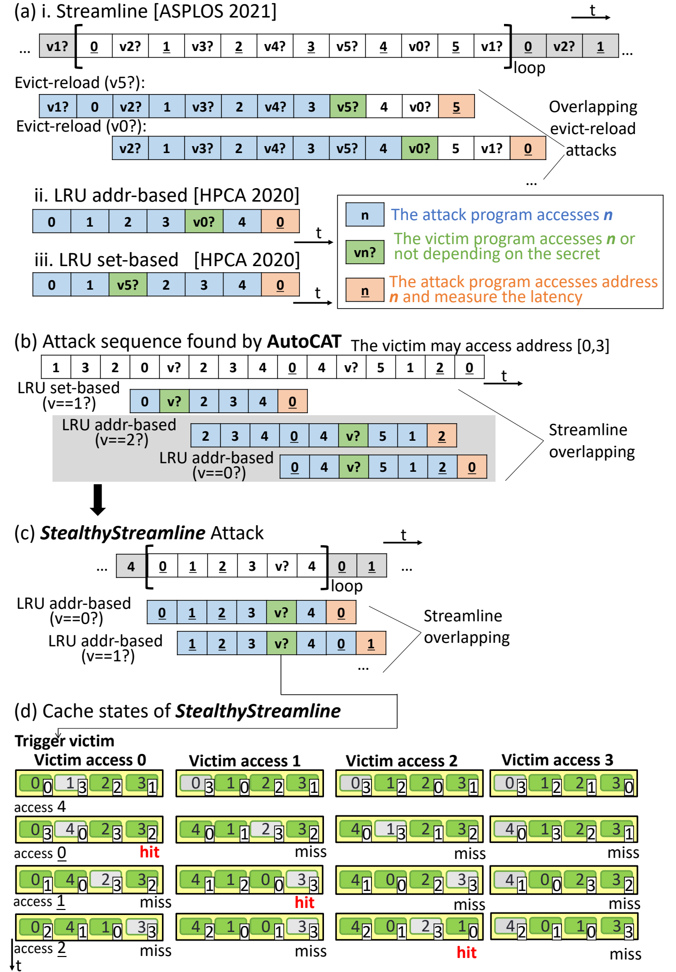

Fig. 4: StealthyStreamline attack. (a) Known attack sequences in the literature. (b) The new attack sequence found by AutoCAT, which can be seen as the combination of the two known attacks. (c) StealthyStreamline attack derived from the attack found by AutoCAT. (d) The cache state changes of the StealthyStreamline attack. The numbers in the right bottom corner indicate the LRU age of the cache line.

The attack sequence can be divided into sub-sequences, which are the LRU set-based or LRU address-based attacks [77]. The two sub-sequences overlap with each other in a way similar to the Streamline attack [58]. Based on the gray part of the sequence generated by AutoCAT in Figure 4(b), we construct a new attack, named *StealthyStreamline*, in Figure 4(c). Compared to the Streamline attack, the new StealthyStreamline attack does not cause cache misses of the victim program and thus is stealthier. Compared to the LRU-based attacks, StealthyStreamline has a higher bit rate by overlapping the steps for multiple bits, effectively transferring multiple bits at a time. Figure 4(d) illustrates the cache state of StealthyStreamline. The attack program observes different timing when the victim program holds different secret values.

TABLE X: Covert channels on real machines.

CPU	μ-arch.	L1D config	OS	Bit Rate[a] (Mbps)		
				LRU	SS.[b]	Impr.
Xeon E5-2687Wv2	IvyBridge	32KB(8way)	Ubuntu18	6.2	7.7	24%
Core i7-6700	Skylake	32KB(8way)	Ubuntu18	3.6	4.5	22%
Core i5-11600K	RocketLake	48KB(12way)	CentOS8	3.4	5.7	67%
Xeon W-1350P	RocketLake	48KB(12way)	Ubuntu20	2.1	3.7	71%

[a] The bit rate when the average error rate $< 5\%$. [b]SS. for StealthyStreamline.

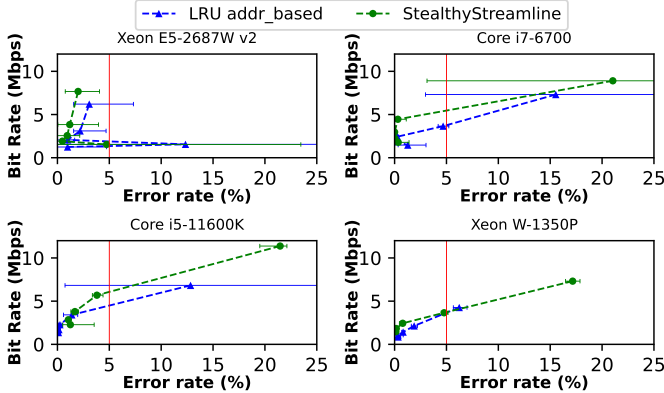

Fig. 5: Bit rate of the StealthyStreamline attack and the LRU address-based attack on four different processors. The horizontal error bars show the range of errors across different transmission runs.

E. Demonstration of Attacks on Real Machines

The attack sequence discovered by AutoCAT captures one essential aspect of real-world cache-timing attacks. However, real-world attacks also have to deal with common practical issues like measurement noise, calibrations, interferences, etc. These practical issues are common in many attacks and people handle them in a similar way regardless of the actual attack sequences. To demonstrate the effectiveness of these attack sequences in a real-world scenario, we put the attack sequence from AutoCAT into an open-source attack assembly template [76], which handles calibration, measurement, cache line access, and forms a covert channel corresponding to the attack sequence. We can then execute the assembly on a real processor under a practical operating environment.

With this method, we tested the *StealthyStreamline* attack sequence for a covert channel in the L1 data cache on four different Intel processors (Table X). Two processors have a 32KB (8-way) L1 data cache, and the other two latest processors employ a 48KB (12-way) L1 data cache. The machines run different versions of Linux.

We generalize the 2-bit *StealthyStreamline* attack sequence in a 4-way cache (in Figure 4) to 8-way and 12-way scenarios by adding extra accesses to the cache lines that map to the same cache set. We implemented both 2-bit (4 possible $addr_{secret}$ values) and 3-bit (8 possible $addr_{secret}$ values) *StealthyStreamline* covert channels.

Bit Rate and Error Rate: We test the bit rate and the corresponding error rate of the covert channel within a process. We do not change any system configuration with the purpose of facilitating the attack, *i.e.*, hardware prefetchers remain enabled. We measure the bit rate by measuring the time of sending a 2048-bit random string 100 times, using `time` in Linux. We evaluate the error rate using the Hamming Distance between the message being sent and the message received. We observe that the 3-bit *StealthyStreamline* has a high error rate due to the tree structure in PLRU, while the 2-bit *StealthyStreamline* has a low error rate.

Figure 5 shows the bit rates and the corresponding error rates for the 2-bit *StealthyStreamline* covert channel and the baseline LRU address-based covert channel. In most of the machines, we observe the LRU address-based covert channel has a larger variation in the error rate across different experiment runs, as shown by the error bars in Figure 5. For the error rate less than 5%, StealthyStreamline has a higher bit rate than the LRU address-based covert channel. In an 8-way cache, StealthyStreamline has up to a 24% higher bit rate. In the latest 12-way cache, the StealthyStreamline has up to a 71% higher bit rate. StealthyStreamline improves the bit rate more for caches with higher associativity because a smaller fraction of memory accesses (4 out of 10 for the 8-way cache vs. 4 out of 14 for the 12-way cache) need to be measured and measuring the latency of access takes more cycles than normal memory accesses.

Spectre Attack using StealthyStreamline: We test Spectre V1 attack [30] with the StealthyStreamline as the covert channel. Compared with the LRU address-based covert channel, the 2-bit StealthyStreamline enables us to encode 4x more symbols with the same cache. Compared with flush+reload or evict+time, StealthyStreamline makes the attack stealthier.

VI. Discussion and Future Work

A. Comparison with Search Algorithms

RL takes fewer steps to find a successful attack than a brute-force search and has the potential to handle much larger search spaces. Consider the prime+probe attack on an N-way cache set as an example. On average, we can find one prime+probe sequence every M sequences, where $M = \frac{2\times(N+1)^{2N+1}}{(N!)^2}$. Since $N! \sim \sqrt{2\pi N}(\frac{N}{e})^{2N}$, we have $M \sim e^{2N}$, which increases exponentially with the number of ways. For $N = 8$, $M \approx 2.05\times 10^7$, it takes about 369 million steps to find an attack, considering each attack sequence takes $2N+2$ steps. Last-level caches usually have more than 8 ways and it will be infeasible for an exhaustive search. With RL, the agent converges within ∼1 million steps.

Compared to RL which learns policy/value on the fly to progressively improve the search quality to find the distinguishing sequence, traditional search techniques may not have sufficient learning capability. For example, a random search does not have learning at all; A* search utilizes a predefined heuristics function, which was not updated during the search process.

B. Factors Affecting Training Speed

Several factors, including ML model architecture and the initialization of the RL environment, affect the training speed and convergence. While we mainly report the results from Transformer models, we also tested other models such as multi-layer perceptron (MLP). MLP is capable of finding attacks for small configurations, but typically takes more time to converge. Also, in our experiments, the cache is initialized by accessing a sequence of addresses randomly sampled from the attack and victim programs' address ranges. Different initialization schemes may change the time to converge during training. We leave finding an optimal initialization scheme for training speed as future work.

C. Future Extensions

One challenge in AutoCAT lies in analyzing many attack sequences produced by an RL agent. In this study, we manually analyzed and categorized the sequences found by AutoCAT. Ideally, we want to automate the attack analysis. For example, we plan to develop a classification model to decide the type of attacks found by AutoCAT.

Given the repetitive structure of a cache, we focus on training AutoCAT with small cache configurations and generalize the attacks to real machines manually. The scalability and generalizability when applying to multiple levels of large caches still need to be further explored. Our initial investigation shows promising results that suggest AutoCAT can be further improved with RL generalization methods [29], [54]. In future work, we plan to investigate extending the AutoCAT approach to more complex cache configurations and other types of microarchitectural timing channels beyond caches.

For training on real hardware, the CacheQuery interface only supports accesses to one cache set, which limits studying attack sequences that involve multiple cache sets. We plan to make CacheQuery support multiple cache sets.

VII. Related Work

Automated timing channel analysis and discovery. Prior studies proposed systematic analysis methods for cache timing channels based on simplified cache models [15], [23], [75]. Our work demonstrates RL as a new way to enable more automated security analysis, which can be easily extended to new systems or defense mechanisms with less human effort. There exist other automated approaches for vulnerability analysis such as exhaustive approach [18], taint analysis [20], fuzzing [19], [42], [73], relational testing [46], and formal methods [67]. CheckMate [67] can only discover attacks that is specified by the given exploit sequence. Information flow tracking technique such as SecVerilog [85] can identify information leakage in the designs but cannot generate exploits automatically. For example, the attack sequences that exploit dirty bits [14] are only reported seven years after the vulnerability is noted by SecVerilog. Compared to formal methods, the RL-based method cannot provide mathematical security guarantees, but can more easily be applied to a system without building a formal model or writing proofs, both of which require significant human efforts [10]. Compared to fuzzing, the RL-based method is more efficient and expressive for complex attack sequences. For example, Osiris [73] explores attack with three instructions, and IntroSpectre [19] relies on predefined attack gadgets.

Cache-timing attack detection and defense. To prevent cache timing-channel attacks, many detection and defense mechanisms were proposed. Detection mechanisms such as cc-Hunter [11] and ReplayConfusion [79] focus on detecting an attack. We show that the RL agent has the potential to automatically generate attacks to evade detection schemes. On the other hand, defenses [16], [28], [45], [53], [59], [66], [78] focus on mitigating or removing the interference that leads to known timing channels. This paper shows that RL also has the potential to automatically evaluate the security of mitigations, and shows that AutoCAT can break PL cache [72] successfully.

ML for security. Machine learning was used in the computer security domain for anomaly detection [33], website fingerprinting [32], [61], and other analysis tasks. However, traditional supervised learning cannot find new attacks without known attack sequences or labels. To address this challenge, we propose to use RL, which can be trained with delayed rewards and whose action trajectories are expressive enough to represent real-world attack sequences.

Reinforcement learning has been used for software security [44], IoT security [68], autonomous driving security [56], power side-channel attacks [55], circuit test generation [49], power side-channel countermeasures [57], and hardware Trojan detection [48]. To the best of our knowledge, our work is the first of its kind in using RL to actively and automatically generate attack sequences in the microarchitecture security domain.

VIII. Conclusion

In this paper, we propose to use reinforcement learning to automatically find existing and undiscovered timing-channel attack sequences. As a concrete example, we build the AutoCAT framework, which can explore cache-timing attacks in various cache configurations and attack/victim program settings, and under different defense and detection mechanisms. Our experimental results show that the RL agent can find practical attack sequences for various blackbox cache designs. The RL agent also discovered the StealthyStreamline attack, which is a novel attack with a higher bit rate on real machines than attacks reported in previous literature. AutoCAT shows RL is a promising method to explore microarchitecture timing attacks in practical systems.

Acknowledgment

This project is partially supported by NSF grant ECCS-1932501 and Commonwealth Cybersecurity Initiative. We thank the anonymous reviewers for their constructive feedback. We thank John G. Harris at Virginia Tech for technical support and Yifang Liu at Cornell University for providing a computer for experiments. We thank Chris Cummins, Hugh Leather, Andrew C. Myers, Benjamin C. Lee, Jiaxun Cui, and Yanqi Zhang for helpful discussions.

REFERENCES

[1] [Online]. Available: https://github.com/auxiliary/CacheSimulator

[2] [Online]. Available: https://openai.com/blog/openai-baselines-ppo

[3] A. Abel and J. Reineke, "Reverse engineering of cache replacement policies in intel microprocessors and their evaluation," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 141–142.

[4] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," *Cryptology ePrint Archive*, 2017.

[5] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[6] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[7] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2006, pp. 201–215.

[8] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+ REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1967–1984.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," 2016.

[10] P. Buiras, H. Nemati, A. Lindner, and R. Guanciale, "Validation of side-channel models via observation refinement," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 578–591.

[11] J. Chen and G. Venkataramani, "CC-hunter: Uncovering covert timing channels on shared processor hardware," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 216–228.

[12] Y. Chen, L. Pei, and T. E. Carlson, "Leaking control flow information via the hardware prefetcher," *arXiv preprint arXiv:2109.00474*, 2021.

[13] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[14] Y. Cui and X. Cheng, "Abusing cache line dirty states to leak information in commercial processors," in *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2022.

[15] S. Deng, W. Xiong, and J. Szefer, "A benchmark suite for evaluating caches' vulnerability to timing attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 683–697.

[16] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid side-channel-resilient caches for trusted execution environments," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 451–468.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[18] M. R. Fadiheh, A. Wezel, J. Muller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz, "An exhaustive approach to detecting transient execution side channels in RTL designs of processors," *IEEE Transactions on Computers*, 2022.

[19] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "IntroSpectre: a pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 874–887.

[20] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures." in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[22] A. Harris, S. Wei, P. Sahu, P. Kumar, T. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 57–72.

[23] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 341–353.

[24] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, "Distributed prioritized experience replay," in *International Conference on Learning Representations*, 2018.

[25] Y. Hsiao, D. P. Mulligan, N. Nikoleris, G. Petri, and C. Trippel, "Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 679–694.

[26] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.

[27] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990.

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

[29] R. Kirk, A. Zhang, E. Grefenstette, and T. Rocktäschel, "A survey of generalisation in deep reinforcement learning," *CoRR*, vol. abs/2111.09794, 2021.

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

[31] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, "SpyDetector: An approach for detecting side-channel attacks at runtime," *International Journal of Information Security*, vol. 18, no. 4, pp. 393–422, 2019.

[32] A. S. La Cour, K. K. Afridi, and G. E. Suh, "Wireless charging power side-channel attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 651–665.

[33] T. D. Lane, *Machine learning techniques for the computer security domain of anomaly detection*. Purdue University, 2000.

[34] A. Lazaric, "Transfer in reinforcement learning: a framework and a survey," in *Reinforcement Learning*. Springer, 2012, pp. 143–173.

[35] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 549–564.

[36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[37] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[38] M. Luo, A. C. Myers, and G. E. Suh, "Stealthy tracking of autonomous vehicles with cache side channels," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 859–876.

[39] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly, "PowerSpy: Location tracking using mobile device power analysis," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 785–800.

[40] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, "Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1124–1137.

[41] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, publisher: Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved.

[42] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1427–1444.

[43] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, "Curriculum learning for reinforcement learning domains: A framework and survey," *arXiv preprint arXiv:2003.04960*, 2020.

[44] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *IEEE Transactions on Neural Networks and Learning Systems*, 2019.

[45] D. Ojha and S. Dwarkadas, "TimeCache: using time to eliminate cache side channels when sharing software," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 375–387.

[46] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: testing black-box cpus against speculation contracts," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 226–239.

[47] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[48] Z. Pan and P. Mishra, "Automated test generation for hardware trojan detection using reinforcement learning." New York, NY, USA: Association for Computing Machinery, 2021.

[49] Z. Pan, J. Sheldon, and P. Mishra, "Test generation using reinforcement learning for delay-based side-channel analysis," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–7.

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[51] W. Perez, E. Sanchez, M. S. Reorda, A. Tonda, and J. V. Medina, "Functional test generation for the plru replacement mechanism of embedded cache memories," in *2011 12th Latin American Test Workshop (LATW)*. IEEE, 2011, pp. 1–6.

[52] A. Petrenko, Z. Huang, T. Kumar, G. Sukhatme, and V. Koltun, "Sample factory: Egocentric 3D control from pixels at 100000 FPS with asynchronous reinforcement learning," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 7652–7662. [Online]. Available: https://proceedings.mlr.press/v119/petrenko20a.html

[53] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 360–371.

[54] R. Raileanu, M. Goldstein, D. Yarats, I. Kostrikov, and R. Fergus, "Automatic Data Augmentation for Generalization in Reinforcement Learning," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 5402–5415.

[55] K. Ramezanpour, P. Ampadu, and W. Diehl, "SCARL: side-channel analysis with reinforcement learning on the ascon authenticated cipher," *arXiv preprint arXiv:2006.03995*, 2020.

[56] I. Rasheed, F. Hu, and L. Zhang, "Deep reinforcement learning approach for autonomous vehicle systems for maintaining security and safety using LSTM-GAN," *Vehicular Communications*, vol. 26, p. 100266, 2020.

[57] J. Rijsdijk, L. Wu, and G. Perin, "Reinforcement learning-based design of side-channel countermeasures," Cryptology ePrint Archive, Report 2021/526, 2021.

[58] G. Saileshwar, C. W. Fletcher, and M. Qureshi, "Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 1077–1090.

[59] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1379–1396.

[60] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[61] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 639–656.

[62] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[63] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[64] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[65] K. So and R. N. Rechtschaffen, "Cache operations by MRU change," *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 700–709, 1988.

[66] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "PhantomCache: Obfuscating cache conflicts with localized randomization." in *Networked and Distributed System Symposium (NDSS)*, 2020.

[67] C. Trippel, D. Lustig, and M. Martonosi, "CheckMate: Automated synthesis of hardware exploits and security litmus tests," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 947–960.

[68] A. Uprety and D. B. Rawat, "Reinforcement learning for IoT security: A comprehensive survey," *IEEE Internet of Things Journal*, vol. 8, no. 11, pp. 8693–8706, 2020.

[69] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[70] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf, "CacheQuery: Learning replacement policies from hardware caches," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 519–532.

[71] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy, "PAPP: Prefetcher-aware prime and probe side-channel attack," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[72] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 494–505.

[73] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated discovery of microarchitectural side channels," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1415–1432.

[74] L. Wei, B. Luo, Y. Li, Y. Liu, and Q. Xu, "I know what you see: Power side-channel attack on convolutional neural network accelerators," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 393–406.

[75] Y. Xiao, Y. Zhang, and R. Teodorescu, "SPEECHMINER: A framework for investigating and measuring speculative execution vulnerabilities," *arXiv preprint arXiv:1912.00329*, 2019.

[76] W. Xiong, S. Katzenbeisser, and J. Szefer, "Leaking information through cache LRU states in commercial processors and secure caches," *IEEE Transactions on Computers*, vol. 70, no. 4, pp. 511–523, 2021.

[77] W. Xiong and J. Szefer, "Leaking information through cache LRU states," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 139–152.

[78] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 347–360.

[79] M. Yan, Y. Shalabi, and J. Torrellas, "ReplayConfusion: detecting cache-based covert channel attacks using record and replay," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–14.

[80] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.

[81] X. Yang, B. Cui, T. Li, and Y. Tian, "RLMeta: A Flexible Framework for Distributed Reinforcement Learning," 1 2022. [Online]. Available: https://github.com/facebookresearch/rlmeta

[82] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International*

Symposium on High Performance Computer Architecture (HPCA). IEEE, 2018, pp. 168–179.
[83] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.
[84] Y. Yuan, Q. Pang, and S. Wang, "Automated side channel analysis of media software with manifold learning," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022.
[85] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ASPLOS '15*, p. 503–516, 2015.
[86] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 118–140.

APPENDIX

A. Abstract

Our artifact contains the AutoCAT framework (a cache simulator + scripts for launching the training and evaluation), as well as code for performing the StealthyStreamline attack on real processors.

B. Artifact check-list (meta-information)

AutoCAT

- **Algorithm:** PPO training cache gueesing game
- **Run-time environment:** Linux
- **Hardware:** multi-core CPU with NVIDIA GPUs (with CUDA support)
- **Disk space required:** About 10GB
- **Time needed to prepare workflow:** About 1hr
- **Time needed to complete experiments:** About 3hr (using checkpoints).
- **Code licenses:** GPLv2

StealthyStreamline Attack

- **Algorithm:** StealthyStreamline cache timing channel attack
- **Run-time environment:** Linux
- **Hardware:** Intel x86 CPU
- **Metrics:** Bandwidth and error rate
- **Disk space required:** About 1GB
- **Time needed to prepare workflow:** About 10 mins
- **Time needed to complete experiments:** About 10 mins

C. Description

Here is a brief description of the artifact. You can find more details in the artifact README file.

1) How to access: The code is available at:
https://github.com/facebookresearch/AutoCAT.

2) Hardware dependencies: The cache simulator training requires one physical machine with NVIDIA GPUs. We use Intel(R) Xeon(R) CPU E5-2687W v2 @ 3.40GHz with NVIDIA Tesla K80 GPUs.

3) Software dependencies:

- Linux 18.04 or higher
- Python 3.8
- Pytorch 1.12
- CUDA 10.2 or higher
- gcc 9.3
- conda environment

D. Installation

In the Linux terminal, the following command downloads the artifact:

```
$ git clone https://github.com/
    facebookresearch/AutoCAT
```

To experiment with the artifact, some dependencies need to be installed manually. We use Conda to manage all the Python dependencies, we assume Conda is already installed. Creating a conda environment:

```
$ conda create --name py38 python=3.8
```

Then press enter when prompt.
Activate the conda environment

```
$ conda activate py38
```

Under the py38 environment, install PyTorch as the following. (note that different machines may have different instructions on how to install PyTorch, please follow https://pytorch.org/get-started/previous-versions/ for specific machines. The following command works on the aforementioned setup.)

```
(py38) $ conda install pytorch==1.12.1
    torchvision==0.13.1 torchaudio==0.12.1
    cudatoolkit=10.2 -c pytorch
```

Install gcc9.3 (which supports C++17, required for building moolib).

```
(py38) $ conda install gcc_linux-64=9.3.0
(py38) $ conda install gxx_linux-64=9.3.0
```

Install and build moolib.

```
(py38) $ pip install scikit-learn seaborn
    pyyaml hydra-core terminaltables pep517
(py38) $ pip install git+https://github.com/
    facebookresearch/moolib.
    git@06e7a3e80c9f52729b4a6159f3fb4fc78986c98e
```

The environment is based on OpenAI Gym. To install it, use the following.

```
$ pip install gym==0.26
```

The RL trainer is based on RLMeta.
Please follow setup process on rlmeta for installing RLMeta.

```
$ git clone https://github.com/
    facebookresearch/rlmeta
$ cd rlmeta
$ git checkout 1057
    fbbf2637a002296afe5071e6ac0e7b630fe0
$ git submodule sync
$ git submodule update --init --recursive
$ pip install -e .
```

These should install all the dependencies needed.

E. Evaluation and expected results

1) **Table IV: attacks patterns found on CacheSimulator** We expect different configurations to have different attack patterns as shown in Table IV.

2) **Table V: RL training with different replacement policies**
We expect the RRIP policy to take more steps/epochs to converge than LRU/PLRU.
3) **Table VI: random replacement policies**
We expect larger step rewards to result in lower guessing accuracy.
4) **Table VII: comparison of PLRU with and without PLCache**
We expect training for PLCache to take more steps/epochs.
5) **Table VIII: bit rate, autocorrection and accuracy for attacks bypassing CC-Hunter**
We expect textbook and RL_baseline to have high max autocorrelation, while RL_autocor to have low max autocorrelation. On the other hand, RL_autocor's bit rate is lower.
6) **Table IX: bit rate, guessing accuracy, and detection rate for attacks bypassing SVM-based detection**
We expect texbook and RL_baseline to have a high SVM detection rate, while RL_SVM to have a low SVM detection rate.
7) **Figure 4: measuring the bit rate and the error rate for the SteathyStreamline attack**
We expect the bit rate of StealthyStreamline to be higher than that of the LRU addr_based attack when the error rate is low (5%).

F. *Methodology*

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html