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

ABSTRACT

The aggressive performance optimizations in modern microprocessors can result in security vulnerabilities. For example, the timing-based attacks in processor caches are shown successfully in stealing secret keys or causing privilege escalation. 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 hardware vulnerabilities efficiently as undiscovered vulnerabilities leave the systems at risk.

In this paper, we propose *AutoCAT*, an automated exploration framework that finds cache timing-channel attacks using reinforcement learning (RL). Specifically, AutoCAT simulates the cache timing-channel attack on a cache model as a guessing game between the attacker program and the victim program holding a secret, which can thus be solved via modern deep RL techniques. Not only can AutoCAT explore attacks in various cache configurations and with different attacker and victim configurations, but also it finds attacks that bypass known detection and defense mechanisms. In particular, AutoCAT discovered *StealthyStreamline*, a new attack that is able to bypass detection based on performance counters and has up to a 71% higher information leakage rate than the known LRU-based attacks on real processors. AutoCAT is the first of its kind using RL for crafting microarchitectural timing-channel attacks and can accelerate cache timing-channel exploration for secure microprocessor designs.

1. 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 [36, 42].

Unfortunately, developing a system that is sufficiently secure and efficient (high-performance) 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 evaluation 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 patterns discovered by humans manually makes it difficult to assess the security of a new microarchitecture design or a defense mechanism. New types of vulnerability are often left unnoticed for a long time, and defense mechanisms are often found vulnerable to new attack patterns 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) [52], flush+reload (2014) [94], flush+flush (2016) [22], etc. New attacks in caches are still being reported recently, e.g., attacks in cache replacement states (2020) [9,88], streamline (2021) [65], and attacks using cache dirty states (2022) [16].

This paper proposes to leverage reinforcement learning (RL) to automatically explore and generate attacks 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 game (e.g., Go and Chess [70], DoTA 2 [5]) without starting from existing human knowledge, via RL training. In this paper, microarchitecture-level timing-channel attack can also be formulated as a *guessing game* between an attacker program and a victim program, and is a good fit for RL. In this case, an RL agent can be learned by self-playing the game many times in a well-defined environment, which is provided by the efficient simulation infrastructures commonly used for architecture studies. Although caches can be quite large in practice, an attack pattern found for a small cache can relatively easily be applied to a larger cache because cache-timing attacks usually exploit the contention in each cache set independently. The experiments show that our RL framework, named *AutoCAT*, can automatically find cache timing-channel attack patterns, including ones that are more effective than known attack methods, and can also adapt to countermeasures.

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 patterns*. For example, ML models with supervised learning are used in side-channel attacks to recover a secret from side-channel traces [38, 44, 45, 85, 91, 95]. Similarly, ML models can be trained with known attack traces for intrusion detection [39]. This paper asks a different question: can RL agent automatically learn and generate new attack patterns that are not explicitly specified by humans? To be widely applicable, the RL agent should also be able to adapt to diverse new system

designs without substantial changes of its implementation details. Our experimental results suggest that such autonomous explorations are indeed possible.

We believe that the proposed RL-based approach has the potential to enable a more systematic and rigorous evaluation of system security. More specifically, 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 attacks for a diverse set of cache configurations and replacement algorithms. AutoCAT also tries to find attack patterns with higher success rates and bandwidth, providing a way to quantitatively compare the effectiveness of attacks across different cache designs. AutoCAT's environment can also be augmented with an explicit protection scheme such as an attack detector, and the RL agent can be asked to find an attack pattern that bypasses the countermeasure. While AutoCAT cannot prove the security of a defense mechanism, testing a countermeasure with AutoCAT will provide a far better measure of its robustness compared to only testing its effectiveness using the known attack patterns 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.
- We demonstrate AutoCAT can find cache attacks with multiple cache configurations, replacement algorithms, prefetchers, and attacker's capability.
- We demonstrate that AutoCAT can bypass several cache-timing defense and detection schemes, such as the partition-locked (PL) cache [83], detection based on the victim misses [4, 15, 37, 46, 96], detection based on autocorrelation [12, 90], and ML-based detectors [25].
- We present a novel cache-timing attack, named *Stealth-Streamline*, discovered by AutoCAT, which bypasses detection based on the miss counts and has an up to 71% higher bit rate than existing LRU-based attacks on real machines.

2. MOTIVATION

2.1 Cache-Timing Attacks and the Cache Guessing Game

The cache timing channel is a widely-studied vulnerability in modern microprocessors, given its practicality and high bit rate. Depending on the threat model, it can be used as a side channel (where an attacker and a victim 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.

In this paper, we treat the cache-timing attack as a guessing game. A victim has a secret, and the memory access pattern of the victim ($addr_{secret}$) depends on the secret. An attacker's goal is to guess the secret without directly accessing the secret. The attacker needs to achieve the goal by contriving memory accesses and measuring the timings of the memory accesses or the victim's external activities, e.g., using the timing/cycle measurement facilities such as `RDTSCP` in x86. The timing of the memory accesses indicates whether a cache line is in the cache or not. When the cache is shared by the attacker and the victim, the victim's memory access pattern can be inferred through the attacker's memory access latencies.

Table 1: Actions/observations in known cache timing attacks.

Attack Category	Attacker's actions	Victim's actions	Observations
prime+probe [43]	access addrs	access an addr	attacker's latency
flush+reload [94]	flush addrs	access an addr	attacker's latency
evict+reload [52]	access addrs	access an addr	attacker's latency
evict+time [7]	access addrs	access addresses	victim's latency

There are two types of cache-timing attacks, *set-based* and *address-based* cache-timing attacks. In set-based cache-timing attacks, the attacker is able to tell if a certain cache set has been touched by the victim, i.e., learn the cache set that $addr_{secret}$ maps to. In address-based cache-timing attacks, the attack can learn which cache line $addr_{secret}$ maps to. In both attacks, the attacker can learn about $addr_{secret}$, but with address-based cache-timing attacks, the attack can learn $addr_{secret}$ with finer-granularity, but usually a shared address is required for such attack.

The recent cache timing-channel models [17, 84] divide the attacks into three essential components, including the attacker's actions, the victim's actions, and the attacker's observations. The attacker's actions include a normal attacker's memory accesses, cache line flushing, etc. The victim'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 attacker's observations, the attack accesses certain cache lines and obtains the timing measurement to gain information about the secret. With correct combinations, the attacker can learn the secret, as demonstrated in known attacks such as prime+probe [43], flush+reload [94], evict+time [52], cache collision [7] attacks, and other recent cache timing-channel attacks. Table 1 summarizes the actions of the common cache timing-channel attack categories[1]. Note that some attacks may require complex sequence of actions [9, 16, 65, 88, 93], making discovery of new attacks hard.

2.2 Reinforcement Learning

Recently, reinforcement learning (RL) has been shown to learn human-level (or even super-human) policies in many game environments [47,69] where there are well-defined rules and win/loss conditions. Those games are relatively light-weight and can be simulated easily (like our cache guessing game). RL even discovers novel tactics, including novel openings in the game of Go [70], which humans have played for thousands of years.

In general, the goal of an RL agent is to maximize the long-term rewards by finding a policy that generates a proper action sequence in a Markov Decision Process (MDP [57]), which includes the formulation of FSM. More specifically, the MDP is specified by the tuple $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{P}, \mathscr{R}, \gamma)$, where $\mathscr{S}$ is the state space, $\mathscr{A}$ the action space, $\mathscr{P}(s'|s,a)$ the probability of transitioning from state $s \in \mathscr{S}$ to state $s' \in \mathscr{S}$, and $\gamma \in [0,1)$ a discount factor. The state and action spaces

[1] In this paper, an attack sequence refers to a specific sequence of actions in one evaluation trajectory. An attack pattern refers to an attack policy that instantiates into multiple attack sequences for different secret values and different cache initial states. An attack category are comparable to the types of attacks that summarized in [17, 27].

can be either discrete or continuous. An "agent" chooses actions $a \in \mathscr{A}$ according to a policy function $a \sim \pi(s)$, which updates the system state $s' \sim \mathscr{P}(s,a)$, yielding a reward $r = \mathscr{R}(s) \in \mathbb{R}$. The goal of the agent is to learn an optimal policy ($\max_\pi \mathbb{E}_\mathscr{P}[\sum_{t=0}^{\infty}[\gamma^t \mathscr{R}(s_t)]$) that takes in states and outputs actions to maximize the cumulative reward over some horizon. In our cache guessing game, the long-term reward is defined as whether the victim's information has been retrieved within a fixed number of steps.

MDP can be solved by dynamic programming [58] if all elements in tuple $(\mathscr{S}, \mathscr{A}, \mathscr{P}, \mathscr{R}, \gamma)$ are known. However, the advantage of RL methods for solving MDPs is that it does not require knowledge of the full FSM. Thus, in RL, we do not need to find a formal model of the cache or reverse engineer the cache FSM. RL can adapt to a distribution over problem instances and find clever heuristics using observations. In our exploration of cache timing vulnerability, the distribution of the cache state (occupancy, LRU state, etc.) can critically affect the patterns of possible attacks. While it takes months or even years for humans to understand the distribution and find corresponding attacks, an RL agent can discover them in a more efficient and fully automatic manner.

There are many algorithms for RL, including proximal policy optimization (PPO) [67], deep Q-learning (DQN) [47], soft-actor critic (SAC) [23], etc. We mainly use PPO since it performs similarly or better than other methods while being much simpler to tune [1].

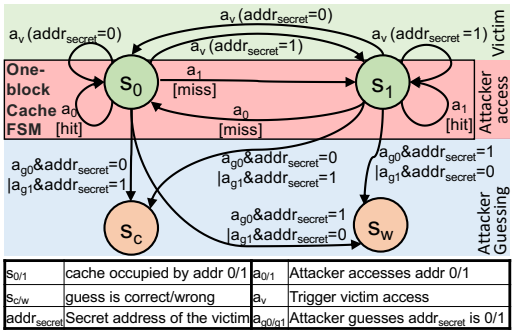
In this paper, we propose to use RL as a way of solving the cache guessing game, as RL has shown its potential to solve similar gaming problems. Here, the RL agent's goal is to find cache-timing attacks, i.e., find policies to play the cache guessing game, in the form of a specific sequence of operations in an FSM, to obtain victim's information.

2.3 Rationale for Using Reinforcement Learning in the Cache Guessing Game

In this section, we discuss how a cache set can be represented as an FSM, and why RL is a good fit for the cache guessing game given the RL method is known to be able to solve an MDP with an FSM. As cache-timing attacks exploit the internal states of a cache, the cache guessing game can be seen as an MDP that optimizes the correct guessing rate. All digital circuits including caches can be seen as FSMs.

Note that we discuss an example FSM formation for the cache guessing game here mainly to show that this problem setting matches the RL method in theory and provide a rational for using RL in the cache guessing game. In practice, however, RL method does not require the knowledge of FSM formulation, and the RL model in AutoCAT directly interfaces with a black box cache model to solve the cache guessing game.

To show the theoretical foundation, here we consider a toy cache example with only one cache block and two cache lines 0 and 1. This simple cache can be represented by a FSM $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{P}\}$ where $\mathscr{S} = \{s_0, s_1\}$ contains the all cache states representing which addr (0 or 1) is in the cache, $\mathscr{A} = \{a_0, a_1\}$ contains actions that access address 0/1, and $\mathscr{P}$ has the transition rules, shown as the arrows in Figure 1.



$s_{0/1}$	cache occupied by addr 0/1	$a_{0/1}$	Attacker accesses addr 0/1
$s_{c/w}$	guess is correct/wrong	a_v	Trigger victim access
$addr_{secret}$	Secret address of the victim	$a_{g0/g1}$	Attacker guesses $addr_{secret}$ is 0/1

Figure 1: FSM of the cache guessing game for an example one-block cache with only two cache lines. [hit/miss] indicates the observation during a transition.

We can formulate the cache guessing game based on the FSM of the cache with the additional state transitions for the victim's accesses (a_v) and the attacker's guess (a_{g0} or a_{g1}). In a typical cache timing-channel threat model, the victim holds a secret address $addr_{secret}$ which is unknown to the attacker. For example, if the secret address is either address 0 or address 1, then we can add an action a_v to $\mathscr{A}$ to represent an access to the secret address. As the attacker does not know the secret address, this action may lead to s_0 or s_1 depending on the secret as shown in Figure 1. We add two actions a_{g0} and a_{g1} to $\mathscr{A}$ indicating that the attacker makes a guess that the secret address is 0 or 1. We also add two additional states s_c and s_w to $\mathscr{S}$ to represent the guess is correct or wrong. The transition $\mathscr{P}$ depends on $addr_{secret}$.

A successful attack sequence that can be represented as a path starting from s_0 or s_1 and ends in s_c in Figure 1, whereas a failed attack sequence ends in s_w. The goal is to find as many successful attacks as possible. For example, assume we start with s_0, and we want to end in s_c. We can reach s_c by taking action a_{g0} or a_{g1} depending on the value of $addr_{secret}$, which is unknown to the attacker. However, $addr_{secret}$ can be inferred by the state after a_v. For example, if after a_v it ends in s_1, then $addr_{secret}$ must be 1. To infer the current state, we can do a_1 and observe the hit/miss response. Thus, if $addr_{secret} = 1$, a successful attack sequence can be: $s_0 \to a_v \to s_1 \to a_1(\text{hit}) \to s_1 \to a_{g1} \to s_c$. Similarly, if $addr_{secret} = 0$, a successful attack sequence can be: $s_0 \to a_v \to s_0 \to a_1(\text{miss}) \to s_1 \to a_{g0} \to s_c$.

This example shows that for all possible values of $addr_{secret}$ and initial states if we can enumerate all paths that lead to s_c, we can find all possible attacks. However, this search approach does not work in practice for the following reasons:

1. It is difficult to know the exact representations of the FSM because the detailed microarchitectural operations are often kept as proprietary confidential know how by chip designers. Reverse engineering effort can be onerous and only reveal limited information [3, 80].
2. The state is not fully visible to the attacker because the state depends on other processes' cache entries.
3. There are multiple FSMs including a cache FSM, a replacement-state FSM [56], cache directory FSM [91], cache prefetcher [14, 82], etc. The composition of these FSMs multiples the number of states and transitions, making optimization-based methods intractable.
4. Microarchitectural operations can be pseudo-random

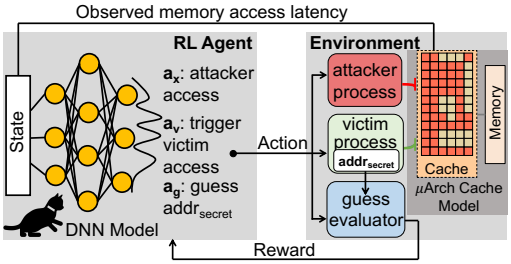


Figure 2: AutoCAT's RL formulation.

such as random replacement algorithms, meaning the transitions inside the FSM are not entirely predictable. Due to these difficulties, directly using a search approach to enumerate all possible paths and find successful attacks is not practical. Instead, we use RL that does not need the knowledge of the full FSM and does not need full observation of the state space. Applying RL to the cache-timing attack, we only need to define the rewards $\mathscr{R}$ and a black box cache model.

3. AutoCAT OVERVIEW

3.1 High-level RL Formulation

Figure 2 shows the RL formulation in AutoCAT. In this setting, we let the RL agent play the attacker role to explore possible attack strategies. It is capable of fully controlling the attacker process, as well as interacting with the victim process and making a guess. The RL agent is a policy parameterized by a deep neural network (DNN) model. The environment contains a cache model, which simulates the memory accesses of both the victim process and attacker process. The environment also contains the $addr_{secret}$ which is the secret of the victim, and a guess evaluator to check if a guess is correct. Below we explain the main elements of this RL formulation.

Episode: In each episode, the environment will generate a random number to be the victim'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 for an attack), we also give a small penalty on each step the agent takes.

Actions:

- a_X—access X where X is the memory address accessible by the attacker. As the attacker, the agent can access a cache line of address X, and observe timing of the access (hit/miss).
- a_v—trigger victim. The attacker can trigger the victim's secret access; the victim accesses $addr_{secret}$, which potentially changes the cache state, we use $addr_{secret\_e}$ to represent the victim makes no access when triggered.
- a_{gY}—guess that the value of $addr_{secret}$ is Y where Y is chosen from address accessible by the victim, and end the current episode.

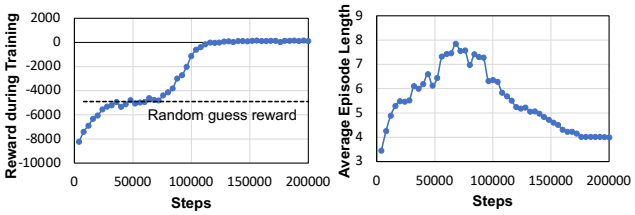


Figure 3: RL training rewards and episode length.

Next action	Victim access 0 when triggered		Victim access 1 when triggered	
Trigger victim	0 1	N.A.	0 1	N.A.
access 3	0 1	miss	0 1	miss
access 1	0 3	**miss**	3 1	**hit**

Attacker can infer $addr_{secret}$ by checking hit or miss

Figure 4: Attacks found by the RL agent for a 2-way cache. Gray blocks indicate the LRU block to be evicted.

Observations: When the agent takes a_X (access X) above, the cache model simulates the memory access, 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), the environment uses the $addr_{secret}$ of the current episode and lets the cache simulator access $addr_{secret}$, while returning N.A. as the latency since the victim access latency is not directly visible to the attacker. With this environment, the agent can explore attacks using a combination of actions.

3.2 A Simple Example

We present a simple example of how AutoCAT's RL formulation can learn to perform an attack automatically. In this simple example, we show a 2-way cache set with the LRU replacement algorithm. The victim's $addr_{secret}$ can be 0 or 1, while the attacker process can access address 0-3. All addresses are mapped to this cache set, and the cache is initially filled with cache blocks of address 0 and 1. The experiments use 200 for the reward for a correct guess and -10,000 for the reward for an incorrect guess. Figure 3 shows the reward and the length of an episode as the RL agent learns. We can see that the reward function increases to a positive number, meaning the agent can make a correct guess with a high probability. Figure 4 shows the actions taken by the RL agent. The attack sequence is $a_v \rightarrow a_3 \rightarrow a_1 \rightarrow a_{g0/1}$[2]. We can see that the attack leverages the LRU state of the cache to control which cache line should be evicted. In section 5, we will show results on realistic 8-way caches.

4. DESIGN AND IMPLEMENTATION

As shown in Figure 2, AutoCAT is composed of an environment that includes a cache model, attacker and victim processes, and an RL agent that chooses actions based on the input state, using a DNN model. AutoCAT is invoked using a Python script, which takes in a configuration file that specifies a cache configuration, attacker's capabilities, action space, the state space, the DNN model, and reward values. We list some of the configuration parameters in Table 2.

[2]We only show the actions but omit the state after each action as the state is secret dependent.

Table 2: AutoCAT configuration parameters.

Option Type	Option Name	Definition	Type	Range
Cache configs	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_alg	replacement algorithm for all cache sets	str	"lru", "plru",...
Attacker and Victim configs	attacker_addr_s	starting address of attacker	integer	0,1,2,3,...
	attacker_addr_e	end address of the attacker	integer	0,1,2,3 ,...
	victim_addr_s	starting address of victim	integer	0,1,2,3,...
	victim_addr_e	end address of the victim	integer	0,1,2,3,...
	flush_enable	whether to enable flush instruction for the attacker	boolean	true, false
	victim_no_access_enable	whether the victim need to make a memory access when triggered by the attacker	boolean	true,false
	detection_enable	whether the episode terminates when the attack detector signals a potential attack	boolean	true,false
State space	window_size	size of the history window in the observation space	integer	1,2,3 ,...
Reward	correct_guess_reward	reward when the attacker makes a guess and the guess is correct	float	$(0, \infty)$
	wrong_guess_reward	reward when the attacker makes a guess and the guess is wrong	float	$(-\infty, 0]$
	step_reward	reward when the attacker make a memory access	float	$(-\infty, 0]$
	length_violation_reward	reward value when the length of episode exceed limit	float	$(-\infty, 0]$
	detection_reward	reward when the attack detector signals a potential attack	float	$(-\infty, 0]$

4.1 Cache Model

We focus mainly on the vulnerabilities of a cache itself. Thus, we only need to simulate the cache instead of simulating the entire processor. We embedded an open-source cache simulator in Python [2] as the cache model the AutoCAT framework, whose number of blocks and number of ways can be configured by setting `num_blocks` and `num_ways` in Table 2. While AutoCAT can simulate a cache with an arbitrary number of sets, given that cache-timing attacks usually exploit the contention in each cache set independently, our experiments focus on exploring attacks on a small cache under different configurations. For simplicity, we currently simulate physically-indexed physically-tagged (PIPT) caches and let the attacker and the victim directly use physical addresses for their accesses. The replacement algorithm can be configured using `rep_alg`, whose values can be `lru`, `rand`, `tree-plru`, and `rrip` to represent LRU, random [41], tree-PLRU [73], and RRIP [29] respectively. The framework can be further extended to multi-level caches, but that is not the focus of this paper. Without loss of generality, we assume the cache block size is one byte. While previous cache timing-channel analysis [17, 27] use relatively simple cache models, AutoCAT works with more complex cache configurations.

4.2 Attacker and Victim Configurations

In addition to the cache configuration, cache-timing attacks also depend on how the memory space is shared between an attacker and a victim, and which cache operations are available. We refer to these choices as the attacker and victim configurations. To allow exploring both types of attacks with and without shared memory space between the attacker and the victim, AutoCAT lets a user configure the address ranges of the victim (`victim_addr_s` to `victim_addr_e`) and the address range of the attacker (`attacker_addr_s` to `attacker_addr_e`). These address ranges determine whether the attack process can touch the same addresses accessed by the victim[3]. Similarly, to explore attacks with and without a cache-line flush instruction (`clflush` in x86), which is not always available, e.g., in JavaScript, we make

[3]The addresses here are the physical addresses used by a physically-indexed physically-tagged cache. We believe that the attack will also be applicable to virtually-indexed caches when virtual addresses are used by the attacker/victim and can be generalized to other cases. For example, mapping a physical address to a virtual address is studied as an orthogonal problem in attacks [81].

```
...
victim_addr_s: 0
victim_addr_e: 3
attacker_addr_s: 0
attacker_addr_e: 3
flush_inst: True
...
```

```
...
victim_addr_s: 0
victim_addr_e: 3
attacker_addr_s: 4
attacker_addr_e: 7
flush_inst: False
...
```

Figure 5: Two different attacker and victim configurations.

`flush_enable` a configuration option. As an example, Figure 5 shows two different AutoCAT configurations. In the left-side example, the attacker and the victim can access the same set of addresses, the attacker can flush these addresses out of the cache. A flush+reload attack is possible in this case. On the other hand, in the right-side example, there is no shared memory between the victim and the attacker and `clflush` is not allowed. The flush+reload attack is not possible, but a prime+probe attack is still possible.

To explore both address-based and set-based attacks, we use the `victim_no_access_enable` option. When this is enabled, the victim either makes or does not make an access depending on a secret; when $addr_{secret}$ is $addr_{secret\_e}$ the victim makes no access when triggered. To explore avoiding an attack detection scheme, we have the configuration option `detection_enable`, which terminates an episode when the pattern is recognized as an attack by the detector.

4.3 RL Agent Action Space

The RL agent can have the attack software take one of the three actions (i.e., a_X, a_v, a_{gY}). If `flush_enable` is set, we extend the action a_X to also include cache line flush denoted by a_{fX}, where X is an address in the attacker's address space. Similarly, if `victim_no_access_enable` is set, then the victim can either access one of the addresses in the victim's address space or make no access. In this case, we use a_{gE} to denote that the agent guesses that the victim made no access after triggered. When the attacker 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 victim: a_v; 3) guess: a_{gY} or a_{gE}, depending on the attacker/victim configuration. As the number of actions is limited, we use one-hot encoding to represent the actions.

4.4 RL Agent State Space

For each memory access a_X, the attacker can directly observe the cache access latency in terms of a hit or a miss. In principle, this access latency alone is enough for the attacker

to infer the secret. However, it does not contain enough information to represent the cache state. For example, if an access results in a cache miss, the corresponding cache set may be empty or occupied by another address; there are many possible cache states that lead to a cache miss. As a result, encoding the state space with only the access latency of the current step makes it challenging to train an agent that can perform attacks with a high success rate. To provide more information to let the RL agent learn the cache state, 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 access latency, actions taken, current steps, and whether the victim 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 a is taken at step i. The step subspace S^i_{step} is defined as $S^i_{step} = \{s_{step1}, s_{step2}, ...\}$. The victim triggering subspace S^i_{trig} is defined as $S^i_{trig} = \{s_t, s_{nt}\}$, representing whether the victim has been triggered or not. The attacker agent needs this information in order to make a guess after the secret-dependent memory access by the victim is triggered. Otherwise, the environment will not depend on the victim's secret, making the attacker's guess random at best. The above is a standard formulation in partially observable MDPs (POMDPs, [32]), where the agent does not have full information about the state it is in, and only has a partial observation instead. Typically, history is incorporated to provide a better estimate of the true state [71]. Each state $s \in S$ can be represented by a $4 \times W$-dimensional vector, which contains enough information for the attacker to decide 1) whether it needs to take more actions and 2) what the secret value is. This $4 \times W$-dimensional vector is further encoded with multiple one-hot vectors where each dimension is one-hot encoded and concatenated.

With the state space S defined by Equation 4.4, the agent is able to know how many steps it already took, what actions it used before, whether the victim's access has been triggered, and what cache access latency is. This information allows the agent to infer the needed cache state, including which addresses are in the cache, what the LRU state is, etc., and decide the next action to take.

4.5 DNN Model for RL Agent

To demonstrate the feasibility of using RL to explore cache-timing attacks, we use simple machine learning models such as a multi-layer perceptron (MLP) [26] to encode our RL policy. More complex recurrent models like LSTMs [28] or RNNs can also be used but may be more difficult to tune. The input states, encoded as one-hot vectors, are sent to two hidden layers, each with 256 neurons and `tanh` activation, followed by a softmax that outputs action probability. In certain scenarios, we also explored Transformer [78] model (input feature dimensions 128, 1 layer encoder, 8-head, FFN dimension 2048) to learn step-wise representation, followed by average-pooling over steps for sequence embedding.

4.6 Reward Values

When the agent makes a guess a_{gY}, 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 patterns. We use the following baseline rewards for the majority of the experiments: `correct_guess_reward` = 200, `wrong_guess_reward` = -10,000, and `step_reward` = -10. We discuss how these rewards affect the training process and result attack sequence in Section 6.1. We also provide other configurable rewards such as `length_violation_ reward`, which is a large negative value when the length of an episode is longer than `window_size`, to discourage the RL agent from taking too many steps without making a guess. When we implemented an attack detection scheme in the environment, there is also the `detection_reward` which is the reward when the pattern is caught by the detector as a potential attack.

5. EVALUATION AND CASE STUDIES

5.1 Experimental Setup

In AutoCAT, we use RLlib [40] v1.9.3 as the reinforcement learning framework to train the RL agent. Our DNN model is implemented in PyTorch [55]. The environment wrapper follows the gym [10] interface. We implement the cache model based on [2]. We use PPO as our learning algorithm. The RL training process is performed on a machine with an Intel Xeon CPU E5-2687W v2 @ 3.40GHz running Ubuntu 16.04 and Nvidia K80 GPUs with CUDA 10.2.

5.2 Attacks on Diverse Configurations

The expressiveness of AutoCAT's environment allows us to represent a variety of different cache and attack configurations. To evaluate how effective the RL agent can be across a broad range of environments, we tested AutoCAT under many different cache and attacker/victim configurations shown in Table 3. These environments use the (true) LRU replacement algorithm by default. Note that the attacker/victim configuration limits the feasible attacks in the environment. For example, if the environment does not allow the cache flush instruction, the flush+reload attack in its original form cannot be produced by the agent. Also, if there is no shared address space, flush+reload or evict+reload is not feasible.

The RL agent could successfully find working attack patterns for all configurations we tested. Table 3 shows one example attack for each configuration that was automatically found by the RL agent. The RL-generated attacks vary for different environment configurations and cover a range of known attack patterns, including prime+probe, flush+reload, and evict+reload. For configuration 1, 3-7, we use the MLP model for the RL agent. For configuration 2, 8-14, we use a Transformer model in `rlmeta` framework [92].

In most cases, the RL agent generates attack patterns that are of the attack type expected for that the configuration. Interestingly, the attack pattern found by the agent can be more

Table 3: The RL environment configurations (cache and attacker/victim configurations) tested, and the example attack patterns generated by AutoCAT (FA: fully-associative, DM:direct-mapped, FR: flush+reload, ER: evict+reload, PP: prime+probe, PFnextline: nextline prefetcher, PFstream: stream prefetcher.).

No.	Cache configuration			Attacker& victim configuration				Expected attacks		Example Attack found by AutoCAT	
	Type	Ways	Sets	Victim addr	Victim no access	Attack addr	Flush inst	Possible attacks	Type	Attack sequence (p indicates prefetch)	Attack Category
1	DM	1	4	0-3	no	4-7	no	PP	set	$5 \rightarrow 4 \rightarrow 7 \rightarrow v \rightarrow 5 \rightarrow 7 \rightarrow 4 \rightarrow g$	PP
2	DM+PFnextline	1	4	0-3	no	4-7	no	PP	set	$6(p7) \rightarrow 4(p5) \rightarrow v \rightarrow 4(p5) \rightarrow 5(p6) \rightarrow g$	PP
3	DM	1	4	0-3	no	0-3	yes	FR	set	$\ldots \rightarrow f1 \rightarrow v \rightarrow 1 \rightarrow f0 \rightarrow v \rightarrow f2 \rightarrow v \rightarrow 2 \rightarrow f3 \rightarrow 0 \rightarrow g$	FR
4	DM	1	4	0-3	no	0-7	no	ER, PP	set	$\ldots \rightarrow 3 \rightarrow 7 \rightarrow 4 \rightarrow 6 \rightarrow v \rightarrow 3 \rightarrow 0 \rightarrow 6 \rightarrow 4 \rightarrow g$	ER and PP
5	FA	4	1	0-0	yes	4-7	no	PP	set	$4 \rightarrow 6 \rightarrow 7 \rightarrow v \rightarrow 5 \rightarrow 4 \rightarrow g$	LRU set-based
6	FA	4	1	0-0	yes	0-3	yes	FR	set	$0 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow f0 \rightarrow 2 \rightarrow v \rightarrow 3 \rightarrow 0 \rightarrow g$	FR
7	FA	4	1	0-0	yes	0-7	no	ER, PP	set	$v \rightarrow 4 \rightarrow 1 \rightarrow 6 \rightarrow 7 \rightarrow v \rightarrow 1 \rightarrow v \rightarrow 5 \rightarrow 6 \rightarrow g$	LRU set-based
8	FA	4	1	0-3	no	0-3	yes	FR	addr	$f3 \rightarrow f2 \rightarrow v \rightarrow 2 \rightarrow 3 \rightarrow f0 \rightarrow v \rightarrow 0 \rightarrow g$	FR
9	FA	4	1	0-3	no	0-7	true	FR	addr	$f0 \rightarrow f2 \rightarrow f1 \rightarrow v \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow g$	FR
10	DM	1	8	0-7	no	0-7	yes	FR	set	$f2 \rightarrow v \rightarrow 2 \rightarrow f4 \rightarrow f0 \rightarrow v \rightarrow 0 \rightarrow 4 \rightarrow f3 \rightarrow f7 \rightarrow v \rightarrow 3 \rightarrow v \rightarrow 7 \rightarrow f1 \rightarrow f6 \rightarrow v \rightarrow 6 \rightarrow 1 \rightarrow g$	FR
11	FA	8	1	0-0	yes	0-7	yes	FR	set	$f0 \rightarrow v \rightarrow 0 \rightarrow g$	FR
12	FA	8	1	0-0	yes	0-15	no	ER, PP	set	$7 \rightarrow 11 \rightarrow 10 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow v \rightarrow 0 \rightarrow g$	ER
13	FA+PFnextline	8	1	0-0	yes	0-15	no	ER, PP	set	$4\ (p5) \rightarrow 9\ (p10) \rightarrow 15\ (p16) \rightarrow 2\ (p3) \rightarrow v \rightarrow 0\ (p1) \rightarrow g$	ER
14	FA+PFstream	8	1	0-0	yes	0-15	no	ER, PP	set	$15 \rightarrow 9 \rightarrow 8 \rightarrow 7(p6) \rightarrow 11 \rightarrow 6 \rightarrow 12 \rightarrow 14 \rightarrow v \rightarrow 0 \rightarrow g$	ER

Table 4: RL training and generated attacks for deterministic cache replacement algorithms.

Repl. alg.	Steps to converge	Episode length	Attack pattern found by AutoCAT
LRU	296,000	6.59	$v \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 0(hit/miss) \rightarrow g0/gE$
Tree-PLRU	340,000	9.14	$1 \rightarrow v \rightarrow 1 \rightarrow 4 \rightarrow v \rightarrow 3 \rightarrow 2 \rightarrow 1(miss/hit) \rightarrow g0/gE$
RRIP	336,000	9.68	$3 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow v \rightarrow 4 \rightarrow 2(miss) \rightarrow g0;\ 3 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow v \rightarrow 4 \rightarrow 2(hit) \rightarrow 0 \rightarrow gE$

Table 5: The RL-generated attacks on the random replacement algorithm with varying step reward values.

Step reward	End accuracy	Episode length	num of patterns
0	1.0	41.00	737
-10	0.87	35.75	171
-20	0.999	34.99	163
-30	0.897	30.17	122

efficient than the attack we expected. In some cases, the RL agent can generate a shorter pattern than expected. For example, for configuration 1 in Table 3, a textbook prime+probe attack will result in the following attack[4]: $4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow v \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow g$. Meanwhile, the attack sequence given by AutoCAT is: $5 \rightarrow 4 \rightarrow 7 \rightarrow v \rightarrow 5 \rightarrow 7 \rightarrow 4 \rightarrow g$. The RL agent removes the unnecessary memory access in the textbook prime+probe attack, i.e., by attacking three cache sets, the attacker can already learn which of the four possible addresses the victim accessed. For configurations 4 and 6 in Table 3, instead of a prime+probe attack, the RL agent finds a shorter attack pattern leveraging the LRU state, similar to the simple example in Section 3.2. However, the attack patterns found by AutoCAT are not always the shortest in length (e.g., configuration 5 in Table 3) and may contain unnecessary accesses; nonetheless, they do capture the key mechanism that enables the attack for each configuration. In some cases, the pattern found by the agent is an interesting combination of different attacks, e.g., configuration 3 in Table 3 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 [72] or a stream prefetcher [31]. The RL agent could find attack patterns even with prefetching.

5.3 Adapting to Replacement Algorithms

We study how the replacement algorithms affect the attack patterns found by the RL agent. Different replacement algorithms have different cache FSMs and internal rules not explicitly visible to the RL agent. It takes varying amount of efforts for the RL agent to "learn" this internal rules in the replacement algorithms and generate timing attacks. The effort can be estimated by the time for the RL training to converge and the length of an attack sequence. In this study, we use a 4-way cache (set) with four replacement algorithms: three deterministic (LRU, tree-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. Tree-pseudoLRU is a way of approximating LRU with less state information, whose behavior will be slightly different from LRU. The attacker's address space is configured to be from 0 to 4 (large enough to fill the 4-way cache set). The victim 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 No. 6 in Table 3. The cache is initially accessed by address 0 and 1 before the attacker performs the attack.

As shown in Table 4, the RL agent can successfully generate valid attacks for all three deterministic policies, which takes less than one hour for each case. For the deterministic replacement algorithms, the RL finds attacks that always make a correct guess (there is no noise). The results also show that tree-PLRU and RRIP need longer training and a longer attack sequence compared to LRU, In the RRIP attack example, the $3 \rightarrow 2 \rightarrow 3 \rightarrow 1$ sequence is needed to ensure that for both address 1 and address 3, RRPV is 0 since they are cache hit and predicted to be immediately re-referenced. For address 2, RRPV is 2 since it is not re-referenced. As shown in Figure 6, the victim access of address 0 will set RRPV=0 for address 0 and leaving address 2 to be replaced.

[4]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.

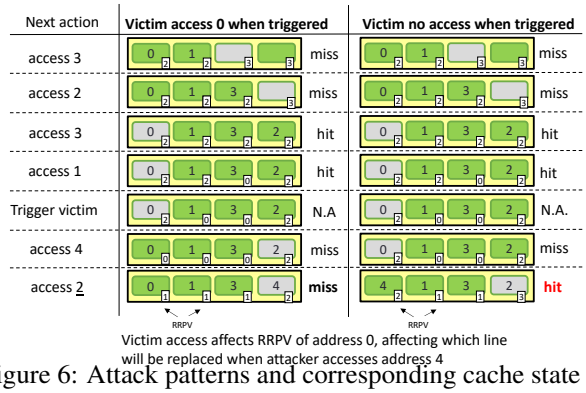


Figure 6: Attack patterns and corresponding cache state for RRIP replacement algorithm and victim access (left)/no access (right) information is inferred by the attacker. Gray blocks indicates the one to be evicted on a cache miss.

Unlike a deterministic replacement algorithm where the next state will be fully determined given the action and current state, the next state is hard to predict in the (pseudo)-random replacement algorithm. Thus, an attack pattern 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 algorithm, there is no single attack pattern that always works in the random replacement algorithm.[5] Instead, we evaluate the attack accuracy of the RL agent over 2,000 evaluation runs. As shown in Table 5, the step reward determines the tradeoff between attack length and accuracy. The evict+reload strategy is similar to the prior attacks on random replacement algorithm [41].

5.4 Bypassing Defense and Detection

To protect against cache timing-channel attacks, a variety of detection and mitigation techniques have been proposed. For example, detection techniques may monitor system behaviors such as memory access patterns and timing using hardware performance counters. Secure cache designs may mitigate timing attacks by reducing interference among different processes or introducing noise or randomness. To understand AutoCAT's ability to adapt to and bypass detection or defense schemes, we implemented three cache timing-channel protection schemes: 1) Partition-locked (PL) cache [83], 2) Hardware Performance Counter (HPC)-based detection [4,15,37,96], and 3) Autocorrelation-based detector similar to CC-hunter [12]. Then, AutoCAT was used to find attack patterns when the protection schemes are deployed.

Partition-Locked (PL) Cache. PL cache [83] 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 attacker program. Further, the victim's access to the locked cache lines will not evict any of the attacker's cache lines. In [27], the formal analysis on a simplified cache model concludes that PLcache is secure against set-based attacks.

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 set-based attack setting and the victim's cache line is pre-installed and locked in the cache. We use a 4-way cache, and the address range of the attacker is 1-5 and the victim either accesses 0 or has no access depending on the secret value. For this setting, training takes about 45 minutes. Table 6 shows the training time (# of steps) and the attack sequence length for a cache with the tree-PLRU replacement algorithm, with and without the PL cache. AutoCAT successfully found an attack that works even with the PL cache, which is shown in Figure 7. In this attack, the victim's cache line (address 0) always stays in the cache, and the victim's behavior (whether the victim makes access or not) does not evict any of the attacker's cache lines. However, the victim's access affects the LRU state. When the attacker made subsequent accesses, it can tell whether the victim accessed address 0 or not by observing if a new block can be brought into the cache. This attack is reported in recent literature [88].

[5]The eviction rate depends on the number and the pattern of memory accesses [41].

Table 6: Comparison of tree-PLRU w/ and w/o PLCache.

Cache	Steps to converge	Episode length
PLCache	232,000	11.66
baseline	176,000	8.58

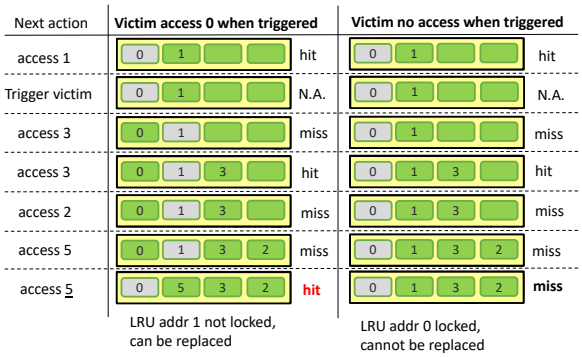


Figure 7: Attack patterns and corresponding cache state for the PL cache where address 0 (owned by victim) is locked and the victim's access (left) / no-access (right) information is inferred by the attacker. Gray blocks indicates the one to be evicted on a cache miss.

Autocorrelation-based Detection. Autocorrelation of cache events have also been proposed as a way to detect the existence of cache-timing channels [12, 90] based on the observation on the common covert-channel patterns. In CC-Hunter [12], two types of conflict miss events (i.e., victim evicting attacker's cache line, $V \rightarrow A$, encoded with "0", and attacker evicting victim's, $A \rightarrow V$, 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 like prime+probe, these two events are interleaved periodically. We can check the autocorrelation C_p at lag p using the following equation: $C_p = \frac{\sum_{i=0}^{n-p}[(X_i-\bar{X})(X_{i+p}-\bar{X})]}{\sum_{i=0}^{n}(X_i-\bar{X})^2}$. If there exists p where $1 \leq p \leq P$ and P is a predefined parameter such that $C_p > C_{threshold}$ (e.g. 0.75), then it is considered an attack.

For example, in a 4-set direct-mapped cache where the victim and attacker'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 attacker, after that one victim access is triggered, and then addresses 4-7 are probed. The event train and the corresponding autocorrelogram is

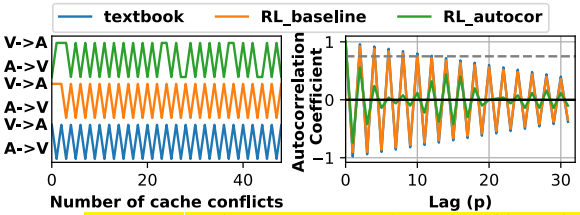


Figure 8: (a) Event train (A→V: attackers' conflict misses with the victim and V→A: victim's conflict misses with the attacker.) (b) Autocorrelograms of the event trains. The dashed-line shows the threshold for detecting an timing channel.

Table 7: Comparison of bit rate, accuracy and autocorrelation.

sampled attacker	bit rate (guess/step)	max autocorrelation	accuracy
textbook	0.1625	0.96	1
RL_baseline	0.236	0.92	0.99
RL_autocor	0.1975	0.55	0.99

shown in Figure 8. The maximum autocorrelation for $p \geq 1$ is 0.96, which is over the threshold.

The AutoCAT generated baseline pattern (*RL_baseline*) can be detected by this autocorrelation-based detector, as shown in the corresponding sampled cache conflict miss event train and the autocorrelogram (Figure 8). The maximum autocorrelation for $p \geq 1$ is 0.92. Note that the event train and the corresponding autocorrelograms will be different for each sample. However, we observe 0.875-0.940 maximxum autocorrelation over 25 distinctive inferences (sampled from randomly generated victim addresses), which are all above the detection threshold.

However, AutoCAT can learn to bypass such autocorrelation 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 autocorrelations, 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 8. The maximum autocorrelation beyond lag 0 is 0.55 in this example. For 25 different samples, we observe the maximum autocorrelation of 0.396-0.772 with only 2 samples over 0.75. The results indicate that the agent (*RL_autocor*) will be able to evade autocorrelagoram detection with high probability.

Table 7 compares the attack samples from the textbook, *RL_baseline*, and *RL_autocor* in terms of bit rate (measured by the number of guesses per step), accuracy, and autocorrelation. All attacks achieve over 0.99 accuracy and the RL agents have higher bit rates than textbook samples. 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 a secret directly 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 the *RL_autocor* agent makes additional accesses to reduce autocorrelation.

ML-based Detection. Machine learning classifiers can also be used for detecting cache-timing attacks. For example, in Cyclone [25], the frequency of cyclic access patterns by different security domains (e.g., $a \rightsquigarrow b \rightsquigarrow a$) for each cache line within each timing 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 pattern counting for each cache line following [25]. We train an SVM classifier 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 is is 98.8%.

Table 8: Comparison of bit rate, guess accuracy and detection rate by the SVM.

sampled attacker	bit rate (guess/step)	attack accuracy	SVM detection rate
textbook	0.1625	1	1
RL_baseline	0.228	0.990	0.907
RL_SVM	0.150	0.964	0.021

We then train the AutoCAT's RL agent (*RL_SVM*) with this SVM detector . If the SVM detector correctly reports the existence of the attack, the RL agent gets a negative reward. We also train a *RL_baseline* agent without detection penalty. We show the result in Table 8. The textbook and the *RL_baseline* can be easily detected by the SVM detector, with the detection rate of 1 and 0.907, respectively. However, when the RL agent is trained with the SVM detection penalty, it can find attack patterns that can bypass the SVM detector, with the detection rate of 0.021, at the cost of a reduced bit rate. This indicates ML-based detector trained on static traces can be easily bypassed by an RL-based attacker and novel training method for ML-based detector is needed.

μarch Statistics-based Detection. Most of the cache-timing attacks cause the victim'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 process and detect an attack at run-time [4,15,37,96]. More recently, [46] 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 the attack is detected when the victim access triggers a cache miss. We set `detection_enable`, so that the episode will be terminated when the victim access is a cache miss. We then assign `detection_reward` (a negative value) as the reward of the episode. This configuration encourages the RL agent to avoid the victim misses, and thus, bypass the statistics-based detection.

Figure 9(b) shows the attack pattern generated by AutoCAT for a 4-way cache with the statistics-based detection scheme. This is a new attack pattern, which can be seen as a novel combination of the two recent attacks in literature (shown in Figure 9 (a)). The attack pattern can be divided into sub-patterns, which are the LRU set-based or LRU address-based attacks [88]. The two sub-patterns overlap with each other in a way similar to the Streamline attack [65]. Based on the gray part of the pattern generated by AutoCAT in Figure 9 (b), we construct a new attack, named *StealthyStreamline*, in Figure 9(c). Compared to the Streamline attack, the new StealthyStreamline attack does not cause cache misses of the victim process, thus is stealthier. Compared

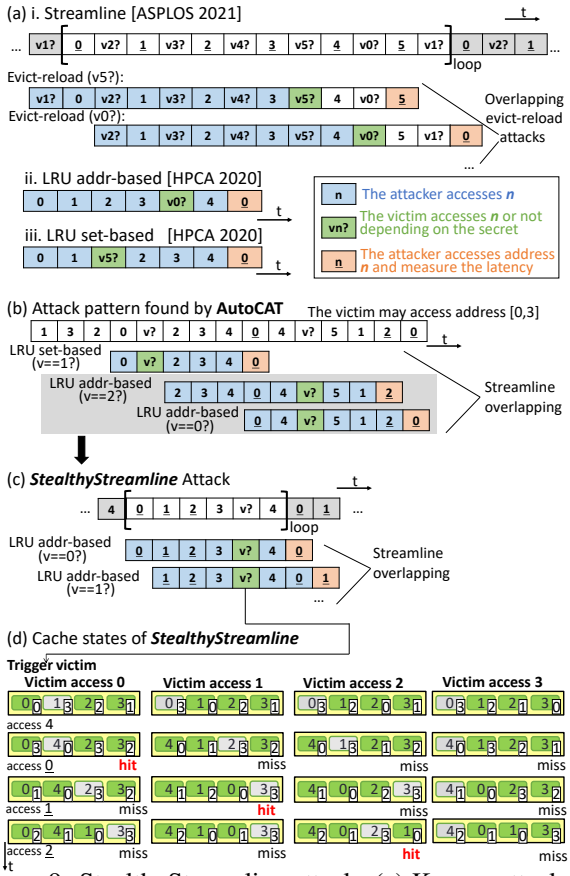


Figure 9: StealthyStreamline attack. (a) Known attack patterns in the literature. (b) The new attack pattern found by AutoCAT, which can be seen as the combination of the two known attacks. (d) StealthyStreamline attack derived from the attack found by AutoCAT. (e) The cache state changes of the StealthyStreamline attack. The numbers in the right bottom corner indicate the LRU age of the cache line.

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 9(d) illustrates the cache state when the StealthyStreamline is transmitting 2 bits. The attacker observes different timing when the victim has 4 different possible secret values.

5.5 Covert Channels on Real Machines

We demonstrate the *StealthyStreamline* covert channel in the L1 data cache on four different Intel processors (Table 9). Two machines have the processors with a 32KB (8-way) L1 data cache, and the other two latest processors employ a 48KB (12-way) L1 data cache. The machines are running different versions of Linux.

We generalize the 2-bit *StealthyStreamline* covert channel in a 4-way cache (in Figure 9) 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. Our implementation is based on the open-source LRU address-based attack in [87].

Table 9: Covert channels on real machines.

CPU	μ-arch.	L1D config	OS	Bit Rate[a] (Mbps)		
				LRU	SS.[b]	Impr.
Xeon E5-2687W v2	Ivy Bridge	32KB(8way)	Ubuntu18	6.2	7.7	24%
Core i7-6700	Skylake	32KB(8way)	Ubuntu18	3.6	4.5	22%
Core i5-11600K	Rocket Lake	48KB(12way)	CentOS8	3.4	5.7	67%
Xeon W-1350P	Rocket Lake	48KB(12way)	Ubuntu20	2.1	3.7	71%

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

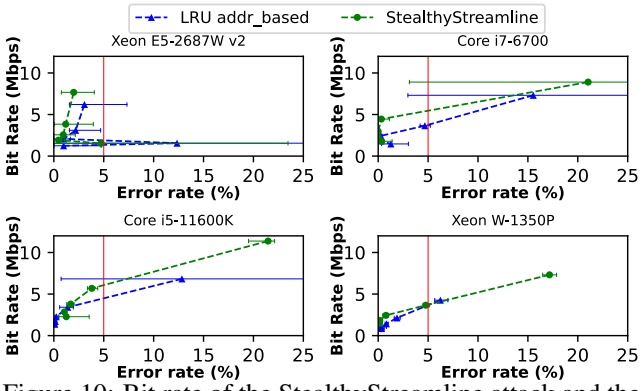


Figure 10: Bit rate of the StealthyStreamline attack and the LRU address-based attack on four different processors. The horizontal error bars show the range of errors across different transmission runs.

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-based PLRU, while the 2-bit *StealthyStreamline* has a low error rate.

Figure 10 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 10. For the error rate of $< 5\%$, 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 a 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 an access takes more cycles than a normal memory access.

Spectre Attack using StealthyStreamline: We demonstrate Spectre V1 attack [36] with the StealthyStreamline as the covert channel. Compared with the LRU address-based covert channel, the 2-bit StealthyStreamline covert channel enables us to encode 4x more symbols with the same cache. Compared with flush+reload or evict+time, StealthyStreamline makes the attack stealthier.

Table 10: Training with different wrong-guess rewards.

	wrong_guess_reward		
	-1,000*	-10,000	-100,000
steps (10^3)	1038 (1000, 1076)	889 (816, 1000)	852 (780, 992)
time (h)	5.1 (4.8, 5.4)	4.5 (4.3, 5.0)	3.4 (2.3, 4.6)
episode length	25.04 (22.27, 27.81)	29.53 (14.97, 37.24)	30.74 (27.40, 35.00)

* Not all the experiments find an attack when wrong_guess_reward = -1,000.

Table 11: Training with different step rewards.

	Step_reward				
	0	-2	-10	-25	-50
steps (10^3)	713 (672, 788)	833 (648, 988)	889 (816, 1000)	1004 (808, 1284)	1977 (464, 3712)
time (h)	2.2 (2.0, 2.5)	3.8 (2.6, 5.3)	4.5 (4.3, 5.0)	4.0 (3.3, 5.3)	7.7 (1.9, 13.8)
episode length	25.94 (23.00, 29.58)	25.23 (16.39, 29.86)	29.53 (14.97, 37.24)	28.29 (19.28, 32.80)	21.62 (13.16, 25.85)

6. DISCUSSION

6.1 Stability of Training and Limitations

In AutoCAT, the agent only get positive reward when guessing correctly. This is defined as *sparse reward*, when most of the rewards received during training are non-positive. Reinforcement Learning algorithms can wander aimlessly or stuck at local optima when dealing with sparse reward. [24,74]. The learning speed and accuracy of RL also have some correlation to reward values, especially for simple MLP models. Reward values using domain knowledge can improve the performance of an RL agent. As our goal is to train an agent that can guess the secret ($addr_{secret}$) correctly with minimal number of steps, we chose a positive value for correct_guess_reward, negative values for wrong_guess_reward and step_reward.

We evaluated the impact of reward values on the performance of the RL agent by changing the reward values using the default three-layer MLP model. When the agent reaches 95% accuracy, we consider an attack pattern is found. We chose three metrics to evaluate the agent: 1) the number of time steps in 10^3 until convergence, 2) clock time in hours until convergence, and 3) episode lengths when the attack is found, i.e., the length of the attack. For each metric, the results of average (minimum, maximum) values are presented to show the variation between the training trials in Table 10 and Table 11. As a baseline, we use the reward values of correct_guess_reward = 200, wrong_guess_reward = -10,000, and step_reward = -10. We evaluated an 8-way cache set. The victim either accesses cache line 0 or makes no access. The attacker address range is 1 to 8. Each configuration has been tested for 3 random seeds.

Scale of rewards. We train the model with different scales (0.01x - 1x) of the rewards but keep the ratio of the reward values the same. The result suggests that the learning performance is relatively stable even though the absolute values of the rewards do impact the performance. This is because we use Adam optimizer [33] whose step formula is not sensitive to the scale of gradients.

Wrong guess reward. We test how training is affected by the wrong-guess reward values. As shown in Table 10, the agent converges slightly faster than the baseline when the penalty is larger. There is no significant difference in episode length. When correct_guess_reward is 200 and wrong_guess_reward is -1,000, the agent sometimes fails to find an attack and converges to a random guess.

Step reward. In Table 11, we show the results when the step reward changes. Learning speed is improved by decreasing the step from -50 to 0. This is because the step penalty discourages the RL agent from exploring longer attack patterns which could lead to a successful attack. However, a larger step penalty results in a shorter attack pattern.

We also found that the Transformer model is less sensitive to the reward value settings. For our experiments with a Transformer model, we used the reward of correct_guess_reward = 1, wrong_guess_reward = -1, and step_reward = -0.01.

Limitations of AutoCAT. Given to the repetitive structure of a cache, we focus on training AutoCAT with a small cache configuration with a few cache sets and generalize the attack to real machines manually. The scalability and generalizability when applying to multiple levels of large caches still needs to be further explored. Our initial investigation showed promising results that suggest AutoCAT can be further improved with new ML model architectures such as Transformers [13,30,79] and RL generalization methods [35,61]. In the future work, we plan to investigate extending the AutoCAT approach complex cache configurations and other types of microarchitecture timing channels beyond cache attacks.

Another remaining challenge in AutoCAT lies in analyzing many attack sequences from the RL agent. We had to manually analyse and categorize the patterns found by AutoCAT. Ideally, we need an automated pattern analysis framework.

6.2 Comparison with Search Algorithms

RL takes a fewer number of steps to find a successful attack than exhaustive 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. For brevity, assume that the attacker can access N addresses and the secret is one bit. There are N possible a_X actions, 2 possible a_{gY} actions and 1 possible a_v action the agent can choose to perform at each step. A prime+probe attack will look like:

$$1 \to 2 \to 3 \to ... \to (N-1) \to N \to v \to$$
$$1 \to 2 \to 3 \to ... \to (N-1) \to N \to g$$

The prime+probe attack takes $2N+2$ steps, and there are $(N!)^2$ such patterns considering the permutations within prime steps and probe steps. For each step, there are $N+1$ actions that do not terminate the episode and 2 actions that terminate the episode. There are $2 \times (N+1)^{2N+1}$ patterns that are exactly $2N+2$ steps. Thus, on average, we can find one prime+probe pattern every M patterns, 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 attack, considering each attack pattern takes $2N+2$ steps. Last-level caches usually have more than 8 ways and it will be infeasible for exhaustive search. With RL, as shown in Section 6.1, the agent converges within $\sim$ 1 million steps.

There may be several intuitions for how RL finds an attack with less steps. First, the RL agent learns a progressively better policy during its exploration, and on-policy exploration in RL proactively avoids low-reward regions and focuses on high-reward ones. This avoids a lot of unnecessary enumeration of ineffective actions, compared to exhaustive search.

Second, the policy is parameterized with neural networks, which can generalize to novel states and make smart decisions. Note that different network architecture could lead to different biases for generalization. For example, as shown in the experiment section, compared to MLP, Transformers may lead to better performance in our applications.

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

6.3 Dynamically Mapped Caches

To thwart set-based cache timing attacks, recent studies [59, 60, 66, 83] proposed changing the cache mapping dynamically and make it hard for the attacker to deterministically co-locate with the victim on desired cache sets. As a preliminary study, we implemented a dynamic remapping scheme similar to CEASER [59], and tested AutoCAT with dynamic remapping. Similar to CaSA [8], we assume an attacker can keep monitoring and learning the cache set mapping and launch an attack when needed. We use a 4-set 2-way cache in this experiment. The victim accesses address 0 or not depends on the secret. The attacker's address range is 1-11. When remapping, address 0-11 are randomly permuted and map to different sets. In Figure 11, we show the accuracy of the agent as the function of the training epochs. We set the remapping period as 2M memory accesses[6]. We found that the attack accuracy drops after a remapping, but the agent can adapt to the new mapping quickly within around 50 epochs in many cases. While fully understanding the effectiveness of RL-based attacks against dynamic remapping will need further studies, the initial result is promising in a sense that it suggests that the RL agent can adapt to a mapping change.

It is also interesting to note that our preliminary result shows that the attack accuracy improves gradually even when an attacker does not have a full eviction set, which is the same observation made by the previous study [8] (there is a trade-off between calibration time and attack accuracy).

7. RELATED WORK

Automated timing channel analysis and discovery. Prior studies proposed systematic analysis methods for cache timing channels based on simplified cache models [17, 27, 86]. 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 [19], taint analysis [21], fuzzing [20, 48, 84], relational testing [51], and formal methods [76]. Compared to exhaustive search or fuzzing-like methodologies, we believe the RL-based method is more efficient and expressive for complex attack sequences (e.g., Osiris [84] cannot find prime+probe attack because it requires multiple accesses). 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, which often require significant human efforts [11].

[6]Note that the RL training process in this preliminary study is not yet optimized to make a full use of these memory accesses.

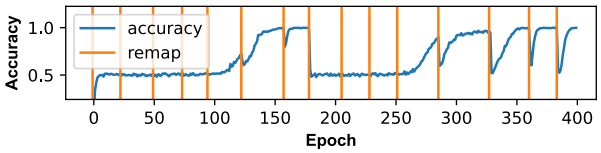


Figure 11: Attack accuracy as a function of training epochs, the orange vertical lines show when the remapping happens.

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 [12] and ReplayConfusion [90] focus on detecting an attack but by themselves do not remove the leakage. We show that the RL agent has the potential to automatically generate attacks to evade detection schemes. On the other hand, defense mechanisms [18, 34, 50, 60, 66, 75, 89] focus on mitigating or removing the interference that leads to timing channels. This paper shows that RL also has the potential to automatically evaluate the security of such defense, and shows that AutoCAT can bypass PLCache [83].

ML for security. Machine learning was used in the computer security domain for anomaly detection [39], website fingerprinting [38, 68], and other analysis tasks. However, traditional supervised learning cannot find new attacks without known attack patterns 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 [49], IoT security [77], autonomous driving security [63], power side-channel attacks [62], circuit test generation [54], power side-channel countermeasures [64], and hardware Trojan detection [53]. To our knowledge, our work is the first of its kind in using RL to actively and automatically generate attacks in the microarchitecture security domain.

8. CONCLUSION AND FUTURE WORK

In this paper, we propose to use reinforcement learning to automatically find existing and undiscovered timing-channel attacks. As a concrete example, we build the AutoCAT framework, which can explore cache-timing attacks in various cache configurations and attacker/victim settings, and under different defense and detection mechanisms. Our experimental results show that the RL agent can find practical attacks within one hour for a 4-way cache and in about 5 hours for an 8-way cache with a simple MLP model. We expect more advanced models can learn even faster. The RL agent also discovered the StealthyStreamline attack, which is a novel attack with a higher bit-rate than attacks reported in previous literature. AutoCAT shows RL is a promising method to explore microarchitecture timing attacks in practical systems.

REFERENCES

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

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

[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. Dębiak, 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."

[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] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "CaSA: End-to-end quantitative security analysis of randomly mapped caches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1110–1123.

[9] 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.

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[11] 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.

[12] 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.

[13] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, "Decision transformer: Reinforcement learning via sequence modeling," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=a7APmM4B9d

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[24] J. Hare, "Dealing with sparse rewards in reinforcement learning," *arXiv preprint arXiv:1910.09281*, 2019.

[25] 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.

[26] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2009, vol. 2.

[27] 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.

[28] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Comput.*, vol. 9, no. 8, Nov. 1997.

[29] 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.

[30] M. Janner, Q. Li, and S. Levine, "Offline reinforcement learning as one big sequence modeling problem," in *Advances in Neural Information Processing Systems*, 2021.

[31] 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.

[32] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1-2, pp. 99–134, May 1998. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S000437029800023X

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR (Poster)*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[34] 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.

[35] R. Kirk, A. Zhang, E. Grefenstette, and T. Rocktäschel, "A survey of generalisation in deep reinforcement learning," *CoRR*, vol. abs/2111.09794, 2021. [Online]. Available: https://arxiv.org/abs/2111.09794

[36] 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.

[37] 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.

[38] 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.

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

[40] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2018, pp. 3053–3062.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

[55] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. fGimelshein, 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. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[56] 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.

[57] M. L. Puterman, "Markov decision processes: Discrete stochastic dynamic programming," *Journal of the Operational Research Society*, 1995.

[58] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[59] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[60] 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.

[61] 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. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/2b38c2df6a49b97f706ec9148ce48d86-Paper.pdf

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

[71] S. Singh, M. R. James, and M. R. Rudary, "Predictive state representations: A new theory for modeling dynamical systems," in *Proceedings of the 20th conference on Uncertainty in artificial intelligence*. AUAI Press, 2004, pp. 512–519.

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

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

[74] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, 2nd ed. MIT press, 2018.

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

[76] 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.

[77] 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.

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

[79] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[80] 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.

[81] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 39–54.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

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

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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

[93] 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.

[94] 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.

[95] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/yuan-yuanyuan

[96] 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.

Key Modifications

We list the main changes in the revision below.

1. **StealthyStreamline on real machines (new Section 5.5)**:
   We moved the results from old Section 5.4-Hardware Performance Counter (HPC)-Based Detection to new Section 5.5. In addition, we measured the covert channel in three more processors and present the results in new Table 9 and new Figure 10.
2. **FSM formulation (Section 2)**:
   Section 2 was reorganized with the order of Section 2.2 and Section 2.3 being switched.
   We added a discussion in Section 2.3 to clarify the purpose of the FSM formulation.
3. **Cache model and simulation environment**
   We added sentences at the end of Section 4.1 to clarify that our cache model uses a full cycle-level simulator instead of simplified models used in previous studies.
4. **ML-based detection scheme (new paragraphs in Section 5.4)**
   We added additional experiments on how AutoCAT can bypass ML-based detection scheme. We implemented a Cyclone-like [25] detector in our cache model and trained a SVM model for detection.
5. **Random Remapping (Section 6.3)**:
   We added a preliminary study of AutoCAT for a small randomized cache with dynamic remapping.
6. **(pseudo)random replacement algorithms (Section 5.3)**:
   We added Table 5 and discussed the result of random replacement algorithms.
7. **Cache with prefetcher (Section 5.2)**:
   We added additional experimental results on caches with a next-line prefetcher and a stream prefetcher. The results are in Table 3 configuration 2,13, and 14.
8. **Known PLcache vulnerability (Section 5.4 – Partition-Locked (PL) Cache**:
   We clarified that AutoCAT found new vulnerability that was not identified in the previous cache timing-channel analysis paper [27].
9. **Terminology: actions, attack pattern, attack category (Section 2)**:
   The differences among attack sequence, attack pattern, and attack category are clarified in footnote 1. Figure 1 is updated.
10. **HPC-based detection (Section 5.4)**:
    We renamed the original “HPC-based detection" to “μarch Statistics-Based Detection" to clarify the scope.
11. **Limitations of AutoCAT (Section 6.1)**:
    We added a discussion on the limitations of AutoCAT.
12. **Rewards (Section 4.6)**:
    We added the first few sentences to clarify when the rewards are assigned.
13. **CC-hunter Results (Section 5.4)**
    We found a bug in our original CC-hunter implementation. We have fixed the bug and updated the experimental results in Section 5.4. The main findings did not change.
14. **Typo in Figure 4**
    Fixed following the comments in Review-B.

New Experimental Results

Here is a short list of new experimental results.

1. **StealthyStreamline on more real-world (Intel) processors** (Section 5.5)
   Table 9 and Figure 10
2. **Bypassing ML-based detection schemes (Cyclone [25])** (Section 5.4)
   Table 8
3. **AutoCAT for dynamic remapping (Section 6.3)**
   Figure 11
4. **Random replacement policy**
   Table 5
5. **Attack exploration with a prefecher enabled**
   Table 3, configuration 2, 13, 14
6. **CC-hunter bug fix**
   Figure 8 and Table 7