WIP: Automatic Transient Execution Attack Detection

ABSTRACT

Transient execution attacks have taken the computer architecture community by surprise, leaving researchers with the challenge to tackle the root source of the vulnerability. Since its public announcement, academia and industry have been working ferociously to defend against these clever types of attacks. While some of these attacks have a well defined pattern, it is extremely difficult to mine that pattern from labelled examples. There are several reasons for this difficulty: 1) The classification algorithms, such as decision tree learning and deep neural networks are not applicable as the length of the traces are unbounded; 2) The classical automata learning algorithms (Angluin's L^* algorithm, Gold's algorithm) are not applicable since the data domain is infinite; and 3) the SMT-based approaches to automata synthesis do not scale to the size of dataset.

We propose the use of *register automata* to capture such pattern and propose a Bayesian learning approach to identify register automata from labelled traces. Our algorithm can readily be implemented inside various probabilistic programming languages to automate the task of learning patterns of transient execution attacks. While our prior attempts to scale a PyMC3 based implementation are not successful to the size of our current dataset, we plan to investigate deep probabilistic programming tools to scale our Bayesian register automata learning algorithm.

KEYWORDS

Computer Architecture Security, Security Verification, Bayesian Learning, Register Automata

1 INTRODUCTION

Transient execution attacks are difficult to identify because they involve both software and hardware properties. One of the most prominent examples of such attacks is Spectre, a processor level vulnerability that relies upon speculative execution to perform intentional branch mispredictions that can leak information through completed, yet not retired, memory instructions [17]. Before the Spectre attack was published in early 2019, transient state was not considered to be a problem given that it was microarchitectural state that was assumed to be invisible to the user. Unfortunately, as the Spectre attack has shown, prior understanding of microarchitectural state was incorrect, and these new complex vulnerabilities have given the hardware community a new perspective about the security and privacy of our machines.

Prior work fails to robustly address these attacks because the underlying root problem is a very effective performance feature which cannot be ignored. Several attempts to patch these types of attacks either suffer from high performance costs or have later on been found to not be robust enough to sustain other variants of the attack. For example, soon after the Spectre attack was published, several defenses involved limiting the state updates of caches by introducing shadow copies which would only be "visible" once the modifying instruction is retired [16, 31]. The problem with this approach is that the first step of the attack, the access to the secret, is not prevented, and other modes of leaking the secret were introduced that did not involve caches, making these types of defenses obsolete [6, 23, 25].

We observe that a lot of proposed defenses do not target the root problem that enables the attacks in the first place: the ability of attackers to access secrets. For performance purposes, modern processors allow the memory access to happen while permission checks happen in parallel. This performance optimization creates the vulnerability that allows transient execution attacks to take place. Since the reasoning about transient state is not easy in practice, the programmer support for security analysis integrating both software and hardware transient state is limited. We address this gap by developing a *Bayesian learning framework* that incorporates components from both the software and the hardware to learn patterns detecting vulnerabilities to this class of attacks.

The key contribution of this work is the insight that the conditions that are required to exploit a transient state vulnerability can be explained via *register automata* (RA) [15]: a generalization of finite state automata with the capability to store and retrieve data using a finite set of data registers. We show an example of how to explain the Spectre attack with a RA and how this RA can be used to recognize potentially vulnerable microarchitectures. Our goal is to automate the discovery of such patterns from labeled systems traces. This *work-in-progress* paper targets the following problem:

Register Automata Learning. *Given a binary-labeled (safe vs. vulnerable) set of system traces (of unbounded length), learn a register automaton discriminating safe and vulnerable traces.*

There are several difficulties in developing such a learning algorithm. First, since the register automata operate on infinite data words, the well-known automata learning algorithms such as L^* [2] or Gold's algorithm [10] are not sufficient to learn such a machine. The same limitation also affects the SMT-based approaches [20]; moreover, the scale of the datasets arising from our application using such exact approaches is too large to handle. Recurrent neural networks [12, 28] can be used to learn patterns in sequences, but are notoriously difficult to train [22].

Our approach is based on Bayesian data analysis [9]. Bayesian data analysis begins by providing a generative model of the underlying computation capturing the prior beliefs of the practitioner captured as a joint probability distribution (prior distribution) on the observable and latent variables. The next step performs a conditioning on the observed dataset to calculate an appropriate posterior distribution refining the generative model to better fit the observed dataset. While for simple models and distributions, such conditioning can be done analytically, when the models are complex we require computational tools to automate this conditioning. Probabilistic programming languages [27] automate the process of numerical conditioning for Turing-complete description of generative models. We propose the use of probabilistic programming in inferring register automata characterizing the transient execution attack pattern is implicit in a given dataset.

In our approach, the practitioner provides a partial sketch of a register automaton where some key information (states, transitions, and register updates) maty be left unspecified. In an extreme case, the algorithm can also work with an upper bound on the number of states and register variables. The algorithm then fills the register automaton sketch by introducing a uniform prior distribution on the missing transitions. It then performs a conditioning on the labelled dataset to compute the posterior distributions and return a maximum a posteriori probability estimate as a register automaton.

In summary, we propose a detection platform to identify microarchitectures that could potentially lead to a transient execution attack by learning a register automaton characterizing patterns that defines the machine conditions to enable these attacks. The goal of our platform is to identify patterns in a processor's execution pipeline indicative of transient state vulnerabilities and to identify problematic software. Due to the evolving nature of these class of attacks, many processors remain at risk to this vulnerability [7] further necessitating an efficient detection process.

2 BACKGROUND

2.1 Transient Execution Attacks

Transient execution attacks are a relatively new class of hardware exploits that rely on pipeline flushes to leak information through microarchitectural transient state. Speculative execution is a very effective performance optimization, commonly used in modern processors, to accelerate the slowest operations required of general purpose machines. Very rarely, the speculated path is found to be incorrect and thus the processor needs to clean-up the speculated state before it is visible in the rest of the system. The clean-up of speculated state requires the flushing of the *speculatively executed* instructions as well as resetting the register values. These flushed instructions, known as *transient instructions*, can create side-effect—also known as transient state—in the microarchitecture, such as cache state, which can then be exploited by an attacker to learn secret information [7].

Spectre is the original version (along with the Meltdown variant [18]) of transient execution attacks that performs calculated branch training to cause branch mis-predictions to speculatively execute unauthorized memory instructions. By training the branch predictor to purposely execute an incorrect instruction sequence, the attacker is able to access secret data which they can then use to access an attacker controlled array. Given that the secret value is used as the index into the attacker array, the fact that a particular bucket of the attacker array is in the cache, even after the speculated instructions are flushed from the pipeline, leaks the actual value of the secret in a side-channel attack. Memory operations that complete during the speculative execution but then get flushed from the pipeline are the most vulnerable. Given the simplicity of this attack, we start by looking at our approach to identify Spectre attacks. In our future work, we plan to extend the approach and generalize it to detect other variants in this class of attacks.

2.2 Grammatical Inference

The classical problem of grammatical inference seeks to learn grammars, or some representation thereof, from a finite number of samples [8, 13]. Approaches for learning such grammars are generally categorized as either active or passive. Passive inference seeks to mine the underlying specification from a static dataset of observed traces. Active methods differ in that the System Under Learning (SUL) can be queried to guide the learning process. The L^* algorithm for learning regular languages [2] is the quintessential example, and assumes the existence of a minimally adequate teacher capable of answering membership and equivalence queries. This approach has been broadly adopted to learn interface automata [1], deterministic Mealy machines [21], automaton representations of recurrent neural networks [29], and MDPs [26]. This approach has also been extended to learn restrictions of register automata [14], but no algorithm exists to handle the full class of register automata.

3 PROBLEM DEFINITION

3.1 Pattern of Spectre Vulnerability

To identify potentially vulnerable microarchitectures one needs to evaluate software and hardware in combination. We propose to look at both the instruction sequence as well as the active pipeline during execution, in combination, in order to determine when branch mispredictions occur and whether or not they flush completed memory operations.

We are able to examine the pipeline at each instruction interval to determine whether or not a given instruction has been completed and has modified any microarchitectural state (*i.e.* cache state) , but not written back to the register file (retiring). In this case, a completion without a retire implies that this instruction has been flushed from the pipeline and is possibly vulnerable to leaking information through microarchitectural side-channels. If the instruction in question is a memory load operation and the pipeline is flushed due to a branch misprediction, then we can identify a Spectre vulnerability. However, if the flushed operation is due to a different reason, such as a page fault, we conclude that the instruction sequence does not have a Spectre vulnerability. To determine the cause of the pipeline flush, we must examine the sequence of software instructions and compare the first flushed instruction to the first retired instruction after the flush. If the program counter matches for each instruction, indicating the re-execution of the same instructions and therefore a sign that the path was correct just entered at the wrong time, then we can determine that the processor flushed the pipeline for reasons other than branch misprediction. However, two distinct program counters imply that a different instruction path is now taken, indicating that the flushed instructions were speculatively executed and incorrectly taken.

This vulnerability pattern is graphically depicted in Figure 1 in an extended finite-state machine like description. Here we have five states q_0, q_1, q_2, q_3 and q_{acc}. Each transitions is labeled with a tuple $\Sigma \times \mathbb{D}$ where Σ is a finite alphabet and $\mathbb{D}$ is an infinite data domain (program counters). This machine also uses a register variable x that can be used to store and compare data values for equality. For instance, the transition from q_0 to q_1 remembers the value of the program counter in the register x and from the state q_3 the data value with the retired instruction is matched with the stored data value to detect the vulnerability. We formalize such extended finite-state machines with register variables via register automata.

3.2 Register Automata

A register automaton (RA) [15] is a tuple $A = (Q, \mathcal{R}, \Sigma \times \mathbb{D}, q_0, F, \delta)$ where Q is a finite set of states, $\mathcal{R} = \{1, 2, \ldots, k\}$ is a finite set of registers, Σ is a finite alphabet, $\mathbb{D}$ is a (potentially infinite) data

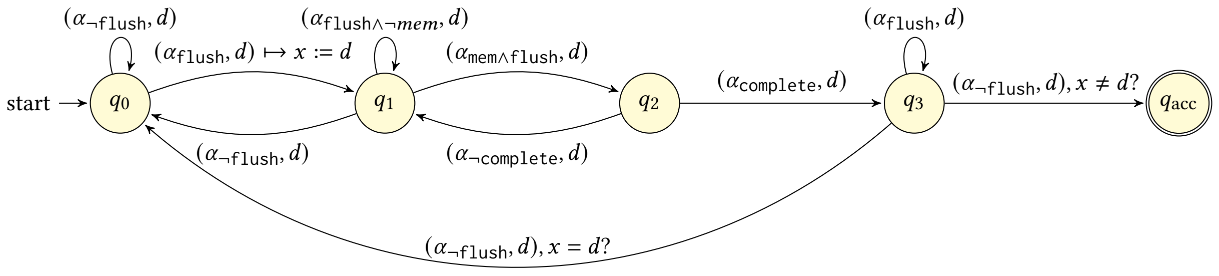
start
q_0
q_1
q_2
q_3
q_{acc}
$(\alpha_{\neg\texttt{flush}}, d)$
$(\alpha_{\texttt{flush}}, d) \mapsto x := d$
$(\alpha_{\texttt{flush}\wedge\neg mem}, d)$
$(\alpha_{\texttt{mem}\wedge\texttt{flush}}, d)$
$(\alpha_{\texttt{complete}}, d)$
$(\alpha_{\texttt{flush}}, d)$
$(\alpha_{\neg\texttt{flush}}, d), x \neq d?$
$(\alpha_{\neg\texttt{flush}}, d)$
$(\alpha_{\neg\texttt{complete}}, d)$
$(\alpha_{\neg\texttt{flush}}, d), x = d?$

Figure 1: A register automaton characterizing SPECTRE vulnerability. Here $\alpha_{\texttt{pred}}$ signifies all alphabets matching the predicate `pred`. Key predicates include `memory`, `flush`, and `complete`. The negation of `flush` is `retire`.

domain, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta \subseteq Q \times (\Sigma \times \mathbb{D}) \times 2^{\mathcal{R}} \times Q \times 2^{\mathcal{R}}$ is the transition relation with the semantics that for a transition $(q, a, d, C, q', D) \in \delta$, the set C indicates the guards on registers requiring that all the registers $x \in C$ must evaluate to d to enable this transitions, and after the transition all of the register in D must be set to the data value d.

A register valuation v is a function $v : \mathcal{R} \longrightarrow \mathbb{D}$. Let V be the set of valuations. Let $\overline{0} \in V$ be the valuation $x \in \mathcal{R} \mapsto 0$. A configuration of a A is a tuple in $Q \times V$ providing a state and a valuation to the register variables. The initial configuration is $(q_0, \overline{0})$. A data word is a word in $(\Sigma \times \mathbb{D})^*$. Given a *data word* $(a_0, d_0), (a_1, d_1), \ldots, (a_n d_n)$, a run of A is a sequence

$$(q_0, v_0), (a_0, d_0), (q_1, v_1), \ldots, (a_n, d_n), (q_{n+1}, v_{n+1})$$

such that (q_0, v_0) is the initial configuration and for every $0 \leq i \leq n$ we have that $(q_i, (a_{i+1}, d_{i+1}), C, q_{i+1}, D) \in \delta$ and for every register $x \in C$ we have that $v_i(x) = d_{i+1}$ and for every register $x \in D$ we have that $v_{i+1}(x) = d_{i+1}$. A run is accepting if it ends in a final state. The language $L(A)$ of a register automaton A is the set of all data words that have an accepting run in A. The emptiness problem for register automata is decidable and is in PSPACE [5, 15].

3.3 Bayesian Register Automata Learning

Given the sets of positive and negative instances of data words $\mathcal{D}_+$ and $\mathcal{D}_-$ and a bound on the states and register variables, find a register automaton A consistent with the dataset, i.e. $\mathcal{D}_+ \subseteq L(A)$ and $\mathcal{D}_- \cap L(A) = \emptyset$. Note that this exact automata learning problem is already intractable for regular languages. We propose a Bayesian learning approach to solve this problem.

We define a probabilsitic extension of register automata $P_A = (Q, \mathcal{R}, \Sigma \times \mathbb{D}, q_0, F, \delta)$ where $Q, \mathcal{R}, \Sigma \times \mathbb{D}, q_0, F$, are define in analogous way to a register automaton, and the transition relation $\delta \subseteq Q \times (\Sigma \times \mathbb{D}) \longrightarrow \textsc{Dist}(2^{\mathcal{R}} \times Q \times 2^{\mathcal{R}})$ provides a probability distribution on guards, next states, and register updates. Let $\Theta(P_A)$ be the set of probability distribution parameters used in P_A. Given the dataset $\mathcal{D}$ of positive and negative examples, we are interested in computing a posterior on the distribution parameters, i.e.

$$p(\Theta \mid \mathcal{D}) = \frac{p(D \mid \Theta) p(\Theta)}{p(D)}.$$

Let A^* be the register automata corresponding to maximum a posteriori probability (MAP) estimate from $p(\Theta \mid \mathcal{D})$. Our goal is to infer A^* using probabilistic programming.

4 METHODOLOGY

To test our conjecture on the register automaton showin in Figure 1, we conducted an preliminary proof-of-concept study. By running known SPECTRE vulnerable and known safe programs on a state of the art processor simulator, we were able to use the processor trace to generate examples/counterexamples to learn register automata.

4.1 Trace Generation

For our data generation, we use the emulation mode of the gem5 simulator [4], a cycle accurate processor simulator. We set the configuration of the simulator to use an x86 out of order processor with LTAGE branch prediction to get the best results. The simulator provides logging support, so we are able to easily produce trace files when running programs on the simulated processor. As the processor is out of order, we enumerated each instruction with a sequence number which was composed as a function of fetch time and micro-operation of the fetched instruction.

We produced vulnerable and safe traces as positive and negative sets $\mathcal{D}_+$ and $\mathcal{D}_-$. The traces in $\mathcal{D}_+$ were generated by running a known vulnerable program, `spectre.c` [19], on the simulator. This program contains a victim function which performs a memory load:

```
void victim_function(size_t x) {
    if(x < array1_size) {
        temp &= array2[array1[x] * 512];
    }
}
```

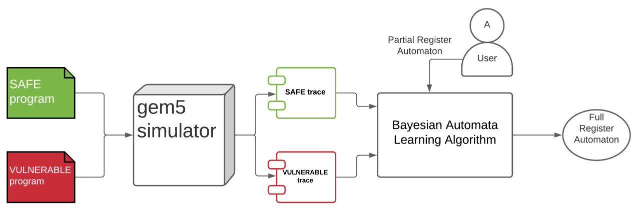

Figure 2: Methodology: Data generation and learning.

This function is vulnerable to Spectre attack because of the memory load of array2 on line 3. Here, array 2 is an attacker controlled array and array 1 is a victim array. If line 3 is speculatively executed before the bounds check on line 2 completes, then an illegal value can be used to index into array1 and leak secret information by indexing into array2 with the secret value. The secret value can then be extracted via a cache side-channel attack.

Safe traces are produced through two distinct fencing approaches: hardware fencing and software fencing. Hardware fencing approaches insert memory fences after conditional branches in the decode stage of the pipeline while software fencing uses fences within the source code of the victim function. Both approaches produce similar traces, so we focus on software fences alone since they are easier to implement. A safe trace is produced by adding a serializing memory fence instruction:

```
void victim_function(size_t x) {
    if(x < array1_size) {
        _mm_lfence();
        temp &= array2[array1[x] * 512];
    }
}
```

By adding a memory fence, Spectre vulnerabilities are mitigated at the software level at the cost of significant execution overhead. However, this is not a concern as our goal is data generation.

4.2 Algorithm Implementation

Instead of developing a purpose-built Bayesian inference engine for our algorithm, we exploited probabilistic programming language (PPL) frameworks to implement our Bayesian learning procedure. PPLs provide rich, structured programming constructs to define generative models and automate the inference procedure by numerically conditioning on the observations [30]. There are several variants of PPL including WebPPL [11], a javascript based probabilistic programming language (PPL), and PyMC3 [24].

Our preliminary results showed that while small finite automata can be learned effectively, such automata are insufficient to capture Spectre vulnerabilities. Our initial attempts at Bayesian learning using PyMC3 allowed us to learn simple acyclic automata. However, this implementation failed to scale for the dataset collected in this work. We posit that PPLs based on approximation architectures (neural networks), such as Pyro [3], will be effective in learning register automata: this is the thrust of our current research.

REFERENCES

[1] Fides Aarts and Frits Vaandrager. 2010. Learning i/o automata. In *International Conference on Concurrency Theory*. Springer, 71–85.

[2] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106.

[3] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. arXiv:1810.09538 [cs.LG]

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* (2011).

[5] Patricia Bouyer, Antoine Petit, and Denis Thérien. 2003. An algebraic approach to data languages and timed languages. *Information and Computation* 182, 2 (2003), 137–162.

[6] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking data on meltdown-resistant cpus. In *ACM SIGSAC Conference on Computer and Communications Security*.

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In {*USENIX*} *Security Symposium*.

[8] Colin De la Higuera. 2010. *Grammatical inference: learning automata and grammars*. Cambridge University Press.

[9] Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. 1995. *Bayesian data analysis*. Chapman and Hall/CRC.

[10] E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and control* 37, 3 (1978), 302–320.

[11] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2021-9-25.

[12] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In {*USENIX*} *Security Symposium*.

[13] James Jay Horning. 1969. *A study of grammatical inference*. Technical Report. STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE.

[14] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. 2012. Inferring canonical register automata. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 251–266.

[15] Michael Kaminski and Nissim Francez. 1994. Finite-memory automata. *Theoretical Computer Science* 134, 2 (1994), 329–363.

[16] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Design Automation Conference (DAC)*. IEEE.

[17] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy (SP)*. IEEE.

[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In {*USENIX*} *Security Symposium*.

[19] Jason Lowe-Power. 2018. Visualizing Spectre with gem5. Blog post: http://www.lowepower.com/jason/visualizing-spectre-with-gem5.html.

[20] Daniel Neider and Nils Jansen. 2013. Regular model checking using solver technologies and automata learning. In *NASA FM Symposium*. Springer, 16–31.

[21] Oliver Niese. 2003. *An integrated approach to testing complex systems*. Ph.D. Dissertation. Technical University of Dortmund, Germany.

[22] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *International conference on machine learning*. PMLR, 1310–1318.

[23] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. 2021. I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches. *International Symposium on Computer Architecture (ISCA)* (2021).

[24] John Salvatier, Thomas Wiecki, and Christopher Fonnesbeck. 2015. Probabilistic Programming in Python using PyMC. arXiv:1507.08050 [stat.CO]

[25] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. Springer.

[26] Martin Tappler, Bernhard K. Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim G. Larsen. 2019. L^*-Based Learning of Markov Decision Processes. In *Formal Methods FM 2019 (Lecture Notes in Computer Science)*. Springer, 651–669. https://doi.org/10.1007/978-3-030-30942-8_38

[27] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. arXiv:1809.10756 [stat.ML]

[28] Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 5247–5256.

[29] Gail Weiss, Yoav Goldberg, and Eran Yahav. 2019. Learning Deterministic Weighted Automata with Queries and Counterexamples. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*. 8558–8569. https://proceedings.neurips.cc/paper/2019/hash/d3f93e7766e8e1b7ef66dfdd9a8be93b-Abstract.html

[30] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2015. A New Approach to Probabilistic Programming Inference. arXiv:1507.00996 [stat.ML]

[31] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. 2019. Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum). In *International Symposium on Microarchitecture (MICRO)*.