Detection of Spectre Attacks Using Hardware-based Cache Access Patterns

An Honors Thesis

in

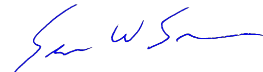
Engineering Sciences

by Wyatt Ellison

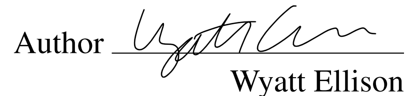
Thayer School of Engineering
Dartmouth College
Hanover, New Hampshire

June 2025

Advisor ____________________
Professor Peter Chin
Thayer School of Engineering



Advisor ____________________
Professor Sean Smith
Dartmouth Computer Science



Author ____________________
Wyatt Ellison

ABSTRACT

Transient execution attack vulnerabilities are built into modern processors as a result of CPU pipelining and out-of-order (O3) execution optimizations. Spectre is a type of transient execution attack that uses predictive components to execute transiently to bypass software-based security policies. These attacks leverage microarchitectural side-channels to leak information from transiently executed instructions. Spectre detection tools exist as both software-based and hardware-based implementations. Software mitigations often look at cache signals using hardware performance counters (HPCs) while hardware mitigations tend to use instructions as features. This project introduces a hardware-based machine learning model that detects Spectre attacks using cache access and transient cache access event patterns. The gem5 simulator was used to collect transient cache access data for several Spectre implementations as well as benign programs, and then used to train a Mamba state space model to detect Spectre. This proof-of-concept model was able to accurately differentiate between Spectre attacks and benign programs for 99.12% of the test cases.

Acknowledgements

This was quite the project, and so I have a lot of people to thank. First are my two advisors, Prof. Sean Smith and Prof. Peter Chin. Prof. Smith – thank you for teaching some of my favorite classes during my time here and helping get me so interested in computer hardware. Prof. Chin – this project would not have been possible without your support from the engineering department. Thank you for always being so encouraging even when I was struggling with the most project objectives.

Dr. Xavier Cadet helped me learn about the practical aspects of the machine learning parts of this project. He helped me find a lot of great libraries and tools to use and greatly advanced the speed at which I was able to get work done right when I was getting down to crunch-time. Thank you, Dr. Cadet.

Sam Barton, Mihir Singh, Noah Dunleavy, and Zach Ellison all provided great technical advice for this project, for which I am grateful. I always appreciated being able to vent when I was stuck on something with this project, and while others might have just given support, you were able to provide me with some great ideas that I could work through, too. Thank you.

My friends and teammates on the lightweight rowing team provided a much-needed break from work constantly. Slimfasts, Saturday morning ergs, jungle rows, Friday heat-training gave me all the motivation I needed to get this done. To Trevor, thank you for knowing when to give me the space I needed to take care of what I needed to outside of the boathouse (for this project and more), and for knowing when to do the opposite instead and take me aside after practice to talk through it.

To all my friends, thank you for sticking with me this whole time. This project unfortunately may conclude the brick-v-bot debate in my case, but I'll always be grateful for the ability to sit with you all and let my academic stress fade away. Ryan, Charlie, Sam, Mihir, Lina, Eliza, Ellie, Danni, Spencer, Tea, Allie, Adam, Ben, Christian, Alan, Elliot, Kent, Michael—thank you.

My siblings have paved the way for me and I'm thankful for the examples they have provided me. Zach, I don't think I would be where I am if I didn't have you to look up to. Thank you for all the (solicited) feedback and advice. Cate, I can only wish that I fail this project so I could stay another year with you up here in Hanover. Thanks for dragging me into Feldberg to work with you. Sabra, I'm sorry I never respond to your incredibly supportive texts and check-ins, but thank you so much for continuing to send them anyways—they always helped.

I'll always be thankful to my parents for supporting me and getting me where I am, but especially these last few months. Mom, thank you for driving up to Hanover to sit with me as I tried to organize my thoughts into an outlne for this paper. Dad, thank you for picking up the late-night phone calls to talk me through my anxieties so often. I'm so grateful for you two and love you both.

Contents

1 Introduction **1**

2 Background **3**
- 2.1 Caches 3
- 2.2 Pipelined CPUs 4
- 2.3 Microarchitectural Side-Channel Attacks 5
- 2.4 The gem5 Simulator 7
- 2.5 Related Work 8

3 Methodology **11**
- 3.1 gem5 Simulation Setup 11
- 3.2 Spectre Attack & Benign Program Implementations 13
- 3.3 Data Visualization 13
- 3.4 Detection Model 15

4 Results **19**
- 4.1 Analysis of Event Patterns 19
- 4.2 Detection Model Results 22

5 Conclusions **25**

A Simulation Graphs **27**

B Project Software Repository **44**

List of Tables

3.1 Example Simulation Output 12
3.2 List of Programs Executed in gem5 14

4.1 Detection Model Results 23
4.2 Detection Model Comparison to Related Work 24

List of Figures

3.1 Project Methodology Flowchart 11
3.2 Spectre v1 Event Patterns 15
3.3 Spectre v1 Cache Miss Interval Histogram 16
3.4 Spectre v1 Rolling-Window Cache Miss Interval Histogram 16
3.5 Spectre Detection Tool Model Architecture 17
3.6 Model Feature Engineering Diagram 17
3.7 Tensorboard model accuracy results 18

4.1 Spectre v1 (onefile) Event Signals 20
4.2 Spectre v1 (onefile) Cache Miss Interval Histogram 20
4.3 freqmine (benign) Event Signals 21
4.4 freqmine (benign) Cache Miss Interval Histogram 21
4.5 Spectre v1 (onefile) Transient Cache Miss Interval Histogram 22
4.6 freqmine (benign) Transient Cache Miss Interval Histogram 23

A.1 blackscholes program simulation results 28
A.2 bst_work program simulation results 29
A.3 btb_sa_ip program simulation results 30
A.4 freqmine program simulation results 31
A.5 monte_carlo program simulation results 32
A.6 normal_workload program simulation results 33
A.7 safeside_spectre_v1_btb_ca program simulation results 34
A.8 safeside_spectre_v1_btb_sa program simulation results 35
A.9 safeside_spectre_v1_pht_sa program simulation results 36
A.10 safeside_spectre_v4 program simulation results 37
A.11 spectre_onefile program simulation results 38
A.12 spectrev2poc program simulation results 39
A.13 string_work program simulation results 40
A.14 swaptions program simulation results 41
A.15 transientfail_btb_sa_ip program simulation results 42
A.16 vector_math program simulation results 43

Chapter 1

Introduction

Computer hardware design can be seen as a balance between optimizing efficiency and maintaining security. Advancements in technology often tip that balance towards increased efficiency, while security works to catch up. As the amount of data spread across computing systems has increased drastically over the last several decades, data privacy has become significantly more important. The use of shared computing resources and execution of foreign processes on local devices mean that programs from bad actors could be running on the same computing system as private data. These malicious programs can leverage microarchitectural vulnerabilities in modern CPUs to obtain private data. These vulnerabilities are direct result of advancements in computing efficiency, and are unintended consequences of the race to make computers increasingly faster.

The innovation that computer engineers made from single-cycle CPUs to pipelined CPUs helped increase the rate at which processors execute instructions, but the introduction to transient execution in pipelined processors opens them up to vulnerabilities in their microarchitectural states. Spectre is one such example of a microarchitectural attack in modern CPUs.

Spectre leverages pipelined, out-of-order (O3) processors to access secrets in memory without leaving any architectural trace [1]. Because Spectre uses pipelined processors de-

signed in hardware to optimize instruction throughput, mitigation of the attack must not compromise processor performance. Simply disabling the pipeline and regressing back to single-cycle execution is not an option, as processors rely on pipelining and out-of-order execution to maintain high rates of instruction execution.

Furthermore, because the vulnerability that allows the attack to happen is an inherent part of the hardware, millions of processors that are already produced are susceptible to this attack, and can only be fixed with software patches. While these patches can help mitigate attacks on current processors, the underlying vulnerability still exists unless changes are implemented to processor designs.

This project examines the underlying cache access behavior of these attacks using the gem5 CPU simulator and uses machine learning to develop a real-time Spectre detection tool. Because of the limitations of gem5 to simulate processor behavior compared to actual hardware, this project serves as a proof of concept for this detection tool, with an ultimate goal of hardware implementation.

Chapter 2

Background

2.1 Caches

Spectre attacks leverage vulnerabilities in the microarchitectural state of a processor, which often tends to be the cache state. Generally, caches exist to serve memory requests from the CPU faster than main memory would be able to. By storing a copy of a subset of memory in the cache state, processors can achieve faster memory access times when trying to access data from that subset of memory. However, when the CPU requests memory that does not exist in the cache, it is fetched from main memory and stored in the cache, changing the cache state to reflect that memory access for the processor. These cache misses incur a time penalty as they fetch from main memory, but ultimately result in faster access times if the data is requested again, which is often the case in normal program flow.

By timing the access times for each cache line, a user can determine which cache lines have been accessed recently. This is a side-channel attack called `Flush+Reload` [2]. `Flush+Reload` can let a program figure out if certain data has been recently accessed in memory by entirely flushing the cache before another process tries to request data. When that process makes a memory access, it will result in a cache miss for that address, which can then be learned by the attacker process by timing cache accesses afterwards. If a cache

access happens quickly, then the request was performed by the victim process. If the cache access happens slowly, then the victim process did not try to access the data.

Different methods exist for flushing the cache. X86 processors implement a user-level instruction called `CFLUSH`, but other ISAs may have to flush the cache with user-level processes by using an eviction set. A complete eviction set is a set of addresses that can entirely fill the cache. Methods exist to find these eviction sets [3], which would let a program flush the cache without use of a `CLFUSH` instruction in order to execute a `Flush+Reload` attack. There are also mitigations against eviction-based attacks, including novel cache replacement algorithms that slow down the eviction process [4].

2.2 Pipelined CPUs

Sequential execution of machine code is a functional abstraction up to which modern CPUs do not actually match. In reality, most processors execute instructions using out-of-order pipelines that optimize efficiency while retaining functional equivalence to a simpler, less efficient in-order implementation. The general steps in the pipeline are (1) fetch, (2) decode, (3) execute, (4) memory request, and (5) commit. By issuing a continuous stream of instructions, the pipeline can minimize idle time and maximize instruction throughput. Of course, some instructions depend on the results of previous execution. The CPU may issue other instructions that can be completed independently to prevent a pipeline stall.

When encountering dependencies in program flow, the CPU makes an educated guess as to what to do to minimize stalling. In the case of conditional jumps, the CPU's Pattern History Table (PHT) will guess whether to jump or not before evaluating the condition. If the actual jump destination is not immediately known, the Branch Target Buffer (BTB) speculates on a jump destination that may or may not be correct. Similarly, when returning from a `call` instruction, the Return Stack Buffer (RSB) contains the return addresses for the N most recent `call` instructions and will provide that return address to predict

the program flow. Even data dependencies such as Store To Load (STL) may also be predicted—CPUs have a memory disambiguator that may speculate which loads can be executed before evaluating the data dependency.

Generally, if these speculative components guess correctly, then the CPU continues executing, but if it guesses incorrectly, then the CPU has to "squash" all of the instructions in the pipeline. Executing instructions that later get squashed is called "transient execution." In order for out-of-order CPUs to remain functionally equivalent to their in-order counterparts, this rollback cannot affect the architectural state, so the program counter and registers must all be reverted back to the correct branch flow. However, the microarchitectural state can change as a result of mispredicted execution. If a cache miss occurs during transient execution and the transient execution window is long enough, the request can be serviced by the cache before the CPU squashes the mispredicted branch and reverts back to the correct one.

2.3 Microarchitectural Side-Channel Attacks

The cache state being affected by missed memory requests and pipelined processors' ability to transiently execute instructions open up modern CPUs to transient execution attacks. One such variant is called Spectre—a transient-execution attack that affects the CPU microarchitecture to allow data to leak through side-channels [1]. The other main type of transient execution attack is Meltdown, which leverages exception handling in out-of-order processors to modify the microarchitectural state to access kernel data from user processes [5].

There are various versions of Spectre that exist and are organized into a taxonomy by Canella et al. based on the microarchitectural components used in the attack as well as prediction mistraining strategies [6]. One example is `Spectre V1`, or `Spectre PHT`, that mistrains the Pattern History Table into transiently executing an illegal memory ac-

cess to uncover secret data. By executing an illegal memory access transiently, the security policies never get raised because the instruction is squashed and never gets committed. However, that access can still affect the microarchitectural cache state. By using transient execution in conjunction with `Flush+Reload`, the secret data can be uncovered by timing cache line access to see what data was accessed. Consider the following code example that Kocher et al. provide [1]:

```
void victim_function(int x) {
    if (x < array1_size)
        y = array2[array1[x] * 4096];
}
```

This so-called victim function takes an integer index into `array1`, which it checks to make sure is in-bounds before using its indexed value to access `array2`, offset by a certain size. This is a reasonable piece of code to find in an everyday library, and even implements a safety check to be sure that the only values of `x` that will work are within `array1_size`. However, if the `x < array1_size` condition is speculated upon, then the `array1` access could happen even with a value of `x` that has not passed the bounds check. If some desired byte lived at `SECRET_ADDRESS`, then by passing in `x = SECRET_ADDRESS - array1` an attacker could access the secret byte under `array1[x]`. The security check is bypassed because the array access happens transiently, but there is no way to recover the data from normal channels because once the branch condition is evaluated the transiently executed instructions will be squashed in the pipeline and nothing will get committed to the architectural state. The result of the `array1` access gets used to index into `array2`, though, which could be used to encode the secret byte into the cache state. If the cache were flushed before this array access happened, then trying to access `array2[SECRET_BYTE * 4096]` would result in a cache miss and the line corresponding to that address would get brought into the cache. This specific line

could then be measured using `Flush+Reload`, which allows us to recover the secrete byte from encoding it into the cache state.

While this example exploits the PHT, other Spectre variants and implementations leverage other microarchitectural components to the same effect. Ultimately the attack can be split into three steps: (1) setup, (2) transient execution, and (3) data recovery. In the setup phase, the targeted microarchitectural components are conditioned to prepare for the attack. This could involve mistraining the branch predictor, flushing the cache, or overflowing the Branch Target Buffer. In the transient execution phase, instructions that would not pass software-defined security policies get executed transiently, which can affect the microarchitectural state even after the instructions get squashed and the CPU begins go execute from the correct program flow. This might look like accessing an array using an out-of-bounds index as seen in the above Spectre PHT example. In the data recovery phase, whatever changes that have been made to the microarchitectural state are measured, as a successful Spectre attack will have encoded the secret data into the microarchitectural state to be recovered using side-channel techniques such as `Flush+Reload`.

The three steps for a Spectre attack result in reading one secret byte, so in order to read a string of bytes the attack must be run multiple times. Furthermore, the data recovery step is not always accurate due to noise in the microarchitectural state encodings from the CPU juggling other processes. To account for this, Spectre implementations run several attack attempts for each desired secret byte to statistically determine the result. This leads to many individual attack attempts for a successful string of secret data to be read, which increases the window of instructions in which Spectre attacks could be detected.

2.4 The gem5 Simulator

In order to collect the cache access and transient instruction data for programs, a controlled CPU environment is required. The gem5 simulator is a cycle-accurate CPU simulator that

is highly configurable [7], [8]. gem5 supports most ISAs (including X86, ARM, RISC-V, SPARC, and more) and can be configured to any number of model architectures. gem5 models each hardware component on top of an event-driven simulation engine to allow for dynamically configurable systems. The CPU models are designed to be ISA-agnostic, but once specified in a configuration, program binaries can be loaded into the simulator to be executed. The primary focus of gem5 in this project is the out-of-order (O3) CPU model, which runs slowly but provides the pipeline behavior that is comparable to real hardware.

gem5 also provides support for collecting simulation statistics for each runtime as well as debug tracing for various components. Because gem5 is open-source, the debug module can be modified to include custom debug flags to trace out specific events. Examination of the event-driven cache and O3 CPU modules show the sections of gem5 where cache misses and instruction squashing happens, which can be logged to the simulation runtime's tracefile using custom debug flags.

Ayoub and Maurice have shown that Spectre implementations can be run on gem5 with comparable results to running on a native machine [9]. While the simulator does not behave exactly the same as on real hardware, the benefits of running on gem5 for security research are proven. They also show that the gem5 pipeline viewer tool (used to visualize each instruction executing in each stage of the O3 CPU) can be used to identify the branch predictor mistraining as well as the transient cache accesses that result in a successful attack run.

2.5 Related Work

Spectre attacks and transient execution attacks in general are well-studied. The introduction of Spectre to the academic community by Kocher et al. [1] led to many papers describing the attack and its variants [6], as well as detection and mitigation tools. These detection tools can be broken down into two camps: software-based and hardware-based.

For software-based detection, researchers primarily analyze cache events using hardware performance counters (HPCs). Depoix and Altmeyer produced a Spectre detection tool by detecting cache side-channel attacks by reading cache access rates from hardware performance counters [10]. They run processes that continuously monitor the relevant hardware performance counters and link them with running processes to determine if any are executing a Spectre attack.

Choudhari et al. produced SpecDefender, which includes branch prediction counters in its HPC feature set along with cache events to detect Spectre [11]. Once Spectre is detected, they demonstrate dynamically instrumenting the binary to linearize the program and eliminate conditional jumps to mitigate Spectre.

Software-based detection tools are a useful field of study because they can be run on the many processors that currently exist with transient execution attack vulnerabilities. However, the underlying problem still exists in the hardware and does not address future processor design. Furthermore, software-based tools require processor time to run and impact processor performance.

Hardware-based tools have used instruction flow to detect Spectre attacks. Zhang and Makris employ an instruction-sequence windowed approach to encoding instruction features into a RNN model in hardware to detect Spectre [12]. They utilize LSTMs to model the sequential data for detection.

McKevitt et al. present SpecCheck, which uses a register-based finite state machine (FSM) as a detection tool to find vulnerable code windows in a program [13]. The input features into this FSM are the program instructions themselves, as SpecCheck considers how instructions, specifically transiently-executed ones, can put the processor microarchitecture in a vulnerable state.

Software-based Spectre detections generally use HPCs related to cache events and branch mispredict events as features in their models, while hardware-based tools utilize program instructions as features. The obvious benefit to hardware-based tools is that they

do not take up processing power from the CPU and instead can run on their own hardware components in parallel, but they would only be implemented on new processors. Furthermore, hardware-based tools can run on instruction-level granularity, whereas HPCs can only be read by software-based detection processes around every 100ms.

This leaves an avenue of work to see if a cache event based detection tool could be implemented in hardware for future processors to detect and mitigate Spectre attacks without much performance overhead. By shifting the processing of cache-based event patterns to hardware components, the processor can remain available for executing user programs rather than securing vulnerabilities. Additionally, the cache and branch mispredict event signals as a detection featureset could result in lower implementation costs than uniquely encoding each instruction.

Chapter 3

Methodology

An overview of the project methodology can be found in Figure 3.1. The gem5 simulator was used to simulate a CPU with cache misses and transient instructions logged to a trace file, which was then scraped using a Python script and used in a MATLAB visualizer and used to train and test a Mamba detection model.

3.1 gem5 Simulation Setup

Both Spectre attack implementations and benign benchmark programs were simulated in gem5, which was modified to include a custom debug flag that traced the unique sequence number of any instruction that resulted in a cache miss or was executed transiently during the simulation runtime. These debug flags were inserted in the cache modules of the memory system and the out-of-order CPU module in gem5 (running an X86 O3 CPU with

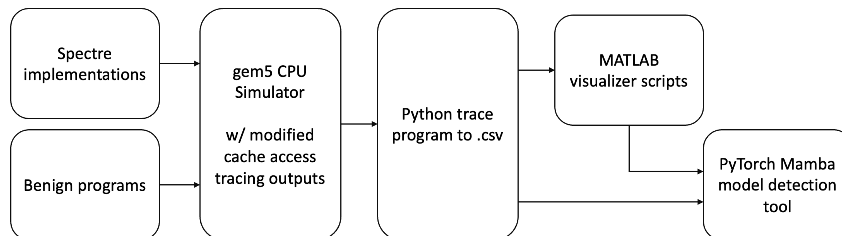


Figure 3.1: Project methodology flowchart

8192MB of memory and an L1 and L2 cache using an LTAGE branch predictor). Because memory requests can be serviced before a branch is squashed, these traces would occur out of order. A Python script was then written to scrape the debug trace file and output a .csv file with a row for every instruction sequence number that resulted in a cache miss or a squashed instruction, and columns to indicate which event occurred (or if both occurred). An example of this output can be seen in Table 3.1. The instruction sequence number can be thought of as a time variable as the program progresses, although obviously the rate at which instructions are executed is not constant with time. However, for the purposes of this model, performing this analysis in the instruction-sequence domain rather than the time-domain is perfectly appropriate. These .csv files were then fed into MATLAB scripts to visualize the cache access and branch prediction patterns as well as imported into Python to use in a Mamba model for a binary classification detection tool.

Table 3.1: Example simulation output from Python tracer. Non-sequence number columns are indicator variables, with label=1 for Spectre attack and 0 for benign program

Sequence Number	Cache Miss	Transient Instruction	Transient Cache Miss	Label
1	0	0	0	0
2	0	0	0	0
3	1	0	0	0
4	0	1	0	0
5	1	0	0	0
...	...	...	...	...
23044432	1	1	1	1
23044433	0	1	0	1
23044434	0	0	0	1
23044435	1	0	0	1

3.2 Spectre Attack & Benign Program Implementations

Spectre attack implementations were collected from various sources. Between Google's Safeside project,[1] Canella et al.'s transientfail project,[2] Erik August's Spectre V1 demo,[3] and Anton Cao's Spectre V2 demo,[4] there were 8 implementations of Spectre variants that were successfully run on gem5 after running on a real X86 machine. To model a detection tool, a control group of benign programs was also required. Sample programs from the PARSEC benchmark suite [14] as well as programs given by a generative AI model[5] were used to control for benign cache access patterns in simple enough processes to run on gem5. 16 programs were run while collecting cache access data, processed into instruction sequence data, and had four (two Spectre, two benign) set aside for testing and the rest for training. These programs are described in Table 3.2.

3.3 Data Visualization

A MATLAB program was written to take the processed instructions sequence data and visualize cache access patterns between Spectre and benign program execution. By visualizing the cache miss events and transient execution events, an intuition for Spectre implementations compared to benign programs could be developed to help inform design choices for the detection tool. The simulated programs execute on the magnitude of 10^6 and 10^7 instructions, so these event indicators plotted over time (e.g. Figure 3.2) show general behavior with low resolution.

In order to understand more about the frequency at which these events occur, the in-

[1] github.com/google/safeside

[2] transient.fail

[3] gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6

[4] github.com/Anton-Cao/spectrev2-poc

[5] An example prompt plugged into OpenAI's o4 Mini after describing project context: "In order to test the detection tool I need a control group of programs that executes everyday mundane code. I have tried CPU benchmark programs but they tend to run one algorithm over and over again. Could you please generate a program in C that can compile into one binary that demonstrates normal execution behavior?"

Table 3.2: Each program used in project, including Spectre variant if Spectre or benign if not, as well as source from GitHub projects, PARSEC, or ChatGPT[5]

Program name	Spectre variant (or benign)	Source	Training or testing
blackscholes	benign	PARSEC [14]	train
bst_work	benign	ChatGPT o4 Mini	train
BTB_sa_ip	v2 (BTB)	transientfail3.2	train
freqmine	benign	PARSEC	train
monte_carlo	benign	ChatGPT o4 Mini	test
normal_workload	benign	ChatGPT o4 Mini	test
safeside_spectre_v1_btb_ca	v2 (BTB)	Safeside3.2	test
safeside_spectre_v1_btb_sa	v2 (BTB)	Safeside	train
safeside_spectre_v1_pht_sa	v1 (PHT)	Safeside	train
safeside_spectre_v4	v4 (STL)	Safeside	test
spectre_onefile	v1 (PHT)	adapted from Erik August3.2	train
spectrev2poc	v2 (BTB)	Anton Cao3.2	train
string_work	benign	ChatGPT o4 Mini	train
swaptions	benign	PARSEC	train
transientfail_btb_sa_ip	v2 (BTB)	transientfail	train
vector_math	benign	ChatGPT o4 Mini	train

struction sequence intervals between events were plotted as a histogram to examine the frequency of events (e.g. Figure 3.3). Spectre attacks can be implemented differently and can have different instruction timing between setup, transient execution, and data recovery phases, but the three steps all have to occur for the attack to be successful. Furthermore, each attack attempt happens many times for each Spectre implementation because attackers are often trying to read each byte multiple times from a string of bytes to statistically determine the correct result. Because of this, the intervals between events could show regular patterns that the event-versus-time graphs showed less easily.

Program behavior also changes over time. The Spectre implementations all have a program setup phase in which the attack is not actually occurring, and benign programs go through different sections that are not all the same. The instruction sequence number interval histograms showed the relative frequency of an event occuring in the program, but it does not show how those intervals change over the course of the program's runtime. To

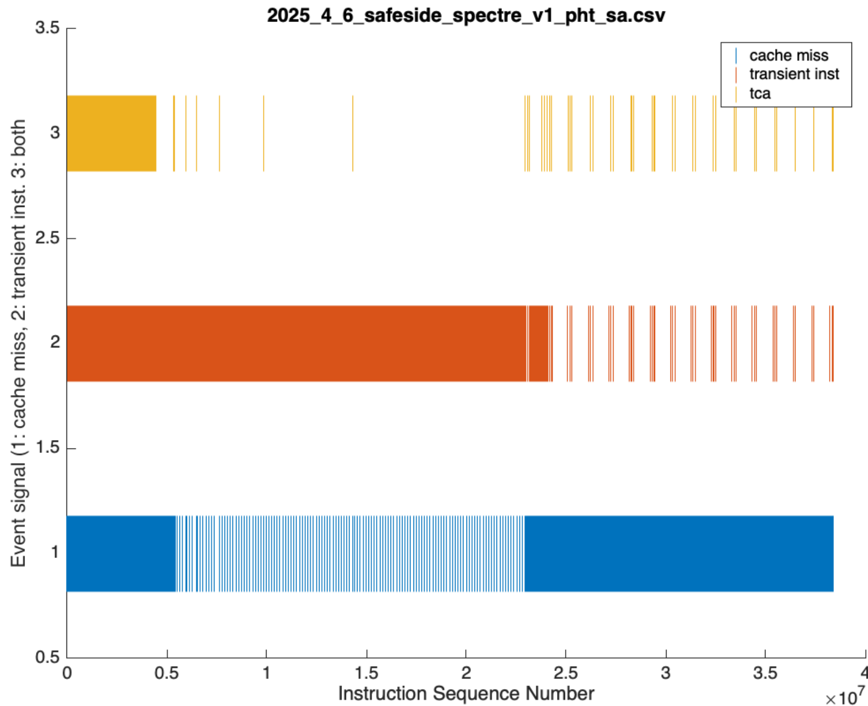


Figure 3.2: Event patterns for Spectre v1 attack showing cache misses, transient instructions, and transient cache misses

account for this, each event was fed into a rolling buffer that stored the interval between the last n events with a relative frequency histogram computed for each window to see the cache access patterns over time. An example of this can be seen in Figure 3.4.

3.4 Detection Model

The detection tool takes streams of sequential input data as the processor runs. Because of the sequential nature of the data, a recurrent neural network (RNN) model was used. The Mamba model was chosen for its performance in modeling time-sequence patterns with efficient training [15]. The Mamba model compared to other RNNs (like LSTMs) generally has faster training times and performs better. The event-driven data were processed in a Python script to zero-fill the tables so that instruction sequence numbers without cache misses or transient execution were represented. These sequences were then labeled benign or malicious for supervised learning in this binary classification task. Using the PyTorch

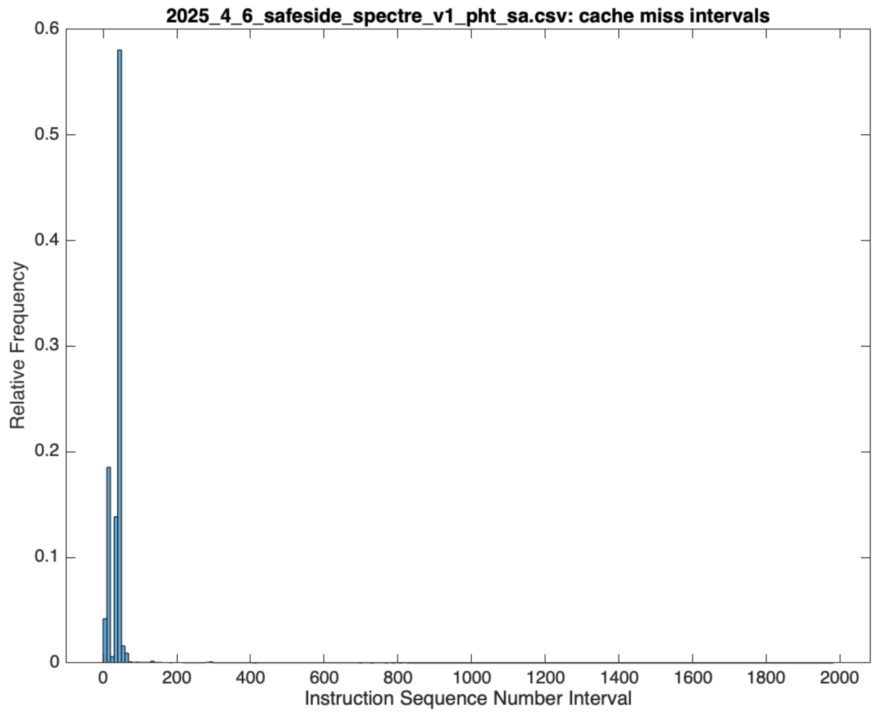


Figure 3.3: Cache miss interval histogram for Spectre v1 attack

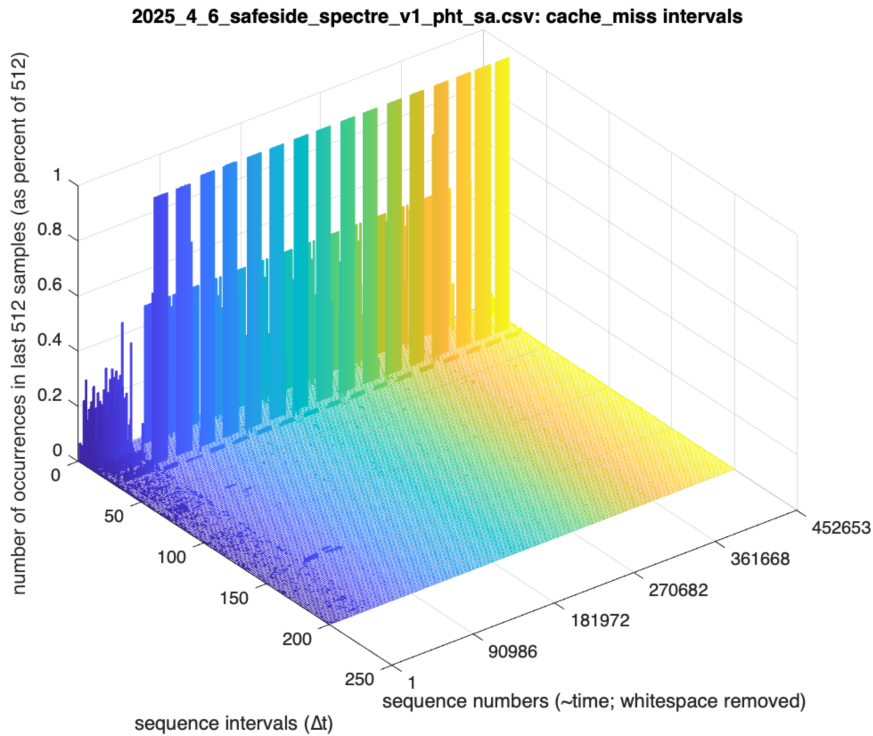


Figure 3.4: Rolling-window cache miss interval histogram for Spectre v1 attack showing cache miss intervals over programs duration

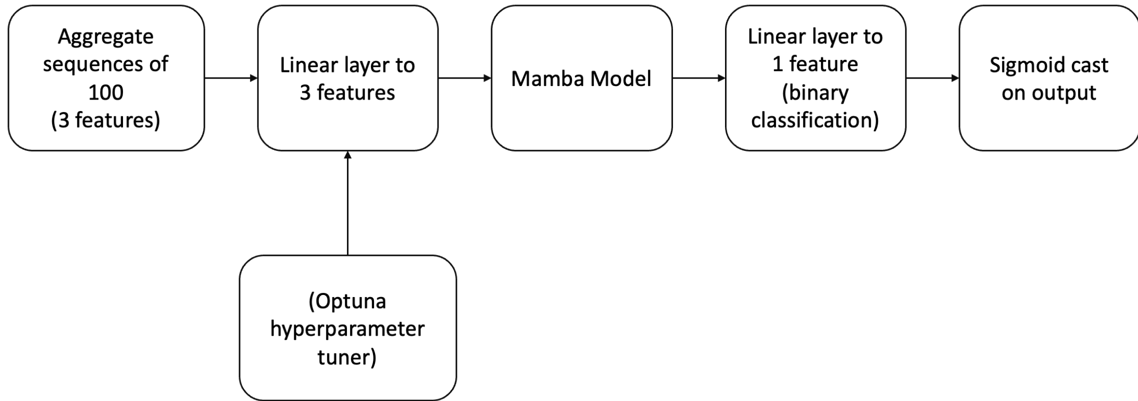


Figure 3.5: Spectre detection tool model architecture

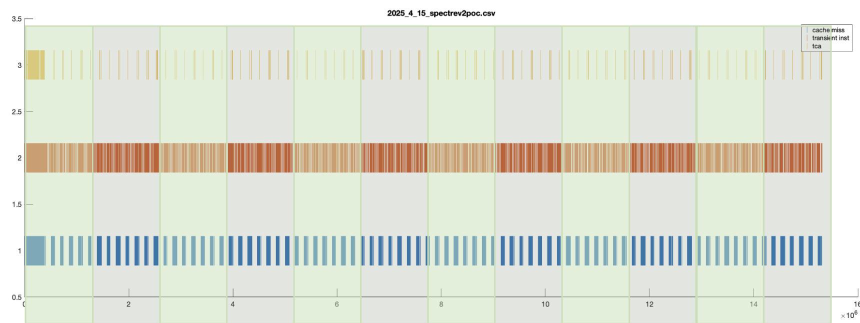


Figure 3.6: Spectre detection model feature engineering showing window-aggregation for each event variable, then normalized by window size

library, these sequences were aggregated and normalized by a certain window size (shown in Figure 3.6) and then fed into a PyTorch Mamba model implementation [16] for training. This model takes in the aggregate events and predicts if a Spectre attack is occurring or not. The model architecture can be found in Figure 3.5.

The 12 simulation programs (6 from Spectre attacks, 6 benign) were used to train the Mamba model while the remaining 4 were used for testing / validation purposes. The aggregate window size and the hidden layer size of the model were the two hyperparameters that were adjusted to find the the best results. PyTorch's lightning module with Tensorboard for rapid model evaluation alongside the Optuna library for hyperparameter tuning was used to find these two values. An aggregate window size of 4096 (shown in Figure 3.7) was found to result in higher accuracy than the 2048-instruction window size that Zhang

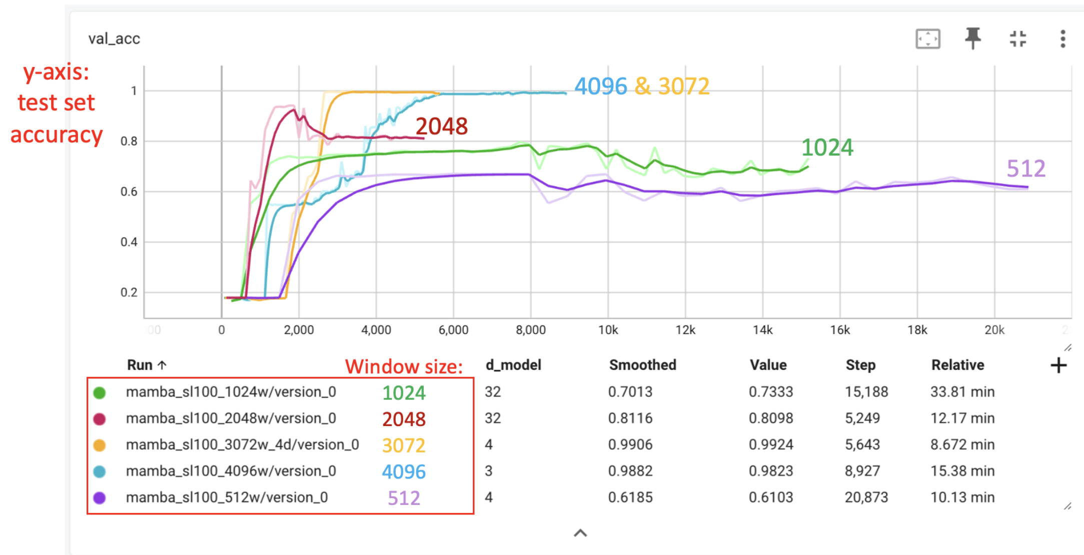


Figure 3.7: Tensorboard screenshot showing test accuracy for Mamba model varying by aggregate window size

and Makris found to be optimal for hardware-based detection of Spectre using instruction sequences as features rather than cache and transient instruction events [12]. A hidden-layer depth of 3 was found to be as accurate compared to higher-degree models, but did not result in any overfitting when trained for longer. In each case, the model was trained until the validation loss function reached a minimum and then rose to diverge from the training loss function values.

Chapter 4

Results

4.1 Analysis of Event Patterns

When examining the graphs produced for each simulation (which can all be found in Appendix A with representative examples provided in the following figures), the differences between Spectre attacks and benign programs are clear. Generally, the Spectre attack implementations follow a much more periodic cache access pattern than the benign programs. This can be seen in Figures 4.1 and 4.2. The cache miss intervals for this Spectre implementation can be attributed to the periodic application of `Flush+Reload` to measure the cache state as well as the constant mistraining of the branch predictor for transient instruction signals. This can be compared to the less regular signals for a benign implementation such as PARSEC's `freqmine` program seen in Figures 4.3 and 4.4. These regular signals are likely attributed to Spectre attacks' tendency to run `Flush+Reload` or other data recovery techniques over and over again to try to measure a string of bytes multiple times. These irregular spikes in cache miss intervals seem to function well as a marker of a Spectre attack for these implementations, but cache misses can occur regularly without any transient execution.

Examining the transient cache miss interval histograms, however, provides less confi-

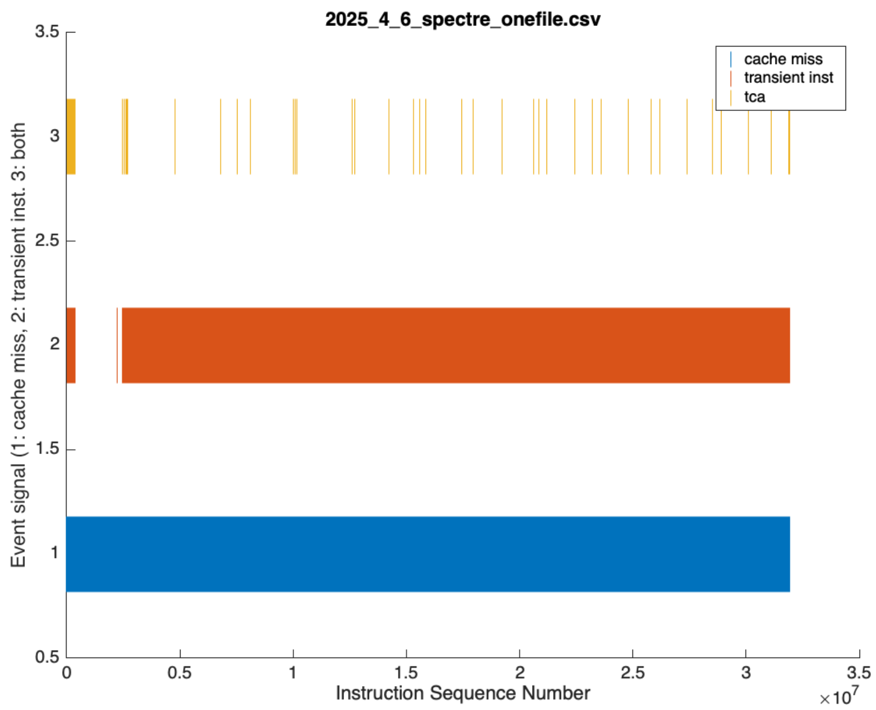


Figure 4.1: Event signals for Spectre v1 implementation

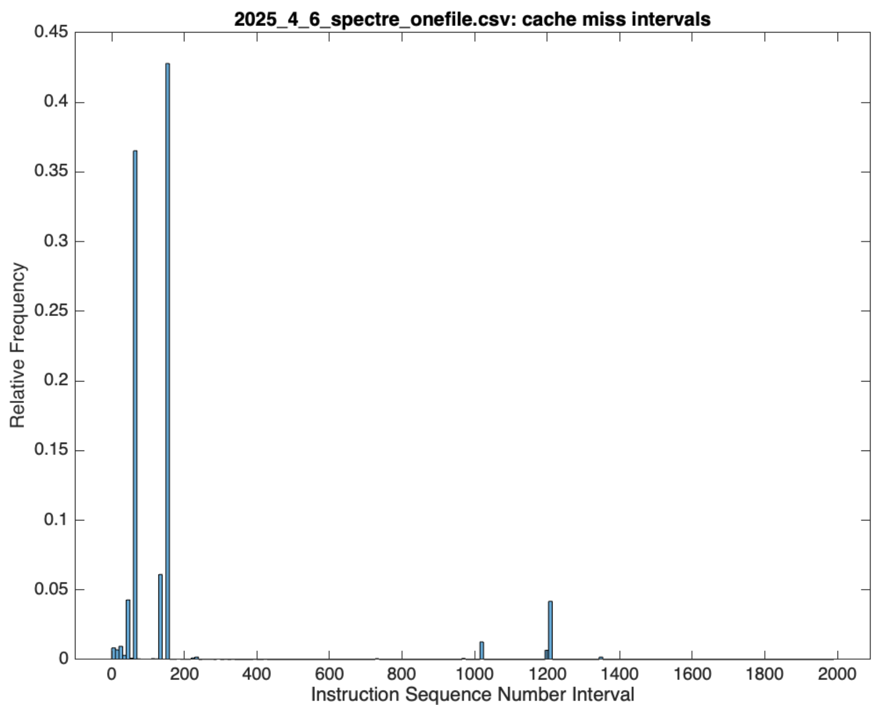


Figure 4.2: Cache miss interval histogram for Spectre v1 implementation

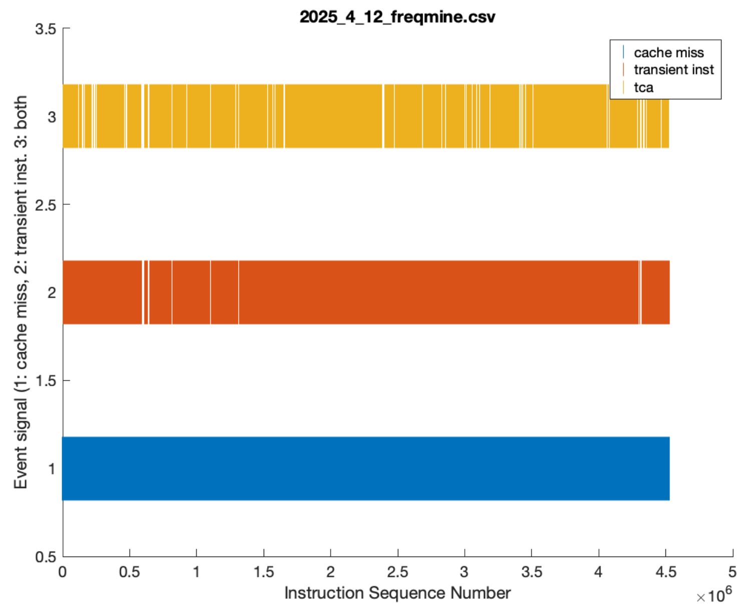


Figure 4.3: PARSEC freqmine benchmark cache miss, transient instruction, and transient cache miss signals

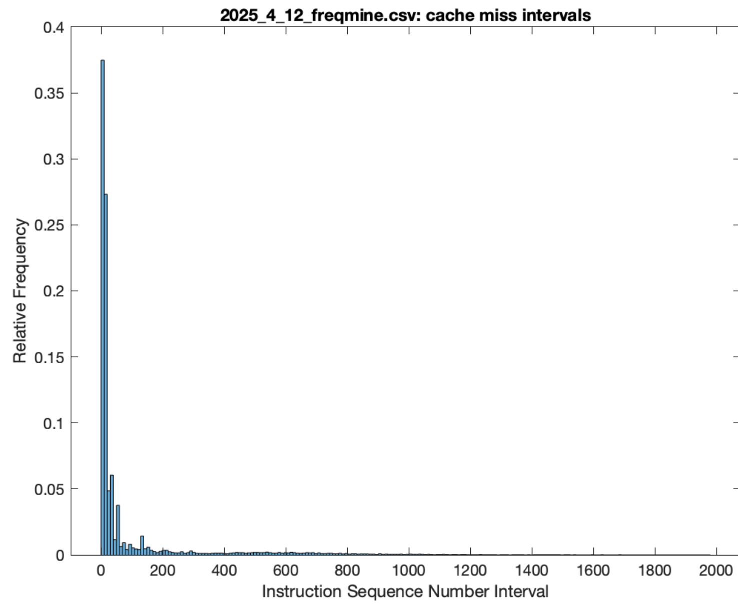


Figure 4.4: PARSEC freqmine benchmark cache miss interval histogram

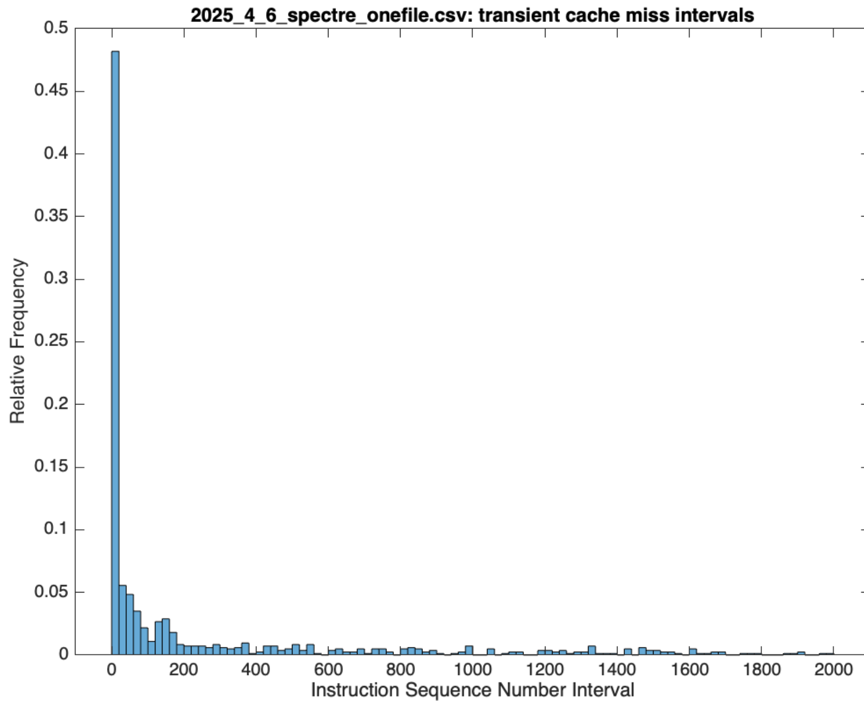


Figure 4.5: Transient cache miss interval histogram for Spectre v1 implementation

dence in its use as a marker for Spectre attacks. Figure 4.5 shows a Spectre implementation's transient cache miss intervals and Figure 4.6 shows a benign program's. While the Spectre implementation's histogram shape rolls off more gently than the benign program's, the clear non-zero interval frequency spikes of the normal cache miss signals are not present when isolated to just transient cache misses. Transient cache misses likely do not occur periodically as often as normal cache misses because they only occur once per Spectre attack attempt. Because the setup phase for each attack attempt involves flushing the cache and then probing its contents, there are many more periodic normal cache misses per Spectre attack attempt than for transient cache misses.

4.2 Detection Model Results

After training the Mamba model using 4096-instruction length window size aggregate features and a 3-deep hidden layer, the model achieved a 99.12% accuracy on the testing data and a 94.91% accuracy on the training data. The confusion matrix for the testing data can

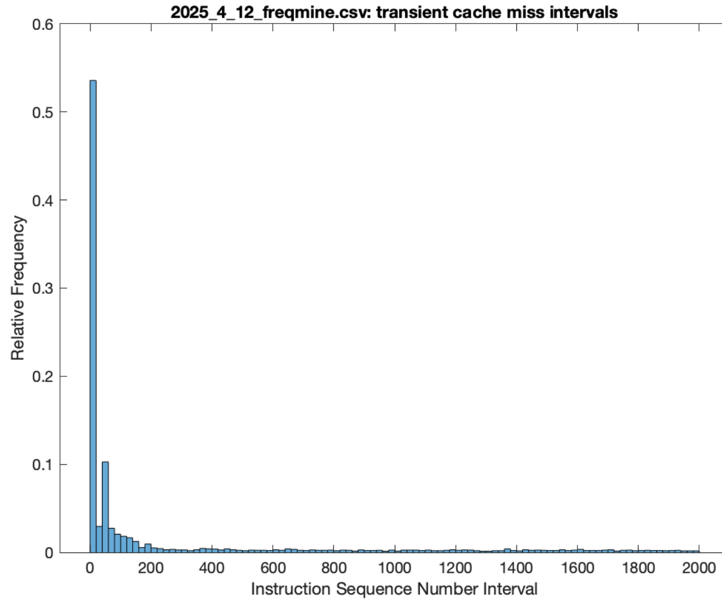


Figure 4.6: PARSEC freqmine benchmark transient cache miss interval histogram

be seen in Table 4.1 and comparisons to similar detection tools can be seen in Table 4.2. After tracing the model predictions for each sequence back to their source programs, it was found that the lower training accuracy is due to the Spectre Safeside BTB (v2) programs used in training were not providing clear results when running the program on gem5. In fact, these simulation runs in general have fewer clear hits using `Flush+Reload` on gem5 compared to when running the implementations on a native X86 machine. This could be due to the cache timing thresholds as well as discrepancies in branch prediction algorithms.

Table 4.1: Confusion matrix results for 4096-window size 3-deep hidden layer Mamba model. (true positive = alarm bells for spectre, false positive = false alarm for spectre, false negative = spectre goes undetected, true negative = no alarm for benign program)

TP: 69	FP: 2
FN: 0	TN: 325

Table 4.2: Reported accuracy for this project compared to related work

Model	Description	Reported Accuracy
SpecCheck [13]	Hardware-based instruction flow FSM	85%
SpecDefender [11]	Software-based HPC model	92.9%
This Project	**Hardware-based transient cache access model**	**99.12%**
Depoix and Altmeyer [10]	Software-based HPC model	99.23%
Zhang and Makris [12]	Hardware-based instruction flow model	99.76%

Chapter 5

Conclusions

The tool developed in this thesis can reliably detect Spectre attacks based on cache access patterns collected from gem5 simulations. One could imagine a component in the processor that collects miss event signals from the cache and mispredict signals from the branch predictor, feeds them into a hardware implementation of the Mamba model, and raises an alarm if a Spectre attack is detected to perform mitigation strategies. There is still work to be done, however, to determine if this component is viable for inclusion on a real processor.

In order to improve model robustness, the dataset could be improved by trying to collect data from less contrived programs than CPU benchmarks and LLM-generated examples. This would require more work to get more complex programs running on gem5. Another option would be to extend the data collection components from a FPGA-based CPU implementation to run programs on actual hardware while being able to collect data and eventually test the detection tool in actual hardware. This would be a clear next step for determining viability before committing to any actual CPU implementations.

Model performance could also be improved by expanding the feature set by breaking into cache-level granularity. The current system does not discriminate between cache levels when considering misses, so expanding the feature set by looking at the cache level hierarchy for each miss signal could feasibly improve the model.

The cache access patterns mostly seem to learn from cache misses rather than the much less frequent transient cache miss signals, so this detection tool could also be examined for application in detecting other side-channel attacks. This avenue also depends on more robust data collection processes, though.

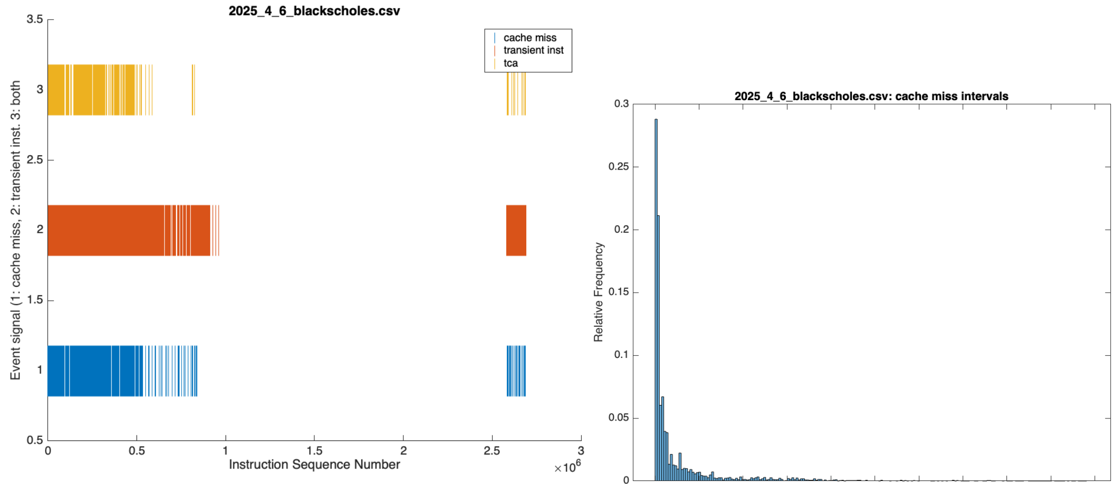
Additonally, the actual Spectre implementations used in this project were all proofs-of-concept. Spectre attacks found in the wild may vary in their `Flush+Reload` timing or try to obfuscate their branch mistraining in a way that is less obvious. This project would benefit greatly from improving the faithfulness of both Spectre attack code and benign code to what is found running on real processors.

Finally, this project only contributes a detection tool. Once Spectre is detected, the processor should take care to mitigate the attack. Mitigation strategies were were out of scope for this project, but future work should explore strategies that effectively balance instruction throughput while minimizing Spectre attack success rates. Accurate detection of Spectre is the first step in mitigating the attack, though, which this project shows is possible.

Appendix A

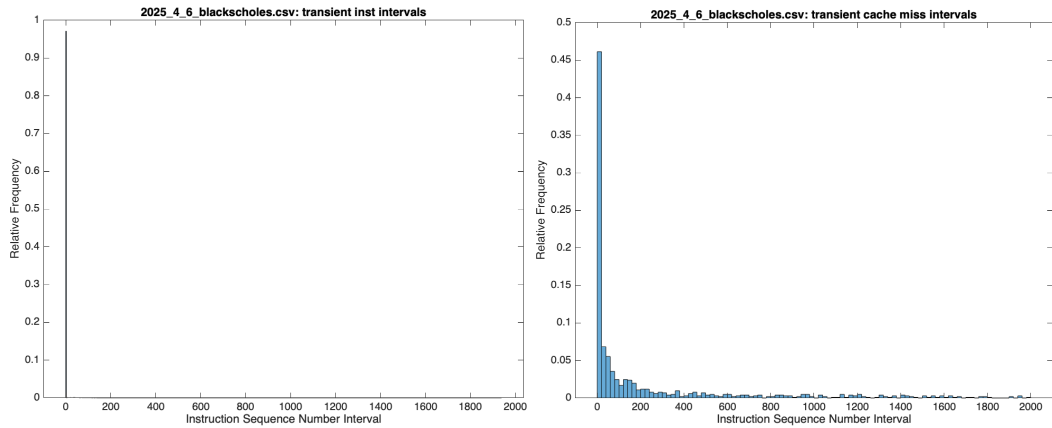
Simulation Graphs

Appendix A shows, for all 16 simulation runs, the overall event vs. instruction sequence number graph of the run (cache misses, transient instructions, and transient cache misses), the histogram of intervals between each of those three events, and the 3D rolling-histogram graph of cache miss intervals.



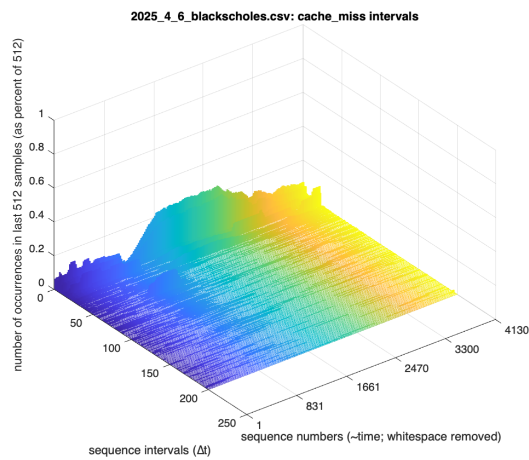
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events

(b) Cache miss interval histogram



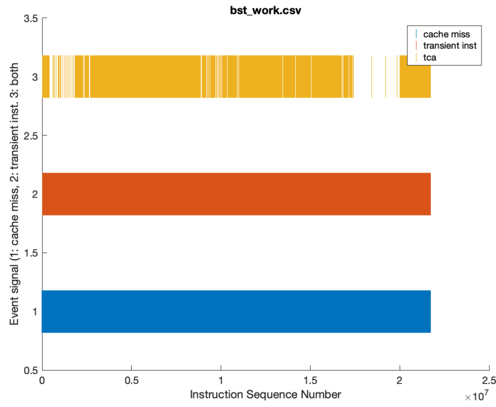
(c) Transient instruction interval histogram

(d) Transient cache miss interval histogram

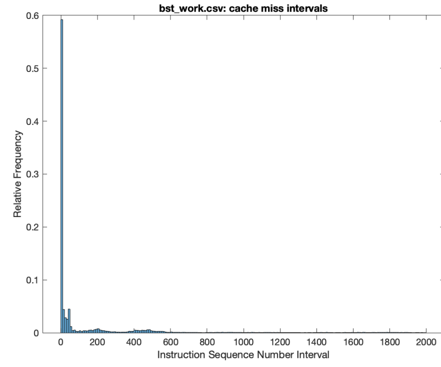


(e) Cache miss interval sliding-window histogram

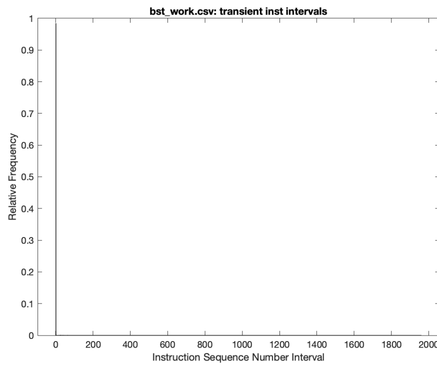
Figure A.1: blackscholes program simulation results



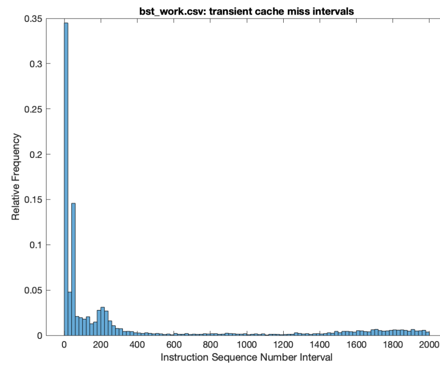
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



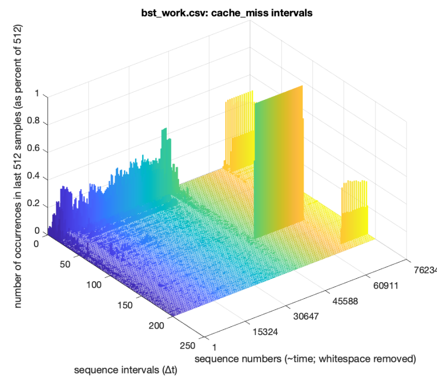
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

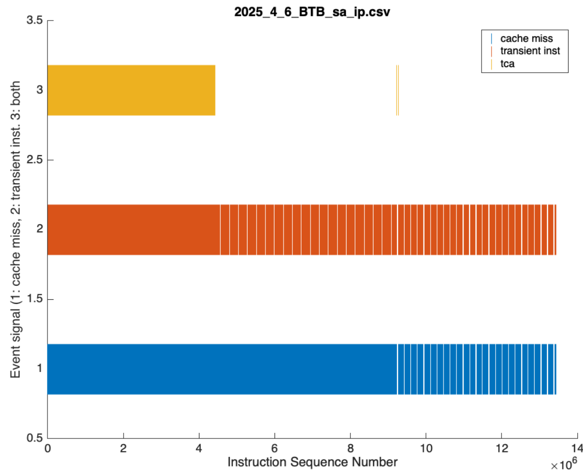


(d) Transient cache miss interval histogram

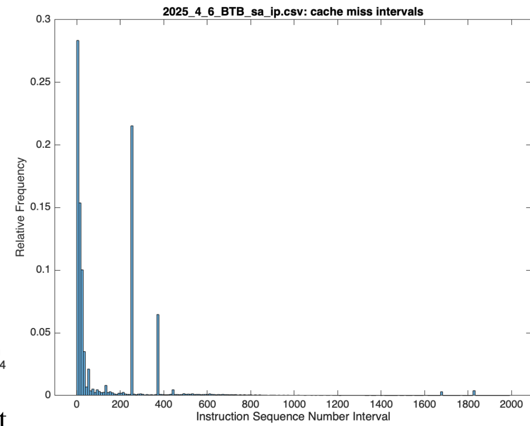


(e) Cache miss interval sliding-window histogram

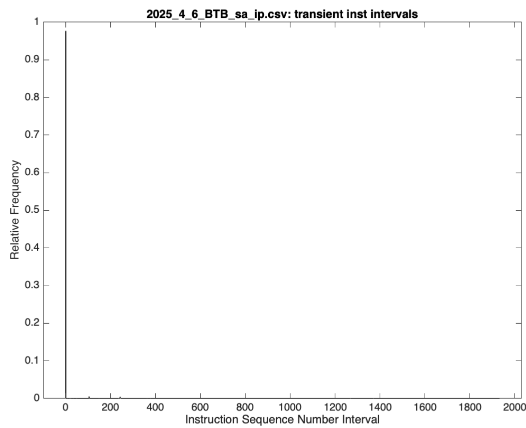
Figure A.2: bst_work program simulation results



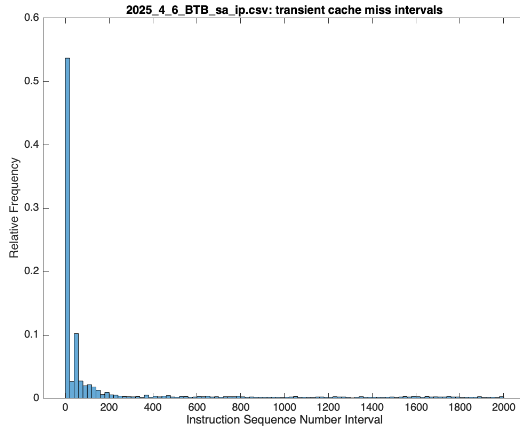
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



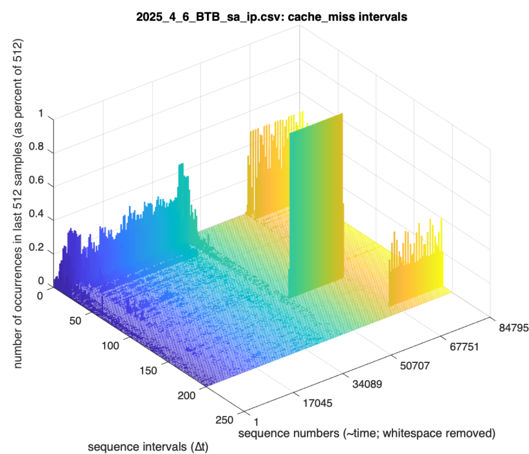
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

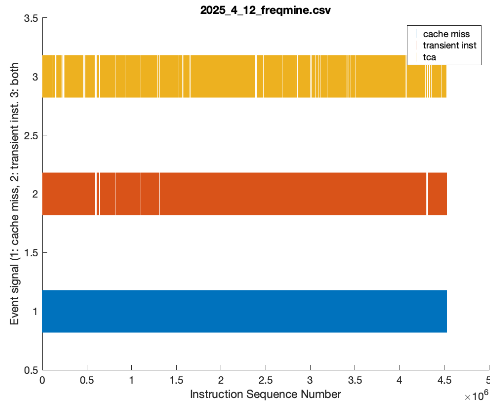


(d) Transient cache miss interval histogram

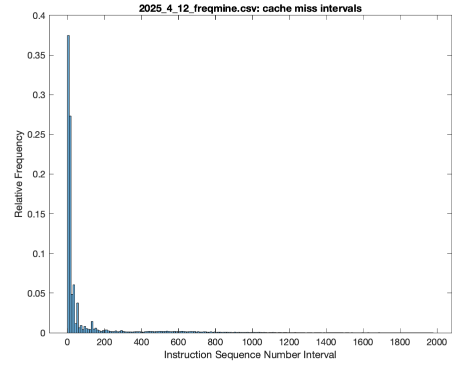


(e) Cache miss interval sliding-window histogram

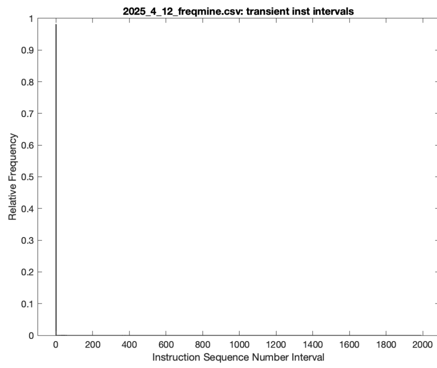
Figure A.3: btb_sa_ip program simulation results



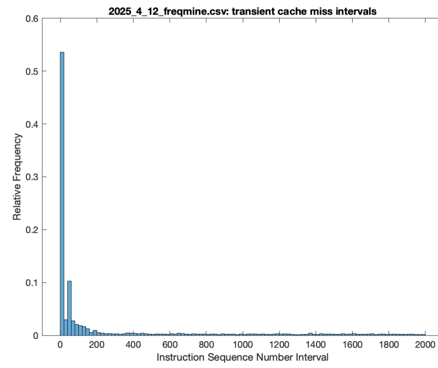
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



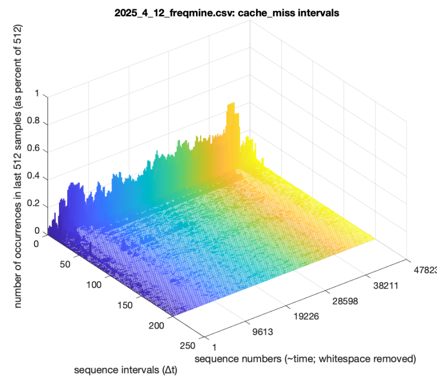
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

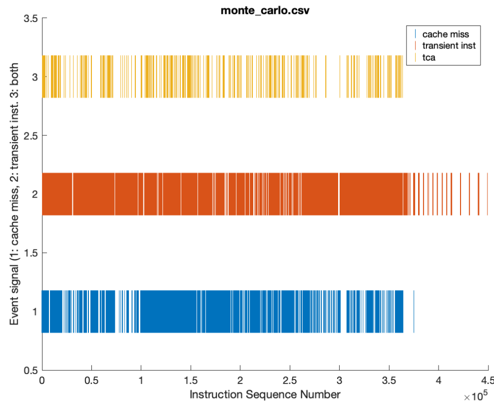


(d) Transient cache miss interval histogram

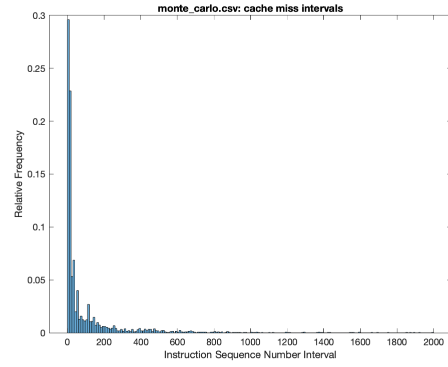


(e) Cache miss interval sliding-window histogram

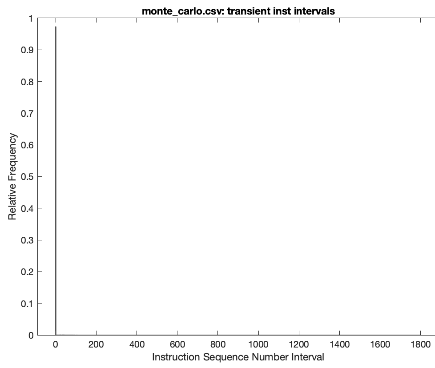
Figure A.4: freqmine program simulation results



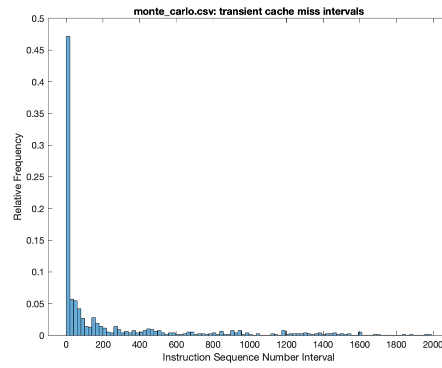
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



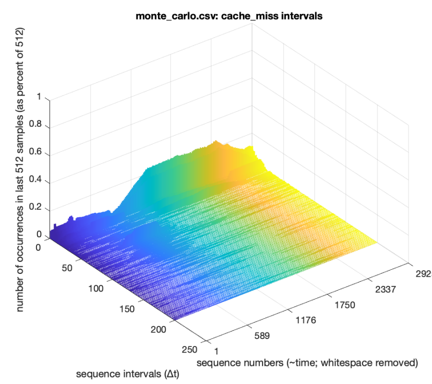
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

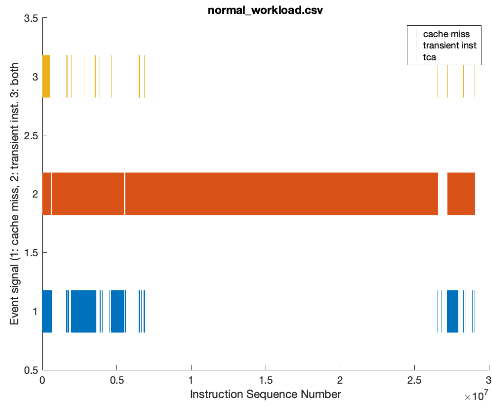


(d) Transient cache miss interval histogram

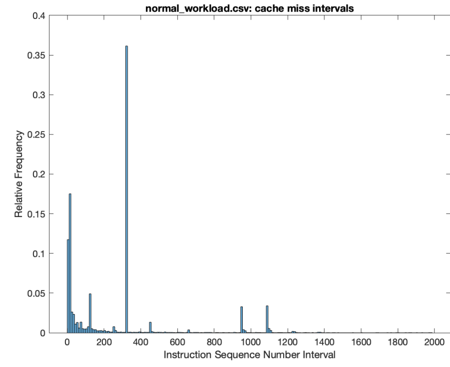


(e) Cache miss interval sliding-window histogram

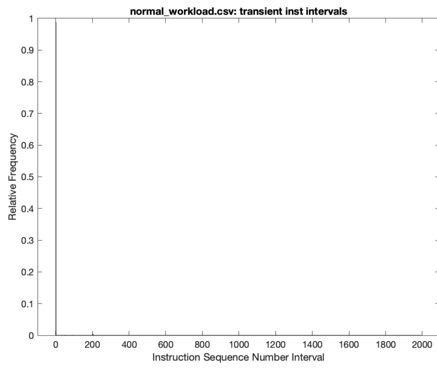
Figure A.5: monte_carlo program simulation results



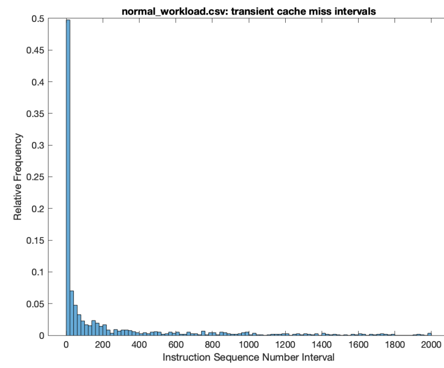
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



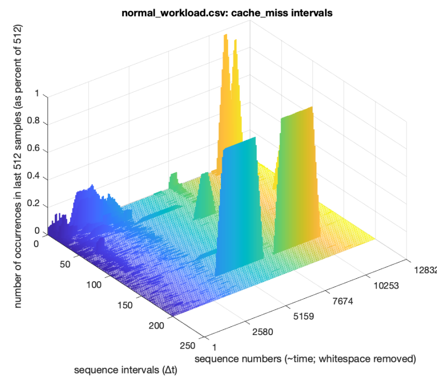
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

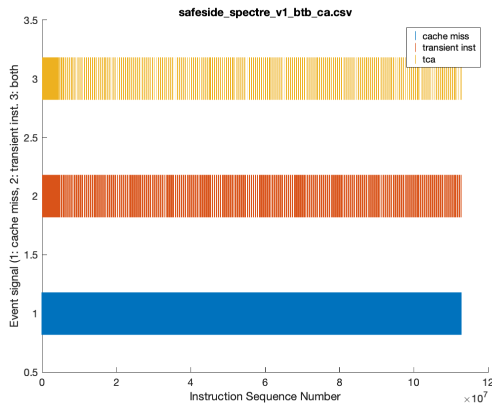


(d) Transient cache miss interval histogram

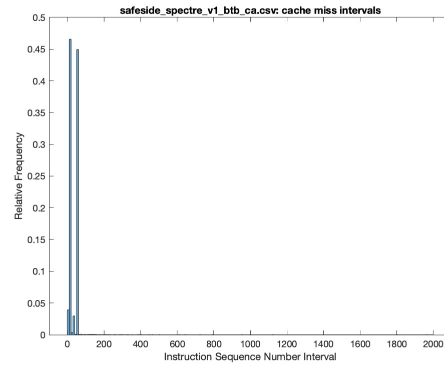


(e) Cache miss interval sliding-window histogram

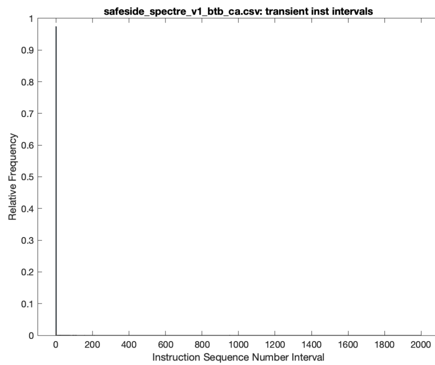
Figure A.6: normal_workload program simulation results



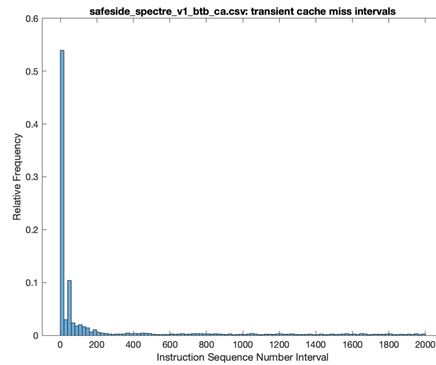
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



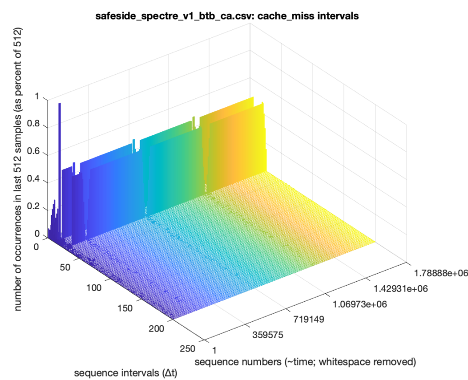
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

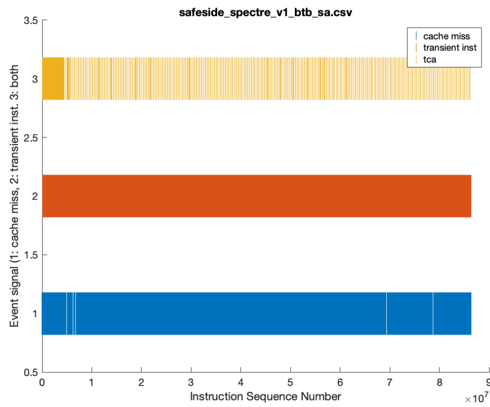


(d) Transient cache miss interval histogram

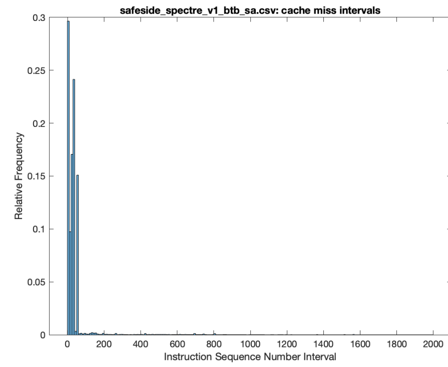


(e) Cache miss interval sliding-window histogram

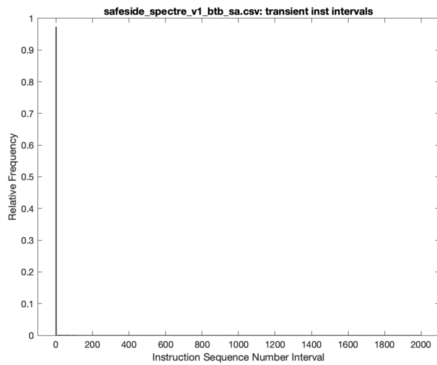
Figure A.7: safeside_spectre_v1_btb_ca program simulation results



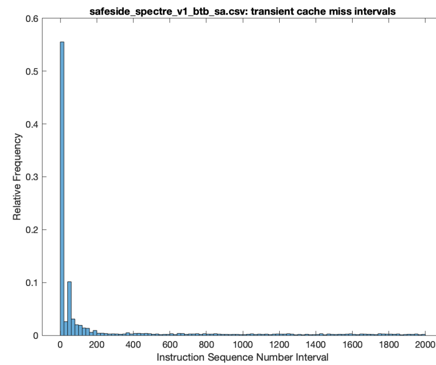
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



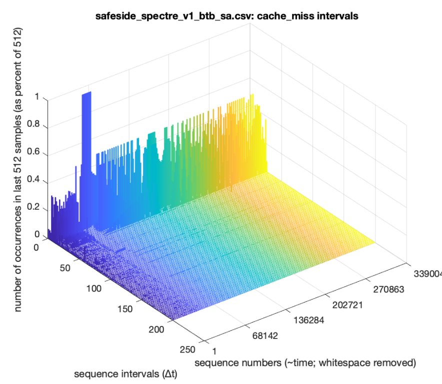
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

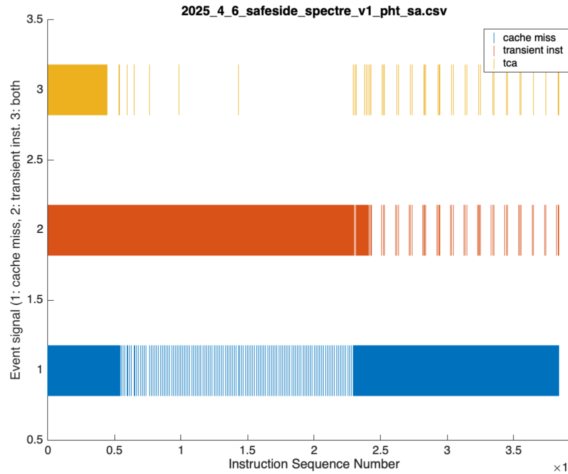


(d) Transient cache miss interval histogram

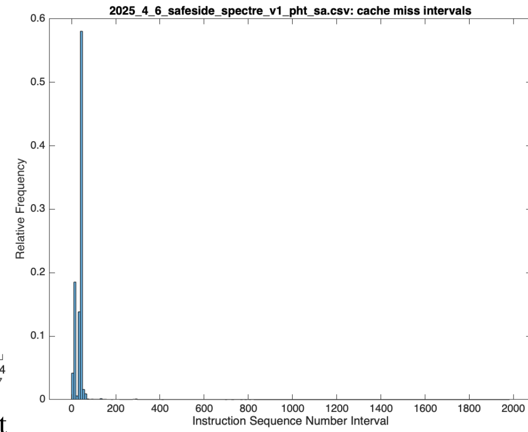


(e) Cache miss interval sliding-window histogram

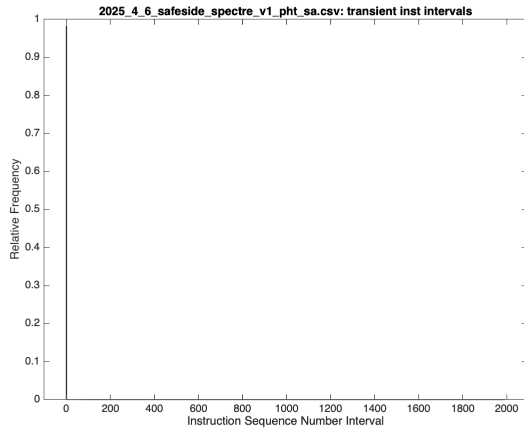
Figure A.8: safeside_spectre_v1_btb_sa program simulation results



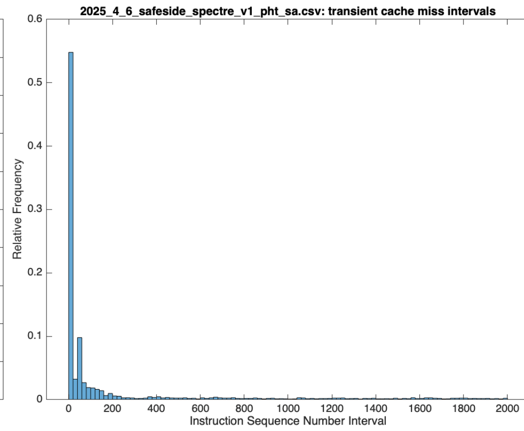
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



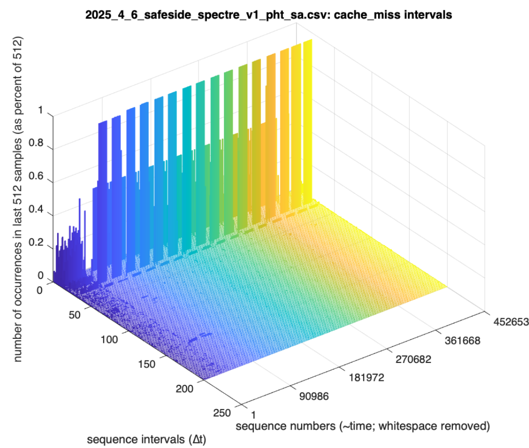
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

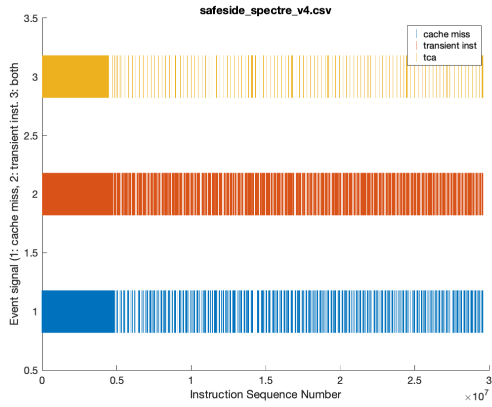


(d) Transient cache miss interval histogram

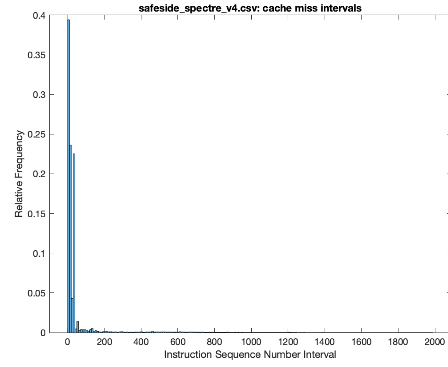


(e) Cache miss interval sliding-window histogram

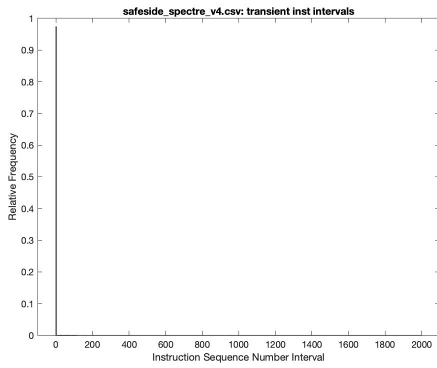
Figure A.9: safeside_spectre_v1_pht_sa program simulation results



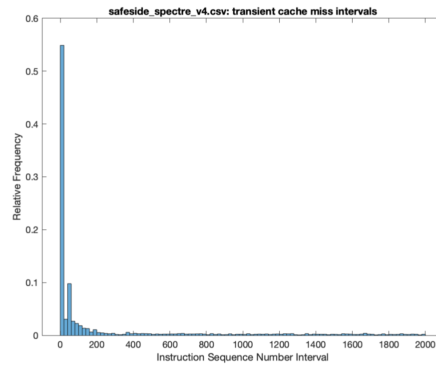
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



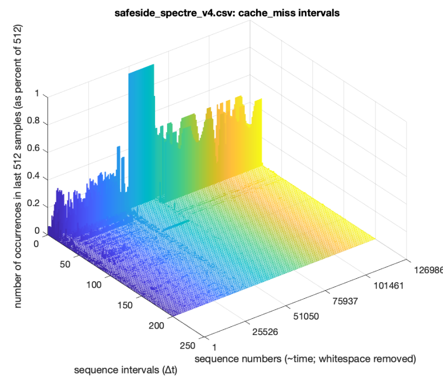
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

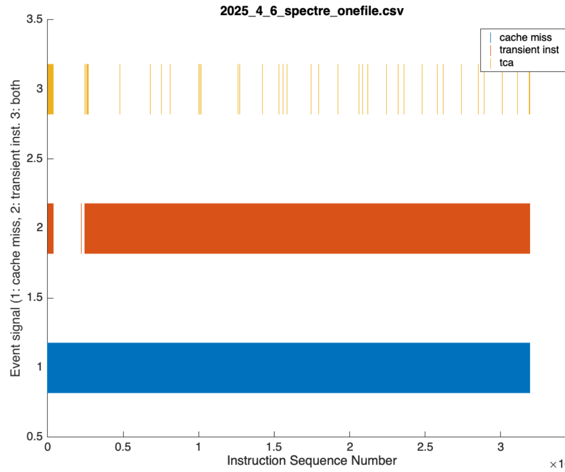


(d) Transient cache miss interval histogram

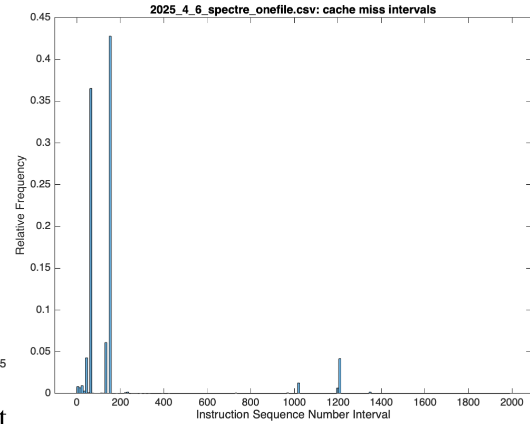


(e) Cache miss interval sliding-window histogram

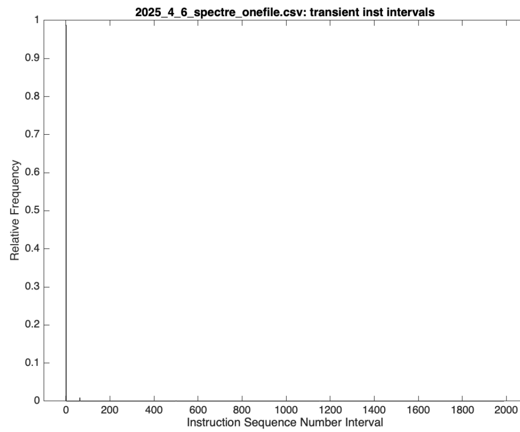
Figure A.10: safeside_spectre_v4 program simulation results



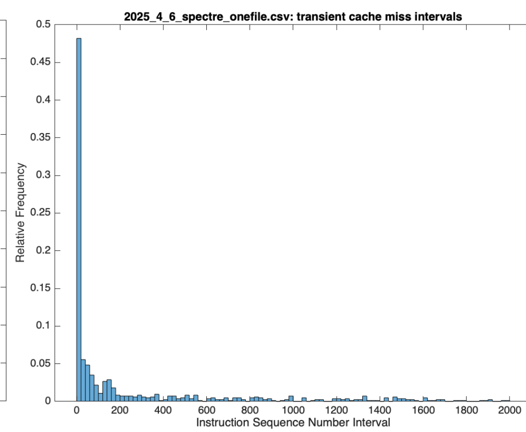
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



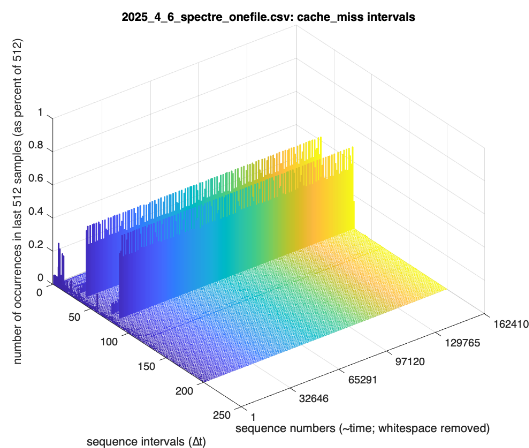
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

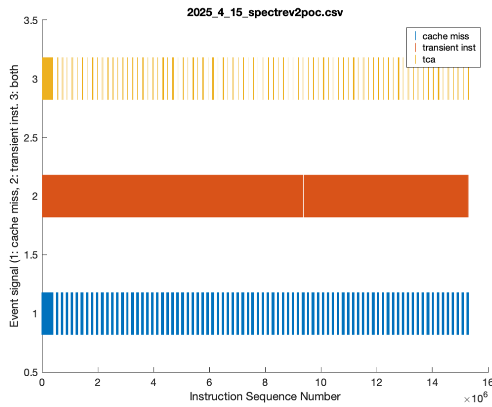


(d) Transient cache miss interval histogram

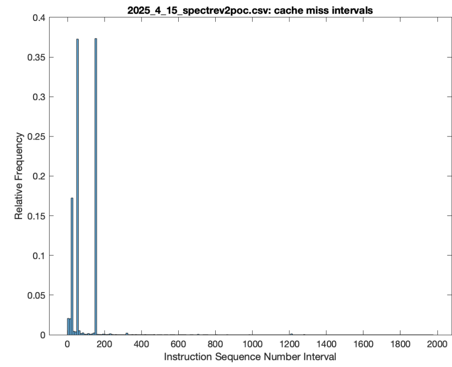


(e) Cache miss interval sliding-window histogram

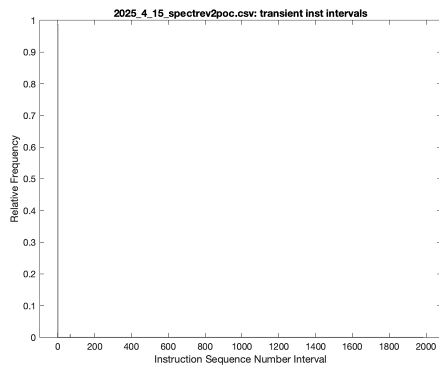
Figure A.11: spectre_onefile program simulation results



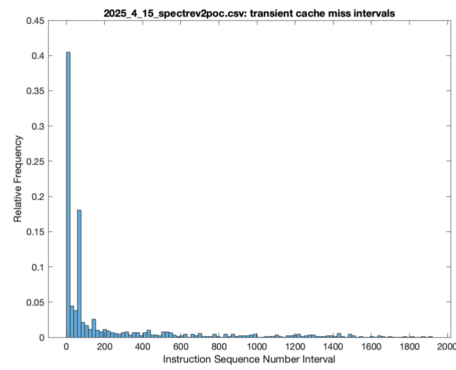
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



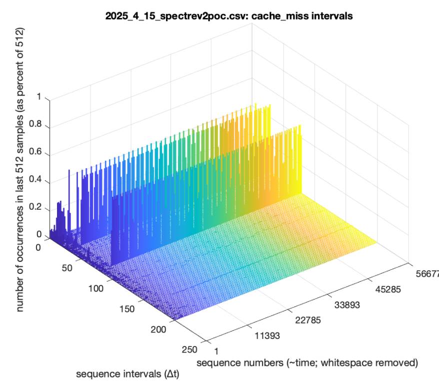
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

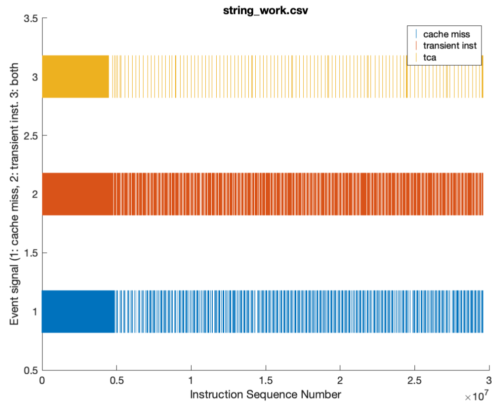


(d) Transient cache miss interval histogram

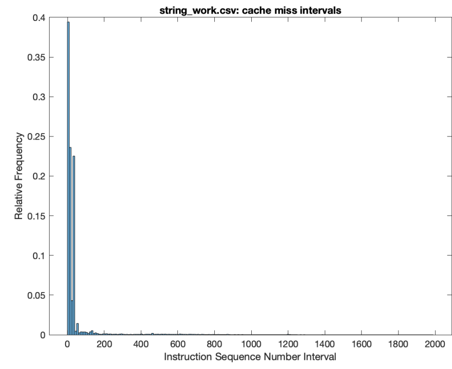


(e) Cache miss interval sliding-window histogram

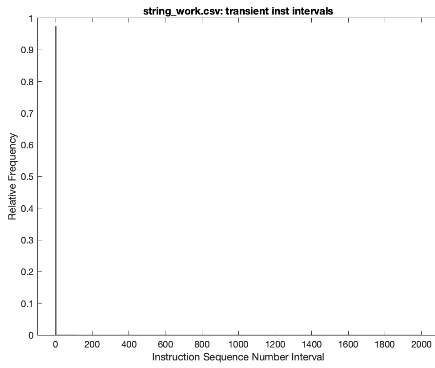
Figure A.12: spectrev2poc program simulation results



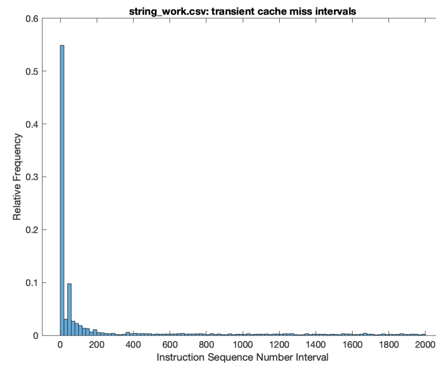
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



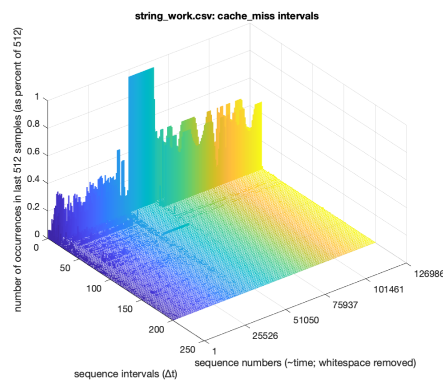
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

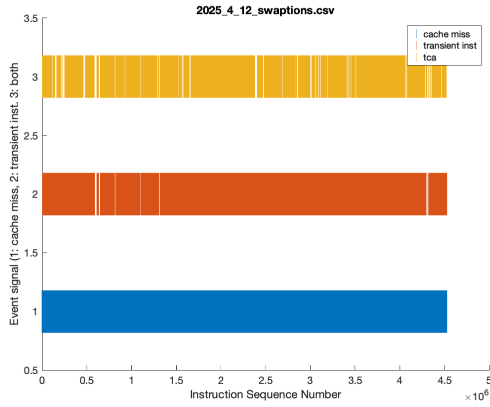


(d) Transient cache miss interval histogram

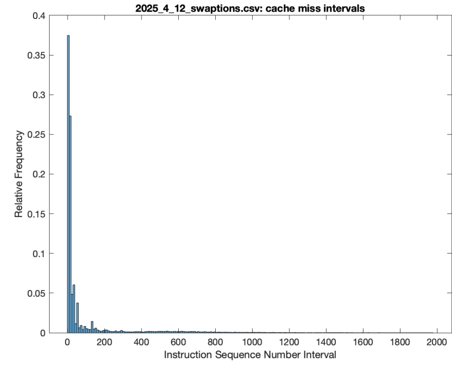


(e) Cache miss interval sliding-window histogram

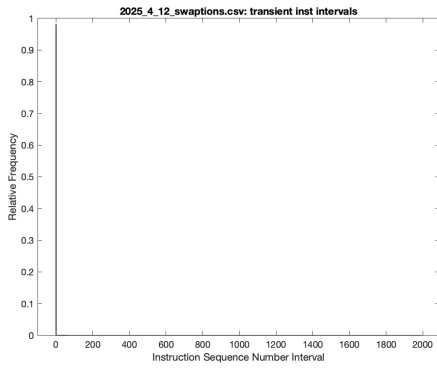
Figure A.13: string_work program simulation results



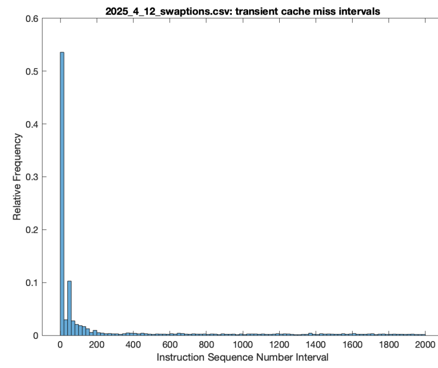
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



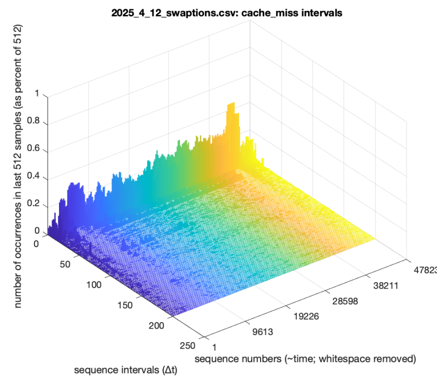
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

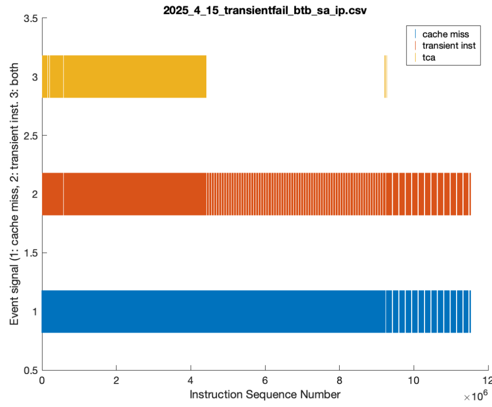


(d) Transient cache miss interval histogram

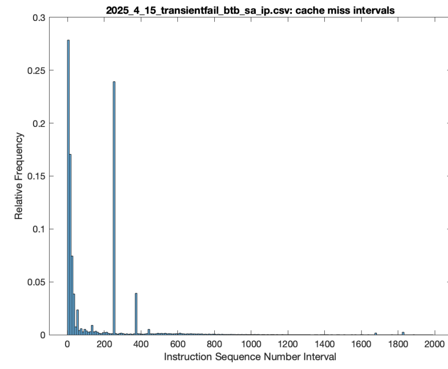


(e) Cache miss interval sliding-window histogram

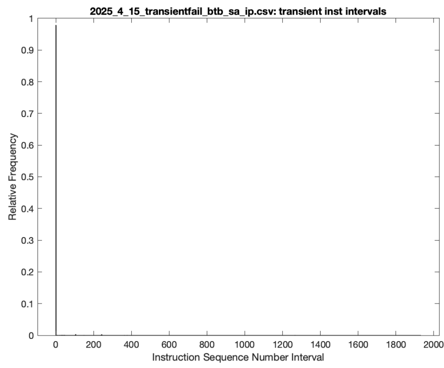
Figure A.14: swaptions program simulation results



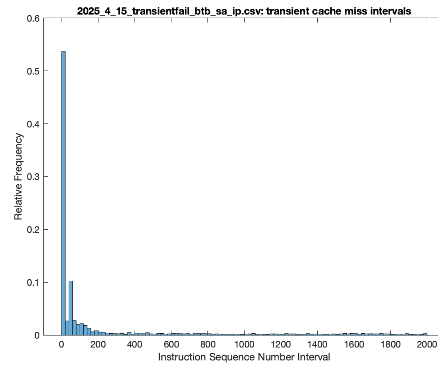
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



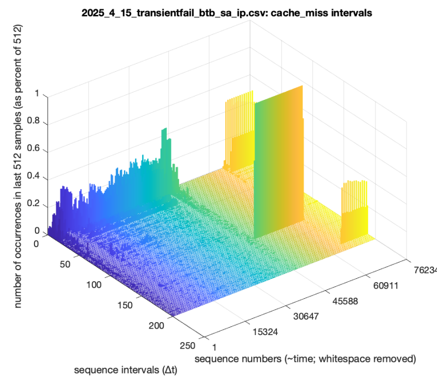
(b) Cache miss interval histogram



(c) Transient instruction interval histogram

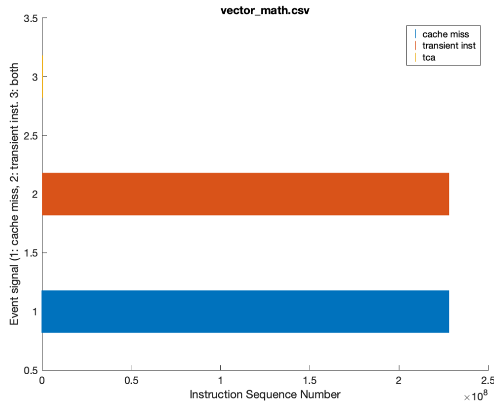


(d) Transient cache miss interval histogram

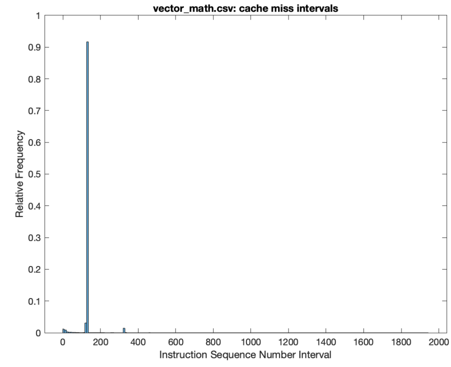


(e) Cache miss interval sliding-window histogram

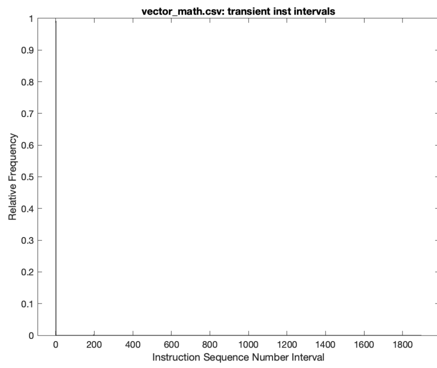
Figure A.15: transientfail_btb_sa_ip program simulation results



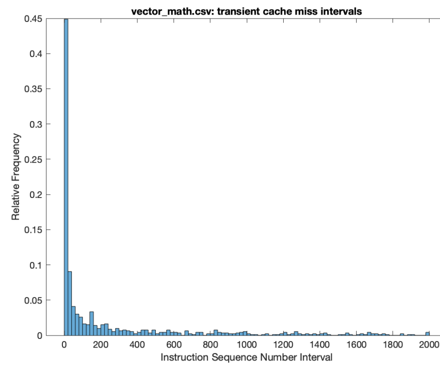
(a) Simulation run showing cache miss, transient instruction, and transient cache miss events



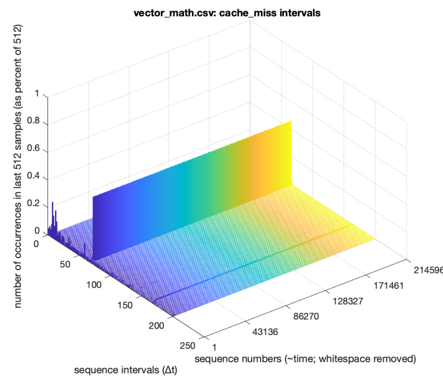
(b) Cache miss interval histogram



(c) Transient instruction interval histogram



(d) Transient cache miss interval histogram



(e) Cache miss interval sliding-window histogram

Figure A.16: vector_math program simulation results

Appendix B

Project Software Repository

Many of the other projects researched for this thesis were difficult to reproduce due to lack of instructions. Because of this, a GitHub repository is presented as a guide for modifying gem5 to include the cache and branch prediction debug traces used in this project, as well as the Python scripts for analyzing those traces, MATLAB programs for producing the graphs, and the notebooks used in training the Mamba model.

The GitHub repository can be found at:

`https://github.com/wyattellison/spectredetector`.

Bibliography

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre Attacks: Exploiting Speculative Execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, May 2019. ISSN: 2375-1207.

[2] Y. Yarom and K. Falkner, “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack,” July 2013.

[3] P. Vila, B. Köpf, and J. F. Morales, “Theory and Practice of Finding Eviction Sets,” 2018. Version Number: 2.

[4] R. Chakravarty, “3Tree: Segmented CPU Caching for Speed and Eviction Set Security,” June 2025.

[5] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, “Meltdown: reading kernel memory from user space,” *Communications of the ACM*, vol. 63, pp. 46–56, May 2020.

[6] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. v. Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, “A Systematic Evaluation of Transient Execution Attacks and Defenses,” in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 249–266, USENIX Association, Aug. 2019.

[7] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. MÃ¼ck, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Ã. F. Zulian, “The gem5 Simulator: Version 20.0+,” Sept. 2020. arXiv:2007.03152 [cs].

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, May 2011.

[9] P. Ayoub and C. Maurice, "Reproducing Spectre Attack with gem5: How To Do It Right?," in *Proceedings of the 14th European Workshop on Systems Security*, (Online United Kingdom), pp. 15–20, ACM, Apr. 2021.

[10] J. Depoix and P. Altmeyer, "Detecting Spectre Attacks by identifying Cache Side-Channel Attacks using Machine Learning," in *Fourth Wiesbaden Workshop on Advanced Microkernel Operating Systems*, (Wiesbaden, Germany), pp. 75–86, Aug. 2018.

[11] A. Choudhari, S. Guilley, and K. Karray, "SpecDefender: Transient Execution Attack Defender using Performance Counters," in *Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security*, (Los Angeles CA USA), pp. 15–24, ACM, Nov. 2022.

[12] Y. Zhang and Y. Makris, "Hardware-Based Detection of Spectre Attacks: A Machine Learning Approach," in *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, (Kolkata, India), pp. 1–6, IEEE, Dec. 2020.

[13] Z. McKevitt, A. Trivedi, and T. S. Lehman, "SpecCheck: A Tool for Systematic Identification of Vulnerable Transient Execution in gem5," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 265–278, Oct. 2023.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, (Toronto Ontario Canada), pp. 72–81, ACM, Oct. 2008.

[15] A. Gu and T. Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces," May 2024. arXiv:2312.00752 [cs].

[16] A. Torres-Leguet, "mamba.py: A simple, hackable and efficient Mamba implementation in pure PyTorch and MLX.," 2024.