Robustness of Microarchitecture Attacks/Malware Detection Tools against Adversarial Artificial Intelligence Attacks

Team: sdmay23-16

Client &/Advisor: Berk Gulmezoglu
Team Members:
Shi Yong Goh
Connor McLoud
Felipe Bautista Salamanca
Kevin Lin
Liam Anderson
Eduardo Robles

Team Email: sdmay23-16@iastate.edu

Team Website: http://sdmay23-16.ed.ece.iastate.edu/

List of Figures

1. Introduction

1.1. Problem Statement

In a technology-centric world, cybersecurity is crucial to ensuring consumer privacy of information. Some microarchitecture-based malware attacks cannot be detected using current existing software – causing a breach of security and loss of privacy. Major chip manufacturers and cloud computing providers need a way to identify and differentiate these attacks from benign signals to ensure

consumer privacy. These attacks can happen at any given time, making them difficult to identify. Thus, our team is creating a software tool that can assess the robustness of an AI-based detector against microarchitecture attacks. This will allow companies to strengthen and improve their own software to better detect and quarantine said attacks.

1.2. Requirements & Constraints

1.2.1. Functional Requirements

The software our team will develop will assess the robustness of security systems that attempt to detect microarchitecture attacks. The robustness will be measured by its ability to detect microarchitecture attacks specially designed to evade detection. The software will generate these evasive adversary attacks by inserting artificial noise into the attack instructions to mimic benign power signatures and exploit the security system's underlying machine-learning model.

Five microarchitecture attack codes will be provided, and all five attacks must be able to execute without detection and without significantly slowing down the attack. The security system cannot report any higher than 20% detection certainty for the attack to be undetected. The power usage should not exceed 2x normal activity, and the attack should not surpass a 5x slower data leak rate than its non-evasive counterpart.

1.2.2. UI Requirements

The software must feature a user-friendly graphical user interface with the following features and functions:

- Define an attack type.
- Upload attack source code
- Specify run parameters.
    - Number of executions
    - Time in between executions (seconds)
- Launch evasive examples.

After running an attack, the GUI must display statistics about the attack, including the data leak the rate in bits per second and the security systems malware detection certainty for the selected model as a percentage.

1.2.3. Resource Requirements

The project will require specialized hardware to pull the necessary CPU power consumption data and achieve the performance needed to run the AI-based microarchitecture attack detector. Our team will be provided and required to use the following experimental setup:

- Intel Comet Lake Microarchitecture
    - CPU Model: Intel(R) Core (TM) i7-10610U CPU @ 1.80GHz
    - OS: Ubuntu 20.04 LTz
    - Linux Kernel: 5.11.0-46-generic
- Server Information
    - Nvidia GeForce RTX 3090 GPU
    - CPU Model: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz

1.3. Engineering Standards

IEEE 12207: This standard provides processes that can be employed for defining, controlling, and improving software life cycle processes within an organization or a project.

IEEE 1074: This standard provides a process for creating a software project life cycle process (SPLCP), and it is primarily directed at the process architect for a given software project.
Software Testing

IEEE 29119: This is a series of five standards for software testing. They define vocabulary, processes, documentation, techniques, and a process assessment model for testing that can be used within any software development lifecycle.

1.4. Intended Users and Uses

1.4.1 Researchers

Researchers focusing on microarchitecture attacks will use our tool to test and further research power-anomaly detection systems.

Key Characteristics

Researchers are the leaders in developing and further exploring microarchitecture vulnerabilities and security solutions. They have a vast knowledge of computer engineering and system security. Researchers are more concerned with the pursuit of knowledge and development of theory than with implementing solutions for economic reasons.

Needs

Researchers need a better way to test new microarchitecture attacks against their deep learning power-anomaly detection systems. They must validate that their security systems hold up against evasive microarchitecture attacks.

How they will benefit

With the tool our team is developing, researchers can quickly generate new, highly evasive microarchitecture attacks. Saving them a lot of time implementing it themselves and providing them with the tools they need to test their systems.

1.4.2. Implementers

Implementers, such as Intel, AMD, Nvidia, and PaaS providers, will use our tool to test their systems against microarchitecture attacks.

Key Characteristics

Implementers are companies that manufacture chipsets or provide access to computer resources. Vulnerabilities in their products can damage their reputation and be very costly, so they tend to invest heavily in cybersecurity.

Needs

Implementers need a tool to test their products and discover potential vulnerabilities.

How they will benefit

Implementers will be able to conduct penetration testing on their products with evasive microarchitecture attacks and, as a result, discover existing vulnerabilities.

1.4.3 End Users

End users indirectly benefit by trusting their data with systems tested by our tool.

Key Characteristics

End users are typical everyday computer users. They don’t have much knowledge of cybersecurity or the inner workings of the system they are using, and they trust the product they use is secure.

Needs

End users need their data to be secure. They need their personal computer or cloud environment not to be vulnerable to microarchitecture attacks.

How they will benefit

End users can be assured that their private information is secured and inaccessible to non-authorized individuals. They can also have better protection on their personal devices and as well as their cloud environments.

1.5 Security Concerns & Countermeasures

1.5.1 Physical Security

Unauthorized users might take over a user's laptop and gain access to it.

- The user needs to be responsible to lock their laptop when they are not close to them.

1.5.2 Cybersecurity

An unauthorized user might be trying to gain access to the servers using the GUI.

- The GUI can only contact the server if an individual SSH key has been created and set up in the laptop's server. Without the SSH key, the GUI cannot create a connection.

2. Project Design Evolvement

2.1 Original Project Design

2.1.1 Overview

Our software will offer a graphical user interface (GUI) or command line interface (CLI) for the user to interact with the application. Navigating the interface, the user can upload attack codes and select a detection model and attack type, which is fed into the instruction insertion algorithm. The algorithm then generates an evasive attack that exploits the detection system's underlying machine learning (ML) model. Running the attack with the program will generate statistics about the attack.

2.1.2 Detailed Design and Visual(s)

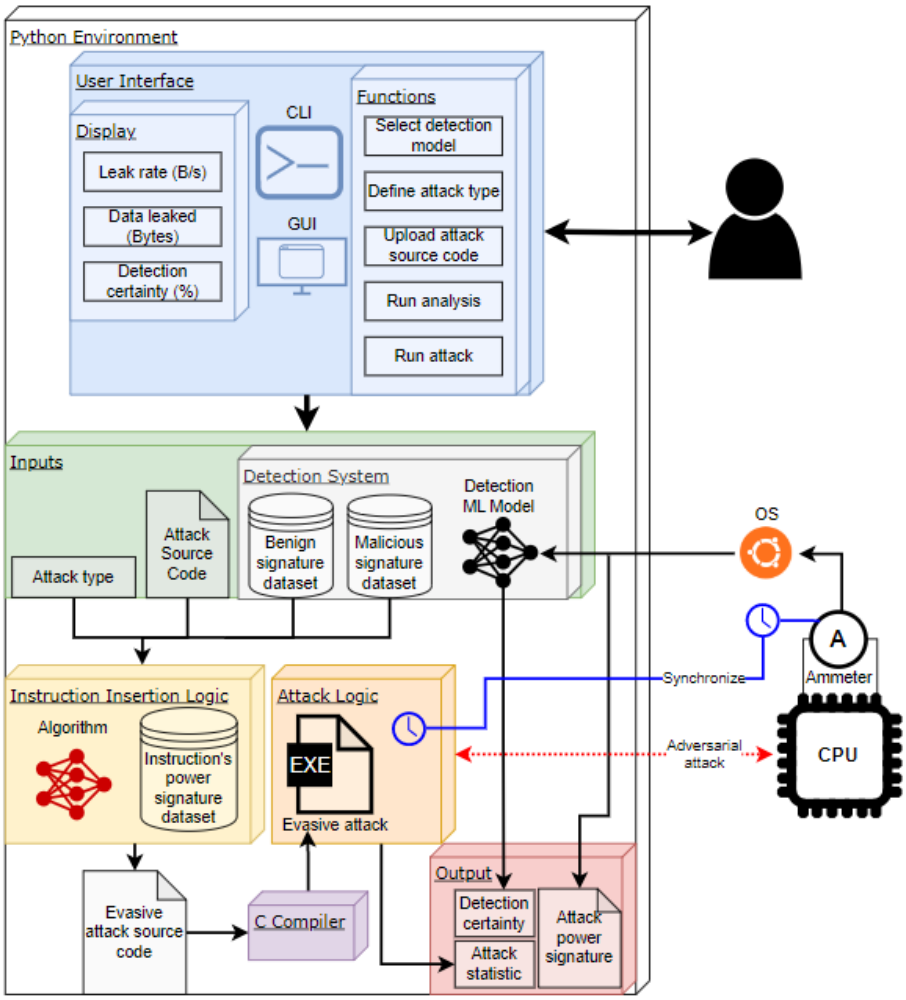

Figure 1: System Architecture Diagram

2.1.2.1 Design Overview

Our system primarily lives in a Python environment. Currently, a machine learning model exists that can differentiate between malignant and benign codes. Our job is to find ways to lower the model's detection certainty to below 20% - if possible.

Users are provided with a GUI and CLI to access the functionality of the software that we will be writing. Users can upload attack source code by navigating the interface and selecting the attack type and detection model. This is fed into the instruction insertion logic, which generates the attack code, but is modified to attempt to avoid the selected machine learning model's detection. Running that attack with the program would create statistics about the attack, including data leaked, certainty, and leakage rate.

2.1.2.2 User Interface

We provide clients with a GUI as well as a CLI, both with the same functionality. Clients can interact with our application through it. They can input models and attack types/codes to receive and output, which is

displayed after the machine learning model analyzes it. This is output to the console/GUI with detection certainty, data leaked, and leakage rate.

2.1.2.3 Inputs

The inputs that are available for the client to put in consist of the attack type, attack code, and detection model. The attack source codes will either be C files or x86 assembly.

2.1.2.4 Instruction Insertion Logic

All the inputs are fed into another algorithm that is compared with the power reading dataset. This generates attack source code that is meant to subserve the selected mode. It then compiles the code and sends this into the attack logic block.

2.1.2.5 Attack Logic

After running the code, it determines whether the code is malware and outputs its certainty percentage and various statistics. This is done by reading the computer's power usage as the attack is running and looking for any abnormalities that have already been quantified in the machine-learning model.

2.1.2.6 Outputs

All this info will be outputted on the CLI or GUI for the client to view.

2.1.3 Functionality

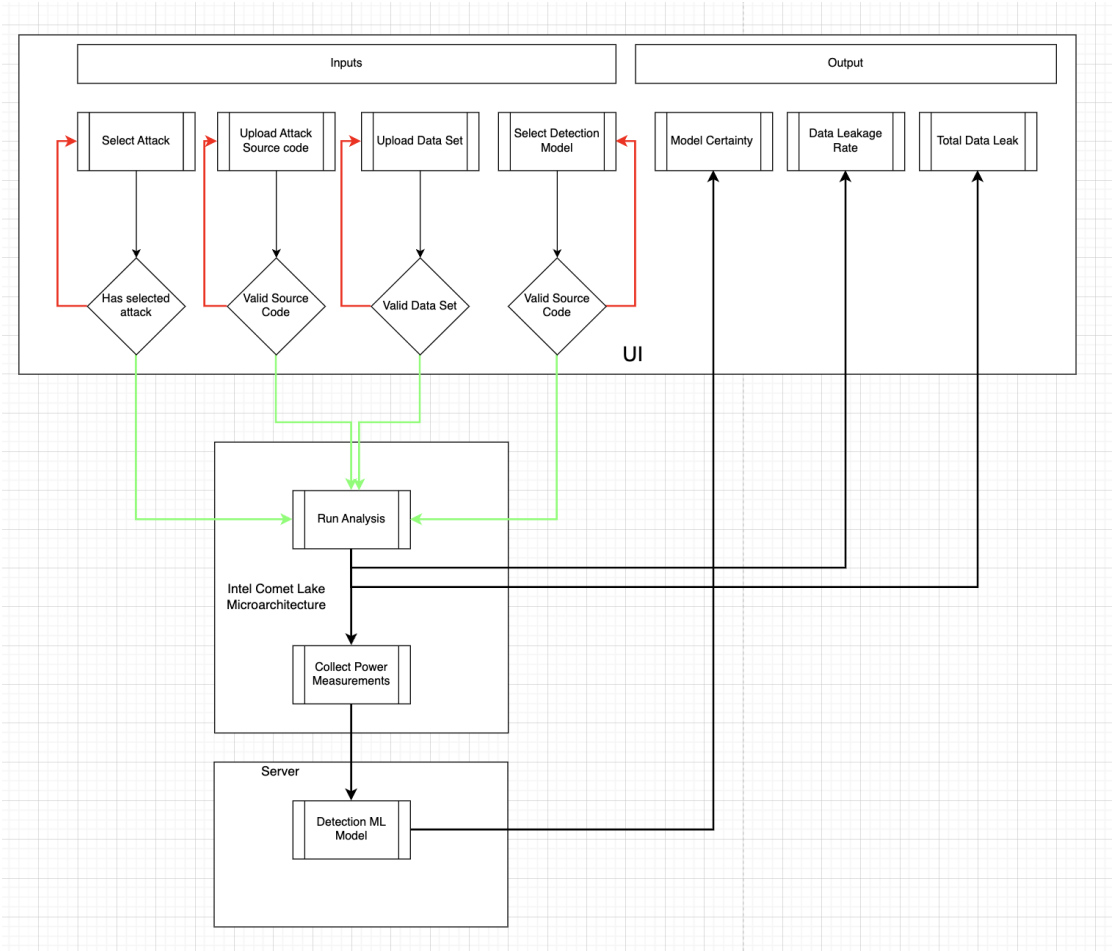

Figure 2: System Functionality Diagram

Our design consists of using the UI as a way for the user to interact with the system as well as to receive information. As shown in the image above, the user can select and add unique system inputs that fit their needs. These inputs are selecting which attack the trained model is looking for, the source code of the attack the user is testing, the data set used, and lastly, selecting which trained model the user wants to test the attack on. Once the user has inputted all these parameters, they can go ahead and run the analysis.

The microarchitecture will run the code and output the rate at which the attack leaked the data as well as the total amount of data that was leaked. This information will be sent to the UI, where the user can get those results. After the code is run, the system will then go ahead and collect the power measurements generated by the uploaded attack on the microarchitecture. These measurements will be sent to the server along with the data set of power measurements generated by the selected attack type. The detection model will run an analysis and output a certainty level on the match between the data set given by the user and the uploaded attack power measurements collected. If the power measurement matches the power measurements of the selected attack, it will result in a higher certainty. This result would also be sent to the UI, where the user can see the result of the analysis.

2.2 Evolved Project Design

The original project design initially expected there be around 3-4 attacks that the team would create adversarial examples against. In addition, we expected to be able to create a ML model that would be used to insert instructions into the attack. Throughout the course of the project, as the team faced more unanticipated complexity, the team was told to focus on working on the Spectre attack. The design was simplified as the team's understanding of the project grew.

3. Implementation

3.1 GUI

3.1.1 Frontend

The GUI was implemented using Python and PyQt6 GUI Framework. Using the Qt toolkit, PyQt6 is a Python GUI framework for developing GUI applications. The GUI allows the users to perform the following actions.

- Select the attack type they are going to test on the system.
- Upload the source code of their evasive example.
- Provide parameters for the execution.
    - Number of times to execute evasive attacks.

        - The wait time between executions (seconds)
    - Executing the evasive example

The GUI can reformat power computation data. This is necessary since the detection ML model required a specific format before it could test the data and determine if it was an attack.

Since the UI halts once it executes the SSH script to open the communication channel to catch errors, the logs generated by the script are logged into a respective log file. Once the script is done executing and the GUI resumes, it will examine the log file generated by the script. It will search for errors that occurred during the execution of the script and handle those errors accordingly.

3.1.2 Backend

The backend logic uses SSH to establish communication channels with machine learning (ML) and a power signature server to collect and analyze data. Two bash scripts were written to open SSH connections with the servers and transfer the needed data between the GUI and the respective server. The GUI could execute the scripts by utilizing the OS library in the Python code.

3.2 Servers Configuration

3.2.1 Test Server (Intel Comet Lake Microarchitecture)

This server is configured with Bash scripts that provide the ability to collect statistics about an evasive example using Intel's built-in tool, **Running Average Power Limit** (RAPL) interface. The tool reports the accumulated energy consumption of various power domains. We use this tool to profile an instruction's power consumption.

3.2.2 Detection Machine Learning Model Server

The server that hosts the Machine Learning Model is configured with Bash scripts that compares the uploaded data to the machine learning model. This can test an attacker's ability to fool the ML model and provide the certainty rate.

3.3 System Communication Diagram

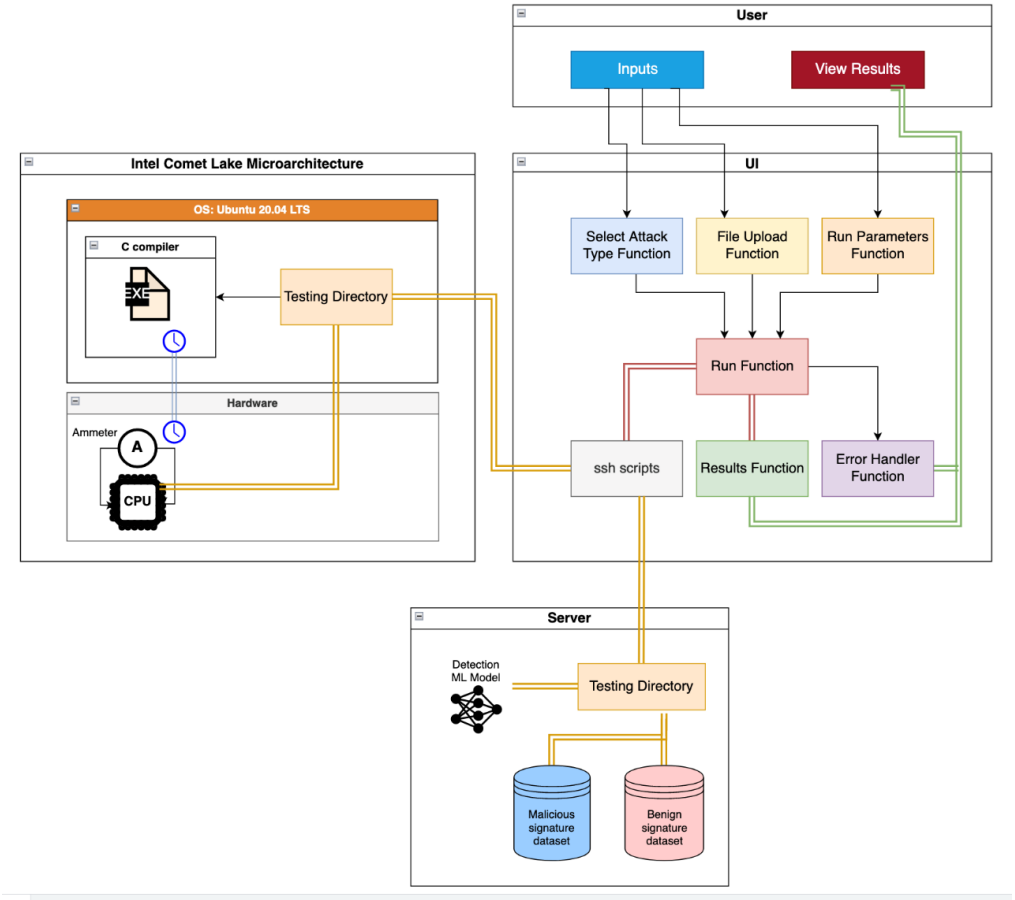

Figure 3: System Communication Diagram

4. Testing Process & Results

4.1 x86 Instructions Testing

Our objective is to identify instructions that consume a higher amount of power while having a lower execution frequency. This approach may assist attackers in evading detection mechanisms that aim to identify abnormal power consumption patterns. By focusing on instructions that consume more power and are executed less frequently, attackers could potentially obscure their activities within the background noise generated by more frequently executed instructions, thereby decreasing the likelihood of detection.

4.1.1 Profile and Find all the Executable Instructions

During the testing phase, we created an automated system to rapidly execute all the instructions documented for Intel Skylake processors (Agner, 2021). We then utilized the system to collect power consumption data for each instruction and identify those that consumed the most power on average.

Instruction	Time (ms)	Instructions (M)	Avg. Power	# Power Values	Inst. / ms (K)
add i r16	458.31	450	7400.04	3917	981.86
add i r32	223.73	450	8493.38	1644	2011.33
add i r64	267.95	450	8828.76	1923	1679.41
dec r64	136.29	450	7656.22	1162	3301.88
cmp i r64	272.54	450	8616.44	1967	1651.16
imul i r64 r64	274.20	450	8760.30	2008	1641.11
mulx r32 r32 m	241.22	960	8382.29	2415	3979.69
mov r64 m	231.46	153	6878.65	2308	661.01
mov r64 r64	259.51	2700	7791.87	2576	10404.09
prefetchw	807.28	200	7479.29	8294	247.74
rdtsc	898.44	136	8112.16	9336	151.37
sleep	1001.32	0	6081.71	9826	N/A
spectre	807.91	0	5939.56	8319	N/A

Figure 4: Table of Individual Instructions' Power Consumption

4.1.2 Testing Methodology

After adding noise to the attack code, we executed the code using the GUI. Each run involved taking 15 measurements to collect 5000 sample points of power consumption, which we gathered using the Intel RAPL built-in software. As RAPL gathers data points as a running average, our backend first calculates the differences between each sample point. Before inputting the data into the machine learning (ML) model, the backend extracted the differences between sample points to determine the power consumed at each point. The detection model then displayed the results as the accuracy of the detection.

It's important to note that the data collection process is susceptible to external interference, such as someone accessing the laptop during the collection phase. To ensure data accuracy, we used Matlab to plot the samples. If the plots were significantly off, we would reboot the laptop and recollect the data.

4.1.3 Sleep() as Noise

The sleep function is added before the victim_function() in the readMemoryByte() function of the attack code. After adding 0.5s of sleep using nanosleep(), the ML detection accuracy is around 7%. Similarly, adding 100us sleep using usleep() also evaded the detection system. Using the usleep() function resulted in a higher power consumption by the processor compared to using the nanosleep function. However, both approaches resulted in an execution time that was twice as slow as the original attack code.

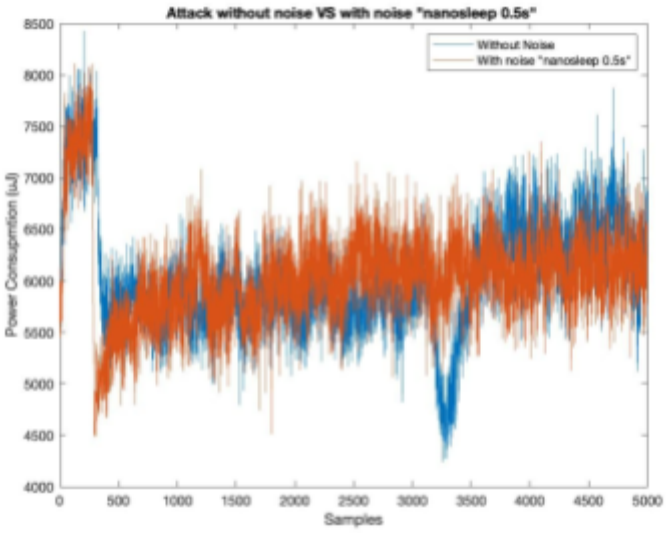

Figure 5: Power Consumption Comparison (nanosleep 0.5s)

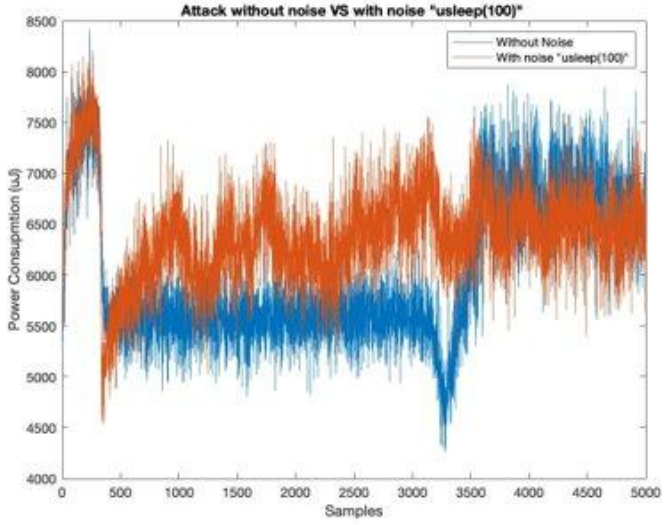

Figure 6: Power Consumption Comparison (usleep 100)

4.1.4 Create Noise that using Different Instructions

4.1.4.1. Floating Point Instructions

We used faddps & flp instructions to generate some noises as shown in Figure 7, as they involve complex mathematical calculations and require more processing resources than simpler instructions, resulting in increased power consumption by the processor. This noise is placed at the beginning of the readMemoryByte() function of the attack code. Based on our experiment, to 100% evade the detection

system, the noise must have at least 200000 loops. The execution time is only 1.03 times of the original attack code's execution time.

```
float a = 1.11, b = 4.77;
    __asm__ (
     ".rept "XSTR(10000000)"\n\t"
    "fld %1;"
    "fld %2;"
    "faddp;"
    "fstp %0;"
    ".endr;"
    : "=m" (a)
    : "m" (a), "m" (b)
    );
```

Figure 7: sample instructions code of faddps & flp (10000000 loops)

As the number of loops increases, the compiler will take a longer time to generate the executable file. In Figure 8, we observed that the sample points from 0 to 1000 of the attack with faddps & flp noises were collected while the C file was compiling. Hence, by examining the sample points between 1000 and 2500, we observed an increase of around 250uJ in the power consumption of the attack with faddps and flp noises.

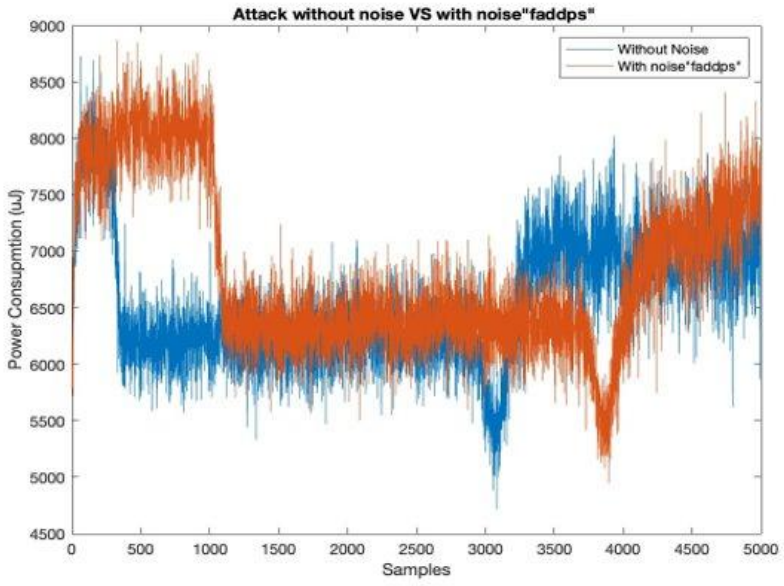

Figure 8: Power Consumption Comparison (faddps&flp-200000 loops)

4.1.4.2. Logical Instructions

Inserting OR x86 instructions into the attack code resulted in the red line shown below in Figure 9. The instructions were added right after the readMemoryByte() section of the code. After adding 300,000 OR instructions the model detected it with 60% accuracy. The attack ran for a total of 2.06 minutes. The resulting graph looks very different with the noise added in. The attack still got through the model with lower accuracy.

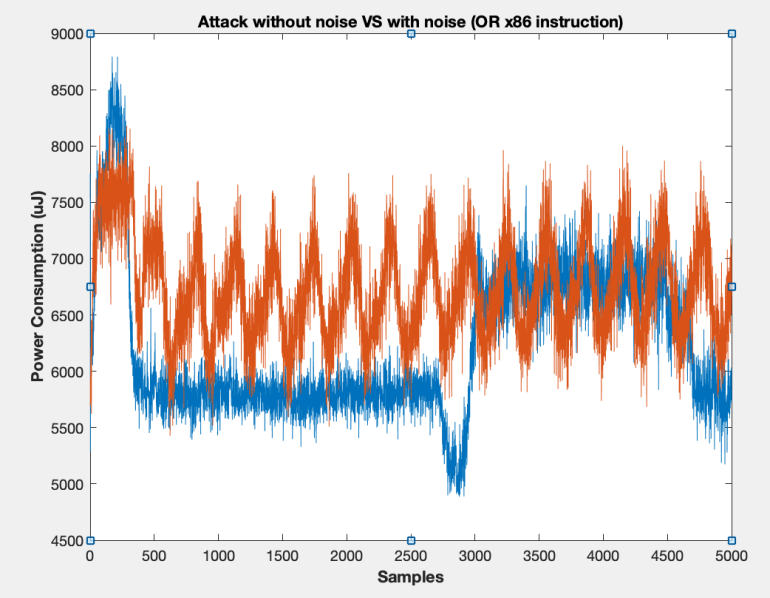

Figure 9: Power Consumption Comparison (OR)

5. Related Products & Literature

5.1 Literature

The team used “Using Power-Anomalies to Counter Evasive Micro-Architectural Attacks in Embedded Systems” by Shijia Wei, Aydin Aysu as a reference for this project. The paper highlights the impact that malware-injected applications can have on the power signatures of computing systems. The authors proposed a unique technique to detect and counter micro-architectural attacks in embedded systems, which involves analyzing the power signatures of a computing system during the execution of a given application.

The paper also demonstrated that malware-injected applications can cause significant alterations to the power signatures of a computing system. However, the proposed technique by the authors can successfully identify the presence of malware-injected applications by detecting power anomalies. Additionally, the authors showcased that their proposed technique is effective in mitigating the effects of micro-architectural attacks by limiting the ability of malware-injected applications to modify the power signatures of the computing system.

Building on the topic of microarchitectural attacks, another paper titled "MAD-EN: Microarchitectural Attack Detection through System-wide Energy Consumption" presents a novel method for detecting such attacks. Authored by Dipta, D.R. and Gulmezoglu, B., this paper proposes a new approach that monitors system-wide energy consumption to identify changes in power consumption caused by microarchitectural attacks.

Microarchitectural attacks are a type of side-channel attack that exploits the behavior of a processor's microarchitecture to extract sensitive information such as passwords, encryption keys, or other data from

a system. The proposed method, called MAD-EN, monitors the energy consumption of a system to detect changes in power consumption caused by microarchitectural attacks. The authors claim that their method can detect a wide range of microarchitectural attacks, including those that are not detected by existing methods such as cache-based attacks and speculative execution attacks.

6. Appendix

6.1 Operational Manual

6.1.1 Prerequisites:

This program requires the Python 3.x and PyQt6 packages.

6.1.2 Makefile:

To install the Required dependencies run the following command in the shell of your choice:

- $ sudo make install

To compile the source files execute the following command:

- $ sudo make

To run the code it's as simple as typing the following into your shell:

- $ spectreGUI

To remove the executable and all unnecessary object files run the following:

- $ sudo make clean

6.1.3 SSH Key Generation:

The Learning Model and Remote Server, both, require the use of SSH for connecting remotley from your personal computer. By setting up an SSH Key Pair, the system will not prompt you for a password every time you attempt to make a connection through the shell. Additionally, without the SSH Key, the GUI will throw several errors when attempting to make a connection. Below, I will list the steps necessary for generating your own SSH key as well as setting up an SSH key pair on both servers.

NOTE: Depending on your personal OS and setup, you may need to use sudo permissions. Do not use them unless necessary here.

NOTE: The IP may change several times during the course of this project.

NOTE: "$" can often denote a new line in the console when referring to Shell commands and occasionally when referring to bash scripting.

SSH KEY PAIR:

Step (01): Generate your own personal SSH Key.

- Run the following command in the shell of your choice.

```
$ ssh-keygen
```

Step (02): Copy your newly generated SSH Key onto the Remote Server.

- Run the following command in the shell of your choice.

```
$ ssh-copy-id sdmay23-16@10.26.XXX.XXX
```

Step (03): Copy your SSH Key onto the Machine Learning Model's Server.

- Run the following command in the shell of your choice.

```
$ ssh-copy-id sdmay23-16@berk.ece.iastate.edu
```

6.1.4 Running the Program:

First open a shell of your choice and change directories until you are in the project folder named "GUI" that actually contains the GUI's python scripts. Do this by executing the following command in the shell of your choice:

- $ cd ~/{file_path_to_sdmay23-16-main}/sdmay23-16-main/GUI

NOTE: {file_path_to_sdmay23-16-main} will vary based on where you have the project saved on your local machine.

To Confirm you are in the correct place, you may run either of the following two commands:

```
$ pwd // pwd - post working directory displays the file path
      // your current location in the shell.
```

or

```
$ ls  // list- shows you the contents of the directory your
      // shell is currently in. If using this method, make
      // sure at the very least the file "main.py" is
      // present.
```

To run the program, execute the following command in the shell of your choice:

- $ sudo python3 main.py

6.1.5 Bash Scripts:

6.1.5.1 Script: model_helper.sh

Description:

This is a bash script that helps in running attacks and collecting data for the machine learning model's creation. The script takes six arguments and three constant values which will be described in more detail below, under the section "Functionality: Arguments & Constant Values".

Usage:

The script first changes into the working directory and then uses the scp command to copy the X_attack_test_15.csv file from the current attack run's directory to the ~/testing_workspace/Data/ directory on the ML server. Next, the script uses the ssh command to log into the ML server and execute the Restored_model_test.py script located in the testing_workspace directory. The >> operator is used to append the output of the Restored_model_test.py script to a file named results_NAME.txt, where NAME is the name of the current attack test run. The deactivate command is then used to exit the virtual environment created by source tensorflow/roy-venv/bin/activate. Finally, the script uses the scp command to copy the results_NAME.txt file from the ML server back to the current attack run's directory.

Functionality: Arguments & Constant Values

Arguments:

- SOURCE_DIR: The working directory for the source files.
- SOURCE_FILE: The name of the source file for the attack, relative to the SOURCE_DIR.
- NAME: The name of the attack test run.
- RUNS: The number of times to run the attack.
- WAIT_TIME: The wait time between each attack for simple attack runs.
- TEST: A boolean value that is 0 if the script should collect data for the ML model, and 1 if the script should run a simple attack.

Constant Variables:

- TL_USERNAME: The username for the remote test laptop.
- MLS_USERNAME: The username for the ML server.
- MLS_HOST: The hostname or IP address of the ML server.

6.1.5.2 Script: ssh_helper.sh

Description:

This is a bash script that assists in the automation of running an attack remotely on a target machine and collecting the results. The script takes in six arguments and five constant values which will be described in more detail below, under the section "Functionality: Arguments & Constant Values".

Usage:

First, the script sets up the ssh info for the remote test laptop and the ML server before it creates a directory for the attack. It then makes the directory on the remote machine server and copies the source code of the attack onto the remote machine. Lastly, this script runs the attack remotely on the target machine and copies the results back to the host machine.

Functionality: Arguments & Constant Values

Arguments:

- - SOURCE_DIR: the directory of the attack source file
- - SOURCE_FILE: the path to the source file for the attack. This is relative to the SOURCE_DIR
- - NAME: the name of the attack test run
- - RUNS: the number of times to run the attack
- - WAIT_TIME: the wait time between each attack for simple attack runs
- - TEST: 0 or 1. Zero means collecting data for the machine learning model, while one means running a simple attack.

Constant Variables:

- - TL_USERNAME: the username for the remote test laptop
- - TL_HOST: the IP address of the remote test laptop
- - MLS_USERNAME: the username for the machine learning server
- - MLS_HOST: the hostname of the machine learning server
- - FILE_NAME: the name of the attack source code file

6.2 References

Agner, F. (2021). Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Retrieved from https://www.agner.org/optimize/instruction_tables.pdf

Dipta, D. R., & Gulmezoglu, B. (2022). MAD-EN: Microarchitectural Attack Detection through System-wide Energy Consumption. arXiv preprint arXiv:2206.00101.

Wei, S., & Aysu, A. (2016). Using power-anomalies to counter evasive micro-architectural attacks in embedded systems. Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1-8. doi: 10.1145/2966986.2967037