μLAM: A LLM-Powered Assistant for Real-Time Micro-architectural Attack Detection and Mitigation

Upasana Mandal
mandal.up98@kgpian.iitkgp.ac.in
Indian Institute Of Technology, Kharagpur
Kharagpur, India

Shubhi Shukla
shubhishukla@kgpian.iitkgp.ac.in
Indian Institute Of Technology, Kharagpur
Kharagpur, India

Ayushi Rastogi
rayushi835@gmail.com
Indian Institute Of Technology, Kharagpur
Kharagpur, India

Sarani Bhattacharya
sarani@cse.iitkgp.ac.in
Indian Institute Of Technology, Kharagpur
Kharagpur, India

Debdeep Mukhopadhyay
debdeep@cse.iitkgp.ac.in
Indian Institute Of Technology, Kharagpur
Kharagpur, India

ABSTRACT

The rise of microarchitectural attacks has necessitated robust detection and mitigation strategies to secure computing systems. Traditional tools, such as static and dynamic code analyzers and attack detectors, often fall short due to their reliance on predefined patterns and heuristics that lack the flexibility to adapt to new or evolving attack vectors. In this paper, we introduce for the first time a microarchitecture security assistant, built on OpenAI's *GPT-3.5*, which we refer to as μLAM. This assistant surpasses conventional tools by not only identifying vulnerable code segments but also providing context-aware mitigations, tailored to specific system specifications and existing security measures. Additionally, μLAM leverages real-time data from dynamic Hardware Performance Counters (HPCs) and system specifications to detect ongoing attacks, offering a level of adaptability and responsiveness that static and dynamic analyzers cannot match.

For fine-tuning μLAM, we utilize a comprehensive dataset that includes system configurations, mitigations already in place for different generations of systems, dynamic HPC values, and both vulnerable and non-vulnerable source codes. This rich dataset enables μLAM to harness its advanced LLM natural language processing capabilities to understand and interpret complex code patterns and system behaviors, learning continuously from new data to improve its predictive accuracy and respond effectively in real time to both known and novel threats, making it an indispensable tool against microarchitectural threats. In this paper, we demonstrate the capabilities of μLAM by fine-tuning and testing it on code utilizing well-known cryptographic libraries such as OpenSSL, Libgcrypt, and NaCl, thereby illustrating its effectiveness in securing critical and complex software environments.

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3676838

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

Microarchitecture Attacks, Attack Detection System, LLMs

ACM Reference Format:
Upasana Mandal, Shubhi Shukla, Ayushi Rastogi, Sarani Bhattacharya, and Debdeep Mukhopadhyay. 2024. μLAM: A LLM-Powered Assistant for Real-Time Micro-architectural Attack Detection and Mitigation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676838

1 INTRODUCTION

In the past decade, numerous new vulnerabilities in microarchitectures have come to light, some causing significant financial impacts on leading CPU manufacturers like Intel. With the emergence of such vulnerabilities, corresponding countermeasures and mitigations have also been developed. Additionally, companies have begun incorporating patches in subsequent processor releases. This field of research is rapidly expanding, though it is complex and typically understood by a small percentage of professionals in the field. However, not all enterprises that depend on these systems have access to experts in microarchitecture security. Given the ongoing handling of sensitive data and transactions, the need for a microarchitecture security assistant has become crucial. Such a tool can assist anyone operating these systems by providing guidance on writing secure code, identifying potential security risks in their software, and detecting live attacks on their system, and in this work we try to build just that by leveraging the power of LLMs.

In a typical microarchitecture attack scenario, an attacker runs a program called spy to compete with victim program for shared hardware resources such as common cache memory [14][4], branch prediction unit [1], a translation look-aside buffer [8], a DRAM [13] [24]. The conflict to use common resources influences spy's execution time in proportion to victim's execution. The correlation can therefore be utilized to recover victim's secret data if its execution depends on it. The majority of defenses against micro-architectural attacks are either hardware or software integrated.

Current attack mitigation strategies include software patching [3], [12], allowing least possible privileges for user that are necessary for running the application [17], adoption of ASLR and KPTI techniques that fortifies the system at application, operating system and hardware level. Hardware options include dividing the shared resources [20] [5], [18]. Redesigning secure chips with less performance overhead and replacing all the prevalent designs is an expensive and impractical solution. At the same time, software patching and other security enforcing adoptions may defend certain vulnerabilities but could not suffice to combat leakages due to speculative execution, out-of-order executions and other subtle optimizations. Hence combating attacks and building necessary defenses remains ardent.

Recent advancements in security research have seen a variety of tools developed for identifying vulnerabilities and detecting attacks, including static and dynamic code analyzers and systems that monitor Hardware Performance Counters (HPCs) for unusual activity. Static code analyzers [11] assess source code at rest without execution, while dynamic analyzers evaluate code during runtime, capturing more dynamic events for discrepancies. However, both types often rely on predefined patterns and lack the adaptability to effectively address newly emerging or complex attack vectors. Similarly, while HPC-based monitoring [2, 6, 16] has proven useful for detecting certain types of attacks, it traditionally struggles with the subtlety and variability of advanced threats, such as camouflaged microarchitectural attacks that deliberately minimize detectable footprints.

In response to the limitations of traditional security tools, in this work we introduce a more dynamic approach using Large Language Models (LLMs) [7]. We present *μLAM*, a sophisticated microarchitecture attack assistant developed with OpenAI's GPT-3.5 [23]. The motivation for using LLMs comes from their exceptional ability to analyze and interpret large volumes of unstructured data, making them ideal for identifying complex, evolving patterns that conventional tools might miss. This ability is essential in microarchitectural security, where attack methods continually change and new vulnerabilities emerge.

μLAM not only identifies vulnerable code segments but also generates custom mitigations tailored to specific system setups and existing security measures. Additionally, it utilizes dynamic HPCs to detect ongoing attacks, enhancing its responsiveness to immediate threats. To fine-tune *μLAM*, we created a comprehensive training dataset that includes diverse system configurations, mitigation strategies for different system generations, dynamic HPC values, and a range of code samples marked as vulnerable or safe. This dataset enables *μLAM* to learn the intricacies of microarchitectural attacks. The fine-tuning process boosts the model's capability to process and respond to real-time data, allowing it to detect ongoing and even stealthy attacks effectively. Thus, by leveraging the advanced capabilities of LLMs, *μLAM* offers a robust, context-aware security solution that far surpasses traditional security analysis tools. To demonstrate the effectiveness of *μLAM*, we performed extensive testing on code from prominent cryptographic libraries, including OpenSSL, Libgcrypt, and NaCl. We specifically chose cryptographic libraries for our use-case due to their vital function in protecting sensitive data across various applications, which also makes them frequent targets for microarchitectural attacks. In summary, *μLAM* demonstrates the following capabilities:

- *Attack Detection:* It utilizes real-time HPC data to detect ongoing attacks, adeptly identifying even camouflaged attacks that are designed to evade traditional detection systems.
- *Vulnerability Identification:* It can further analyze given code alongside system and processor configurations, pinpointing sections vulnerable to specific microarchitectural attacks. This analysis includes consideration of existing mitigations and patches within the system configurations.
- *Attack Mitigation:* Upon detecting vulnerabilities, *μLAM* suggests mitigations for secure code alternatives, thereby actively contributing to system hardening against potential attacks.

This paper is organized as follows: First, we provide a brief background on microarchitecture attacks and LLMs in Section 2. Then, we delve into the design and training details of *μLAM* in Section 3, outlining the methodology and techniques employed in developing the microarchitecture assistant. In Section 4, we demonstrate the capabilities of *μLAM* and also compare its performance with other traditional methods, showcasing its effectiveness in detecting vulnerabilities and mitigating microarchitecture attacks. Lastly, we conclude our paper in Section 5 with brief discussion on potential future research directions.

2 BACKGROUND

2.1 Microarchitecture Attacks

Microarchitectural attacks exploit the intricate design of modern computer processors to stealthily extract sensitive information from applications in execution. These sophisticated attacks focus on the subtle workings and shared components of CPUs—including caches, branch predictors, and speculative execution paths—to surreptitiously infer or steal protected data. By manipulating these components, attackers can observe discrepancies in computational timing or data residues left in shared resources, which reveal details about the operations and data handled by other processes on the same machine. As CPUs evolve to deliver higher performance and efficiency, their growing complexity inadvertently expands the attack surface, making them more susceptible to such exploits. This escalation in potential vulnerabilities requires continual advancements in security strategies to mitigate these risks effectively. There are three broad Categories of Microarchitectural Attacks:

- **Cache Attacks** [26]: These exploit the CPU's cache system to deduce data access patterns of other processes. Techniques like Flush+Reload [29] involve flushing data from the cache and timing how long it takes to reload it, revealing if another process has accessed it. The Prime+Probe [21] method fills the cache with the attacker's data, then monitors for displacements caused by other processes, effectively spying on their activities.
- **Speculative attacks** [28]: These utilize the speculative execution feature of CPUs to force premature execution of instructions, allowing attackers to access temporary data in caches or registers. Notable examples, Spectre [15] and Meltdown [19], manipulate this mechanism to extract sensitive information, such as passwords and encryption keys, from protected memory areas.

- **Timing Attacks** [30]: These attacks involve measuring the duration of certain operations to deduce sensitive information, such as encryption keys. The variation in execution times typically arises from code that does not execute in constant time, often due to conditional branches or other operations that vary with input. By scrutinizing these timing discrepancies, attackers can identify patterns that expose hidden secrets within cryptographic and other sensitive algorithms.

Specialized microarchitectural attacks like Flush + Reload, Flush + Flush [9], Prime+Probe, and their advanced versions, use CPU cache mechanisms to secretly monitor and gather data. These complex techniques require a deep understanding of microarchitecture, a specialized skill that few experts possess. Due to the limited number of such experts and the widespread deployment of these systems, there is a significant challenge in adequately defending against these threats. To address this, we have developed a specialized Language Model (LLM) assistant. This LLM acts as a stand-in expert, offering real-time analysis and guidance for mitigating as well as in real-time detecting such attacks, based on the most current security research and deep insights into CPU architecture.

2.2 Large Language Models

Large Language Models (LLMs) such as GPT, LLaMa and Gemini are built on the foundation of the transformer architecture [27], which fundamentally relies on attention mechanism. The attention mechanism allows the model to dynamically focus on different parts of the input text, determining which aspects are most relevant for generating a response or making a prediction at any given moment. This is achieved through a series of calculations that assign weights to different input tokens, effectively allowing the model to "attend" to more important tokens more than others.

One of the core components of the attention mechanism is the self-attention module. This module enables the model to consider the entire input sequence simultaneously, contrasting with older models like RNNs [25] or LSTMs [10] that process inputs sequentially. This simultaneous processing allows for a better understanding of the context, as the output at each position can directly depend on the entire input sequence. The transformer also utilizes positional encodings to inject information about the order of tokens in the sequence, which is crucial since the model itself does not inherently process data in order.

2.2.1 Evaluating LLM's capability on System Security Analysis: Current ChatGPT models are trained on vast amounts of data, enabling them to provide a wealth of information and guidance on various topics. However, it's important to note that these models are not equipped for dynamic analysis of codes or real-time system monitoring for vulnerability and attack detection. As demonstrated in *ChatBox 1* and *ChatBox 2*, ChatGPT lacks the capability to actively detect ongoing attacks or assess the vulnerabilities in a code provided by the user. Nevertheless, ChatGPT can still provide valuable mitigations for known attacks based on established best practices.

To build on its current capabilities and also improve the areas where it is lacking, we have developed a customized LLM using GPT-3.5, referred to as μLAM. This model leverages the extensive training data of ChatGPT to provide not only information and guidance on various topics but also enhanced capabilities for detecting vulnerabilities and ongoing attacks in systems. While ChatGPT may not possess real-time monitoring abilities or dynamic code analysis, μLAM fills this gap by employing fine-tuning techniques to analyze time-sampled HPCs, system configurations, and code snippets. By utilizing the attention-based structure of LLMs, μLAM can detect subtle patterns and contextual anomalies in HPC data, enabling it to provide insights into system security that traditional models might overlook.

ChatBox 1: Attack detection

User: Hi! Can you tell me whether my system is currently undergoing any attacks?
The HPC values for all the processes currently executing parallel on my system are: [720, 6079, 6498, 640· · ·]
System configuration: 8th Generation Intel(R) Core(TM) 17-8700 CPU @ 3.20GHz
Code = [libgcrypt's RSA decryption code]
ChatGPT: It seems like you're asking about potential attacks on your system based on HPC values and the code being executed. However, without more context, it's challenging to determine if your system is undergoing any attacks solely based on the information provided.

ChatBox 2: Vulnerable Code detection and mitigation

User: I'm currently executing libgcrypt's RSA decryption code on my Intel 8th Generation system. Could you please check my code for vulnerabilities?
ChatGPT: I can't directly review or analyze specific code, but I can provide guidance on common vulnerabilities to watch out for when working with cryptographic libraries like libgcrypt's RSA decryption code.
User: Could you please provide some mitigations?
ChatGPT: Mitigate RSA decryption vulnerabilities by employing proper padding schemes, constant-time algorithms, rigorous input validation, secure key management, and protection against side-channel attacks. Regular security audits, protocol design considerations, and staying updated on cryptographic best practices are essential for robust RSA decryption code.

3 LLM-BASED APPROACH TO ENHANCE MICRO-ARCHITECTURAL SECURITY

In this section, we present our LLM-based microarchitectural assistant, which is built on the GPT-3.5 platform. We'll begin by outlining its extensive capabilities before diving into the intricacies of its design and the fine-tuning process. Key features of our model include:

❶ **Attack detection:** The user provides our μLAM assistant with the system configuration on which they are currently working, the HPC values of every process running on their system measured in sampling rate, and the sensitive code they are trying to run on their system. Depending on the HPC values given as input by the user, μLAM decides whether any attack is currently getting mounted on the user's system.

❷ **Vulnerable code detection:** The user submits the code they intend to execute on their system to μLAM. μLAM has the power to analyze the code, give a response to the user about whether the input code is vulnerable to any attack, and also identify the specific section of the code that might leak

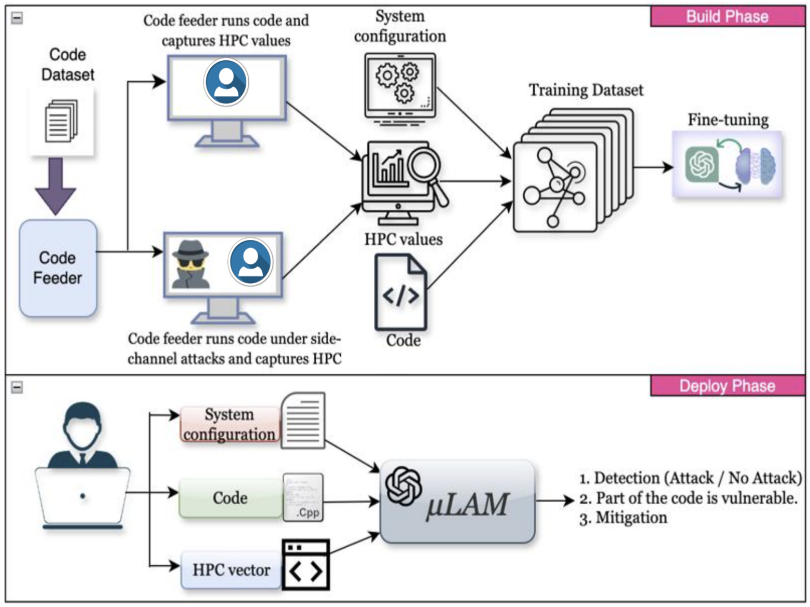


Figure 1: Build and Deploy Phases of *μLAM*

sensitive data when an attack is currently getting mounted on the user's system.

❸ **Attack mitigation:** On the basis of whether the code provided by the user is vulnerable or not, *μLAM* gives an output. If part of the code is vulnerable, then the assistant provides a mitigation.

3.1 *μLAM* Design

μLAM is a microarchitecture security assistant developed on top of OpenAI's GPT-3.5, designed to help in enhancing system security. This advanced tool is equipped to detect the presence of micro-architectural side-channel attacks, which are known to leak sensitive information from computer systems inadvertently. Beyond mere detection, *μLAM* offers a detailed analysis by identifying specific vulnerabilities within the system's code base. It also provides comprehensive mitigation strategies, tailored to each identified vulnerability, which enable users to implement effective countermeasures. The novel micro-architectural assistant is developed in two stages: ❶ build and ❷ deploy phase, shown in Figure 1.

3.1.1 ***Build phase:*** We have leveraged the OpenAI developer platform to construct and implement our virtual assistant. During the build phase, we fine-tuned the *gpt-3.5-turbo-0125* model, enabling it to point out the vulnerabilities in the user's code that could result in data breaches, suggest appropriate countermeasures, and also identify potential security attacks. To fine-tune the base model, it is necessary to generate a dataset comprising the necessary information. The dataset assembly will involve the following steps:

❶ **Collection of HPC Values:** This involves gathering data for various system processes under two distinct scenarios:

Scenario a. System under attack

Scenario b. No ongoing attacks on the system

❷ **System Configuration Details:** Documentation of the system configuration.

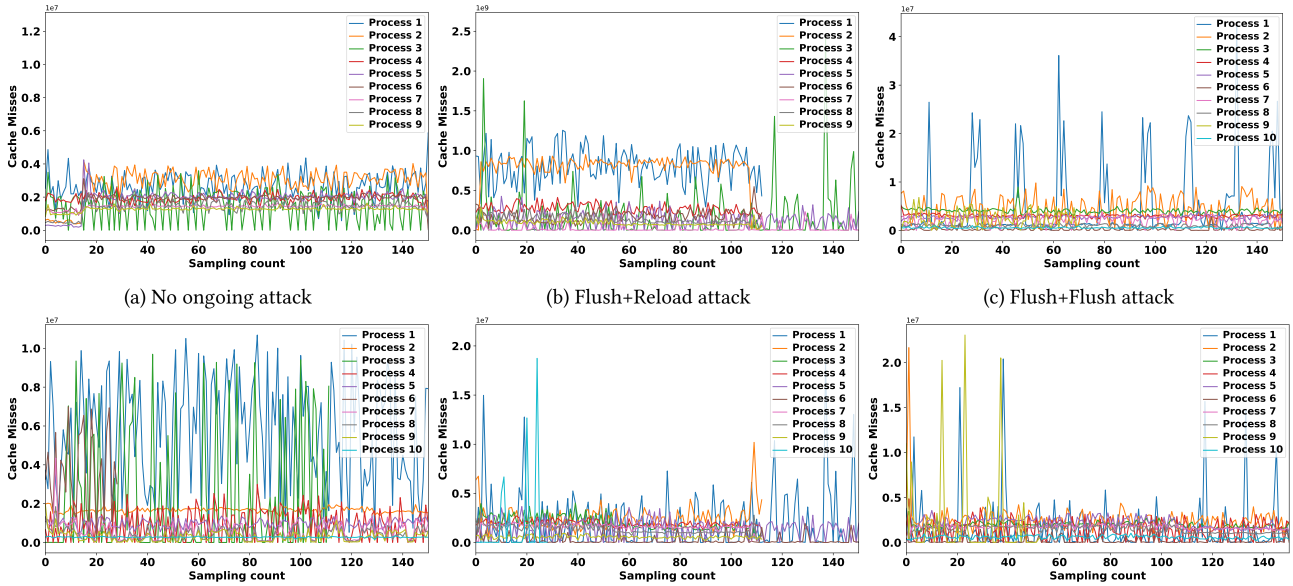
❸ **Sensitive Execution Code:** Acquisition of the code for the victim process that requires safeguarding.

Generation	Model Name	Code Name
13th Generation	Intel(R) Core(M) 17-13700	Raptor Lake
12th Generation	Intel(R) Core(M) 15-12500	Alder Lake
11th Generation	Intel (R) Core(TM) i5-11320H @ 3.20GHz	Tiger Lake
9th Generation	Intel(R) Core(M) 15-9500 CPU @ 3.00GHz	Coffee Lake
8th Generation	Intel(R) Core(TM) 17-8700 CPU @ 3.20GHz	Coffee Lake

Table 1: Processor details for systems used for HPC collection

In our study, we targeted cryptographic libraries, as they are crucial components in the realm of software development and security, serving as the backbone for implementing cryptographic algorithms that secure data. We executed the cryptographic routines found in libraries such as OpenSSL, NaCl, and libgcrypt, etc and we gathered the HPC metrics for all system processes that were running concurrently with these cryptographic operations. Data collection occurred under two distinct scenarios: one where the system was under attack, shown in Figures 2b-2f, and another where the system was operating normally without any attacks as shown in Figure 2a. We have gathered the HPC values from different Intel systems, as shown in Table 1. The dataset utilized for fine-tuning the model is a *jsonl* file that adheres to predefined patterns as provided below.

```
{ "messages":
[{"role": "system", "content": "You are a sophisticated
    and highly skilled chatbot designed to detect any
    side-channel attacks on a system, identify
    vulnerabilities in user-provided code, and suggest
    appropriate mitigations."},
{"role": "user", "content": "I have an Intel 12th Gen
    Alder Lake processor and I want to find out if any
    side-channel attack is being carried out on my
    system and also check the vulnerabilities?"},
```



(a) No ongoing attack

(b) Flush+Reload attack

(c) Flush+Flush attack

(d) Prime+Probe

(e) Umprime+Umprobe attack on 12th Gen

(f) Umprime+Umprobe attack on 13th Gen

Figure 2: Graphs illustrating deviations in HPC values when an attacker process is not executing (a) vs when it is executing (b-f).

```
{"role": "assistant", "content": "Sure! It seems you're
      interested in determining whether any side-channel
      attack is occurring, and to accurately assess that,
      it's necessary to have additional details such as
      HPC values and the relevant code Please provide the
      HPC values and the code that you are currently
      running; only then I can tell you."},
{"role": "user", "content":  "HPC values = [], Code = [],
       Is attack going on?"},
{"role": "assistant", "content": "Yes! Uprime + Umprobe
      attack is taking place currently in our system"},
{"role": "user", "content": "Why is the attack taking
      place?"},
{"role": "assistant", "content": "The primary cause of
      this attack is the Umonitor and Umwait instructions
      found in the newer Intel generations following
      Tremont and you are using Intel 12th Generation
      system", "weight": 1}]}
```

The aforementioned message serves as a single training instance designed to inform the *GPT* model about diverse micro-architectural attacks and corresponding countermeasures on different generations of systems. Likewise, we generate 250 different training instances.

Once we have our dataset, we then fine-tune the *GPT-3.5*. Fine-tuning a model like *GPT-3.5* involves adjusting its internal parameters based on the provided data. After the initial training, fine-tuning is done to tailor the model more closely to specific tasks or datasets. This involves continuing the training process but typically on a smaller, more specialized subset of data. Fine-tuning allows the model to perform better on tasks that may not have been well-represented in the original training data. After the fine-tuning phase of the model is complete, the next critical step involves constructing the micro-architectural assistant. This requires specifying key elements:

❶ **Name :** μLAM
❷ **Instructions:**

- Purpose: To detect potential side-channel attacks and identify vulnerabilities in user-submitted code, using HPC data and system configuration details.
- Input Requirements: System Configuration, HPCs, Source Code.
- Tasks: Analyze Performance Data, Scrutinize Code, Recommend Mitigations
- Output: Detailed report outlining detected vulnerabilities, potential impacts, and recommended mitigation strategies.

❸**Model:** *ft:gpt-3.5-turbo-0125:personal::9FEvcYz*
❹ **Functions**

```
   "name": "analyze_vulnerabilities",
   "description": "Analyze source code, hardware
     performance counter data, and system
     configuration to detect side-channel micro-
     architectural attacks and suggest mitigations.
     ", "parameters": {"type": "object","properties
     ": {"source_code": {"type": "string", "
     description": "The source code in a supported
     programming language."}, "performance_counters
     ": {"type": "object", "description": "
     Dictionary containing key performance counter
     metrics such as coach misses, branch misses",
     "additional properties": {"type": "array"}}, "
     system_config": {"type": "object","description
     ": "Dictionary detailing the system
     configuration, including CPU model.","
     properties": {"CPU": {"type": "string",
 "description": "Model of the CPU."}, "OS": {"type"
     : "string", "description": "Operating system
     installed on the system."},"required": ["
     source_code", "performance_counters", "
     system_config"]
```

After configuring the assistant parameters, our micro-architectural assistant is operational. Next, we move to the deployment phase, where users can pose questions to the assistant and receive answers.

3.1.2 ***Deploy Phase:*** In the deployment phase, there are three distinct steps: 1) Detecting the attack, 2) Identifying the vulnerable section of code, and 3) Providing the appropriate mitigation.

❶ **Attack detection:** The fine-tuning of our micro-architectural assistant leverages HPCs, system configurations, and user-provided code to identify potential micro-architectural side-channel attacks. The detection of potential attacks on the system primarily relies on analyzing the values from the HPCs and the system configuration.

The determination of whether an attack is underway can be influenced by the system configuration. Modern generations of Intel systems are equipped with countermeasures against Flush + Reload, Prime + Probe, and Flush + Flush attacks. Additionally, the Umprime + Umprobe attack is specifically applicable to these newer Intel generations, as the requisite umonitor and umwait instructions are absent in older models.

To detect side-channel micro-architectural attacks using HPCs the model analyzes the patterns and anomalies in HPC data that deviate from established baselines of normal operation. By fine-tuning on datasets comprising detailed logs of HPC values during both typical execution and simulated attacks, the model learns to identify specific signatures indicative of side-channel vulnerabilities, such as unusual cache miss rates, timing discrepancies, or abnormal execution paths. This helps the model to learn and detect whether any attack is getting mounted on the system.

Hyperparameter	Values
Base Model	*gpt-3.5-turbo-0125*
Batch size	1
Learning rate multiplier	0.001
Number of epochs	30

Table 2: Hyperparameters for fine-tuning GPT-3.5

❷ **Vulnerable code detection:**

For the detection of the part of the code that is vulnerable to micro-architectural side-channel attacks, we have fine-tuned the *GPT-3.5* model. This model has been specifically fine-tuned to identify vulnerable sections within a codebase, utilizing both pre-trained data and intrinsic model knowledge. In the realm of side-channel attacks, the control flow of code, particularly branching operations, is crucial as it can inadvertently leak sensitive information through physical channels. Such attacks exploit variations in execution time, power consumption, electromagnetic emissions, and other measurable side effects that differ based on the executed code path. The micro-architectural assistant, assists by pinpointing areas of the code that incorporate such control flows, potentially predisposing them to prevalent attacks like Flush+Reload, Flush+Flush, and Prime+Probe and extract sensitive data running on a user's system.

❸ **Attack Mitigation:** Our *μLAM* assistant provides a mitigation against that attack if it predicts that an attack is getting mounted on the user's system depending on the input given by the user. It leverages a pre-defined set of mitigation strategies, complemented by its inherent large language model (LLM) knowledge base, to offer real-time protective measures. This proactive approach allows the assistant to not only detect threats but also suggest tailored countermeasures to safeguard the user's system effectively. Our model assistant uses a combination of trained knowledge and it's own generative capabilities to provide mitigations.

4 EXPERIMENTAL RESULTS

In this section, we present an evaluation of *μLAM* focusing on its performance in detecting various cache-based attacks such as Flush+Reload, Flush+Flush, Prime+Probe, and the newly introduced Umprime+Umprobe. Notably, *μLAM* has been specifically fine-tuned to recognize these attacks, along with vulnerabilities stemming from speculative execution and branch prediction. Furthermore, we have fine-tuned *μLAM* using HPC data from five generations of Intel processors (cf. 1), including the latest Alderlake and Raptorlake processors. This comprehensive evaluation assesses *μLAM*'s effectiveness across systems with different configurations and processor architectures. Furthermore, we delve into how *μLAM* surpasses traditional analysis and detection tools, showcasing its advanced capabilities and efficacy in identifying and mitigating security threats.

As outlined in Section 3, our *μLAM* provides: (a) attack detection code, (b) vulnerability detection and (c) mitigation strategies. This section presents the experimental results, offering proof of concept for *μLAM*.*μLAM* starts by checking if the user's system is currently under attack by using the HPC values provided in the user's query. If an attack is detected, it alerts the user. Following this, the model examines the code and system configuration for vulnerabilities. If vulnerabilities are found, the model proceeds to mitigate them. Otherwise, it awaits further queries. This approach ensures immediate action in the event of an attack and proactive vulnerability assessment.

We illustrate the comprehensive functionality of our micro-architectural assistant, *μLAM*, through user interactions. These interactions showcase its capability to detect ongoing attacks, find vulnerabilities in source codes, and provide necessary mitigation.

User Interaction 1: Evaluation on Well-known Attacks

The first user interaction example (*ChatBox 3* and *ChatBox 4*) is a conversation between the user and our micro-architectural assistant, *μLAM*. These are the same queries which we had earlier made to ChatGPT (*ChatBox 1* and *ChatBox 2*) which proved unsuccessful in both detecting attacks and identifying vulnerabilities in the provided code.. In this interaction once again the user supplies *μLAM* with HPCs at the sampling rate for all processes currently running on their system, along with the sensitive code and system configuration. After analyzing the HPC values and system configurations provided by the user, our model successfully determines that the user's system is currently under a Flush+Reload attack as shown in *Chatbox 3*.

Further, *Chatbox 4* delineates the comprehensive interaction process between the user and *μLAM*, wherein the user inputs the code currently executing on their system along with the system specifications. Subsequently, the assistant identifies any vulnerabilities within the code. In this particular interaction, the user shares libgcrypt's RSA decryption code and the system configuration with *μLAM*, then asks it to evaluate if the code is vulnerable. *μLAM* responds that the code is vulnerable as Libgcrypt's RSA implementation relies on branches to manage the Montgomery ladder scalar multiplication. This makes it susceptible to side-channel attacks. In this kind of attack, an adversary leverages CPU cache-based timing to monitor memory accesses that reveal the execution path taken, specifically which branches were executed during scalar multiplication. By inspecting these accessed cache lines, the attacker can infer the values of bits in the scalar (private key), which ultimately leads to the compromise of the entire cryptographic system. The user then asked for the necessary mitigations, to which *μLAM* responded with recommendations such as cache flushing, constant-time implementations, introducing noise, and other countermeasures. Other than Libgcrypt's RSA we also successfully tested *μLAM* on both OpenSSL's and NaCl's RSA and ECC as well.

ChatBox 3: Attack detection

User: Hi! Can you tell me whether my system is currently undergoing any attacks?
The hardware performance counter values for all the processes currently executing parallel on my system are: [720, 6079, 6498, 640· · ·]. System configuration: 8th Generation Intel(R) Core(TM) 17-8700 CPU @ 3.20GHz
Code = [libgcrypt's RSA decryption code]
***μ*LAM:** Yes, currently the Flush+Reload attack is getting mounted on your system.

ChatBox 4: Vulnerable code detection and Mitigation

User: I'm currently executing libgcrypt's RSA decryption code on my Intel 8th Generation system. Could you please check my code for vulnerabilities?
***μ*LAM:** Libgcrypt's RSA implementation uses branches to control the Montgomery ladder scalar multiplication. The many side-channel attacks can distinguish which branch is taken by monitoring memory accesses. By probing the accessed memory lines, the attack can deduce the value of the bits in the scalar, thereby finally compromising the cryptosystem. Also vulnerable to other side-channel attacks.
User: Could you please provide some mitigations?
***μ*LAM:** Sure!! Cache flushing, addition of noise, random delays, and constant-time RSA implementations.

In this example, we demonstrate how *μLAM* effectively detects a Flush+Reload attack. Similarly, *μLAM* is also capable of detecting other side-channel attacks like Prime+Probe and Flush+Flush. These side-channel attacks exploit vulnerabilities in speculative execution, inadvertently exposing sensitive data through unintended channels. Speculative execution attacks, like Spectre and Meltdown, leverage transient execution paths to access sensitive data, with cache timing techniques serving as instrumental tools for data extraction during the attack process. As such, the utilization of cache timing in both Flush+Reload attacks and speculative execution attacks results in visibility of similar patterns in HPCs. By successfully identifying Flush+Reload attacks, *μLAM* demonstrates its proficiency in recognizing these characteristic patterns associated with cache-timing attacks, thus also capable of discerning speculative attacks which use cache-timing attacks as tools.

User Interaction 2: Evaluation on Recent Advanced Version of Known Attacks

In this user interaction example, we demonstrate *μLAM*'s capability to detect an newly introduced version of Prime+Probe attack called the *Umprime + Umprobe* attack. The following user interaction example provides a concise, end-to-end representation of how *μLAM* operates and detects the *Umprime + Umorobe* attack.

As demonstrated in *Chatbox 5* The user asks *μLAM* if any attack is currently active on their system, providing the necessary system configuration and HPC values for all processes running concurrently on their system. After examining the HPC values and system configurations, *μLAM* concludes that the user's system is undergoing a Umprime+Umprobe attack.

ChatBox 5: Attack detection

User: Hi! Can you tell me whether my system is currently undergoing any attacks?
System configuration: 12th Gen Intel(R) Core(TM) i512500
The hardware performance counter values for all the processes currently executing parallel on my system are: [9218, 66955, 8118, 9927, 17034 · · ·]
***μ*LAM:** Yes, currently Umprime + Umprobe attack is getting mounted on your system.

As represented in *Chatbox 6*, the user provides *μLAM* with a code. The code scenario is: that two processes are running on two sibling cores, both sharing a common variable. Process 1 reads the shared variable, waits for a while, and then process 2 modifies the variable. Subsequently, process 1 re-accesses the shared variable. By providing the code, the user inquires if *μLAM* identifies any vulnerabilities. *μLAM* detects that the code is vulnerable to Umprime + Umprobe attack, as according to the Umprime + Umprobe attack if process 1 accesses the shared variable after process 2's update, it triggers the cache coherency protocol, updating the variable. If there is a spy process executing in Core 1 that sets the monitor to the address of the shared variable the spy can set up a covert channel. The write by process 2 followed by process 1's access to the shared variable, wakes the processor from its sleep state, signaling a transmission of '0' from the sender to the receiver. If Core 1 does not access the shared variable after Core 2's update, the processor wakes up due to an operating system timeout. This event signals a transmission of '1' from the sender to the receiver. The user then queries for the mitigation. The user outputs that the primary cause of the Umprime+Umprobe attack is the presence of user-level instructions in newer Intel systems, thus specific mitigations do not currently exist. Umprime + Umprobe attack could only be mounted on Intel's new generation after Tremont. *μLAM* model is also capable of providing this granular information to the user, represented in *Chatbox 7*.

μLAM **vs Traditional Security Analysis Tools:**
In Table 3 we compare *μLAM* with the well-known detection tool *Medusa*, the advanced language model *ChatGPT*, *Machine Learning models*, and our micro-architectural assistant, *μLAM*. ***ChatGPT*** can assist the user by analyzing the problem and providing mitigations when the user's code is found to be vulnerable, but it does not have capabilities for direct vulnerability detection and attack detection. ***Medusa*** [22] and ***T-test*** can detect vulnerabilities by analyzing HPCs, simulating attacks, and providing insights into architectural vulnerabilities, but they cannot detect code vulnerabilities and provide necessary mitigations. ***ML models*** [2] can be effective in real-time attack detection, recognizing attack patterns, but it cannot detect code vulnerabilities and provide necessary

ChatBox 6: Vulnerable code detection and Mitigation

User: Please let me know if my code is vulnerable or not.

```
pthread_t t1, t2;
cpu_set_t cpu1, cpu2;
CPU_ZERO(&cpu1);
CPU_ZERO(&cpu2);
CPU_SET(coreid1, &cpu1);
CPU_SET(coreid2, &cpu2);
pthread_create(&t1, NULL, &proc1_func, NULL);
pthread_setaffinity_np(t1, sizeof(cpu_set_t), &cpu1);
sleep(1); pthread_create(&t2, NULL, &proc2_func, NULL);
pthread_setaffinity_np(t2, sizeof(cpu_set_t), &cpu2);
void *proc1_func()
    pthread_mutex_lock(&mutex);
    while (thread_to_execute != 1)
        pthread_cond_wait(&condition, &mutex);
    shared_variable[0] = shared_variable[0] + 1;
    thread_to_execute = 2;
    pthread_cond_signal(&condition);
    pthread_mutex_unlock(&mutex);
    pthread_mutex_lock(&mutex);
    while (thread_to_execute != 4)
        pthread_cond_wait(&condition, &mutex);
    y = shared_variable[0];
    thread_to_execute = 5;
    pthread_cond_signal(&condition);
    pthread_mutex_unlock(&mutex);
void *proc2_func()
    pthread_mutex_lock(&mutex);
    while (thread_to_execute != 2)
        pthread_cond_wait(&condition, &mutex);
    shared_variable[0] = shared_variable[0] + 1;
    shared_variable[0] = shared_variable[0] + 100;
    thread_to_execute = 3;
    pthread_cond_signal(&condition);
    pthread_mutex_unlock(&mutex);
```

and my system configuration is 12th Gen Intel(R) Core(TM) i512500.

μLAM: Yes!! The provided code is vulnerable.
The code provided involves two processes running on separate cores, sharing a variable, and accessing it sequentially. This part is potentially vulnerable to Umprime and Umprobe attacks.

User: Please suggest mitigation.

μLAM: Currently, there's no mitigation for the Umprime and Umprobe attacks; it relies on the umonitor and umwait instructions found in Intel's newer generation processors.

ChatBox 7: Attack detection

User: Hi! Can you tell me whether my system is currently undergoing Umprime+Umprobe attack?
System configuration: 8th Intel(R) Core(TM) 17-8700 CPU @ 3.20GHz. The hardware performance counter values for all the processes currently executing parallel on my system are: [10291, 11234, 9289, 11928, 10829 $\cdots$]

μLAM: The Umprime+Umprobe attack relies on user-level instructions found only in the newer generations of Intel processors. Therefore, this attack could not be executed on your system.

mitigations. On the other hand, *μLAM* is empowered with the capabilities to detect code vulnerabilities, provide proper mitigations, and attack detection.

	ChatGPT	*Medusa*	*ML models*	*T-test*	*μLAM*
Code Vulnerability Detection	✗	✗	✗	✗	✓
Mitigation	✓	✗	✗	✗	✓
Attack Detection	✗	✓	✓	✓	✓

Table 3: Comparing the performance features of *ChatGPT, Medusa, ML models* and *T-test* with *μLAM*

We evaluated *μLAM* with nearly 1000 distinct queries, some of which are demonstrated within this paper. Similar to other language models, *μLAM* exhibits sensitivity to the phrasing and context of queries but still provides correct responses in approximately 96% of cases. Moreover, we found that by imposing additional constraints on the language used in queries, we can achieve accuracy levels approaching 100%. Our micro-architectural assistant *μLAM* stands out as a comprehensive solution, equipped to address the limitations found in existing approaches. It excels at detecting code vulnerabilities, providing necessary mitigations, and detecting attacks, thus delivering a versatile framework for enhancing system security.

5 DISCUSSION AND CONCLUSION

In this work, we make a significant contribution to the field of microarchitectural security by introducing a dynamic microarchitecture assistant using Large Language Models (LLMs). The introduced tool, named *μLAM*, leverages OpenAI's GPT 3.5 to not only identify vulnerable code segments but also generate custom mitigations tailored to specific system setups and existing security measures. Furthermore, it utilizes dynamic HPCs to detect ongoing attacks, enhancing its responsiveness to immediate threats. The fine-tuning process, facilitated by a comprehensive training dataset collected from multiple systems of different architectural configurations, enables *μLAM* to learn the intricacies of microarchitectural attacks, making it capable of processing and responding to real-time data effectively. This robust, context-aware LLM assistant far surpasses traditional security analysis tools in its ability to detect and mitigate microarchitectural attacks. Importantly, *μLAM* stands out as the first tool to combine all these capabilities into a single, comprehensive solution that can be even used by people who are not experts in microarchitecture security.

While our current research demonstrates the effectiveness of fine-tuning a GPT3.5 model for microarchitectural security, there are still many things we don't completely understand about the process. We are particularly interested in how fine-tuning impacts the internal workings of the model, such as which specific parts of the LLM architecture are modified and characteristics of our training data, such as its diversity and complexity, shape these changes. Unfortunately, GPT models are often considered black-box systems, making it difficult to directly explore these questions with GPT3.5. However, gaining insights into these aspects is crucial for further optimizing our model's fine-tuning process and improving its performance. Therefore, in future, we are eager to delve deeper into these questions using open-source Large Language Models (LLMs) like LLama, which offer more transparency. By doing so, we hope to uncover valuable insights that will not only enhance our own model but also contribute to a better understanding of fine-tuning mechanisms in LLMs for microarchitecture security more broadly.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their suggestions for improving the paper. They would also like to thank the Information Security Education and Awareness (ISEA) organization for partially funding this research.

REFERENCES

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*. Springer, 225–242.

[2] Manaar Alam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. 2021. Victims can be saviors: a machine learning–based detection for micro-architectural side-channel attacks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 17, 2, 1–31.

[3] Ruxandra Bălucea and Paul Irofti. 2023. Software mitigation of risc-v spectre attacks. In *International Conference on Information Technology and Communications Security*. Springer, 51–64.

[4] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings 8*. Springer, 201–215.

[5] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 715–733.

[6] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. 2018. Cacheshield: detecting cache attacks through self-observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 224–235.

[7] Yupeng Chang et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15, 3, 1–45.

[8] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: defeating cache side-channel protections with {tlb} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, 955–972.

[9] Daniel Gruss, Clémentine Maurice, and Klaus Wagner. 2015. Flush+flush: A stealthier last-level cache attack. *CoRR*, abs/1511.04594. http://arxiv.org/abs/1511.04594 arXiv: 1511.04594.

[10] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory (lstm). *Neural Computation*, 9, 8, 1735–1780.

[11] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2018. MASCAT: preventing microarchitectural attacks before distribution. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*. Ziming Zhao, Gail-Joon Ahn, Ram Krishnan, and Gabriel Ghinita, (Eds.) ACM, 377–388. DOI: 10.1145/3176258.3176316.

[12] Marc Joye and Michael Tunstall. 2007. Securing openssl against micro-architectural attacks. In *SECRYPT*, 189–196.

[13] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42, 3, 361–372.

[14] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, 388–397.

[15] Paul Kocher et al. 2019. Spectre attacks: exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1–19. DOI: 10.1109/SP.2019.00002.

[16] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. [n. d.] Sok: can we really detect cache side-channel attacks by monitoring performance counters?

[17] Doowon Lee, Opeoluwa Matthews, and Valeria Bertacco. 2018. Low-overhead microarchitectural patching for multicore memory subsystems. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 17–25.

[18] Tamara Silbergleit Lehman, Andrew D Hilton, and Benjamin C Lee. 2016. Poisonivy: safe speculation for secure memory. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[19] Moritz Lipp et al. 2018. Meltdown: reading kernel memory from user space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. William Enck and Adrienne Porter Felt, (Eds.) USENIX Association, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[20] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. Catalyst: defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 406–418.

[21] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 605–622. DOI: 10.1109/SP.2015.43.

[22] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. 2020. Medusa: microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Srdjan Capkun and Franziska Roesner, (Eds.) USENIX Association, 1427–1444. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa.

[23] OpenAI. 2022. ChatGPT. https://openai.com/chatgpt. Version 3.5. (2022).

[24] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. {Drama}: exploiting {dram} addressing for {cross-cpu} attacks. In *25th USENIX security symposium (USENIX security 16)*, 565–581.

[25] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Recurrent neural network (rnn). *Nature*, 323, 533–536.

[26] Chao Su and Qingkai Zeng. 2021. Survey of CPU cache-based side-channel attacks: systematic analysis, security models, and countermeasures. *Secur. Commun. Networks*, 2021, 5559552:1–5559552:15. DOI: 10.1155/2021/5559552.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, (Eds.), 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[28] Wenjie Xiong and Jakub Szefer. 2022. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 54, 3, 54:1–54:36. DOI: 10.1145/3442479.

[29] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Kevin Fu and Jaeyeon Jung, (Eds.) USENIX Association, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[30] Jiliang Zhang, Congcong Chen, Jinhua Cui, and Keqin Li. 2024. Timing side-channel attacks and countermeasures in cpu microarchitectures. *ACM Computing Surveys*.