POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Adversary Virtual Platform in embedded system environment for RED TEAM

Supervisors

Prof. SAVINO ALESSANDRO

Prof. STEFANO DI CARLO

Dott. FRANCO OBERTI

Candidate

Ismail SANAN

March 2024

Abstract

Nowadays The embedded systems in technological landscape has profound implications for companies, as these systems form the backbone of various critical operations. However, their widespread adoption also exposes companies to a myriad of security risks, necessitating robust measures to safeguard against potential attacks. This thesis explores the implementation of Adversary Virtual Platforms (AVPs) within embedded systems, focusing on the Cortex-M3 architectures simulated on the gem5 ARM platform. The primary objective is to demonstrate the feasibility of conducting adversarial attacks in such environments, with a specific emphasis on the Spectre attack. Leveraging gem5 ARM for virtual platform emulation, coupled with Kotana for pipeline visualization, the research successfully simulates and validates the Spectre attack. The findings underscore the significance of AVPs for assessing security vulnerabilities in embedded systems and offer insights into the efficacy of defense mechanisms against adversarial threats. By advancing the understanding of adversarial techniques in embedded system environments, this research enhances Red Team operations and fortifies cybersecurity strategies, thus enabling companies to better mitigate the risks associated with the use of embedded systems.

Acknowledgements

This thesis is the result of not only my own efforts, but also the support and encouragement of the people who have been by my side throughout this journey. I owe my deepest gratitude to my mom and dad, who have always believed in me and supported every step I have taken. I am equally grateful to my brothers Mostafa and Ahmad, and to my sister Maya, for the strong bond we share. Their love and encouragement have lifted me through difficult times and made the joyful moments even more meaningful. They have shaped not only this academic milestone but also the person I continue to be.

I would also like to thank my friends Meriem,leonardo,vicenzo, Hadi and Nadim who have stood by me through it all. Their encouragement and friendship have been a constant source of strength, and the memories we share remind me that this journey was never one which I walked alone.

Table of Contents

Chapter 1

General Introduction

1.1 Introduction to Embedded Systems

The evolution and role of embedded systems in technology can be traced to various sources about the history of embedded systems [1]. From consumer electronics to industrial automation, these systems play a pivotal role in driving innovation and efficiency. Embedded systems are specialized computing devices designed to perform specific functions within a larger system. Unlike general-purpose computers, which are versatile and adaptable, embedded systems are optimized for dedicated tasks, often operating in real-time environments with resource constraints.

1.1.1 Evolution and Proliferation of Embedded Systems

The evolution of embedded systems dates back to the early days of computing. Over time, advances in semiconductor technology and miniaturization have led to the development of smaller and more powerful embedded devices. The rise of embedded systems has been boosted by several factors, including the growing demand for smart devices, the rise of the Internet of Things (IoT), and the convergence of hardware and software technologies.

Embedded systems have undergone a significant transformation since their inception, which can be traced back to the initial advancements in computing and automation. As early as the 20th century, the concept of automating tasks through preprogrammed instructions began to take shape, laying the groundwork for the development of more advanced embedded systems in the subsequent decades.

The widespread acceptance of embedded systems in daily items was greatly influenced by the increase in consumer electronics during the second half of the twentieth century [2]. Embedded systems, integrated into a variety of devices, ranging from household appliances to personal gadgets, became essential elements, improving their functionality, ease of use, and convenience for users. Technological

advances such as component miniaturization, progress in semiconductor technology, and economies of scale played a role in making computing power widely available to the general public.

Embedded systems played an important role in the transformation of the automotive and transportation sectors [3], revolutionizing vehicle design, performance, safety, and efficiency. Electronic Control Units (ECUs) and in-vehicle networks enabled advanced features such as engine management, fuel injection, anti-lock braking, airbag deployment, navigation, and infotainment. Embedded systems facilitated the transition from mechanical control systems to electronic control systems, paving the way for autonomous vehicles, electric vehicles, connected cars, and smart transportation systems.

The integration of embedded systems into medical devices, diagnostic equipment, and patient monitoring systems triggered a revolutionary shift in the healthcare sector [4]. Implantable devices like pacemakers, defibrillators, and insulin pumps relied heavily on embedded controllers to administer treatment, monitor vital signs, and ensure patient safety. In addition, portable devices such as blood glucose meters, pulse oximeters, and electrocardiographs enabled patients to actively engage in managing their health, addressing chronic diseases, and improving their overall quality of life. Beyond individual devices, the widespread adoption of embedded systems facilitated significant advancements in healthcare delivery. By enabling remote monitoring, telemedicine, and personalized medicine, embedded systems have fundamentally transformed the way healthcare is accessed and provided. Patients now have greater access to care regardless of geographical constraints, and healthcare professionals can deliver tailored treatments and interventions based on real-time data, ultimately improving patient outcomes and enhancing the overall efficiency of healthcare delivery systems .

Advancement in aerospace and defence

Embedded systems play a critical role in advancing aerospace and defense technologies [5], serving as the backbone for avionics, navigation, surveillance, weapons, and unmanned aerial vehicles (UAVs). They provide real-time control, monitoring, and communication functions critical for ensuring mission success and safety in aircraft, spacecraft, and military systems .

The aerospace and defense sectors demand highly reliable, durable, performant, and secure embedded systems. This requirement drives continuous innovation in embedded hardware and software, aimed at withstanding extreme conditions, maintaining operational integrity, and protecting sensitive data .

Integration of embedded systems enables advanced functionalities like autonomous flight control, precision navigation, target acquisition, and real-time data analysis, significantly enhancing aerospace and defense capabilities. Overall,

embedded systems not only reshape aerospace and defense technologies but also contribute significantly to national security and scientific exploration beyond Earth.

Emergence of the Internet of Things (IoT)

The emergence of the Internet of Things (IoT) introduced a new era of connectivity [6], where embedded systems could communicate, collaborate, and share data over networked environments . IoT ecosystems encompassed a vast array of devices, sensors, actuators, and gateways, interconnected through wired and wireless networks. Embedded systems powered IoT devices such as smart sensors, connected appliances, industrial machines, environmental monitors, and smart city infrastructure, enabling automation, optimization, and intelligence in diverse domains.

The shift towards edge computing architectures redefined the role of embedded systems in processing and analyzing data at the network edge, closer to the point of data generation and consumption. Edge devices, equipped with embedded processors, memory, and connectivity, performed data processing, analytics, and decision-making in real-time, reducing latency, bandwidth consumption, and reliance on centralized cloud infrastructure. Edge computing enabled embedded systems to deliver low-latency responses, autonomous operation, and adaptive behavior in dynamic environments such as industrial automation, autonomous vehicles, smart grids, and remote monitoring applications.

Future Trends in Embedded Systems and its cybersecurity concerns

The integration of AI and its capabilities into embedded systems will unlock new possibilities for intelligent decision-making, predictive analytics, and adaptive behavior. Embedded AI algorithms will enable features such as predictive maintenance, anomaly detection, natural language processing, computer vision, and autonomous control, enhancing the intelligence, efficiency, and autonomy of embedded systems across various applications.

Advancements in wireless communication technologies, such as 5G, Wi-Fi 6, and LPWANs, will enable high-speed, low-latency connectivity for embedded systems, facilitating real-time data exchange, immersive experiences, and mission-critical applications. Enhanced connectivity will support applications such as remote monitoring, augmented reality (AR), virtual reality (VR), autonomous vehicles, and smart infrastructure, driving innovation and connectivity in the digital ecosystem.

With the increasing connectivity and complexity of embedded systems, cybersecurity and trustworthiness will be paramount concerns for ensuring the integrity [7], confidentiality, and availability of data and services. Embedded systems will incorporate hardware-based security features, secure boot mechanisms, cryptographic

algorithms, and intrusion detection/prevention systems to mitigate cyber threats and ensure system resilience in the face of evolving attacks.

The demand for customized, domain-specific embedded solutions will continue to grow as industries seek tailored solutions to address their unique requirements, challenges, and opportunities. Embedded systems will be designed and optimized for specific applications such as industrial automation, healthcare, automotive, aerospace, defense, smart agriculture, environmental monitoring, and smart infrastructure, leveraging domain expertise, technology integration, and co-creation partnerships.

1.1.2 Importance and Applications of Embedded Systems

The importance of embedded systems in modern society cannot be overstated. These systems are integral to a wide range of applications across various industries, including healthcare, automotive, aerospace, telecommunications, and manufacturing. In healthcare, embedded systems power medical devices such as pacemakers, insulin pumps, and diagnostic equipment, facilitating patient monitoring, diagnosis, and treatment. In the automotive industry, embedded systems control critical functions such as engine management, navigation, and driver assistance systems,enhancing vehicle safety and performance

Embedded systems play a vital role in modern technology, powering a wide range of devices and applications across various industries. Their importance stems from several key factors:

Ubiquity: Embedded systems are ubiquitous in everyday life, embedded within numerous consumer electronics, industrial equipment, automotive systems, medical devices, and smart appliances. From smartphones and smartwatches to industrial robots and IoT devices, embedded systems are integral to the functioning of countless products and services.

Functionality: Embedded systems provide essential functionality and control capabilities in diverse application domains. They enable automation, monitoring, communication, and control of physical processes, systems, and devices. Embedded systems are essential for implementing critical functions such as sensing, actuation, data processing, communication, and user interaction.

Efficiency: Embedded systems are designed for efficiency, optimizing resource utilization, power consumption, and performance to meet the requirements of the target application. They often operate in resource-constrained environments with limited processing power, memory, energy, and space, requiring careful optimization of hardware and software components.

Reliability: Embedded systems are engineered for reliability, robustness, and resilience, particularly in safety-critical and mission-critical applications. They undergo rigorous testing, validation, and certification to ensure compliance with

stringent reliability and safety standards. Reliability is paramount in applications such as automotive systems, aerospace, medical devices, and industrial control.

Innovation: Embedded systems drive innovation by enabling the development of new products, services, and business models. They support advances in technology such as IoT, artificial intelligence (AI), machine learning (ML), edge computing, and autonomous systems. Embedded systems facilitate the integration of advanced features, connectivity, and intelligence into a wide range of devices and applications.

Embedded systems find applications across various industries and sectors, each with unique requirements, challenges, and opportunities:

Consumer Electronics: In the consumer electronics industry, embedded systems power devices such as smartphones, tablets, smart TVs, gaming consoles, digital cameras, wearable devices, and smart home appliances. These systems provide user interfaces, connectivity, multimedia capabilities, and advanced features such as voice recognition, facial recognition, and augmented reality. Automotive and Transportation: Embedded systems play a critical role in automotive and transportation systems, controlling functions such as engine management, vehicle dynamics, infotainment, driver assistance, navigation, telematics, and vehicle-to-everything (V2X) communication. Embedded systems enable safety features such as anti-lock braking systems (ABS), electronic stability control (ESC), adaptive cruise control, and collision avoidance systems.

Industrial Automation: In industrial automation and manufacturing, embedded systems are used in programmable logic controllers (PLCs), distributed control systems (DCS), supervisory control and data acquisition (SCADA) systems, robotics, motion control, and process automation. Embedded systems optimize production processes, improve efficiency, monitor equipment health, and enable predictive maintenance.

Healthcare and Medical Devices: Embedded systems are integral to healthcare and medical devices, including patient monitors, infusion pumps, medical imaging systems, diagnostic equipment, wearable health monitors, and implantable devices. These systems enable remote patient monitoring, diagnosis, treatment, and personalized healthcare delivery.

Aerospace and Defense: In aerospace and defense applications, embedded systems control avionics, navigation systems, flight control systems, communication systems, radar systems, unmanned aerial vehicles (UAVs), and military equipment. Embedded systems ensure the safety, reliability, and performance of aircraft, spacecraft, missiles, and defense systems. Energy and Utilities: Embedded systems are used in energy and utility infrastructure for monitoring, control, and optimization of power generation, transmission, distribution, and consumption. They enable smart grid technologies, renewable energy integration, demand response, energy management, and grid stability.

Smart Cities and Infrastructure: In smart cities and infrastructure projects,

embedded systems support applications such as intelligent transportation systems (ITS), smart meters, traffic management, public safety, environmental monitoring, waste management, and building automation. These systems enhance urban efficiency, sustainability, and quality of life.

Looking ahead, several trends and emerging technologies are expected to shape the future of embedded systems:

Internet of Things (IoT): The proliferation of IoT devices and connected systems will drive demand for embedded systems with enhanced connectivity, intelligence, and interoperability. IoT ecosystems will encompass a wide range of devices, sensors, actuators, and gateways, generating massive amounts of data for analysis and decision-making.

Artificial Intelligence (AI) and Machine Learning (ML): Integration of AI and ML capabilities into embedded systems will enable intelligent decision-making, predictive analytics, and autonomous operation.

Embedded AI/ML algorithms will support functions such as object recognition, anomaly detection, predictive maintenance, natural language processing, and adaptive control.

Edge Computing: The adoption of edge computing architectures will decentralize computation and data processing, moving computing resources closer to the source of data generation. Edge devices and gateways will host embedded systems capable of real-time processing, local decision-making, and reduced latency for time-sensitive applications.

5G and Wireless Technologies: The rollout of 5G networks and advancements in wireless communication technologies will enable high-speed, low-latency connectivity for embedded systems. 5G-enabled embedded systems will support applications such as real-time video streaming, augmented reality (AR), virtual reality (VR), remote monitoring, and mission-critical communications. Cybersecurity and Safety.

Biomedical and Wearable Devices: Embedded systems will continue to advance biomedical and wearable devices for healthcare monitoring, diagnostics, and personalized medicine. Miniaturized, low-power embedded systems will support continuous health monitoring, disease management, drug delivery, and rehabilitation, empowering individuals to take control of their health.

Sustainable and energy-efficient design principles will drive innovation in embedded systems, promoting resource conservation, environmental responsibility, and energy efficiency. Embedded systems will incorporate power-saving features, low-power components, energy harvesting techniques, and intelligent power management algorithms to optimize energy consumption, extend battery life, and minimize environmental impact in battery-operated and energy-constrained applications

1.1.3 Security Challenges in Embedded Systems

Embedded systems are characterized by their heterogeneity, with a diverse array of hardware architectures, operating systems, communication interfaces, and application domains. This heterogeneity complicates security management and interoperability, as each embedded device may have unique security requirements and compatibility constraints. Interconnecting heterogeneous embedded systems in IoT ecosystems further increases the attack surface and introduces new security risks, such as protocol vulnerabilities, misconfigurations, and unauthorized access points. Standardizing security protocols, communication interfaces, and interoperability frameworks can help mitigate these challenges but requires coordination among industry stakeholders and regulatory bodies.

One of the primary challenges in securing embedded systems is their inherent resource constraints. Embedded devices often operate with limited computational power, memory, storage, and energy resources, making it challenging to implement sophisticated security mechanisms. Cryptographic operations, secure communication protocols, and intrusion detection mechanisms can impose a significant overhead on system resources, leading to performance degradation and increased power consumption. Balancing security requirements with resource constraints is a delicate trade-off that requires careful consideration and optimization.

Legacy System and software

Many embedded systems deployed in the field are based on legacy hardware and software platforms that lack built-in security features and receive limited or no firmware updates. These legacy systems may contain known vulnerabilities, deprecated cryptographic algorithms, and obsolete protocols that are susceptible to exploitation by attackers. Retrofitting security measures onto legacy systems can be complex and costly, requiring modifications to hardware components, firmware updates, and backward compatibility testing. Furthermore, the long lifecycle of embedded devices, particularly in industrial, automotive, and aerospace applications, exacerbates the challenge of maintaining security over extended periods.

Lack of secure boot and vulnerabilities

Many embedded systems lack secure boot mechanisms and firmware integrity checks, making them vulnerable to bootloader exploits, rootkits, and firmware tampering. Without secure boot, attackers can compromise the initial boot process, inject malicious code into the firmware, and gain persistent control over the device. Similarly, the absence of firmware integrity checks allows attackers to modify firmware images, inject malware payloads, and bypass authentication mechanisms without detection. Secure boot and firmware integrity verification are essential

security features that protect embedded systems against unauthorized modifications and ensure the trustworthiness of the boot process.

They often rely on communication channels such as wired interfaces (e.g., Ethernet, USB) and wireless protocols (e.g., Wi-Fi, Bluetooth, Zigbee) to exchange data with external devices and networks. These communication channels are susceptible to eavesdropping, man-in-the-middle attacks, packet sniffing, replay attacks, and unauthorized access. Insecure communication protocols, weak encryption algorithms, and inadequate authentication mechanisms can compromise the confidentiality and integrity of data transmitted by embedded systems. Implementing secure communication protocols, encryption algorithms, and mutual authentication mechanisms can mitigate the risks associated with vulnerable communication channels.

Physical security is often overlooked in embedded systems, despite being a critical aspect of overall system security. Many embedded devices are deployed in uncontrolled or hostile environments where they are exposed to physical tampering, theft, vandalism, and environmental hazards. Attackers can gain unauthorized access to embedded systems by bypassing physical barriers, extracting sensitive information from memory chips, and tampering with hardware components. Physical security measures such as tamper-evident seals, enclosure locks, intrusion detection sensors, and secure mounting can help deter and detect unauthorized access attempts, safeguarding embedded systems against physical attacks.

The globalized nature of the supply chain introduces inherent risks to the security of embedded systems. Manufacturers often source components, subsystems, and software modules from third-party suppliers and subcontractors, increasing the risk of counterfeit components, malicious implants, and supply chain attacks. Counterfeit components may contain malicious firmware, backdoors, or hardware trojans that compromise the security and reliability of embedded systems. Supply chain security practices such as vendor vetting, component authentication, tamper-evident packaging, and secure distribution channels can mitigate the risks associated with counterfeit components and supply chain attacks.

Embedded systems deployed in safety-critical, mission-critical, or regulated industries such as automotive, healthcare, aerospace, and defense must comply with industry-specific security standards, regulations, and certification requirements. Achieving regulatory compliance and certification entails rigorous testing, documentation, audit trails, and third-party assessments to demonstrate adherence to security best practices and industry standards. Failure to meet regulatory requirements can result in legal liabilities, financial penalties, reputational damage, and loss of market competitiveness. Furthermore, regulatory compliance may impose additional overhead on embedded system development, leading to increased costs and time-to-market pressures

Embedded systems undergo a lifecycle comprising design, development, manufacturing, deployment, operation, maintenance, and end-of-life phases, each presenting unique security challenges. Secure lifecycle management involves implementing security controls and best practices at each stage of the embedded system lifecycle to mitigate risks and ensure continuous protection against evolving threats. End-of-life issues such as software vulnerabilities, unsupported hardware, and lack of security patches can leave embedded systems vulnerable to exploitation long after they have been retired from service. Proper disposal, decommissioning, and secure erasure of sensitive data are essential to prevent unauthorized access and data breaches during the end-of-life phase

Human factors and social engineering pose significant security risks to embedded systems, as attackers exploit human vulnerabilities such as ignorance, carelessness, and trust to bypass technical security measures. Phishing attacks, social engineering scams, insider threats, and human error can compromise the confidentiality, integrity, and availability of embedded systems by tricking users into disclosing sensitive information, clicking on malicious links, or performing unauthorized actions. Security awareness training, user education, role-based access control, and multi-factor authentication can help mitigate the risks associated with human factors and social engineering attacks

Embedded systems face sophisticated and persistent threats from advanced adversaries, including nation-state actors, cybercriminal organizations, and hacktivist groups. Advanced Persistent Threats (APTs) targeting embedded systems involve stealthy, long-term campaigns aimed at infiltrating, persisting, and exfiltrating sensitive information from targeted organizations. Nation-state actors may conduct cyber espionage, sabotage, or disinformation campaigns against critical infrastructure, government agencies, military organizations, and industrial facilities using advanced techniques such as zero-day exploits, supply chain attacks, and targeted malware. Detecting and mitigating APTs require advanced threat intelligence, network monitoring, anomaly detection, and incident response capabilities to identify and neutralize malicious activities before they cause significant harm.

1.1.4 The Cortex-M3 Processor Architecture

At the heart of many embedded systems lies the Cortex-M3 processor, a versatile and energy-efficient architecture developed by ARM Holdings. The Cortex-M3 is widely used in microcontroller applications due to its high performance, low power consumption[8]. Featuring a 32-bit RISC architecture, the Cortex-M3 combines high performance with low power consumption, making it ideal for battery-powered and resource-constrained devices. With features such as a Harvard architecture, single-cycle multiply-accumulate (MAC) instruction, and efficient interrupt handling, the Cortex-M3 offers a balance of performance and power efficiency, making it

well-suited for a wide range of embedded applications.

The Role of the Cortex-M3 Processor Architecture

The Cortex-M3 processor architecture, developed by ARM Holdings, represents a significant advancement in the field of embedded systems design. Renowned for its versatility, efficiency, and scalability, the Cortex-M3 architecture has become the architecture of choice for a wide range of embedded applications. At its core, the Cortex-M3 architecture embodies a set of design principles aimed at optimizing performance, energy efficiency, and ease of development.

Performance Optimization

The Cortex-M3 architecture is characterized by its 32-bit Reduced Instruction Set Computing (RISC) design philosophy, which prioritizes simplicity, efficiency, and predictability in instruction execution. Unlike Complex Instruction Set Computing (CISC) architectures, which feature a large and diverse instruction set, RISC architectures such as Cortex-M3 streamline instruction execution by focusing on a small set of fundamental operations. This approach enables faster instruction decoding, execution, and pipelining, resulting in improved overall performance for embedded applications.

Furthermore, the Cortex-M3 architecture incorporates advanced features such as single-cycle multiply-accumulate (MAC) instructions, barrel shifters, and efficient branch prediction mechanisms, which further enhance computational throughput and reduce latency. These features make the Cortex-M3 architecture well-suited for computationally intensive tasks commonly found in embedded applications, such as digital signal processing, control algorithms, and data processing.

Energy Efficiency

In addition to performance considerations, the Cortex-M3 architecture prioritizes energy efficiency, making it ideal for battery-powered and energy-constrained embedded devices. By employing a combination of architectural optimizations, low-power modes, and dynamic voltage and frequency scaling (DVFS) techniques, the Cortex-M3 processor minimizes energy consumption without compromising performance.

Key features contributing to energy efficiency include:

Power gating: The Cortex-M3 architecture incorporates power gating mechanisms that selectively power down unused functional units and peripheral interfaces, reducing static power consumption during idle periods. Clock gating: Dynamic clock gating techniques are employed to disable clock signals to unused logic blocks and peripherals, further reducing dynamic power consumption during low-utilization

scenarios. Sleep modes: The Cortex-M3 architecture supports multiple low-power sleep modes, allowing the processor to enter a low-power state when not actively processing instructions. Sleep modes enable significant energy savings by reducing clock frequency and disabling non-essential peripherals while maintaining system state integrity. By optimizing energy consumption at both the architectural and microarchitectural levels, the Cortex-M3 architecture enables embedded system designers to extend battery life, increase system uptime, and reduce operating costs in power-constrained environments.

Scalability and Flexibility

One of the key strengths of the Cortex-M3 architecture is its scalability and flexibility, allowing it to address a wide range of embedded applications spanning various market segments and performance requirements. The Cortex-M3 processor is available in a range of configurations, including single-core, multi-core, and asymmetric multi-core variants, each tailored to specific application domains.

Furthermore, the Cortex-M3 architecture supports a rich ecosystem of development tools, software libraries, and middleware components, facilitating rapid prototyping, software development, and system integration. This ecosystem includes integrated development environments (IDEs), compiler toolchains, debuggers, and simulation environments, empowering developers to efficiently design, debug, and optimize embedded software for Cortex-M3-based systems.

Additionally, the Cortex-M3 architecture offers extensive support for industry-standard interfaces and protocols, including inter-integrated circuit (I2C), serial peripheral interface (SPI), universal asynchronous receiver-transmitter (UART), and controller area network (CAN), enabling seamless integration with external sensors, actuators, and communication modules.

Security Features

In response to the growing threat landscape facing embedded systems, the Cortex-M3 architecture incorporates a range of security features designed to safeguard system integrity, confidentiality, and availability. These security features include:

Memory Protection Unit (MPU): The Cortex-M3 processor includes an optional Memory Protection Unit (MPU) that enables fine-grained memory access control, allowing developers to enforce memory regions and access permissions to prevent unauthorized access and buffer overflow attacks. TrustZone for Cortex-M3: Advanced security features such as TrustZone technology can be integrated with Cortex-M3-based systems to provide hardware-based isolation of secure and non-secure software components [9], protecting sensitive data and cryptographic keys from unauthorized access. Secure Boot and Secure Firmware Update: The

Cortex-M3 architecture supports secure boot and secure firmware update mechanisms, ensuring the integrity and authenticity of firmware images loaded during system initialization and updating. By incorporating these security features into the hardware architecture, the Cortex-M3 processor provides a robust foundation for building secure and resilient embedded systems capable of withstanding sophisticated cyber threats

Development and Optimization Tools

To facilitate software development and optimization for Cortex-M3-based systems, ARM provides a comprehensive suite of development tools, including:

ARM Development Studio: A fully integrated development environment (IDE) for Cortex-M3 software development, offering advanced debugging, profiling, and optimization capabilities. ARM Compiler: A high-performance optimizing compiler tailored for Cortex-M3 architectures, providing efficient code generation and execution for embedded applications. ARM Performance Libraries: Pre-optimized software libraries for common embedded algorithms and functions, enabling developers to accelerate application development and improve performance. These development tools, combined with extensive documentation, tutorials, and community support, empower developers to harness the full potential of the Cortex-M3 architecture and deliver high-quality, reliable embedded software solutions.

1.1.5 The Need for Adversary Virtual Platforms (AVPs)

Evolving Threat Landscape

The rapid proliferation of embedded systems across various industries has coincided with an increase in cybersecurity threats targeting these devices. Embedded systems are now interconnected through networks and interfaces, forming the backbone of critical infrastructure and consumer devices alike. However, this interconnectedness exposes them to a wide range of cyber threats, including malware infections, data breaches, and remote exploits.

Furthermore, the complexity and diversity of embedded systems make them challenging to secure effectively. These systems often incorporate proprietary software, legacy components, and third-party libraries, each with its own set of vulnerabilities and attack vectors. Traditional security assessment methods, such as penetration testing and code review, may not be sufficient to identify and mitigate all potential security risks in embedded systems.

Limitations of Traditional Testing Methods

Traditional security testing methods rely heavily on physical access to hardware devices or access to source code, which may not always be feasible or practical. Moreover, conducting real-world attacks against embedded systems can be costly, time-consuming, and risky, especially when dealing with safety-critical or mission-critical applications.

Additionally, traditional testing methods often lack the ability to simulate complex attack scenarios or assess the impact of vulnerabilities in a controlled environment. As a result, organizations may struggle to accurately assess their security posture and prioritize mitigation efforts effectively.

Advantages of Adversary Virtual Platforms (AVPs)

Adversary Virtual Platforms (AVPs) offer a novel approach to security evaluation in embedded systems by providing a controlled and realistic environment for simulating adversarial attacks. Unlike physical testing methods, AVPs allow researchers to conduct virtual experiments in a safe and reproducible manner, without the need for expensive hardware or risking operational disruptions.

Key advantages of AVPs include:

Realistic Simulation: AVPs accurately emulate the behavior of real-world adversaries, allowing researchers to model a wide range of attack scenarios and assess their impact on embedded systems. This realism enables organizations to identify and prioritize security vulnerabilities based on their potential impact on system integrity, confidentiality, and availability. Scalability: AVPs can simulate large-scale networks of embedded devices, enabling researchers to evaluate the resilience of complex, interconnected systems against coordinated attacks. This scalability is essential for assessing the security posture of embedded systems deployed in IoT ecosystems, industrial control systems, and smart infrastructure. Cost-Effectiveness: By eliminating the need for physical hardware and infrastructure, AVPs offer a cost-effective alternative to traditional testing methods. Organizations can conduct virtual experiments using off-the-shelf hardware and open-source software, reducing capital expenditures and operational costs associated with security evaluation. Reproducibility: AVPs provide a reproducible environment for security testing, allowing researchers to repeat experiments and validate results with confidence. This reproducibility is critical for verifying the effectiveness of security controls, assessing the impact of software updates, and evaluating the resilience of embedded systems over time.

AVPs have diverse applications across various industries and domains, including:

Product Development: Organizations can use AVPs during the product development lifecycle to identify and mitigate security vulnerabilities early in the design phase. By integrating security testing into the development process, companies

can ensure that embedded systems are resilient to cyber threats before deployment. Threat Intelligence: Security researchers and threat analysts can leverage AVPs to study emerging cyber threats and develop countermeasures against new attack techniques. By simulating realistic attack scenarios, researchers can gain insights into adversary tactics, techniques, and procedures (TTPs), helping organizations better defend against evolving threats. Training and Education: AVPs can serve as valuable training tools for security professionals, allowing them to gain hands-on experience with real-world attack scenarios in a controlled environment. By providing interactive simulations and practical exercises, AVPs help bridge the gap between theoretical knowledge and practical skills in cybersecurity.

As embedded systems continue to evolve and become increasingly interconnected, the need for effective security evaluation techniques will only grow. Future advancements in AVPs may include:

Integration with Machine Learning: AVPs could leverage machine learning algorithms to automatically generate and adapt attack scenarios based on real-world threat intelligence. By continuously learning from new data and evolving adversary tactics, AVPs can provide more accurate and dynamic security assessments for embedded systems. Support for Hardware-in-the-Loop (HIL) Testing: AVPs could be integrated with Hardware-in-the-Loop (HIL) testing platforms to simulate realistic hardware interactions and environmental conditions. This integration would enable researchers to assess the impact of hardware vulnerabilities and physical attacks on embedded systems more comprehensively. Standardization and Interoperability: Efforts to standardize AVP frameworks and protocols would promote interoperability between different platforms and tools, allowing researchers to share attack scenarios, benchmarks, and best practices more easily. Standardization would also facilitate collaboration and knowledge sharing within the security community, accelerating innovation and advancing the state-of-the-art in embedded system security.

Chapter 2

The Virtual Platform

2.1 Gem5

In recent years, computer architecture has been passing through an exciting "golden age"[1] as some experts have called it. This period is characterised by rapid innovation, driven by the increasing computational demands of applications in artificial intelligence, big data analytics, and high-performance computing. To keep up with these demands, researchers are developing new hardware architectures that are more efficient, adaptable, and powerful. One of the main ways they advance these architectures is through software-based modelling and simulation. The "Golden Age for Computer Architecture" is an exciting era marked by significant advancements in artificial intelligence, machine learning, and specialized hardware like GPUs and TPUs.[2] This period is transformative because it emphasizes the need for both performance and energy efficiency, leading to innovative designs that move away from traditional CPU-centric models. This shift has revitalized the field, enabling new computing capabilities across a wide range of applications. It's a time of great potential and discovery, where tailored solutions are pushing the boundaries of what computers can achieve.

A key tool in this process is the gem5 simulator, a widely used and community-supported framework that has become essential in computer architecture research. Since its debut in 2011, gem5 has become a cornerstone in the field, frequently cited in top computer architecture publications [3]. The gem5 framework is designed

---

[1]Keckler, S. W. (2022, January 1). A new golden age for computer architecture. Communications of the ACM. Retrieved from

[2]A new golden age for computer architecture by John L. Hennessy David A. Patterson

[3]Ben-Nun, T., Hoefler, T. (2020). Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. HAL-Inria. Retrieved from

to be flexible, supporting a wide range of simulated components, from processing cores to complex memory systems and network interfaces. This versatility allows researchers to conduct detailed system-level performance analyses, gaining a deep understanding of how different hardware configurations impact computational workloads. Additionally, gem5 supports hardware-software co-design, enabling researchers to test and optimize the interactions between hardware architectures and software applications within a single simulation environment. The gem5 simulator has seen continuous development over the past nine years since its initial release. During this period, more than 250 contributors have made over 7000 commits, enhancing the simulator with new features, bug fixes, and improved code quality. This paper provides an overview of how gem5 is used, its features, the current state of the simulator, and the significant changes that have been made since its debut[4]. As, we will also highlight the continued improvement of gem5 simulator for the future of computer architecture research. What makes the gem5 project truly special is its strong, collaborative community, which includes academic researchers, industry professionals, and students. This community is committed to maintaining a stable yet adaptable code base that evolves to meet the needs of its users. Through continuous updates and contributions, the gem5 project incorporates new features and optimizations that reflect the latest advancements in computer architecture. This collective effort ensures that gem5 remains a powerful and relevant tool for exploring complex design spaces, providing researchers with a robust platform to experiment with architectures that address the computational challenges of today.

2.1.1 Gem5 Architecture

At its core, the gem5 simulator is built upon a modular and extensible architecture, purposefully designed to meet a broad spectrum of computer architecture research needs. The architecture of gem5 leverages both C++ and Python [5], combining the performance benefits of C++ with the flexibility and accessibility of Python. This dual-language foundation enables a robust and adaptable simulation framework that can accommodate complex research requirements. Each component within gem5's architecture is thoughtfully crafted to streamline the process of modeling and experimenting with various hardware configurations, thereby supporting a wide range of academic and industrial research goals. The primary structural foundations of gem5's architecture are outlined below.

[4]Ibid.

[5]Binkert, N., Beckmann, B., Black, G., et al. (2011). The gem5 simulator. ACM SIGARCH Computer Architecture News, 39(2), 1-7.s

The gem5 simulator is a modular platform for researching computer-system architecture, covering both system-level and processor microarchitecture [6]. It's a community-driven project with an open governance model. Initially created for academic research in computer architecture, gem5 has expanded its reach [7]. It's now widely used in both academia and industry for research purposes, as well as in educational settings. Additionally, Gem5 serves as an abstraction layer that supports the simulation of various system components, enabling researchers to model and experiment with diverse architectural designs without the need to create physical prototypes[8]. Through this platform, gem5 provides the foundational infrastructure for accurately simulating the behavior of a computer system's components, allowing for detailed exploration of design choices and their performance implications. The Platform is divided into several key components, each contributing to the overall flexibility and functionality of the gem5 simulator:

- Simulator Core: At the heart of gem5 lies the simulator core, which is responsible for coordinating the entire simulation process and facilitating seamless communication among the various system components. Implemented in C++, the simulator core provides a high-performance infrastructure essential for modeling the complex interactions and behaviors within computer systems [9]. This core ensures that each component operates within a cohesive simulation environment[10], thereby allowing researchers to execute extensive simulation experiments that capture the nuances of real-world system interactions [11]. By handling critical simulation tasks and managing time progression and event scheduling, the simulator core supports the accurate emulation of diverse system behaviors.The simulator core in gem5 handles essential tasks like time progression and event scheduling, ensuring that operations occur in the correct sequence during simulations. By managing the execution of instructions, memory requests, and I/O operations, the core allows SimObjects to work together seamlessly within the simulation environment. This event-driven architecture provides precise timing and synchronization, enabling researchers to run detailed simulation experiments. These experiments can capture the complexities of real-world system interactions and help explore the performance impacts of

[6]gem5. (n.d.). The gem5 Simulator. Retrieved from

[7]Ibid

[8]gem5. (n.d.). gem5 Simulator. Retrieved from

[9]System Architecture Model and Simulation Platform: Iteration 1. NEUROPULS Project. (2024, March 29). Deliverable 5.9 Report.

[10]Ben-Nun, T., Hoefler, T. (2020). Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. HAL-Inria. Retrieved from

[11]Ibid.

various architectural designs and configurations.

- SimObjects: One of the most distinctive elements of gem5's architecture is its use of SimObjects, which represent the primary building blocks of the simulated system. SimObjects encapsulate individual components within the system, such as processors, memory units, and peripheral devices, and they provide a unified interface for configuration, initialization, and interaction. Designed to be modular and highly configurable, SimObjects allow researchers to tailor each component's behavior to meet specific research needs. This modularity is a critical feature, as it enables researchers to model only the components they are interested in, thus reducing computational overhead and allowing for focused experimentation. Furthermore, SimObjects' design promotes extensibility, making it straightforward to integrate additional components or modify existing ones. In the gem5 simulator, SimObjects are essential building blocks that represent different hardware components like CPUs [12], caches, and memory controllers. Each SimObject has its own set of properties and behaviors, which users can customize and configure using Python scripts. This modular approach makes it easy for researchers to create and adjust SimObjects, enabling them to explore various architectural setups and their effects on performance. By ensuring that SimObjects can communicate through well-defined interfaces, gem5 accurately simulates the complex interactions within computer architectures. This makes it a valuable tool for both academic research and industry applications.

- Python Interface: The Python interface in the gem5 simulator is a key component that significantly enhances the usability and flexibility of the simulation environment. It allows users to configure systems [13], run experiments, and manage simulation setups with ease. Through Python scripts, users can define simulation parameters and analyze results, making the process both accessible and powerful. This interface enables researchers to create and modify SimObjects, which represent various hardware components, by setting parameters and defining system configurations in a straightforward manner[14]. Python simplifies the scripting process, facilitating rapid prototyping and experimentation. Users can easily adjust parameters and rerun simulations without needing extensive recompilation. Additionally, the Python interface supports a wide range of functionalities, including managing the simulation

[12]Ibid.

[13]gem5 (n.d.) Simple config: Part 1 of learning gem5. Retrieved from

[14]Ben-Nun, T., Hoefler, T. (2020). Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. HAL-Inria. Retrieved from

lifecycle—starting, stopping, and saving the state of simulations. This is particularly beneficial for researchers conducting iterative experiments to analyze the effects of different architectural choices on performance metrics[15]. By enhancing usability and accessibility, the Python interface empowers researchers to interact with gem5 more intuitively and flexibly. It also allows for advanced features like statistical analysis and visualization, enriching the research experience. Leveraging Python's strengths, gem5 offers a robust and user-friendly platform for both academic and industry researchers to explore and innovate in computer architecture.

- Parameterized Models: At the core of the gem5 framework are its parameterized models, which provide researchers with extensive flexibility in configuring simulations. These models encapsulate the behavior and characteristics of various hardware components within a computer system, offering a rich set of customizable options for each element [16]. This customization allows researchers to conduct detailed experiments and tailor the simulation environment to meet specific research needs, enhancing the relevance and applicability of their findings. Parameterized models [17] enable the simulation of systems with a wide range of configurations and performance characteristics. Researchers can modify parameters related to cache sizes, processor configurations, memory hierarchies, and interconnect structures. This capability allows them to explore and analyze a broad spectrum of architectural designs, from simple single-core processors to complex multi-core systems with intricate memory management strategies. By adjusting these parameters, users can create tailored simulation scenarios that reflect real-world conditions or hypothetical architectures, facilitating a deeper understanding of how different design choices impact overall system performance. The flexibility of parameterized models also supports thorough design-space exploration, allowing researchers to systematically evaluate the performance trade-offs associated with various architectural choices. This exploration is crucial for identifying optimal configurations that balance factors such as speed, power consumption, and resource utilization. By leveraging the parameterized models within gem5, researchers can conduct comprehensive studies that highlight the strengths and weaknesses of specific designs and contribute to the development of innovative solutions in computer architecture. Ultimately, the ability to fine-tune parameters and simulate diverse configurations empowers researchers to push

[15]Ben-Nun, T., Hoefler, T. (2020). Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. HAL-Inria. Retrieved from

[16]Ibid.

[17]Ibid.

the boundaries of architectural research and drive advancements in the field.he gem5 architecture is meticulously constructed to support a wide array of research objectives in computer architecture. By integrating a C++-based simulator core, modular SimObjects, a Python-based user interface, and versatile parameterized models, gem5 provides a comprehensive, user-friendly environment for the simulation and analysis of complex computer systems. The framework's extensible and customizable design empowers researchers to push the boundaries of computer architecture innovation, supporting the development of cutting-edge hardware solutions that meet the computational demands of modern applications. Through its combination of performance and flexibility, gem5 has become an indispensable tool in the field, widely recognized for its utility in both academic research and industry applications.

2.1.2 Gem5 Simulation Parameters

Gem5 simulates computer systems by carefully modeling the behavior of various components and running a sequence of events to capture how these parts interact over time. Through this detailed modeling, researchers can better understand system performance and behavior under different configurations and workloads. Key parameters for setting up simulations in gem5 include the following [18] :

- CPU Model: You can simulate two types of processor families. The first type is based on the SimpleCpu implementation. This class models in-order, non-pipelined processors and includes functions for handling interrupts, setting up fetch requests, advancing the program counter (PC), and implementing the ExecContext interface, which manages read/write operations on memory, the arithmetic logic unit (ALU), PC, and cache. Within this family, there are two subclasses: AtomicSimpleCPU and TimingSimpleCPU. The key difference between these subclasses is how they handle memory access. AtomicSimpleCPU uses atomic memory access, meaning it processes memory operations in a single, indivisible step. TimingSimpleCPU, on the other hand, uses timing memory access, which is more detailed and involves two one-way messages to simulate the realistic timing of memory operations. This makes TimingSimpleCPU more accurate in reflecting how memory access timing works in real systems. The second family of processors is more complex and includes the InOrder CPU and the O3 CPU. The InOrder CPU is a configurable, in-order pipelined processor, meaning it processes instructions in the order they are received but can handle multiple stages of instruction processing simultaneously. The O3 CPU is the most complex, simulating an out-of-order

[18] Ibid.

pipelined processor with a reorder buffer. This allows it to execute instructions out of order for better performance and supports superscalar architectures and hardware multi-threading, making it capable of handling multiple instructions at once. Moreover, Gem5 offers a wide range of CPU models, from in-order to out-of-order processors, enabling researchers to select models that best suit their studies. By adjusting parameters like pipeline depth, execution units, and cache hierarchies, users can customize simulations to explore different architectural designs. For example, modifying pipeline depth allows for analysis of instruction-level parallelism, while tuning execution units provides insights into how processing power affects performance.

- Memory System: Memory system is implemented through MemObject implementing a memory structure (CacheObject for example) and ports to communicate with other SimObjects through a master/slave interface. Gem5's memory system is highly customizable, making it possible to explore memory configurations in detail. Parameters such as cache size, associativity, and replacement policies can be finely adjusted to simulate various memory architectures. This flexibility enables researchers to analyze memory behavior and performance closely, investigating how memory design choices influence latency, bandwidth, and system efficiency. By experimenting with diverse configurations, researchers can identify optimal designs tailored to specific workloads.

- Execution Mode: In gem5, there are two execution modes. The first mode is Full Simulation, which simulates a bare-metal environment running a Linux Kernel image. This mode provides the most complete and accurate simulation experience. The second mode is System-call Emulation. In this mode, whenever a system call occurs, it is trapped and passed to the host system running gem5. Gem5 supports multiple execution modes, including full system simulation and syscall emulation. Full system mode can boot an entire OS, providing realistic application execution for comprehensive performance analysis. In contrast, syscall emulation allows applications to run without the full OS overhead, useful in experiments where system-level details are less critical. This adaptability allows researchers to choose the mode that aligns best with their study goals.

- ISA (Instruction Set Architecture): With support for ISAs such as ARM, x86, and MIPS, gem5 facilitates comparisons across architectures. This capability allows researchers to explore trade-offs and examine how different ISAs impact system design and performance. Simulating different ISAs provides insights into how architectural choices affect efficiency, power usage, and execution speed, which is essential for studying the broader implications of ISA selection.

Additionally, With gem5, it's possible to simulate different Instruction Set Architectures (ISAs) without needing to specialize each object for each ISA. This is achieved through an ISA Domain-Specific Language (DSL), which unifies the decoding of binary instructions. The system uses a C++ base class and derived classes for each ISA, which override methods like execute() to implement specific opcodes (e.g., ADD, SUB).

- Devices Modelling: Gem5 allows the integration of I/O devices, including network interfaces and storage controllers, enabling researchers to assess the impact of these devices on system behavior. Modeling these devices helps analyze how different configurations influence data transfer rates, latency, and overall throughput, essential for understanding the role of peripherals in modern systems. Besides, it is possible to add any kind of I/O device(DMA, NIC), but if it is used in FS mode it is needed to having working drivers.

- External API: Gem5 supports external APIs, which allows integration with additional tools and libraries. Researchers can use these APIs to expand gem5's functionality, incorporating analysis tools, visualization libraries, or custom hardware models. This capability enhances the simulation environment and enables a wider range of experimental possibilities.

- GPU Compute: Gem5 also supports GPU compute simulations, making it suitable for heterogeneous computing environments. This feature allows researchers to study the interactions between CPUs and GPUs, examining parallel processing and the effects of offloading tasks to GPUs. By simulating GPU compute, researchers can gain insights into the performance of parallel algorithms and the impact of GPU integration on overall system performance.

The rich set of configuration options in gem5 enables researchers to conduct in-depth experiments, exploring various architectural designs and performance characteristics. This versatility is crucial for advancing computer architecture research and developing new solutions to modern computing challenges.

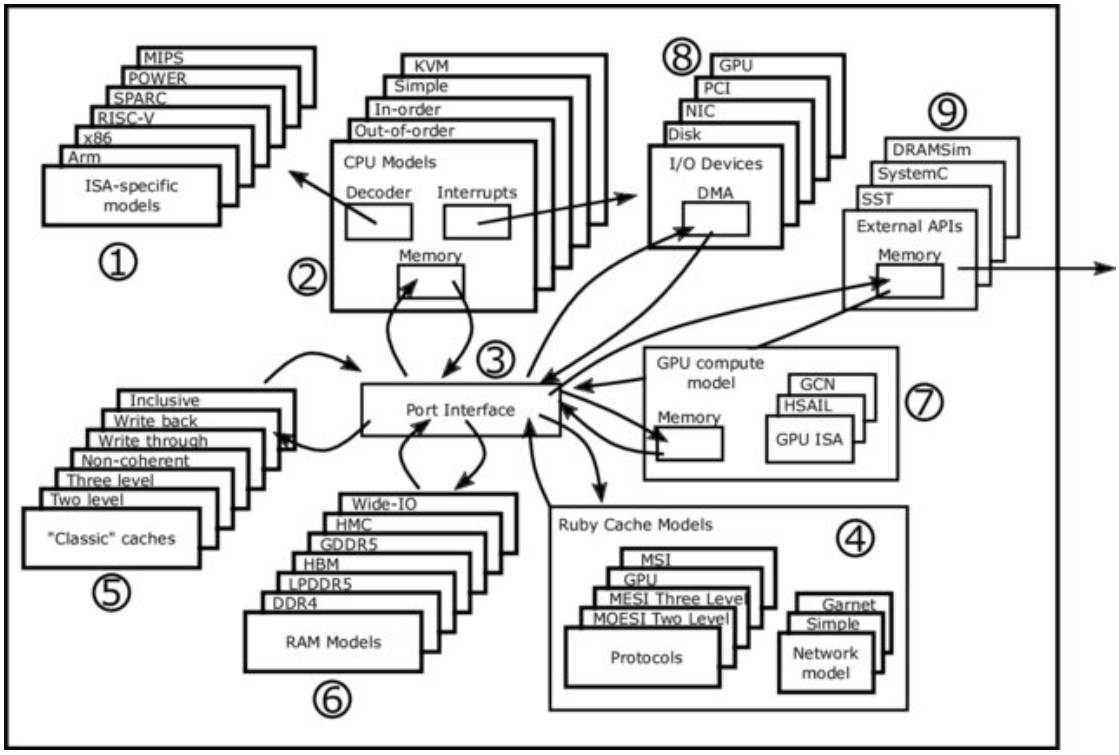

Figure 2.1: An overview of gem5's architecture. Its modular components allow any of each model type to be used in system configuration via Python scripts. Users can choose the fidelity of the memory system, CPU model, etc. while being able to select any ISA, devices, etc. The port interface allows any memory component to be connected to any other memory component as specified by the Python script.

Thus, The diagram shows the architecture of gem5, emphasizing its modular components and the flexibility they offer for system configuration. Each model type in gem5 can be used in various setups through Python scripts, which are the main way users define their simulation environments. This modularity is a key feature of gem5, allowing researchers to customize their simulations to meet specific experimental needs and goals.

Modular Components

The architecture includes several modular components:

- CPU Models: Users can choose from a range of CPU models, like in-order

[18]Figure adapted from "The gem5 Simulator: Version 20.0+ – A new era for the open-source computer architecture simulator", Jason Lowe-Power et al., arXiv:2007.03152.

and out-of-order processors, designed to simulate different execution strategies. This variety lets users tailor simulations based on specific research needs, helping them explore how different CPU architectures affect overall system performance and efficiency.

- Memory Systems: The memory subsystem is highly configurable, allowing users to select different cache architectures and memory types, such as DRAM and SRAM. This flexibility is crucial for studying how memory hierarchies impact performance, as different setups can lead to significant variations in latency and throughput, which are key factors in computer architecture research.

- I/O Devices: Gem5 includes various I/O device models, enabling the simulation of interactions between the CPU, memory, and peripheral devices. This feature allows researchers to examine how different I/O configurations affect system performance and to assess the implications of various device architectures on overall system behavior.

Fidelity and Configuration

Users can choose the fidelity of the memory system, CPU model, and other components in their simulation environment. This means researchers can opt for high-fidelity models that provide detailed timing and performance characteristics, essential for accurate analysis, or lower-fidelity models that run faster but offer less detail. This flexibility is vital for balancing accuracy with simulation speed, allowing efficient exploration of different architectural configurations without being limited by a single model.

Instruction Set Architecture (ISA) Support

Gem5 supports multiple ISAs, such as ARM, x86, MIPS, and RISC-V. This support is crucial for researchers who want to investigate how different instruction sets impact system performance. By enabling comparisons and evaluations of various architectural designs, gem5 allows users to study how different ISAs influence workload execution and overall system efficiency.

Port Interface

The port interface is a significant feature of gem5's architecture, allowing any memory component to connect to any other memory component as specified in the Python script. This modularity enhances the system's flexibility, enabling researchers to create custom memory topologies and interconnects. These capabilities are particularly useful for testing new designs and configurations without

extensive reconfiguration, streamlining the research process and facilitating rapid prototyping of novel architectural ideas.

Simulation Modes

Gem5 offers two distinct simulation models:

- System Emulation (SE): This mode meticulously emulates entire hardware systems at the instruction level, providing a high degree of accuracy. It prioritizes accuracy, making it ideal for full-system workloads and operating system simulations. SE mode is particularly useful for researchers who need to understand detailed interactions within a complete system, allowing comprehensive analysis of system behavior under realistic conditions.
- Functional Simulation (FS): In contrast, FS mode focuses on the functional behavior of the system. While it sacrifices some accuracy for faster simulations, it is ideal for quick design explorations. This mode allows researchers to rapidly iterate on design ideas and evaluate different architectural configurations without the overhead of detailed timing simulations, making it a valuable tool for early-stage design and experimentation.

  The architecture of gem5, as shown in the figure, highlights its modular components and the flexibility they offer for system configuration through Python scripts. The ability to choose fidelity, support multiple ISAs, and use a port interface for component connections enhances gem5's research capabilities. Additionally, the two simulation modes—System Emulation and Functional Simulation—cater to different research needs, allowing for both detailed analysis and rapid design exploration. This versatility makes gem5 a powerful tool for researchers in computer architecture, enabling them to conduct a wide range of experiments and analyses.

2.1.3 Configuration Script

In gem5, configuration scripts play a crucial role in setting up simulations. They harness Python's capabilities along with the SimObject abstraction to create flexible and dynamic system configurations. This setup allows researchers to define complex architectures and experiment with various hardware parameters. Using Python in gem5 provides a powerful scripting interface, making it easy for users to configure and control their simulations. The SimObject abstraction is a key feature, representing different components of the simulated system, such as CPUs, caches, memory controllers, and I/O devices. Each SimObject can be instantiated and configured through Python scripts, enabling a modular and extensible approach to system design. Basically, the provided code snippet is a simplified version of

the actual script used for ARM architecture simulations that are mentioned in the next chapters. Setting up configuration scripts is quite straightforward thanks to Python and the SimObject abstraction. Below is a simplified version of the actual script. This version omits details specific to the x86 implementation and other build options for clarity. In essence, this script creates a FS interface , which will be used later to run the simulations. This script create a dynamic system FS "Full System", It defines predefined CPU configurations using a dictionary named CPU types. Each CPU configuration tuple consists of different CPU class and cache classes options (L1 instruction cache, L1 data cache, L2 cache)

```
import os
import m5
from m5.util import addToPath
from m5.objects import *
from m5.options import *
import argparse

m5.util.addToPath("../..")

from common import SysPaths
from common import ObjectList
from common import MemConfig
from common.cores.arm import O3_ARM_v7a, HPI

import devices


default_kernel = "vmlinux.arm64"
default_disk = "linaro-minimal-aarch64.img"
default_root_device = "/dev/vda1"


# Pre-defined CPU configurations. Each tuple must be ordered as : (cpu_class,
# l1_icache_class, l1_dcache_class, l2_Cache_class). Any of
# the cache class may be 'None' if the particular cache is not present.
cpu_types = {
    "atomic": (AtomicSimpleCPU, None, None, None),
    "minor": (MinorCPU, devices.L1I, devices.L1D, devices.L2),
    "hpi": (HPI.HPI, HPI.HPI_ICache, HPI.HPI_DCache, HPI.HPI_L2),
    "o3": (
        O3_ARM_v7a.O3_ARM_v7a_3,
        O3_ARM_v7a.O3_ARM_v7a_ICache,
        O3_ARM_v7a.O3_ARM_v7a_DCache,
        O3_ARM_v7a.O3_ARM_v7aL2,
    ),
}
```

```
37
38
39 def create_cow_image(name):
40     """Helper function to create a Copy-on-Write disk image"""
41     image = CowDiskImage()
42     image.child.image_file = SysPaths.disk(name)
43
44     return image
45
46
47 def create(args):
48     """Create and configure the system object."""
49
50     if args.script and not os.path.isfile(args.script):
51         print(f"Error: Bootscript {args.script} does not exist")
52         sys.exit(1)
53
54     cpu_class = cpu_types[args.cpu][0]
55     mem_mode = cpu_class.memory_mode()
56     # Only simulate caches when using a timing CPU (e.g., the HPI
   model)
57     want_caches = True if mem_mode == "timing" else False
58
59     system = devices.SimpleSystem(
60         want_caches,
61         args.mem_size,
62         mem_mode=mem_mode,
63         workload=ArmFsLinux(object_file=SysPaths.binary(args.
   kernel)),
64         readfile=args.script,
65     )
66
67     MemConfig.config_mem(args, system)
68
69     # Add the PCI devices we need for this system. The base system
70     # doesn't have any PCI devices by default since they are
   assumed
71     # to be added by the configuration scripts needing them.
72     system.pci_devices = [
73         # Create a VirtIO block device for the system's boot
74         # disk. Attach the disk image using gem5's Copy-on-Write
75         # functionality to avoid writing changes to the stored
   copy of
76         # the disk image.
77         PciVirtIO(vio=VirtIOBlock(image=create_cow_image(args.
   disk_image)))
78     ]
79
```

```
    # Attach the PCI devices to the system. The helper method in the
    # system assigns a unique PCI bus ID to each of the devices and
    # connects them to the IO bus.
    for dev in system.pci_devices:
        system.attach_pci(dev)

    # Wire up the system's memory system
    system.connect()

    # Add CPU clusters to the system
    system.cpu_cluster = [
        devices.ArmCpuCluster(
            system,
            args.num_cores,
            args.cpu_freq,
            "1.0V",
            *cpu_types[args.cpu],
            tarmac_gen=args.tarmac_gen,
            tarmac_dest=args.tarmac_dest,
        )
    ]

    # Create a cache hierarchy for the cluster. We are assuming that
    # clusters have core-private L1 caches and an L2 that's shared
    # within the cluster.
    system.addCaches(want_caches, last_cache_level=2)

    # Setup gem5's minimal Linux boot loader.
    system.realview.setupBootLoader(system, SysPaths.binary)

    if args.dtb:
        system.workload.dtb_filename = args.dtb
    else:
        # No DTB specified: autogenerate DTB
        system.workload.dtb_filename = os.path.join(
            m5.options.outdir, "system.dtb"
        )
        system.generateDtb(system.workload.dtb_filename)

    if args.initrd:
        system.workload.initrd_filename = args.initrd

    # Linux boot command flags
    kernel_cmd = [
        # Tell Linux to use the simulated serial port as a console
        "console=ttyAMA0",
```

```
126         # Hard-code timi
127         "lpj=19988480",
128         # Disable address space randomisation to get a consistent
129         # memory layout.
130         "norandmaps",
131         # Tell Linux where to find the root disk image.
132         f"root={args.root_device}",
133         # Mount the root disk read-write by default.
134         "rw",
135         # Tell Linux about the amount of physical memory present.
136         f"mem={args.mem_size}",
137     ]
138     system.workload.command_line = " ".join(kernel_cmd)
139 
140     if args.with_pmu:
141         for cluster in system.cpu_cluster:
142             interrupt_numbers = [args.pmu_ppi_number] * len(
    cluster)
143             cluster.addPMUs(interrupt_numbers)
144 
145     return system
146 
147 
148 def run(args):
149     cptdir = m5.options.outdir
150     if args.checkpoint:
151         print(f"Checkpoint directory: {cptdir}")
152 
153     while True:
154         event = m5.simulate()
155         exit_msg = event.getCause()
156         if exit_msg == "checkpoint":
157             print("Dropping checkpoint at tick %d" % m5.curTick())
158             cpt_dir = os.path.join(m5.options.outdir, "cpt.%d" %
    m5.curTick())
159             m5.checkpoint(os.path.join(cpt_dir))
160             print("Checkpoint done.")
161         else:
162             print(f"{exit_msg} ({event.getCode()}) @ {m5.curTick()
    }")
163             break
164 
165 
166 def arm_ppi_arg(int_num: int) -> int:
167     """Argparse argument parser for valid Arm PPI numbers."""
168     # PPIs (1056 <= int_num <= 1119) are not yet supported by gem5
169     int_num = int(int_num)
170     if 16 <= int_num <= 31:
171         return int_num
```

```
172     raise ValueError(f"{int_num} is not a valid Arm PPI number")
173
174
175 def main():
176     parser = argparse.ArgumentParser(epilog=__doc__)
177
178     parser.add_argument(
179         "--dtb", type=str, default=None, help="DTB file to load"
180     )
181     parser.add_argument(
182         "--kernel", type=str, default=default_kernel, help="Linux
    kernel"
183     )
184     parser.add_argument(
185         "--initrd",
186         type=str,
187         default=None,
188         help="initrd/initramfs file to load",
189     )
190     parser.add_argument(
191         "--disk-image",
192         type=str,
193         default=default_disk,
194         help="Disk to instantiate",
195     )
196     parser.add_argument(
197         "--root-device",
198         type=str,
199         default=default_root_device,
200         help=f"OS device name for root partition (default: {
    default_root_device})",
201     )
202     parser.add_argument(
203         "--script", type=str, default="", help="Linux bootscript"
204     )
205     parser.add_argument(
206         "--cpu",
207         type=str,
208         choices=list(cpu_types.keys()),
209         default="atomic",
210         help="CPU model to use",
211     )
212     parser.add_argument("--cpu-freq", type=str, default="4GHz")
213     parser.add_argument(
214         "--num-cores", type=int, default=1, help="Number of CPU
    cores"
215     )
216     parser.add_argument(
217         "--mem-type",
```

```
218         default="DDR3_1600_8x8",
219         choices=ObjectList.mem_list.get_names(),
220         help="type of memory to use",
221     )
222     parser.add_argument(
223         "--mem-channels", type=int, default=1, help="number of
    memory channels"
224     )
225     parser.add_argument(
226         "--mem-ranks",
227         type=int,
228         default=None,
229         help="number of memory ranks per channel",
230     )
231     parser.add_argument(
232         "--mem-size",
233         action="store",
234         type=str,
235         default="2GB",
236         help="Specify the physical memory size",
237     )
238     parser.add_argument(
239         "--tarmac-gen",
240         action="store_true",
241         help="Write a Tarmac trace.",
242     )
243     parser.add_argument(
244         "--tarmac-dest",
245         choices=TarmacDump.vals,
246         default="stdoutput",
247         help="Destination for the Tarmac trace output. [Default:
    stdoutput]",
248     )
249     parser.add_argument(
250         "--with-pmu",
251         action="store_true",
252         help="Add a PMU to each core in the cluster.",
253     )
254     parser.add_argument(
255         "--pmu-ppi-number",
256         type=arm_ppi_arg,
257         default=23,
258         help="The number of the PPI to use to connect each PMU to
    its core. "
259         "Must be an integer and a valid PPI number (16 <= int_num
    <= 31).",
260     )
261     parser.add_argument("--checkpoint", action="store_true")
262     parser.add_argument("--restore", type=str, default=None)
```

```
263
264     args = parser.parse_args()
265
266     root = Root(full_system=True)
267     root.system = create(args)
268
269     if args.restore is not None:
270         m5.instantiate(args.restore)
271     else:
272         m5.instantiate()
273
274     run(args)
275
276
277 if __name__ == "__m5_main__":
278     main()
```

This configuration script is a great example of how gem5 leverages Python and the SimObject abstraction to build dynamic and customizable simulation environments. By defining CPU configurations and instantiating various system components, researchers can delve into different architectural designs and assess their performance impacts within a full system simulation. This flexibility is one of the key strengths of gem5, making it an invaluable tool for computer architecture research.

2.1.4 Role of the Script in gem5 Full-System Simulation

The provided script plays a critical role in the configuration and execution of full-system simulations in gem5 [10]. It is designed to simulate ARM-based systems [19] using the gem5 platform. This script defines essential parameters for the CPU, memory system, and I/O devices, enabling detailed simulations that model real- world system behavior. In better wording, The script used for setting up full-system simulations in gem5 which is a key part of the simulation environment, especially for ARM-based systems. Its importance comes from its ability to define and manage the complex parameters that control how different parts of the system, like the CPU, memory, and I/O devices, behave [20]. By using Python and the SimObject abstraction, the script allows for a modular and flexible way to configure the system, letting researchers customize simulations to fit their specific needs. In gem5, this script acts like a blueprint for the simulated environment, detailing the

[19] gem5, n.d

[20] Ibid.

architecture's setup and operational parameters [21]. For example, it specifies the type of CPU to be simulated—whether it's a simple, minor, or out-of-order (O3) CPU—along with the cache configurations [22]. This level of detail is crucial for accurately modeling the system's performance characteristics [23].The choice of CPU type affects the execution model, influencing factors like instruction throughput, cache hit rates, and overall system latency [24]. By offering a range of CPU models, the script enables researchers to explore different architectural designs and their performance implications [25]. Additionally, the script helps integrate I/O devices, which are crucial for simulating full-system behavior [26].Configuring and instantiating various I/O components allows researchers to study the interactions between the CPU, memory, and peripheral devices [27]. This comprehensive approach is essential for understanding how different system configurations affect overall performance, especially in scenarios where I/O operations are a significant bottleneck. The gem5 scripting interface's flexibility also allows for dynamic parameter adjustments during simulation, enabling researchers to conduct sensitivity analyses and explore how architectural changes affect performance metrics [28]. This capability is particularly valuable in research, where the ability to quickly iterate and test various configurations can lead to deeper insights into system behavior. The script is crucial for configuring and running full-system simulations in gem5, particularly for ARM architectures (ARM, n.d.). By defining key parameters for the CPU, memory system, and I/O devices, the script enables detailed and realistic simulations that mimic real-world system behavior. Its modularity and flexibility allow researchers to explore a wide range of architectural configurations, helping them understand the performance implications of different design choices [29]. Thus, the script is not just a tool for simulation; it's an integral part of the research process that drives innovation in computer architecture [30].

[21] Binkert et al., 2011

[22] ARM, n.d.

[23] Binkert et al., 2011)

[24] gem5, n.d.

[25] ARM, n.d.

[26] gem5, n.d.

[27] ARM, n.d.

[28] gem5, n.d.

[29] Binkert et al., 2011

[30] ARM, n.d.

2.1.5 CPU and Memory Configuration

In the gem5 simulation framework, setting up CPU models and memory hierarchies is crucial for ensuring accurate and relevant simulation results [31]. This setup process allows researchers to customize the simulated environment to meet their specific experimental needs, especially when studying complex issues like security vulnerabilities. As it is explained in the following lines:

```
cpu_class = cpu_types[args.cpu][0]
mem_mode = cpu_class.memory_mode()
# Only simulate caches when using a timing CPU (e.g., the HPI
    model)
want_caches = True if mem_mode == "timing" else False
```

Choosing the right CPU model is a key part of the simulation setup. Researchers can select from various CPU architectures, such as in-order, out-of-order (O3)[32], or simple atomic models. Each model has unique characteristics that affect how instructions are executed and how the CPU interacts with memory. For example, out-of-order processors, like the O3 model, execute instructions as resources become available rather than strictly following the original program order. This approach maximizes performance but also introduces complexities that can be exploited by certain attacks, such as the Spectre vulnerability. When studying the Spectre attack, the O3 ARM CPU model is particularly important [33]. Speculative execution, a feature of modern out-of-order processors, allows the CPU to predict and execute instructions ahead of time based on expected future paths. This behavior is central to the Spectre vulnerability, as it can lead to unintended information leakage through side channels. Using the O3 model, researchers can accurately replicate speculative execution, providing valuable insights into how these vulnerabilities occur and can be exploited. Another important aspect of the setup is configuring the memory mode, which depends on the chosen CPU type. The interaction between speculative execution and the memory hierarchy—comprising caches and main memory—plays a crucial role in the performance and security implications of attacks like Spectre and ret2ns. The memory mode determines how memory accesses are handled during execution, affecting factors like cache coherence, latency, and the timing of memory operations. Understanding these interactions is essential for analyzing how speculative execution can be manipulated to leak sensitive information. Additionally, the simulation environment can be adjusted based on

[31] gem5, n.d

[32] gem5, n.d

[33] ARM, n.d.

the chosen CPU model, especially regarding cache simulations. Caches are vital components of modern computer architectures, designed to speed up memory access by storing frequently accessed data closer to the CPU. The behavior of caches can significantly impact the timing of memory operations, which is a key consideration in timing attacks. By enabling or disabling cache simulations based on the CPU model, researchers can create a more realistic representation of how memory access patterns influence the effectiveness of security vulnerabilities. Configuring CPU and memory in gem5 simulations is essential for enhancing the accuracy of the simulated environment. By allowing the selection of various CPU models and dynamically adjusting memory modes, researchers can create simulations that accurately reflect the complexities of modern processor architectures. This capability is particularly important when investigating vulnerabilities like Spectre, as it enables a comprehensive understanding of the interactions between speculative execution and memory access patterns. Careful configuration of these parameters not only improves the validity of simulation results but also provides a robust platform for exploring potential mitigations and defenses against such attacks. Through this detailed approach, researchers can gain deeper insights into the security implications of architectural features and contribute to developing more secure computing systems.

2.1.6 System Creation and Boot Configuration

The boot configuration expands the current kernel command line to support additional key-value data when booting the kernel in an efficient way[34]. The script is all about setting up and starting the full system simulation, including the operating system (OS). It outlines how to configure the virtual ARM-based system, making sure all the necessary elements are in place to simulate a complete OS environment. Here's how it is created and booted:

```
system = devices.SimpleSystem(
    want_caches,
    args.mem_size,
    mem_mode=mem_mode,
    workload=ArmFsLinux(object_file=SysPaths.binary(args.kernel)),
    readfile=args.script,
)
```

In this section, the script configures the simulated ARM system by loading the OS kernel—in this case, Linaro Minimal Linux for AArch64—and defining key system

[34]The Linux Kernel Documentation. (n.d.). Boot configuration.

parameters. This setup is crucial for full-system simulation, enabling the testing of entire OS-level workloads, such as booting the Linux kernel. By specifying parameters like the kernel file and memory size, I could recreate realistic hardware conditions needed to test how the system might respond under scenarios where attacks are executed. This approach enables the simulation of complex interactions within the OS and memory system, which is vital for accurate security assessments. By configuring the system this way, the script allows for a full-system simulation. This means the entire OS, including the Linux kernel, is booted within the simulated environment. Such a setup is essential for testing workloads that operate at the OS level, as it provides the necessary infrastructure to observe and analyze system behavior under various simulated conditions. Setting the kernel and memory size helps the simulation recreate conditions similar to real hardware. This is crucial for realistically testing potential security attacks, such as those targeting memory access patterns. This setup is particularly valuable for security research, as it allows for examining complex interactions within the OS, CPU, and memory hierarchy. Understanding these interactions is vital for figuring out how such attacks can be launched and exploited in real systems.

2.1.7 PCI Devices and Disk Images

It's important to define the Peripheral Component Interconnect (PCI) to understand how this device works. The Peripheral Component Interconnect (PCI) interface, introduced by Intel in 1992, was a standard connection interface for computer motherboards. It was widely used throughout the late 1990s and early 2000s [35]. This interface enabled the attachment of hardware components such as sound cards, video cards, and modems, playing a crucial role in enhancing system functionality during its era. While PCI has been largely replaced by PCI Express (PCIe) slots in modern systems, its historical significance in computer architecture is notable. A PCI device refers to any hardware component designed to connect to the motherboard through a PCI slot. At its inception, PCI operated as a 32-bit, 33 MHz interface capable of data transfer speeds up to 133 MBps. An upgraded version introduced 64-bit functionality and 66 MHz speeds, enabling transfers up to 533 MBps. Further development yielded PCI Extended (PCI-X), which supported up to 133 MHz and data transfer rates of 1064 MBps. However, the introduction of PCI Express (PCIe) in 2004 marked a significant advancement, offering greater speed, efficiency, and scalability, effectively making PCI and PCI-X obsolete [36]. Despite its declining use, PCI remains relevant in older systems.

[35] Inspired eLearning. (n.d.). What is a PCI device?

[36] Ibid.

The plug-and-play (PnP) architecture of the PCI bus and subsequent interfaces allows compatibility with newer slots using adapters. However, enabling a PCI slot may sometimes require BIOS configuration adjustments. PCI slots were versatile, enabling connection to various devices such as network cards, sound cards, graphics cards, controller cards, RAID cards, modems, and scanners. A typical motherboard of that era featured multiple PCI expansion slots, often five or more, allowing users to upgrade their systems by adding essential hardware components [37]. This modular design provided users with the flexibility to enhance features like audio capabilities, graphics performance, wireless connectivity, and storage expansion. In modern systems, PCI devices are largely unnecessary, as contemporary computers predominantly utilize PCIe or USB for peripheral connectivity. Nonetheless, PCI devices still find use in legacy systems or specialized environments where older hardware remains operational. Additionally, as it is provided in the bellowing script, this part of the script is responsible for attaching PCI devices, specifically including block storage devices, with the following code:

```
system.pci_devices = [
    PciVirtIO(vio=VirtIOBlock(image=create_cow_image(args.
   disk_image)))
]
```

In this part of the script, we create a VirtIO block device, which acts as the main disk image for the simulated operating system. The VirtIO Block function attaches the disk image, and the createcowimage function generates a copy-on-write (COW) version of the disk image. This means any changes made during the simulation won't affect the original disk image. For my simulation, I used the Linaro Minimal AArch64 disk image, which provides the OS and mounts the necessary file system. The VirtIO block device is crucial for loading and running important files, like the compiled binary of the Spectre exploit in my case. This setup ensures that the simulated environment has the storage infrastructure needed to execute and interact with the OS and filesystem just like it would on real hardware. By using the VirtIO block device, the script simplifies the configuration process by automatically handling PCI and I/O device setup. This configuration is essential for experiments like the Spectre attack, as it creates a realistic environment where the interactions between the OS, storage, and attack binaries can be observed and analyzed.

[37] Ibid.

2.1.8 Cache Hierarchy and Clustering of CPUs

This section of the script is responsible for defining the CPU cluster configuration and its cache hierarchy, specifically setting up clusters of CPU cores along with their respective caches. The relevant code is as follows:

```
system.cpu_cluster = [
    devices.ArmCpuCluster(
        system,
        args.num_cores,
        args.cpu_freq,
        "1.0V",
        *cpu_types[args.cpu],
        tarmac_gen=args.tarmac_gen,
        tarmac_dest=args.tarmac_dest,
    )
]
```

This block configures the CPU cluster (the group of cores used in the simulation) and their associated cache hierarchy (L1 instruction and data caches, L2 cache). By setting up multiple CPU cores, I was able to simulate multi-core interactions, which are essential for modeling speculative execution vulnerabilities like Spectre. In this part of the script, we use the Arm CPU Cluster function to create and configure a group of CPU cores, known as a CPU cluster. This configuration does not sets up only the CPUs but also defines the cache hierarchy for each CPU core cluster. The cache hierarchy is defined within each CPU cluster, encompassing the L1 instruction and data caches, and the L2 cache. The L1 caches are designed for rapid access to frequently used instructions and data, while the L2 cache acts as a larger, intermediate storage to bridge the gap between the L1 caches and the main memory. This hierarchical structure helps in reducing latency and improving overall system performance. The cache hierarchy typically includes L1 instruction and data caches, as well as an L2 cache shared among the cores in the cluster. Configuring the cache hierarchy is crucial for simulations because it affects data access speeds and models real-world cache interactions. By setting up multiple CPU cores in clusters, this configuration allows for the simulation of multi-core interactions, which are essential for accurately modeling behaviors like speculative execution. When studying vulnerabilities such as the Spectre attack, having a multi-core setup helps replicate the real-world conditions under which these attacks exploit interactions between cores and caches. This setup enables researchers to explore how speculative execution vulnerabilities arise in environments with multiple cores and caches, providing a comprehensive view of these complex interactions.

2.1.9 Bootloader Setup and Kernel Command-Line Arguments

The following lines of code are responsible for setting up the bootloader and passing boot-time parameters to the Linux kernel. The Linux kernel can be defined as the core mechanism responsible for managing requests between software and hardware efficiently, ensuring that the system resources, such as the processor and memory, are utilized effectively. Every program requires at least a processor and memory to run, and the kernel acts as the intermediary, handling the large number of hardware-related requests generated during program execution. The Linux kernel is designed as a monolithic kernel with a modular architecture. Being monolithic means that the entire operating system operates at the kernel level, providing high performance and direct control over hardware resources. Meanwhile, its modular design allows functionalities to be extended dynamically at runtime, enabling the addition of new features or drivers without the need to reboot or recompile the kernel. This configuration is essential for making sure the kermel gets the right instruction about where to send console output, and other key parameters.

```
system.realview.setupBootLoader(system, SysPaths.binary)
kernel_cmd = [
    "console=ttyAMA0",
    f"root={args
    "lpj=19988480",
"norandmaps",
f"mem={args.mem_size}",
]
system.workload.command_line = " ".join(kernel_cmd)
```

In this block, the script is designed to set up the Linux kernel bootloader and provide the necessary command-line arguments for booting the simulated system. This includes instructions for setting the console output to a simulated serial port, specifying the root device for the disk image, and defining the memory size. The final string combines all specified parameters into the exact format that the Linux kernel expects on its command line during initialization. For the Spectre attack simulations, this setup is critical. It ensures that the Linux kernel boots correctly and mounts the disk image containing the Spectre exploit. Additionally, by disabling ASLR with the norandmaps parameter, the setup creates a predictable environment, which is essential for the success of speculative execution attacks. This predictability allows researchers to study the behavior and impact of such attacks in a controlled and repeatable manner

2.1.10 Simulation Execution and Checkpointing

The script provided presents a loop used to manage a simulation using the Gem5 simulation framework. Thus, the script manages the simulation execution and allows for the creation of checkpoints:

```
while True:
    event = m5.simulate()
    exit_msg = event.getCause()
    if exit_msg == "checkpoint":
        cpt_dir = os.path.join(m5.options.outdir, "cpt.%d" % m5.
    curTick())
        m5.checkpoint(os.path.join(cpt_dir))
```

This loop runs the actual simulation, calling ‘m5.simulate()‘ in a continuous loop until the simulation ends or a specific event (such as a checkpoint) occurs. Highlighting that the command event = m5.simulate() triggers the simulation to run. The simulate() function in the Gem5 framework advances the simulation and returns an event object. This object indicates the cause of the simulation pause or termination, such as a timer interrupt, checkpoint request, or a simulation exit condition. Checkpoints are useful because they save the state of the simulation at specific points in time, which can be restored later without restarting the entire process. This was particularly helpful during long simulations like the Spectre attack, where I could pause and resume the simulation efficiently. In other words, This script ensures that the simulation runs continuously, pausing and saving checkpoints whenever specified. These checkpoints allow the simulation to be resumed from specific points, which is particularly useful for long-running simulations, debugging, or analyzing specific segments of the process without having to restart the entire simulation from scratch. This functionality enhances efficiency and flexibility, making it easier to manage and study complex simulation scenarios.

2.1.11 Integration of PMU and Debugging with Tarmac Traces

This script presents an approach where the Performance Monitoring Unit (PMU) and Tarmac trace generation functionalities into the simulated system, enhancing its debugging and performance analysis capabilities.

```
if args.with_pmu:
    for cluster in system.cpu_cluster:
        interrupt_numbers = [args.pmu_ppi_number] * len(cluster)
        cluster.addPMUs(interrupt_numbers)
```

The PMU was used to monitor various hardware events (such as cache hits, misses, and branch predictions) during the attack simulations. By enabling the PMU in the script, I was able to collect performance data, which helped analyze how the system responded to speculative execution and branch prediction misbehavior [7]. This integration was achieved by iterating through each CPU cluster in the simulation system and assigning interrupt numbers to the PMUs corresponding to the cores within each cluster. The assigned interrupt numbers allowed the PMU to monitor and track the performance events specific to each core, such as cache behaviors and branch prediction outcomes. This configuration facilitated precise event monitoring, enabling insights into the system's performance at a granular level. Additionally, Tarmac trace generation was enabled in the script through the 'tarmacgen'optionprovidedadetailedlogof executedinstructions. This feature provided a detailed execution log of instructions, which allowed for in-depth debugging and profiling. The generated Tarmac traces offered a comprehensive view of the execution paths taken by the processor, making it possible to analyze the sequence of operations and identify potential performance bottlenecks or vulnerabilities. By combining PMU data and Tarmac traces, I was able to gain a detailed understanding of how the system behaved under conditions of speculative execution and branch prediction misbehavior. The performance data collected by the PMU highlighted the impact of hardware-level events, such as cache misses and mispredicted branches, while the Tarmac traces provided a step-by-step log of executed instructions. This combination proved invaluable for evaluating the system's vulnerabilities and overall performance during attack simulations, offering critical insights into both expected and anomalous behaviors.

2.1.12 Finalizing the Simulation

The final lines of the script complete the setup and initiate the simulation process by defining the root system, handling checkpoint restoration, and starting the simulation.

```
root = Root(full_system=True)
root.system = create(args)
if args.restore is not None:
    m5.instantiate(args.restore)
else:
    m5.instantiate()
run(args)
```

This part instantiates the full-system simulation, loading either from a checkpoint or from the start of the simulation. The Root object is created to define the entry

point of the system, and the system configuration is assigned based on the user-provided arguments. If a checkpoint is specified, the simulation state is restored using m5.instantiate(args.restore); otherwise, the simulation starts fresh with m5.instantiate(). The m5.instantiate() call initializes the gem5 system, after which the run(args) function starts the simulation loop. For the Spectre attack, after configuring the system and loading the disk image, the simulation is started, and the exploit binary is executed inside the simulated environment. This modularity and flexibility in setting up simulations through the script allowed me to easily configure and test different attack scenarios in gem5. By enabling checkpoint restoration, I could save simulation progress and resume it as needed, facilitating efficient experimentation and analysis of various configurations and vulnerabilities.

2.2 Using the Simulation Script in gem5

The starterfs.py script is an essential tool for kicking off and running full-system simulations within the gem5 framework, especially for ARM architectures. By using this script, you can simulate intricate system setups, including multi-core CPUs, custom kernels, and disk images. This capability enables a deep dive into system behavior, performance profiling, and testing specific scenarios like debugging or analyzing speculative execution vulnerabilities. The example below shows how to set up and run a simulation with this script, showcasing gem5's flexibility and power for research and development. Thus, The starterfs.py script is used to run a full-system simulation in gem5, with the command below providing a practical example:

```
./build/ARM/gem5.opt \
  ./configs/example/arm/starter_fs.py \
  --cpu=o3 \
  --num-cores=3 \
  --kernel=./system/arm/binaries/vmlinux.arm64 \
  --disk-image=./system/arm/linaro-minimal-aarch64.img \
  --debug-file=pipeview.txt \
  --debug-start=13062347000
```

2.2.1 Explanation of the Command

- `./build/ARM/gem5.opt`: This specifies the gem5 binary compiled for ARM systems, using the optimized version (gem5.opt) for simulation performance. It is located in the build/ARM/ directory and is used to run full-system simulations of ARM-based architectures.

- `./configs/example/arm/starter_fs.py`: This is the Python configuration script responsible for setting up the full-system simulation in gem5. It initializes the simulation components, including CPU, memory, kernel, and disk image, ensuring the system is ready for execution.
- `-cpu=o3`:This flag specifies the CPU model. In this case, it is an Out-of-Order (O3) CPU, which is suitable for simulating complex workloads, including those involving speculative execution, such as the simulations used for analysing the Spectre attack.
- `-num-cores=3`: This flag sets the number of CPU cores to be simulated. In this example, three CPU cores are simulated. This multi-core configuration allows the simulation to capture interactions between cores, which is crucial for testing scenarios like speculative execution vulnerabilities.
- `-kernel=./system/arm/binaries/vmlinux.arm64`: This specifies the ARM Linux kernel image that gem5 will use during the simulation. The kernel boots into the operating system, and this last is essential for booting into the operating system within the simulated environment.
- `-disk-image=./system/arm/linaro-minimal-aarch64.img`: This flag points to the root filesystem image used during the simulation. It provides the necessary software environment, including system libraries and tools, for running programs in the simulation.
- `-debug-file=pipeview.txt`: This flag enables detailed debug output and directs the output logs to a file named pipeview.txt. The debug logs are particularly valuable for tracing low-level CPU activities, such as cache hits/misses and speculative execution behavior.
- `-debug-start=13062347000`: This specifies the simulation tick at which debugging will begin. Delaying the debug output until a specific tick helps avoid unnecessary logs from the early stages of the simulation. This makes the debug information more focused and manageable, allowing for clearer insights into the simulation's behavior. This command sets up a full-system simulation with the ARM architecture in gem5. By combining kernel and disk images, it creates a bootable environment, and the multi-core configuration allows for thorough analysis. The debug options provide detailed event tracing, helping you better understand and profile the simulation's behavior.

2.2.2 Example Use Case in Simulation

In my experiments, this script was employed to set up a full-system simulation using the ARM AArch64 architecture. This simulation environment provided a robust

framework for evaluating how speculative execution vulnerabilities manifest and impact processor behavior. After compiling the exploit specifically for the ARM architecture, the exploit was executed within the simulated environment, enabling controlled testing and analysis of speculative execution phenomena. The parameters of the script were carefully adjusted to simulate a three-core system using the O3 CPU model. The O3 model, with its capability to emulate out-of-order execution, proved to be an essential choice for accurately representing the behavior of modern processors during speculative execution scenarios. This configuration ensured that the interactions between cores, speculative paths, and hardware-level events were captured realistically, thereby aligning with the requirements of analyzing vulnerabilities such as Spectre. The script's modular design allowed significant flexibility in customizing various simulation parameters. Adjustments such as memory configuration, number of CPU cores, and kernel specifications could be seamlessly incorporated to tailor the environment for specific experimental needs [10]. Additionally, the integration of debugging tools further enriched the analytical capabilities of the simulation. By enabling the debug-file and debug-start flags, I was able to collect pipeline-level data that detailed how the processor handled speculative execution paths, including critical events like branch predictions and cache behavior. This granular level of data was instrumental in understanding the impacts of speculative execution on system security and performance, shedding light on the vulnerabilities exploited by Spectre. The example script shown below illustrates the essential configuration for simulating such environments:

```
./build/ARM/gem5.opt \
  ./configs/example/arm/starter_fs.py \
  --cpu=o3 \
  --num-cores=3 \
  --kernel=./system/arm/binaries/vmlinux.arm64 \
  --disk-image=./system/arm/linaro-minimal-aarch64.img \
  --debug-file=pipeview.txt \
  --debug-start=13062347000
```

This setup created a thorough simulation environment, allowing for detailed observation and analysis of all critical aspects of the system's speculative execution behavior.

Chapter 3

Spectre Simulation

3.1 Introduction to spectre

In today's world, we rely heavily on a wide range of interconnected technological systems, making it crucial to ensure they run efficiently and securely. As these systems handle more and more sensitive data, protecting them from potential threats has become incredibly important. Computer security is a constant battle to prevent malicious actors from exploiting weaknesses in system design. Among the many threats to modern computing, Spectre attacks are particularly dangerous. Over the years, efforts to boost hardware performance have sometimes created unintended vulnerabilities. One such vulnerability is in the hardware architecture itself, which can be exploited to leak sensitive information through hidden side-channels. Side-channel attacks take advantage of subtle, unintended behaviours in hardware to extract data without the knowledge of running programs or operating systems. By observing things like cache behaviour, timing differences, or power consumption variations, attackers can gather valuable information from small, repeatable changes in system performance. The Spectre vulnerability is a prime example of this, using fundamental hardware optimizations like speculative execution to access and expose data in unexpected ways. Understanding and defending against Spectre attacks is not just a challenge for researchers and engineers; it's essential for protecting the technological systems that our modern lives depend on. Since the Spectre family of attacks became widely known in January 2018, there has been a flurry of activity among researchers, manufacturers, and enthusiasts to develop defenses against them. However, there has also been significant effort to bypass these defenses and

discover new side-channels and methods for executing Spectre-like attacks. This thesis presents and demonstrates a proof of concept for the well-known Spectre v1 vulnerability through simulation. The work focuses on analyzing and implementing the attack within two simulation environments: gem5 and QEMU. While prior work has demonstrated Spectre on gem5 using ARM architecture [1]and on x86 architectures [2], this thesis additionally contributes a Spectre simulation on QEMU, which, to the best of our knowledge, has not been previously demonstrated in this context.

The thesis first introduces the underlying principles of the Spectre vulnerability, detailing how speculative execution and cache-based side channels can be exploited to leak sensitive information. It then elaborates on the implementation and behavior of the Spectre attack in both gem5 and QEMU, highlighting the key challenges, differences, and opportunities in using each simulator. The experimental results demonstrate that Spectre-style information leakage can indeed be observed in both environments, reinforcing the effectiveness of simulation tools for studying microarchitectural attacks.

The Spectre vulnerabilities represent a groundbreaking and transformative discovery in the field of computer security, revealing a class of architectural flaws that stem from the very optimizations designed to improve processor performance. First disclosed by Google Project Zero alongside Meltdown, Spectre significantly shifted the industry's perspective on speculative execution and trust boundaries within modern CPUs.

While Meltdown primarily affects Intel processors, Spectre has a much broader reach, impacting CPUs from multiple vendors including Intel, AMD, and ARM. At its core, Spectre exploits speculative execution, where processors predict and execute instructions ahead of time to maintain high performance. Spectre cleverly manipulates this behavior to transiently execute unauthorized instructions and extract data via side-channel observations, such as cache timing differences.

The broader implications of Spectre go beyond its technical mechanisms, as it compels the computing industry to re-examine decades of performance-driven architectural decisions. Spectre reveals the tension between speed and security, and raises important questions about how future processors should balance efficiency with robust isolation guarantees

[1]Reproducing Spectre Attack with gem5, How To Do It Right?

[2]Spectre Attacks: Exploiting Speculative Execution

3.1.1 Our Technique

In this chapter, we show a new class of attacks which we call spectre attacks. In a Spectre attack, the goal is to bypass the usual protections that keep different processes from accessing each other's memory. This is done by taking advantage of how modern CPUs perform speculative execution, which is when the CPU guesses what instructions to execute next to improve performance. At its core, a Spectre attack breaches memory isolation by combining speculative execution with data exfiltration using microarchitectural covert channels. The process begins with the attacker identifying a specific sequence of instructions within the target process's address space. When executed, this sequence serves as a covert channel transmitter, enabling the leakage of sensitive information such as the victim's memory or register contents. The attacker then manipulates the CPU into speculatively and incorrectly executing this sequence. Even though the processor later detects the error and reverts the speculative execution[3], changes to certain microarchitectural components, like cache states, remain intact. These lingering effects allow the attacker to extract the leaked information by analysing the covert channel, effectively bypassing the safeguards meant to enforce memory isolation. The covert channel used in this implementation employs a Flush+Reload cache-based method, as described by Yarom and Falkner [Yarom2014]. This technique relies on observing cache state changes induced by speculative execution to infer secret data. Exploiting Conditional Branches involves manipulating the branch predictor to mis-predict the branch's direction[4]. Optimizing the transient execution window, as suggested by Ayoub and Maurice, improves reproducibility by increasing the duration of speculative execution through slow operations like division [5]. let's Consider the following exploitable code snippet

```
1 if (x < array1_size)
2 y = array2[array1[x] * 256];
```

Here, the variable x is under the attacker's control. The 'if' statement leads to a branch instruction, verifying that x's value falls within the legal bounds, ensuring the validity of the array1 access.

[3]Yarom, Y., Falkner, K. (2014). Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.

[4]P. Kocher et al. «Spectre Attacks: Exploiting Speculative Execution». In: Communications of the ACM 62.7 (2019), pp. 93–101 (cit. on pp. 3, 31, 32, 36, 37).

[5]Pier Ayoub and Clémentine Maurice. «Reproducing Spectre Attack with gem5: How To Do It Right? » In: 14th European Workshop on Systems Security (EuroSec'21). 2021, pp. 1–16. doi: 10.1145/3447852.3458715. url: (cit. on pp. 31, 37, 38, 41)

A key component of this attack lies in exploiting the behavior of conditional branches to induce branch misprediction. By manipulating the branch predictor, an attacker can deceive the processor into executing instructions outside the intended operational bounds.

In the described example, the attacker initially executes the relevant code using valid inputs to train the branch predictor to consistently anticipate a specific condition, such as $x <$ array1_size. Subsequently, the attacker provides an out-of-bounds value for x and ensures that the array1_size variable remains un-cached.

As a result, the processor CPU speculatively executes the instruction array2[array1[x] * 256]‖ with the malicious x, leading to unintended memory access.

Although the speculative execution is eventually invalidated when the branch condition is re-evaluated, the alterations to the processor's cache state caused by this speculative execution persist. By systematically varying the values of x and monitoring the resulting cache behaviour through techniques like timing analysis, the attacker can infer and reconstruct sensitive data from the victim's memory. This approach highlights the vulnerability of speculative execution mechanisms in modern processors to subtle yet highly effective forms of attack. In addition to conditional branches, Spectre attacks can target indirect branch instructions by leveraging the Branch Target Buffer (BTB) [6]. This component of the processor predicts the target of indirect branches based on historical execution patterns. An attacker selects a "gadget," a sequence of instructions within the victim's address space that can be executed speculatively to leak sensitive information. The BTB is then mis-trained by the attacker through repeated indirect branches from their own address space to the address of the gadget. While the content at this address in the attacker's space is inconsequential, the mis-training causes the victim's branch predictor to misdirect speculative execution to the attacker-chosen gadget. Although the speculatively executed instructions are eventually discarded, their impact on the cache remains, enabling data leakage through timing analysis. Exploiting indirect branches, inspired by return-oriented programming (ROP), involves using a victim's address space to speculatively execute a gadget. Unlike traditional ROP, which needs a vulnerability in the victim's code, this method bypasses such dependencies by manipulating the BTB. The attacker does this by training the BTB to mispredict an indirect branch instruction, redirecting its execution to the gadget's address. Although the instructions executed during this speculative phase are eventually discarded, their residual effects on the cache remain. These residual cache effects act as a side channel, allowing the gadget to leak sensitive information. By carefully choosing a suitable gadget, this approach can

[6]Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Yarom, Y. (2018). Spectre attacks: Exploiting speculative execution. Defiant. Retrieved from

access arbitrary memory regions within the victim's address space. To effectively mistrain the BTB, the attacker identifies the virtual address of the target gadget within the victim's address space and initiates indirect branches to this address [7]. This training process is done entirely from the attacker's address space and does not require executable code at the corresponding gadget address in the attacker's own space. Crucially, only the destination virtual address of the branch matters, regardless of whether actual code exists there. This means the attack can still work even if no code is mapped at the specified virtual address, as long as the attacker can manage exceptions effectively. Additionally, a precise match between the source address of the attacker's training branch and the address of the victim's targeted branch is not necessary. This flexibility gives attackers significant freedom in crafting their training mechanism, enhancing the feasibility and stealth of the method. To improve the efficiency and reliability of these attacks, the transient execution window the duration in which speculative execution occurs can be extended. Techniques such as introducing deliberately slow operations, such as division, into the speculative execution path have been shown to enhance the ability to observe and manipulate cache state changes. These methods amplify the success of attacks by providing a longer time frame for speculative effects to take place and persist. The described techniques underscore the inherent vulnerabilities in speculative execution and branch prediction mechanisms. By exploiting covert channels like cache timing and leveraging both conditional and indirect branches, Spectre attacks represent a sophisticated method of breaching memory isolation and accessing sensitive information, exposing the critical need for architectural and software-based defenses in modern processors. It's necessary to understand here that Spectre attacks rely on exploiting the speculative execution capabilities of modern CPUs to bypass memory isolation and gain access to sensitive information. The attacker begins by locating specific instruction sequences within the victim's process that can serve as covert channels for leaking confidential data. By manipulating the CPU into speculatively executing these instructions, the attacker gains temporary access to data that would otherwise remain securely protected. Although the CPU is designed to roll back incorrect speculative execution to preserve its normal state, changes to underlying microarchitectural components, such as cache contents, often persist. These residual effects enable the attacker to extract the leaked information by analysing the altered microarchitectural state. This method underscores the critical vulnerabilities posed by speculative execution and its implications for data security in modern computing systems.

[7] Ibid

3.2 Spectre Exploit

The Spectre exploit reveals a significant vulnerability in modern processors that rely on speculative execution and branch prediction to enhance performance. Speculative execution enables CPUs to predict the likely path of future instructions and execute them pre-emptively, minimizing delays caused by waiting for data retrieval from memory. While this optimization boosts efficiency, it can be exploited by attackers to execute instructions that deviate from the intended behaviour of a program. By inducing the CPU to perform these unintended speculative operations, attackers can leak sensitive information from other processes' memory, effectively compromising the isolation mechanisms designed to protect confidential data. The initial Spectre exploit code used in this chapter was sourced from ErikAugust's GitHub repository [8]. However, as the original code was crafted for the x86 architecture, it required significant modifications to adapt it for ARM assembly. These adaptations included rewriting components such as the spectre.h library to align with ARM's architectural specifications. The provided code illustrates a typical Spectre exploit scenario. In this case, the attacker aims to leak sensitive information by exploiting the processor's speculative execution capabilities. The exploit consists of two main components: the victim code and the analysis code. The victim code includes a function called victim function, which accesses an array (array1) based on an index provided by the attacker. Due to insufficient validation of the index, the attacker can exploit this function to access out-of-bounds memory locations, thereby exposing data that should remain protected. The analysis code complements the attack by leveraging timing side-channel techniques to infer the values of memory accessed by the victim function. By measuring the time, it takes to access different memory locations, the attacker can reduce the value stored in the victim's array. This combination of speculative execution manipulation and timing analysis demonstrates the critical security vulnerabilities exposed by Spectre attacks and highlights the importance of addressing such threats to safeguard sensitive information. Thus, the core mechanism of the Spectre exploit revolves around transient instructions—operations executed speculatively and subsequently discarded when the CPU identifies a prediction error. Attackers craft scenarios to trigger branch mispredictions, causing the CPU to execute these transient instructions. By carefully controlling the input to a victim program, they manipulate the branch predictor to execute instructions that access sensitive data within the victim's memory space. This data is then exfiltrated through side channels, such as cache timing variations, which reveal the values of leaked information based on the time taken to access certain memory locations. The

[8]Erik August. Spectre PoC Exploit Code. (cit. on p. 37).

implications of the Spectre exploit are far-reaching, as it undermines fundamental security assumptions underlying techniques like sandboxing, process isolation, and memory safety. The exploit's cross-platform and cross-architecture applicability compounds the threat, exposing vulnerabilities in a wide range of systems that employ speculative execution. Addressing these risks requires a comprehensive approach, encompassing software updates and architectural changes to prevent speculative execution from inadvertently leaking sensitive information. The Spectre exploit underscores a critical tension between performance optimization and security in modern computing systems. By exploiting the complexities of speculative execution, attackers can bypass traditional security defenses, potentially leading to severe data breaches. As cybersecurity challenges continue to evolve, effectively mitigating such vulnerabilities will be paramount to ensuring the protection of sensitive information in an increasingly interconnected digital landscape.

3.3 Simulating Spectre Attacks with gem5

In the simulation of Spectre attacks, we utilized the gem5 simulator, specifically version 20.0, to conduct a full-system simulation [9]. The primary operating system for this setup was Linaro Minimal Linux, tailored for the AArch64 architecture, which is compatible with ARM processors. The Linux kernel version employed in this simulation was vmlinux.arm64, providing a suitable environment for testing the exploit. To effectively simulate the Spectre attack, we incorporated the source files named spectre.c and spectre.h [10], which contain the necessary code to execute the attack scenarios. This setup allowed us to analyse the behaviour of the Spectre exploit within a controlled environment, facilitating a deeper understanding of its mechanics and potential impacts on system security.

3.3.1 Compilation of Spectre Exploit

To compile and run the Spectre exploit within the gem5 simulation environment, we followed the steps outlined by Ayoub. This process ensures is properly adapted for ARM-based architectures. Compiling the code correctly is a crucial step in preparing the exploit to function within the simulation framework and ensuring it behaves as intended during execution.

[9]Binkert, G. N.and Beckmann B.and Black, Reinhardt, S. K., Saidi, A. A.and Basu, and M. Shoaib. «The gem5 simulator». In: ACM SIGARCH Computer Architecture News 39.2 (2011), pp. 1–7 (cit. on pp. 29, 35, 38, 47, 56)

[10]Pier Ayoub and Clémentine Maurice. «Reproducing Spectre Attack with gem5: How To Do It Right?» In: 14th European Workshop on Systems Security (EuroSec'21). 2021, pp. 1–16. doi: 10.1145/3447852.3458715. url: (cit. on pp. 31, 37, 38, 41)

```
aarch64-linux-gnu-gcc -Wall -g3 -march=armv8-a -static -O0 spectre
    .c -o spectre
```

Here, the aarch64-linux-gnu-gcc compiler is used, which is tailored for generating binaries for the ARM architecture (AArch64). The -Wall flag enables all common warnings to assist in debugging and code quality checks, while the -g3 option includes maximum debugging information in the binary, aiding in the analysis of the exploit's behaviour during runtime. The -march=armv8-a parameter specifies the target architecture as ARMv8-A, ensuring the generated code is optimized for this specific processor design. The -static flag produces a static binary, which embeds all necessary libraries, making it self-contained and simplifying execution in the gem5 simulation environment. Lastly, the -O0 optimization level is chosen to disable compiler optimizations, ensuring the exploit code remains as close to the original source as possible for predictable and consistent behaviour. The successful execution of this command generates a static binary named spectre, which is specifically compiled for the ARM architecture. This binary serves as the executable for simulating the Spectre attack within the gem5 framework. Adhering to this compilation process is essential to ensure the exploit functions correctly, providing the necessary foundation for studying its behaviour and implications in a controlled environment.

3.3.2 Initiating the Simulation

To start the gem5 simulation for the Spectre exploit, we used a structured approach designed to configure the environment accurately and ensure compatibility with the ARM architecture. The process involves navigating to the appropriate gem5 directory and executing a command to launch the simulation. This command specifies various parameters that enable the simulation of a system capable of exposing the vulnerabilities exploited by Spectre. The steps to initiate the simulation are as follows:

```
cd Gem5_directory
./build/ARM/gem5.opt ./configs/example/arm/starter_fs.py --cpu=o3
    --num-cores=3 --kernel=./system/arm/binaries/vmlinux.arm64 --
    disk-image=./system/arm/linaro-minimal-aarch64.img
```

This command sets up and runs the gem5 simulation with specific parameters tailored to the ARM architecture. Below is a detailed breakdown of the command and its components

Breakdown of the Command

- `./build/ARM/gem5.opt:` This specifies the optimized gem5 binary built for the ARM architecture. The optimized build is essential for running simulations efficiently and accurately
- `./configs/example/arm/starter_fs.py:` This script initializes the full-system simulation for ARM-based systems, providing a foundational setup for running the Spectre exploit.
- `-cpu=o3:` This flag selects the Out-of-Order (O3) CPU model, which is critical for simulating speculative execution. Since Spectre exploits speculative execution behaviour, using the O3 model ensures that the simulation accurately represents the processor's vulnerabilities.
- `-num-cores=3:` This parameter configures the simulation to use three CPU cores, offering a multi-core environment that aligns with modern processor designs.
- `-kernel=./system/arm/binaries/vmlinux.arm64:` This flag specifies the Linux kernel binary for the simulation. The vmlinux.arm64 kernel is prebuilt for the ARM architecture and provides the necessary operating system support for the simulation.
- `-disk-image=./system/arm/linaro-minimal-aarch64.img:` This option points to the disk image used in the simulation. The Linaro Minimal AArch64 image provides a lightweight operating system environment suitable for testing and analysing the Spectre exploit.

The default values in the command are enough to run the simulation effectively. However, you can adjust additional parameters to change the simulation environment, such as modifying memory configurations or system topology. These adjustments provide more flexibility to study different aspects of the exploit or meet specific research needs. By following these steps, you can successfully initiate the gem5 simulation environment. This setup creates a controlled setting to analyse the Spectre exploit, which is crucial for observing how speculative execution behaves in a simulated ARM system. It helps in gaining a deeper understanding of the exploit's mechanisms and its potential impact on processor security.

3.3.3 Booting the System and Running the Attack

Before running the Spectre attack in the simulated environment, you need to boot the system within gem5. This booting process can take a significant amount of time, usually up to one hour. Once the system is fully booted, it will be ready to execute the Spectre exploit.

Mounting the Disk Image

The first step in preparing for the attack involves mounting the disk image to transfer the compiled Spectre binary. This is accomplished using the gem5img utility:

```
python3 gem5img.py mount ../system/arm/linaro-minimal-aarch64.img
    /mnt/
```

After mounting the image, copy the compiled Spectre binary into the mounted directory. This step ensures that the exploit is available to the simulated system when it boots. Once the binary is successfully copied, unmount the disk image to save the changes:

```
python3 gem5img.py umount /mnt/
```

Compiling the Spectre Exploit

To ensure compatibility with the simulated ARM architecture, the Spectre exploit must be compiled using the ARM-specific compiler:the Spectre exploit using the following command:

```
aarch64-linux-gnu-gcc spectre.c spectre.h -o spectre_me
```

After compilation, the binary must be made executable to allow its execution within the simulation:

```
sudo chmod +x spectre_me
```

This process creates an executable binary named `spectre_me`, which is transferred to the simulated system during the disk mounting phase.

3.3.4 Running the Simulation

Once the disk image is prepared and the system is booted up, the next thing to do is connect to the simulated environment and run the exploit. To do this, you'll use m5term, a handy terminal tool for interacting with gem5 simulations. Open a new terminal window, start m5term, and set it to listen on port 3456:

```
./util/term/m5term 3456
```

As the system boots up, you'll see initialization messages in the terminal. Once the boot sequence is complete, the system will be fully operational and ready to run the Spectre attack. At this point, you can execute the prepared `spectre_me` binary to exploit the vulnerabilities simulated in the gem5 environment. These steps ensure that the simulated system is set up correctly and can demonstrate the Spectre attack's behaviour in a controlled setting. This comprehensive setup allows for an in-depth analysis of the exploit and its implications for processor security.

3.3.5 Creating and Restoring Snapshots

To make the simulation process smoother and skip the lengthy boot sequence for future runs, it's a good idea to create a snapshot of the system once it's fully booted. This snapshot captures the system's current state, so you can restore it later without going through the entire initialization again. The snapshot is saved in the m5out/cpt.ticknumber directory, where ticknumber is the simulation tick count at the time the snapshot was taken. After the system has booted and is ready, you can terminate the simulation by pressing Ctrl+C. This will save the current system state into the snapshot directory. The snapshot contains all the necessary data to resume the simulation exactly where it left off. To restore the system from the saved snapshot, use the following command:

```
./build/ARM/gem5.opt ./configs/example/arm/starter_fs.py --cpu=o3
   --num-cores=3 --kernel=./system/arm/binaries/vmlinux.arm64 --
   disk-image=./system/arm/linaro-minimal-aarch64.img --debug-file
   =pipeview.txt --debug-start=13062347000 --restore=./m5out/cpt
   .1958869111750/
```

Here, the command is similar to the one used for initializing the simulation, with additional parameters is included to restore the snapshot Here, the command is similar to the one used for initializing the simulation, with additional parameters is included to restore the snapshot.

- `-debug-file=pipeview.txt`: This parameter specifies the file to store debugging information, which can be useful for analysing system behaviour during restoration.
- `-debug-start=13062347000`: This parameter indicates the simulation tick count at which debugging begins, enabling targeted analysis from the moment the snapshot is restored.

- `-restore=./m5out/cpt.1958869111750/`: This parameter contains points to the directory for the snapshot to be restored. The tick number in the folder name must correspond to the tick count of the saved snapshot.

Thus, can be understood that you can only restore a snapshot if the system configuration is the same as when you created it. If you change any critical components, like the kernel binary, disk image, or CPU model, the snapshot won't work. In that case, you'll need to reboot the system and create a new snapshot after it finishes booting. Using snapshots makes the simulation process much more efficient. It cuts down on downtime and allows for quick iterations during testing and analysis. This is especially useful for long simulations, as it saves a lot of time while keeping the system state consistent.

3.3.6 Executing the Spectre Exploit

Once the operating system within the simulated environment has successfully booted and the prepared Spectre binary has been transferred to the system, the exploit can be executed. This step demonstrates the core functionality of the Spectre attack within the simulation. To initiate the exploit, run the compiled binary using the following command within the simulated environment:

```
./spectre_me
```

This command launches the Spectre attack, leveraging the simulated CPU's speculative execution capabilities to breach memory isolation boundaries and demonstrate the hack. The binary `spectre_me` is a program that contains the code necessary to manipulate the processor's speculative execution behaviour and eliminate sensitive data using side-channel techniques.

3.3.7 Limitations of SE Mode for Spectre Attacks

In gem5's system-emulation (SE) mode only user-level code is executed and system calls are emulated by the simulator rather than handled by a real kernel, so there is no true OS context or scheduling. Consequently, many OS-driven features are highly simplified: for example, in SE mode page tables are kept in internal data structures and "there is no such thing as a pagewalk" on a TLB miss diva-portal.org . Gem5's own documentation explicitly notes that SE mode "ignores the timing of many system-level effects including system calls, TLB misses, and device accesses".[11] Spectre attacks rely on precise speculative-execution and cache-timing interactions

[11]The gem5 Simulator

within a full OS-managed environment (including real page walks, context switches, interrupt handling, etc.), so the stripped-down SE environment (incomplete page-table/TLB modeling, simplified syscalls, no real kernel) can prevent the attack from functioning. In contrast, the full system (FS) mode boots an actual OS kernel and fully models address translation and interrupt behavior, providing the realistic microarchitectural context that Specter exploits require.

3.4 Optimizing Simulation Time and Visualizing Pipeline Behavior

In the exploration of Spectre attacks, understanding the intricate behaviours of CPU pipelines during speculative execution is crucial. This section focuses on optimizing simulation time and visualizing pipeline behaviour while using the gem5 simulator to analyse the Spectre exploit. Given the complexity of modern processors, it is essential to efficiently manage system resources to ensure that simulations yield meaningful insights without overwhelming the system. To achieve this, we recommend running the Spectre binary for only a brief period. Extended execution can lead to the generation of excessively large debug files, which can hinder analysis and consume significant computational resources. After successfully booting the operating system and executing the Spectre binary, observing the attack's output and pipeline behaviour becomes the next critical step. However, optimizing the simulation time is essential to avoid excessive resource consumption while capturing the necessary data for analysis. Executing the Spectre attack for an extended period can produce an overwhelming amount of debug information, potentially generating files as large as 5 GB within a few seconds. Such large files can be challenging to process and analyse. Therefore, a more efficient approach is to run the attack briefly, long enough to capture the desired pipeline activity, and then terminate the simulation.
Steps to Optimize Simulation Time

- Execute the Spectre Binary Briefly: Run the binary `spectre_me` binary within the simulated environment, allowing the attack to proceed for only a few seconds.

- Terminate the Process: After observing sufficient activity, stop the execution by pressing `Ctrl+C`. This action halts the simulation and ensures the debug file remains at a manageable size.

By limiting the execution time, you can capture essential information about the attack's impact on the processor pipeline without overwhelming system resources or generating excessively large debug files. Benefits of this Approach

- Efficient Data Management: Restricting the simulation time helps maintain debug file sizes that are easier to analyse and process.
- Focused Insights: The shortened runtime ensures that only the most relevant pipeline activities are recorded, facilitating targeted examination of speculative execution and cache behaviour.
- Resource Optimization: Minimizing the simulation duration conserves computational resources, allowing for faster iteration and analysis. This method allows researchers to quickly identify and visualize key aspects of the Spectre attack, such as speculative execution patterns and side-channel activities. By efficiently capturing debug data, this approach enhances understanding of the attack dynamics while preserving system usability.

In summary, by optimizing simulation time and focusing on pipeline visualization, we can enhance our understanding of how Spectre attacks operate, ultimately contributing to the development of more robust security measures against such vulnerabilitie

3.5 Visualizing Spectre with gem5

To visualize the Spectre attack within the gem5 simulation environment, a systematic approach involving trace generation, pipeline visualization, and code analysis was employed. The process began with executing the following command to initiate a trace of the simulation:

```
./build/ARM/gem5.opt --debug-file=pipeview.txt -debug-start
   =13062347000
./configs/example/arm/starter_fs.py --cpu=o3 --num-cores=3
--kernel=./system/arm/binaries/vmlinux.arm64
--disk-image=./system/arm/linaro-minimal-aarch64.img
```

To ensure a clear and precise trace, the –debug-file and –debug-start flags were included. These parameters excluded unnecessary boot initialization steps, focusing the trace on the Spectre attack execution. The specific value for the –debug-start parameter was determined by terminating the gem5 simulation using Ctrl+C after booting, which displayed the relevant tick number in the terminal. This tick value was then used as the starting point for debugging. Allowing the simulation to run for approximately one minute produced a trace file of about 3.1 GB. Despite its size, this file provided critical insights into the pipeline's behaviour during the Spectre attack, enabling detailed examination of speculative execution patterns and side-channel vulnerabilities. The resulting trace was visualized using Kotana,

which offered a detailed view of the pipeline's behaviour during the Spectre attack. This visualization allowed for a meticulous examination of the attack patterns and their impact on the processor's microarchitecture. Subsequently, I conducted an object dump of the Spectre binary to gain insight into the disassembled code. This step was essential in verifying that the pipeline executed as expected. To accomplish this, I executed the following command:

```
sudo aarch64-linux-gnu-objdump -d spectre > objectdump-spectre.txt
```

The command I executed, which generated the file objectdump-spectre.txt, was instrumental in our analysis of the Spectre attack. The resulting objectdump-spectre.txt file provided a human-readable representation of the binary's assembly instructions. This step was critical for verifying the sequence of executed instructions and for identifying the interaction between the attack code and the speculative execution mechanisms of the processor. By parsing through this file, I gained valuable insights into the sequence of instructions executed by our gem5 machine during the simulation of the Spectre attack. Analysing the disassembled code allowed me to trace the precise path of execution within the Spectre binary. Each assembly instruction provided a glimpse into how the attack was orchestrated at the machine code level. By following the flow of instructions, I could discern the specific operations performed by the attack code, including memory accesses, branch predictions, and speculative execution paths.

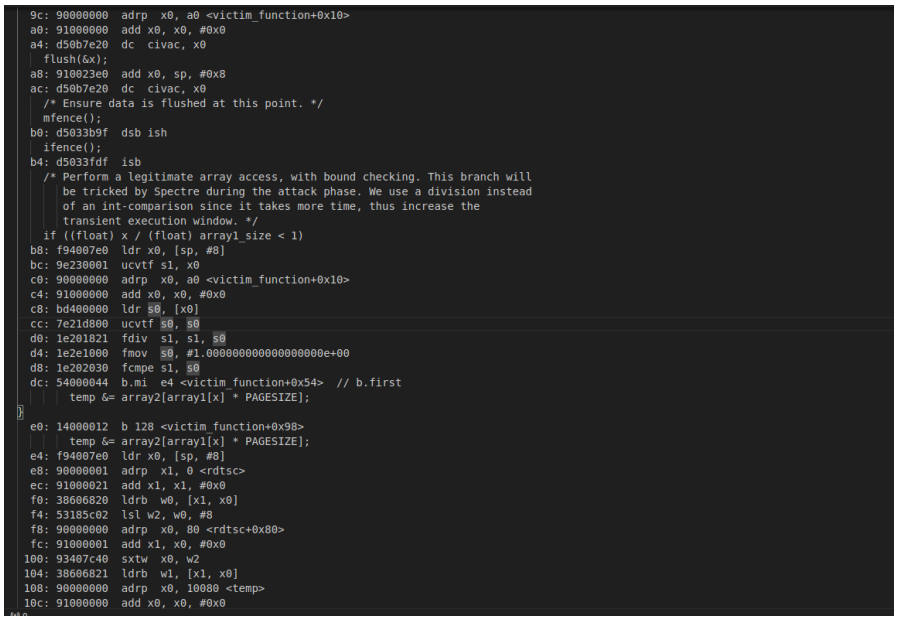

Figure 3.1: figure of my disassembled Spectre code

To conduct a comprehensive analysis of the Spectre attack, the investigation focused on the main loop, which serves as the central component of the exploit. This deliberate emphasis enabled an in-depth examination of the attack's operational mechanisms and its interaction with the underlying system architecture. By concentrating on this critical segment of the code, valuable insights were obtained into the intricate execution patterns of the attack. The conditional branches within the main loop emerged as particularly significant points of interest. These branches dictated the flow of execution and offered vital information about the attack's behavior. A detailed examination of these branches revealed the nuanced conditions that trigger speculative execution and the speculative paths that the processor follows during this phase. Further analysis involved a deeper exploration of the disassembled code to identify and evaluate specific attack patterns. This detailed investigation illuminated the technical intricacies of Spectre's operations, demonstrating how the attack exploits speculative execution to breach memory isolation and access privileged information. By analysing these patterns, a nuanced understanding of the attack's methodology was developed, particularly its reliance on branch prediction and cache timing side-channel vulnerabilities. Through meticulous scrutiny of the main loop and a granular dissection of specific attack patterns in the disassembled code, the core mechanics of the Spectre attack were elucidated. This thorough analysis not only enhanced understanding of the attack's behaviour but also highlighted the broader implications of speculative execution vulnerabilities in modern processor architectures. The findings underscore the critical need to address these vulnerabilities to ensure robust security in contemporary computing environments.

Figure 3.2: Visualization of the main loop of the Spectre attack.

Additionally, the analysis extended to examining specific attack patterns within the disassembled code, offering a more detailed understanding of the underlying mechanics of the Spectre attack. This process involved a methodical review of

the disassembled binary to identify key operational sequences and their roles in facilitating the attack. By focusing on these patterns, the nuanced interplay between speculative execution and microarchitectural vulnerabilities was further clarified. For example, the disassembly revealed the function of the time stamp counter (mr cntvct-el0), a critical component that provides fine-grained timing information. The observation and analysis of this function within the context of the disassembled code were pivotal in validating the execution of the attack code within the gem5 simulation environment. By correlating the behaviour of the time stamp counter with other attack elements, the analysis confirmed that the simulation faithfully replicated the intended operations of the Spectre exploit, ensuring that the results were both accurate and representative of real-world scenarios. Moreover, the output generated from the binary Spectre execution underwent a detailed examination. This output not only confirmed the successful execution of the attack but also provided quantitative insights into its effectiveness. The scoring of the attack, derived from its ability to retrieve sensitive information, was carefully analysed to evaluate the precision and reliability of the exploit. Each data point in the output offered a clearer picture of how the attack leveraged speculative execution to bypass memory isolation and extract privileged data. By integrating the insights gained from scrutinizing the disassembled code, observing the time stamp counter's function, and analysing the execution output, a holistic understanding of the Spectre attack's functionality was achieved. This extended analysis highlighted the interplay of timing, speculative execution paths, and microarchitectural vulnerabilities, offering a detailed perspective on the attack's technical foundation. These findings underscore the necessity of addressing such vulnerabilities to safeguard the integrity and confidentiality of data in modern computational environments.

For instance, by scrutinizing the disassembly, I could observe the function of the time stamp counter (`mr cntvct-el0`). This examination allowed me to corroborate the execution of our code within the gem5 simulation environment.

Looking at the results from running the Spectre binary, this gave us important information about how well the attack worked. The following example shows that we successfully retrieved sensitive information:

```
1
2
3 Reading 40 bytes:
4 Reading at malicious_x = 0xfffffffffffffeb6... 999 999
5 Reading at malicious_x = 0xfffffffffffffeb7... 999 999
6 Reading at malicious_x = 0xfffffffffffffeb8... 999 999
7 Reading at malicious_x = 0xfffffffffffffeb9... 999 999
8 Reading at malicious_x = 0xfffffffffffffeba... 999 999
9 Reading at malicious_x = 0xfffffffffffffebb... 999 999
```

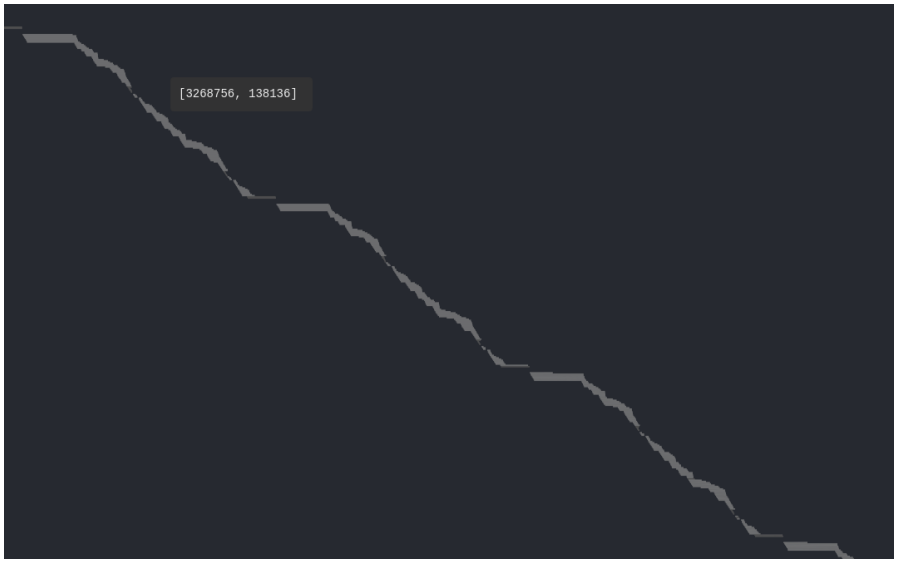

Figure 3.3: Identification of attack patterns within the disassembled code.

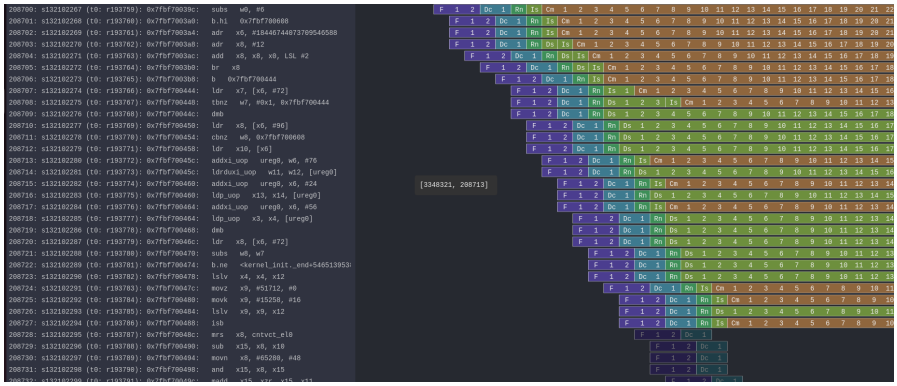

Figure 3.4: Verification of code execution within the gem5 simulation environment.

This detailed analysis provided valuable information on the execution and success of the Specter attack within the gem5 simulation environment. By analyzing traces, visualizing the pipeline, and disassembling the binary, we gained a complete understanding of the Specter attack. These insights revealed the details of speculative execution vulnerabilities and highlighted the bigger security issues in modern processors. This careful study of the Spectre attack shows how important it is to fix these vulnerabilities to protect data confidentiality in today's computing environments.

3.6 Simulating Spectre Attacks with QEMU User-Mode

We adapted the original Spectre V1 proof-of-concept to run on AArch64 and executed it under QEMU's user-mode emulation. In user-mode, QEMU launches a Linux process for one architecture on another via dynamic binary translation qemu. We cross-compiled the C exploit for ARM, producing a statically linked ARM executable. This ARM binary relies only on standard syscalls and cache-flush instructions, which the QEMU user mode can translate and execute on the host CPU.
In our setup, running:

```
qemu-aarch64 ./spectre
```

launches the exploit in QEMU without a full guest OS, relying on the real processor's pipeline to perform speculative execution.

Execution in QEMU. Under QEMU user-mode, the exploit consistently recovers the address of the targeted memory with the score based on timing attack. Figure shows sample output: the program prints repeated lines such as

```
Reading at malicious_x = 0xfffffffffff4f64... 999 999
```

The score "999" indicates that the predicted byte matches the secret and receives the maximum score (999 out of 999). In the PoC, a high score denotes a successful cache hit on the secret value (following the Flush+Reload side channel) These results confirm that the Spectre gadget executed as intended: after training the branch predictor, the malicious out-of-bounds access speculatively loads the secret into cache, and the subsequent timing measurements yield the highest score on the secret index.

Comparison with gem5 SE Mode. By contrast, running the same ARM binary in gem5's syscall-emulation (SE) mode did not complete successfully. In our gem5 experiment, the simulation aborted with a segmentation fault, The backtrace reveals a GDBSignal::SEGV libc abort indicating that gem5 terminated when the exploit code attempted the speculative access or cache flush. This suggests that under gem5 SE, the speculative load either did not occur or was not reflected in architectural state.

In practice, gem5 can achieve cycle-accurate microarchitecture simulation (with branch predictors and caches), but in its default SE setup our Spectre exploit crashed before revealing any data.

Simulation Fidelity Discussion. These results highlight the trade-off between simulator speed and accuracy. QEMU user-mode is extremely fast (it simply runs the translated code on the host CPU), but it does not guarantee cycle-accurate timing or model internal CPU state. It succeeded here only because it leveraged the real processor's speculative execution.
Gem5, conversely, is a cycle-accurate ARM simulator (with out-of-order cores, branch predictors, and caches) and is often used for security research. In principle, gem5 can reproduce Spectre. Our gem5 SE run (with a DerivO3CPU) failed, likely due to unmodeled hazards or lack of forwarded cache state.

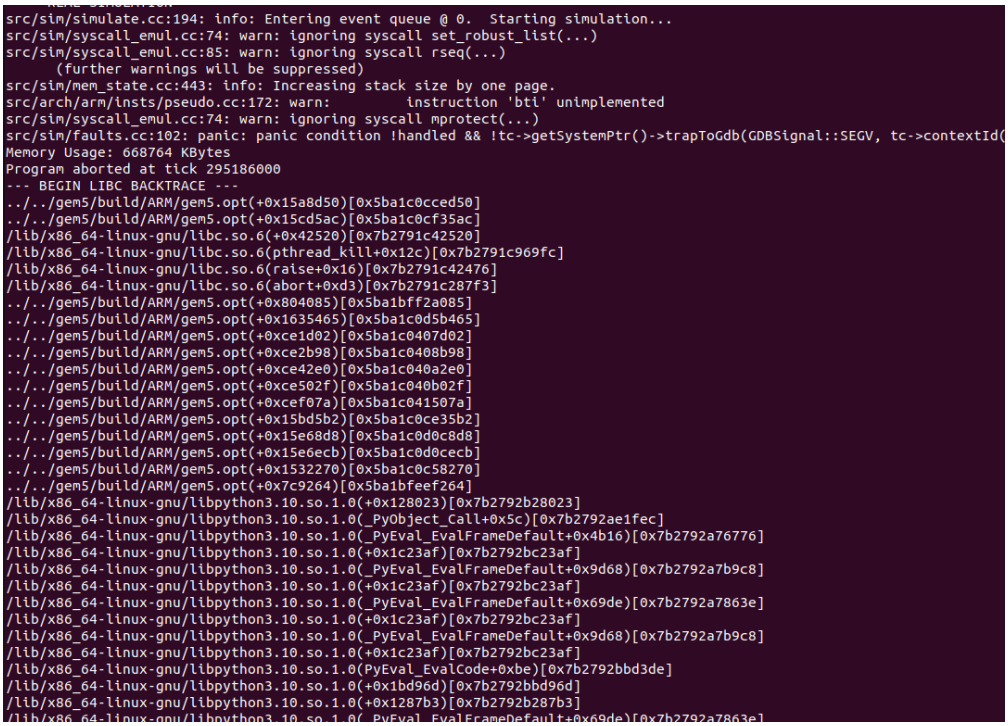

Figure 3.5: Gem5 SE-mode execution of the Spectre PoC on ARM, aborting with a segmentation fault. The libc backtrace shows a `GDBSignal::SEGV` trap at an invalid address, indicating that the exploit did not complete under gem5's SE mode.

Compilation. We compiled the Spectre C code for ARM using a cross-compiler `aarch64-linux-gnu-gcc` with appropriate flags. A command such as:

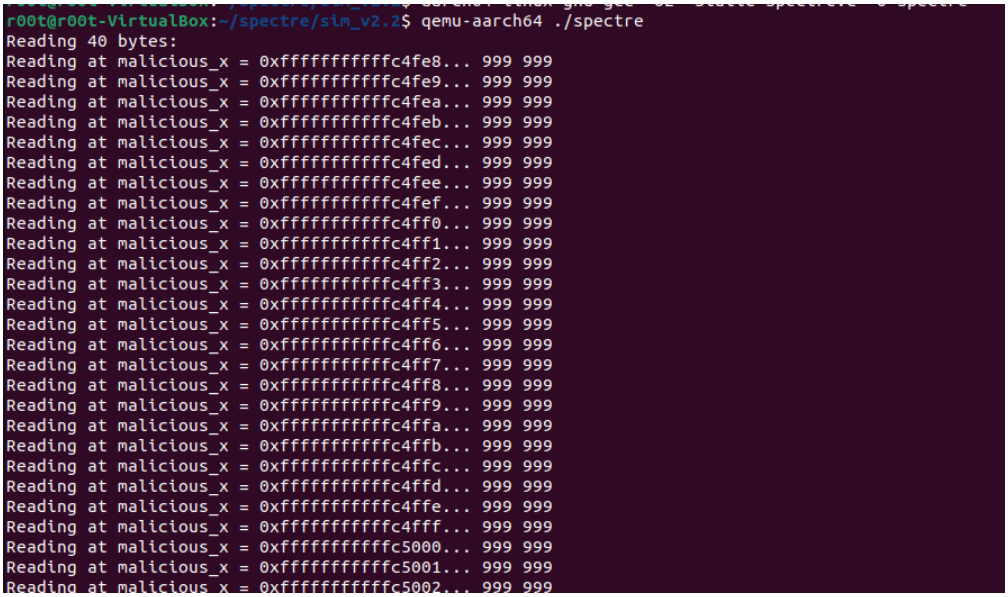

Figure 3.6: Output of the Spectre PoC under QEMU user-mode (AArch64). Each line shows the guessed byte and its score; the repeated "999 999" indicates that the secret byte was correctly identified with maximum confidence in each trial (demonstrating successful Flush+Reload timing attacks).

```
aarch64-linux-gnu-gcc -O2 -static spectre.c -o spectre
```

produces an AArch64 ELF binary. The code uses standard Linux syscalls and ARM equivalents of `mfence`/`dsb` plus cache flush (`dc civac`) instructions, so it runs unmodified on a Linux/AArch64 process under QEMU.

Execution. We launched the compiled binary directly via QEMU:

```
qemu-aarch64 ./spectre
```

The program outputs the number of bytes to read, then performs the Spectre attack loop. In each iteration it flushes `array2[]` from cache, trains the branch predictor, and invokes the victim access. It then probes all indices to find which cached line has low access time. The result line reports the index with the highest score. As shown in Figure 3.6, the guessed index matches the secret (printed after

`malicious_x`) with a score of 999 each time, confirming the secret was leaked. This matches the Spectre attack pattern: the secret data index repeatedly yields a cache hit in Flush+Reload.
In summary, QEMU user-mode provided fast, functional execution (revealing the secret), but it sidesteps microarchitectural fidelity; gem5 provides accuracy (in principle) but in our simple SE setup it failed to reproduce the attack without deeper configuration. These findings align with the broader conclusion that fast functional emulators can execute Spectre code but may not faithfully model the side-channel dynamics, which require cycle-accurate simulation to analyze fully.

3.6.1 Accuracy of Spectre Attack Simulation in gem5 Compared to Real Hardware

Simulating Spectre attacks in gem5 provides a valuable research tool for exploring speculative execution vulnerabilities in modern CPUs. However, assessing the accuracy of these simulations compared to real hardware is critical to understanding the reliability and limitations of the results. Several factors influence the accuracy of Spectre attack simulations, including how well gem5 models speculative execution, cache behavior, and memory systems in comparison to actual processors.

Speculative Execution and Branch Prediction

Real Hardware: Modern CPUs use complex branch prediction mechanisms that are essential for speculative execution, which is a key element of Spectre attacks[12]. These predictors adapt based on the history of past branch decisions, allowing CPUs to speculatively execute instructions before branches are resolved. The effectiveness of this speculation, however, depends on the processor's ability to make accurate predictions, which is not always perfect and can lead to speculative execution vulnerabilities like Spectre.
Gem5: Gem5 models speculative execution using the DerivO3CPU, which supports out-of-order execution and basic branch prediction mechanisms. However, the branch prediction model in gem5, typically a Tournament Branch Predictor, is simpler than the highly sophisticated ones in modern processors[13]. As a result, while gem5 can simulate speculative execution, the attack's effectiveness may vary due to these simplified prediction models. This discrepancy can impact the accuracy

[12]Modern processors from Intel and AMD use dynamic branch prediction systems such as the Tournament Branch Predictor, which is more advanced than the static approaches found in simulators.

[13]Gem5's simpler branch predictors may not fully replicate the dynamic nature of real hardware's prediction systems, leading to inaccuracies in the simulation of Spectre attacks.

of Spectre attack simulations, as the success of such attacks depends on the CPU's ability to predict and speculatively execute instructions.

Cache Timing and Side-Channel Attacks

Real Hardware: Cache side-channel attacks, such as Flush+Reload and Prime+Probe, rely on fine-grained differences in the timing of cache accesses to infer sensitive data during speculative execution[14]. Real hardware's cache hierarchies, which include L1, L2, and L3 caches, have non-uniform access times that are crucial for the success of these attacks. In addition, real hardware's precise timing measurements, typically on the order of nanoseconds, enable high accuracy in detecting timing differences that reveal secret data.
Gem5: Gem5's ability to model cache behavior is relatively accurate, but the timing granularity in gem5 is limited by the simulator's clock cycles, which are typically less precise than the nanosecond-level timing found in real hardware[15]. Moreover, gem5's cache models are highly configurable, but they may not replicate all the complexities of real hardware, such as cache bank conflicts or the specific effects of hardware prefetching. As a result, the accuracy of Spectre attack simulations that rely on cache timing might be lower in gem5 than on real hardware, leading to potential discrepancies in attack performance.

Microarchitectural State and Memory System Behavior

Real Hardware: Modern CPUs are equipped with sophisticated memory systems, including multi-level caches, DRAM controllers, and prefetchers, all of which affect how memory is accessed during speculative execution. These interactions play a crucial role in the timing and success of Spectre attacks[16]. Real hardware optimizations, such as store-to-load forwarding and memory disambiguation, can influence the timing of speculative execution, making it difficult to simulate all behaviors with perfect accuracy.
Gem5: While gem5 can model memory systems, its representations are often simplified compared to real hardware[17]. For example, gem5's memory model might

[14]Cache timing attacks, particularly in the context of Spectre, exploit the non-uniform access times of various cache levels in modern CPUs.

[15]Gem5 operates with a configurable clock cycle resolution, but it may not capture the fine-grained, nanosecond-level precision of real hardware's timing mechanisms.

[16]Complex interactions between CPU caches, DRAM, and memory controllers significantly affect the success and timing of Spectre attacks in real hardware, making simulations more challenging.

[17]Gem5 provides configurable memory system models, but it may not fully capture low-level

not fully replicate the impact of DRAM row buffer conflicts or the exact behavior of memory controllers, which could affect the results of Spectre attack simulations. These differences in memory system behavior contribute to the inaccuracies in simulating Spectre attacks in gem5.

Impact of Simplified Hardware Models on Attack Accuracy

Real Hardware: Real hardware has a wealth of fine-grained optimizations and system-level interactions that are often proprietary and not easily replicated in simulation. This makes it difficult for simulators like gem5 to achieve perfect accuracy in predicting Spectre attack outcomes[18]. The real-world performance of Spectre attacks is influenced by many such details, including hardware-specific noise and low-level optimizations that may not be captured in gem5.
Gem5: While gem5 is a powerful research tool, its simplified models of CPU behavior and memory systems do not capture all of the intricate details present in real hardware. This discrepancy means that Spectre attack simulations in gem5 are not always representative of real-world performance. The lack of real-world noise, for example, makes Spectre attacks appear more predictable and successful in gem5 than they would be on actual hardware[19].
In conclusion, while gem5 is an excellent platform for simulating Spectre attacks and providing valuable insights into the vulnerabilities of modern CPUs, there are inherent limitations when comparing its results to real hardware. Differences in branch prediction models, cache timing granularity, and microarchitectural detail contribute to inaccuracies in Spectre attack simulations. Researchers should take these limitations into account when interpreting results from gem5 and consider validating their findings with real hardware experiments whenever possible.

3.7 Future Work

3.7.1 Spectre Vulnerability Research

As hardware security continues to evolve, future research on Spectre and related side channel attacks must expand to encompass diverse CPU architectures and explore innovative mitigation strategies. The study of speculative execution vulnerabilities

optimizations like DRAM row buffer management or intricate memory controller behaviors that are present in real hardware.

[18]Gem5 simulations lack the randomness and real-world interference (e.g., interrupts, multitasking) that are present in real hardware, which can affect the outcome of Spectre attacks.

[19]Gem5 simulations lack the randomness and real-world interference (e.g., interrupts, multitasking) that are present in real hardware, which can affect the outcome of Spectre attacks.

like Spectre has shown major challenges in modern processor security. To move forward, future research should focus and address the following directions:

- Exploration of Diverse Architectures Investigate Spectre-like vulnerabilities in a variety of CPU architectures, including custom processors, low-power designs, and specialized architectures. This will help uncover new vulnerabilities and understand their impact in different contexts.
- Enhanced Speculative Execution Models Create detailed models of speculative execution to capture more attack surfaces. These models should include the complexity of real-world platforms like mobile devices, embedded systems, and edge computing. This will provide deeper insights into potential security gaps.
- Adaptive Mitigation Strategies Design hardware-agnostic, dynamic defenses to make systems more resilient against speculative execution attacks. This could include runtime adaptation techniques like anomaly detection using machine learning or improved speculative execution controls in CPU designs.
- Focus on Emerging Architectures Research emerging architectures like RISC-V to see how they handle Spectre-like attacks compared to traditional platforms. This will help develop best practices for secure architecture design and identify areas needing attention in new instruction sets.

3.7.2 Improvements to gem5 for Broader Adaptation

The gem5 simulation platform has been incredibly useful in hardware security research, especially for studying Spectre-like attacks. To make the most of its potential and tackle future challenges, we should consider the following improvements:

- Support for Advanced CPU Models Expand gem5 to include more advanced CPU models, like heterogeneous processors and systems with complex memory hierarchies. This would allow for more accurate simulations of modern processors and help study speculative execution vulnerabilities.
- Integration with Standard Boards and SoCs Increase compatibility with popular boards and system-on-chip (SoC) platforms, such as Raspberry Pi, NVIDIA Jetson, and IoT-specific hardware. This would broaden gem5's use in embedded systems and IoT research, enabling a deeper analysis of vulnerabilities in real-world scenarios.
- System-Level Simulation Components Add advanced system-level components, like high-fidelity networking models, realistic peripheral devices, and more

comprehensive I/O subsystems. This would extend gem5's utility in complex simulation environments, especially for large-scale distributed systems research.

- Compliance with Open Standards Align gem5 with current open standards, such as the ARM Platform Security Architecture (PSA), Trusted Firmware-A (TF-A), and emerging standards like RISC-V. This would enhance its relevance in the evolving hardware security landscape and attract a broader research community.
- Improved Modularity and Third-Party Integration Focus on making gem5 more modular to allow seamless integration with third-party tools and frameworks. This flexibility would make gem5 a more versatile platform for hardware design, validation, and verification, fostering interdisciplinary collaboration in system security research.

These future improvements will help make gem5 a more powerful and adaptable platform, allowing researchers and engineers to simulate a wider range of systems with greater flexibility. In other words, the proposed advancements in Specter vulnerability research and gem5 simulation capabilities will create a strong foundation to tackle speculative execution vulnerabilities in modern processors. These improvements will help researchers uncover new vulnerabilities across different platforms and architectures, develop and test innovative defense mechanisms, and enhance the accuracy and efficiency of system-level simulations in security studies. In addition, as speculative execution vulnerabilities remain a significant challenge, addressing these areas is crucial to improve the security and reliability of current and future computing systems. The insights gained from this work will not only deepen our understanding of hardware security but also guide the design of more resilient processor architectures and secure software solutions.

Chapter 4

Return-to-Non-Secure (ret2ns) Attack

4.1 Introduction to Return2nonsecure Attack

ARM TrustZone technology, available as an optional Security Extension in ARMv8-M processors, provides a powerful foundation for improving system security in a wide range of embedded applications. This extension builds on the TrustZone technology that has been used in ARM Cortex-A processors for years, adapting its concepts to better suit the specific needs of smaller, energy-efficient devices. At its core, TrustZone technology for ARMv8-M works in a way that's similar to its Cortex-A counterpart. Both systems divide the processor's operations into two states: Secure and Non-secure. Non-secure software is limited to accessing Non-secure resources, while Secure software can access both Secure and Non-secure areas. What sets ARMv8-M apart is its optimization for embedded systems. Instead of relying on a dedicated Secure Monitor handler, as Cortex-A processors do, it uses a memory map-based approach to manage transitions between Secure and Non-secure states, with these transitions happening automatically during exception handling. There are also some key differences in how TrustZone is implemented in ARMv8-M. For example, it supports multiple Secure entry points, offering more flexibility in controlling Secure resources. In contrast, Cortex-A processors rely on a single Secure Monitor handler for all Secure transitions. ARMv8-M also allows Non-secure interrupts to be handled even while Secure functions are running, which is particularly valuable for real-time applications that require responsiveness. Thus, This technology is designed with microcontroller systems in mind, where low power consumption and quick response times are critical. By allowing interrupts to be handled efficiently during Secure operations, TrustZone ensures that real-time tasks aren't delayed. Shared register banks between Secure and Non-secure states

help reduce power usage, keeping energy efficiency on par with previous ARM microcontroller architectures. The ability to switch seamlessly between Secure and Non-secure states with minimal overhead also supports frequent interaction between these two domains, which is common in embedded applications where Secure firmware handles tasks like graphics or communication protocols. TrustZone essentially splits a system into two worlds: Secure and Normal (Non-secure). Secure software has full access to both worlds, but Non-secure software is restricted to its own. These Secure and Non-secure states integrate smoothly with the processor's existing modes, allowing tasks in both states to operate independently. By incorporating TrustZone into ARMv8-M processors, developers can create systems that meet the high demands of modern embedded applications—balancing performance, power efficiency, and security. For a depth understanding, ARM Cortex-M microcontrollers, widely deployed in embedded and Internet of Things (IoT) applications, have become integral to modern computing environments. These microcontrollers are equipped with ARM TrustZone, a security extension that partitions system resources into secure and non-secure states as it has been mentioned bellow. This partitioning allows the execution [1] of sensitive operations within a protected environment, while non-secure tasks are handled separately. However, despite the security enhancements offered by TrustZone-M, the rapid state-switching mechanism between these domains introduces vulnerabilities that adversaries can exploit [ret2ns]. One such vulnerability is the Return-to-Non-Secure (ret2ns) attack, which takes advantage of the seamless and rapid transitions between secure and non-secure states. Unlike its counterpart, ARM Cortex-A, which requires a secure monitor call (SMC) to transition between states, ARM Cortex-M allows transitions through simple function calls and returns. This simplicity increases the risk of unauthorized code execution. In a ret2ns attack, an adversary compromises a secure-state program and redirects execution to non-secure state code, potentially gaining arbitrary control over the system [2]. The ret2ns attack is a notable security flaw in the ARM Cortex-M architecture, particularly in systems utilizing the TrustZone extension. This attack exploits the fast state-switch mechanism in Cortex-M, which permits direct control-flow transfers from secure state programs to non-secure state userspace programs. By redirecting execution from a secure context to a potentially malicious non-secure context, the attack enables arbitrary code execution with elevated privileges in the non-secure state. The attack typically begins with interactions between the attacker's userspace program and the secure state program, leveraging non-secure state system calls executed via the supervisor

[1] Binkert, G. N.and Beckmann B.and Black, Reinhardt, S. K., Saidi, A. A.and Basu, and M. Shoaib. «The gem5 simulator». In: ACM SIGARCH Computer Architecture News 39.2 (2011), pp. 1–7 (cit. on pp. 47, 56, 65).

[2] ARM Ltd. Trusted Firmware-A (TF-A) Documentation. . 2023 (cit. on p. 56).

call (SVC) instruction. Vulnerabilities in either the userspace or kernel space of the secure state can serve as entry points for the attacker. A primary objective is to corrupt code pointers used by specific instructions, such as bxns or blxns, within the secure state program. By manipulating these pointers, the attacker can redirect the control flow to a userspace program under their control. Once the control flow is redirected, the attacker can execute code with privileged access, usually limited to secure state operations. This escalation of privileges poses a severe threat as it enables actions that compromise the system's integrity and confidentiality. Ret2ns attacks can originate in two primary ways: handler-mode-originated attacks, commonly found in real-time operating systems (RTOS), and thread-mode-originated attacks, which are typical in security-enhanced bare-metal systems supporting privilege separation [3]. The broad state transition surface in Cortex M processors makes these attacks easier to execute compared to analogous attacks on architectures like x86 or Cortex-A [4]. The ret2ns attack underscores critical security vulnerabilities in the ARM Cortex-M TrustZone architecture. It emphasizes the need for robust defense mechanisms to mitigate these risks and safeguard against potential exploitation.

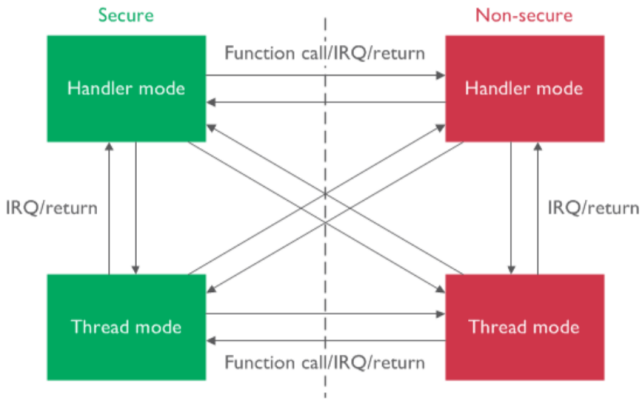

Figure 4.1: High-level view of TrustZone-M profile's fast state-switching mechanism.[5]

The ret2ns attack underscores the vulnerabilities inherent in the design of TrustZone-M, especially in systems that do not incorporate advanced security features like a Memory Management Unit (MMU) or the Privileged Execute-Never (PXN) attribute. These limitations create an environment where such attacks can be executed more easily, posing significant risks to system integrity and security.

[3]National Science Foundation. (n.d.). Overview of ret2ns attack.

[4]Ibid.

Consequently, it is imperative to implement effective defense strategies that can counteract these exploits while maintaining optimal performance and functionality of the system. Robust security measures are essential to safeguard against potential threats without hindering the operational capabilities of the embedded systems utilizing TrustZone-M.

N.B. Understanding State Transitions in TrustZone-M Processors

1. **Secure Thread to Secure Handler or Non-secure Thread to Non-secure Handler**

   In this case, there's no shift in the processor's security state. The process for handling exceptions mirrors the familiar stacking mechanism found in Cortex-M processors. The Interrupt Service Routine (ISR) simply runs in the current security state, whether it's Secure or Non-secure.

2. **Non-secure Thread to Secure Handler or Non-secure Handler to Secure Handler**

   Here, the processor transitions from a Non-secure state to a Secure one. Despite this shift, the exception handling process remains largely similar to the standard Cortex-M stacking mechanism. During this transition, the ISR runs in the Secure state to maintain a secure processing environment.

3. **Secure Thread to Non-secure Handler or Secure Handler to Non-secure Handler**

   When moving from a Secure state to a Non-secure state, the processor takes extra steps to prevent information leakage:

   - It saves all general-purpose registers to the Secure stack and clears their contents before running the Non-secure ISR.
   - Once the ISR is complete, the processor restores the registers from the Secure stack.

   This added layer of security does introduce a slight delay in interrupt latency, but it ensures that no sensitive data from the Secure state is exposed. The ISR, in this case, operates within the Non-secure environment.

4. **Switching Between Secure and Non-secure Threads (Privileged or Unprivileged)**

   Transitions between Secure and Non-secure states are also possible while staying at the same privilege level (Privileged or Unprivileged). These shifts often occur during function calls and returns. Importantly, interrupts can occur in either Secure or Non-secure code without restriction, as long as they follow the defined priority levels.

Key Takeaways

TrustZone-M processors are designed to handle state transitions in a way that keeps Secure and Non-secure domains isolated while maintaining performance. Whether it's managing interrupts or ensuring data integrity during transitions, these mechanisms are critical for protecting embedded systems from vulnerabilities, especially in scenarios where both security domains must interact.

4.1.1 Threat Model and System Assumptions

The threat model for ARM Cortex-M processors with TrustZone technology highlights significant vulnerabilities in the interaction between secure and non-secure states. One critical attack, known as the Ret2User attack, targets the logical separation that TrustZone-M is designed to enforce. This attack assumes the existence of a memory corruption vulnerability in the secure state, which an attacker in the non-secure environment can exploit. Such vulnerabilities may exist in either the kernel or user space of the secure state, providing opportunities for malicious exploitation. In a typical Ret2User attack, the attacker uses a userspace program under their control to make system calls to the secure state. By exploiting the memory corruption vulnerability, the attacker can manipulate critical code pointers used by control flow instructions, such as bxns or blxns. This manipulation redirects execution to a specific location in the non-secure user space, allowing the attacker to execute code with elevated privileges that would normally be restricted. The primary aim of the Ret2User attack is not to execute arbitrary code within the secure state itself but to use the secure state's privileges to gain control over the system. This is achieved by corrupting control flow mechanisms that facilitate transitions between secure and non-secure states. Such redirection of execution undermines the fundamental security goals of TrustZone. The effectiveness of this attack increases in systems lacking advanced security features, such as a Memory Management Unit (MMU) or the Privileged Execute-Never (PXN) attribute. These features are essential for enforcing strict access controls and preventing unauthorized code execution. Without them, the system becomes more vulnerable, making Ret2User attacks easier to execute. To address these vulnerabilities, it is essential to develop robust defense mechanisms. A detailed understanding of the threat model and the architectural assumptions of TrustZone-based systems is necessary for designing effective countermeasures. Such efforts are critical for reducing the risks of these attacks while maintaining the performance and functionality of embedded systems that rely on TrustZone technology.

4.1.2 Attack Overview and Methodology

The Ret2NS attack is a critical vulnerability targeting systems that use the TrustZone-M architecture. Its goal is to compromise the system by exploiting a memory corruption vulnerability in the secure state. By doing so, the attacker can manipulate the system's execution flow and redirect it to a non-secure memory region they control. This relies on the assumption that the secure state contains an exploitable issue, such as a buffer overflow or improper pointer management. These vulnerabilities allow attackers to corrupt key control-flow structures, such as return addresses or function pointers, which are essential for safely transitioning between secure and non-secure states. In the TrustZone-M architecture, operations are distinctly divided into secure and non-secure domains, with the secure domain being inherently more trusted. The Ret2NS attack specifically targets the transition mechanisms, particularly the BXNS and BLXNS instructions, which are designed to facilitate the flow of execution between these domains while maintaining system integrity. When an attacker successfully corrupts the relevant control-flow pointers, they can manipulate these instructions to redirect execution to a non-secure address of their choosing, rather than the intended safe location. This manipulation allows the attacker to execute code within the non-secure domain while still retaining the elevated privileges associated with the secure domain, effectively circumventing the security assurances provided by TrustZone-M [6]. When successful, this attack redirects execution from the secure state to the non-secure state without losing the elevated privileges of the secure domain. This enables the attacker to execute arbitrary code in the non-secure domain while bypassing the security guarantees that TrustZone-M is supposed to enforce.

Key Conditions for the Ret2NS Attack:

The success of the Ret2NS attack depends on several specific conditions:

Memory Corruption Vulnerability: The attack requires a memory corruption flaw in the secure state. This could be caused by improper input validation, poor memory management, or errors in pointer handling. Such a flaw enables the attacker to modify or overwrite critical data, making it possible to redirect control flow to malicious code.

Lack of Control-Flow Protections:To prevent attacks like Ret2NS, systems need strong control-flow integrity (CFI) measures that monitor and restrict how execution flow is redirected. Without these safeguards, attackers can manipulate control flow and redirect execution to unsafe locations without being detected.

Reliance on TrustZone-M Transition Instructions:The Ret2NS attack takes advantage of how TrustZone-M transitions between secure and non-secure states using BXNS and BLXNS instructions. These instructions are designed

[6]ARM Ltd. Trusted Firmware-M (TF-M) Documentation. 2023 (cit. on pp. 11, 58)..

for seamless state transitions but become vulnerable when paired with corrupted control-flow pointers. This makes them a primary target for exploitation in the Ret2NS attack.

Privilege Escalation Opportunities: The attack is particularly effective in systems where transitions from secure to non-secure states retain the elevated privileges of the secure state. If the system does not strictly downgrade privileges during these transitions, attackers can use this oversight to execute privileged operations in the non-secure domain. The Ret2NS attack illustrates a fundamental flaw in TrustZone-M's mechanism for separating secure and non-secure states. By exploiting memory corruption and taking advantage of the absence of robust control-flow protections, attackers can redirect execution and bypass the intended security of the system. This not only compromises sensitive operations in the secure domain but also puts the overall system at risk. To address these challenges, systems must implement stronger defenses, such as enforcing control-flow integrity, improving memory safety, and implementing privilege management policies that de-escalate permissions during transitions. Without these measures, TrustZone-M systems remain exposed to significant threats like the Ret2NS attack, which can undermine the very foundation of their security architecture.

Non-Secure Code:

```
1
2  /*-------------------------------------------------------------
3    Attacker-controlled function: should only execute in
      unprivileged level
4    address: 0x00200620
5   -------------------------------------------------------------*/
6  void attacker_controlled(void);
7
8
9
10 /*-------------------------------------------------------------
11   print_LCD function: this function registers the user message and
       performs an SVC call to enter the secure state and invoke a
      secure function (print_LCD_nsc()).
12  -------------------------------------------------------------*/
13     void print_LCD(char *msg)
14 {
15     register char *r0 __asm("r0") = msg;
16     __asm volatile("svc #1"
17                    :
18                    : "r"(r0));
19 }
```

```
20
21 /*-------------------------------------------------------------
22   SVC handler function: The SVC handler checks the SVC number and
      dispatches the call to print_LCD_nsc() in the secure state with
       the user-supplied message.
23   -------------------------------------------------------------*/
24
25 void SVC_Handler_Main(uint32_t exc_return_code, uint32_t msp_val)
     {
26     .....
27
28         case 1:  // extracted number of the svc will be handled
     uppon
29          print_LCD_nsc((char *)svc_args[0]);
30          break;
31
32       }
```

Secure Code:

```
1
2 /*-------------------------------------------------------------
3   print LCD Non Secure callable  function: This function checks if
      the LCD is ready and then uses sprintf to concatenate the
     message with a timestamp and system status. The buffer buf is
     vulnerable to overflow if msg is too large..
4   -------------------------------------------------------------*/
5     int32_t print_LCD_nsc(char *msg) __attribute__((
     cmse_nonsecure_entry));
6     int32_t print_LCD_nsc(char *msg)
7     {
8          char buf[MAX_LEN] = {0};
9          int32_t val = -1;
10         if (_driver_LCD_ready()){
11         sprintf(buf, "%s %s: %c%c%c%c", _TIME_STAMP,
     _SYSTEM_STATUS, msg[0], msg[1], msg[2], msg[3]); //buffer
     Overflow
12
13         val = _driver_LCD_print(buf); //BXNS return
14          }
15    }
```

In a TrustZone-M architecture, secure and non-secure states interact to manage hardware resources securely. Consider a scenario where a secure function, `print_LCD_nsc`, is responsible for handling display operations on an LCD. This setup ensures that sensitive LCD peripheral registers, accessible only in the secure state, remain protected. To enable non-secure code to use the LCD, the system employs an intermediary library function, `print_LCD()`, which utilizes a Supervisor Call (SVC) to transition into the secure state and invoke the `print_LCD_nsc` function.

The process begins with the `print_LCD()` function in the non-secure state initiating an SVC call. The call includes a specific identifier to signal the transition into the secure state. During this transition, the Secure Vector Handler (SVC Handler), which operates within the secure state, receives the address of a user-provided message via register `R0`. The SVC Handler then calls the secure function `print_LCD_nsc`, passing along the pointer to the message.

The `print_LCD_nsc`function, defined with the `cmse_nonsecure_entry` attribute, processes the message. It first checks the LCD's readiness using a driver function. Once confirmed, it concatenates the user message with additional information, such as a timestamp and system status, using sprintf. This formatted data is stored in a buffer. However, the function lacks safeguards to ensure the message fits within the buffer's allocated size, making it vulnerable to a buffer overflow. If the user-supplied message exceeds the buffer's capacity, the overflow can overwrite critical data on the stack, including the Link Register (LR). The LR holds the return address of the function and determines the control flow after the function completes. An attacker can exploit this vulnerability by crafting a malicious message that corrupts the LR value. When `print_LCD_nsc` returns to the non-secure state via the BXNS instruction, the corrupted LR causes the system to execute arbitrary non-secure code, such as a function chosen by the attacker. This scenario demonstrates how a buffer overflow in a secure function can be weaponized to manipulate the return path, granting control over non-secure code execution. It highlights the critical importance of implementing robust input validation and secure coding practices. Properly sanitizing user inputs, carefully managing buffer sizes, and employing additional runtime protections are essential to mitigating such vulnerabilities in TrustZone-M systems and ensuring the security of the interaction between secure and non-secure states. The scenario provided reveals a significant weakness in TrustZone-M systems, where inadequate input validation within secure functions, such as `print_LCD_nsc`, opens the door to critical exploits like buffer overflows. These vulnerabilities allow attackers to manipulate essential control flow mechanisms, such as the BXNS instruction, enabling arbitrary code

execution in the non-secure domain. Such exploits not only blur the boundary between secure and non-secure states but also fundamentally erode the security that TrustZone-M is designed to provide. Addressing these challenges requires a proactive approach to security, with an emphasis on rigorous input validation, careful memory management, and adherence to secure coding standards. Techniques such as bounds checking, stack canaries, and control-flow integrity enforcement can provide critical layers of defense. Additionally, integrating tools for formal verification and static analysis into the development process can help identify vulnerabilities early, ensuring that TrustZone-M systems remain resilient against sophisticated attacks. These measures are vital to safeguarding embedded systems, particularly as their role in critical applications continues to expand.

4.2 Challenges in Simulating ret2ns Attacks with gem5 and Buildroot

4.2.1 Introduction to Buildroot

Simulating the ret2ns attack requires a carefully constructed and representative embedded environment, and Buildroot serves as a crucial tool in this process. Buildroot is a powerful tool that automates the creation of complete embedded Linux systems, including cross-compiling toolchains, kernel builds, and root filesystems. Its flexibility and configurability make it invaluable for preparing environments tailored to TrustZone-M simulations, particularly when using gem5, a highly flexible architecture simulator. One of the notable features of Buildroot is its ability to cross-compile essential components like TrustedFirmware-M (TF-M). TF-M is a robust, open-source firmware designed to handle secure state transitions in ARM TrustZone-M systems. When it comes to simulating TrustZone-M, Buildroot is set up to create the necessary software stack for both secure and non-secure environments. This setup provides a realistic platform for assessing security vulnerabilities, such as the ret2ns attack. By seamlessly integrating various software elements, Buildroot helps build a highly customizable simulation environment, which is vital for testing how secure and non-secure states interact in TrustZone-M systems. The simulation of attacks targeting the secure/non-secure state transitions in TrustZone-M requires a specialized environment to replicate the behaviour of ARM Cortex-M processors. Buildroot facilitates this by cross-compiling key components such as TrustedFirmware-M (TF-M), which is crucial for managing secure-state operations[7]. Through Buildroot, researchers can generate a minimal root filesystem that supports the coexistence of secure and non-secure components, simulating the

[7]Buildroot. (n.d.).

dual-world architecture of TrustZone-M. This is particularly important for simulating ret2ns attacks, which exploit vulnerabilities in state transition mechanisms between the secure and non-secure regions. Additionally, Buildroot automates the configuration of the necessary bootloaders, such as U-Boot, and the Linux kernel for ARM Cortex-M systems [8]. These components are vital for accurately representing state transitions, peripheral access, and simulating attack scenarios that rely on memory vulnerabilities. Therefore, Buildroot not only simplifies the process of building a secure and non-secure system but also ensures that the environment is configured to reflect real-world embedded systems, which is crucial for the accurate simulation of ret2ns attacks [9].

4.2.2 Setting Up Buildroot for TrustZone-M Simulations

Setting up Buildroot to simulate TrustZone-M environments involves a series of carefully executed steps to configure components essential for accurately demonstrating secure and non-secure state interactions. This setup enables a detailed examination of security vulnerabilities, such as the ret2ns attack, in a controlled environment. The key stages in preparing Buildroot for TrustZone-M simulations include cross-compiling TrustedFirmware-M (TF-M), configuring the root filesystem, and setting up the kernel and bootloader.

- **Cross-Compiling TrustedFirmware-M (TF-M)**:Buildroot simplifies the process of cross-compiling TrustedFirmware-M (TF-M), an open-source framework designed for managing secure-world operations on ARM Cortex-M processors. TF-M plays a crucial role in these simulations as it provides runtime services needed for secure state management, including secure boot processes, secure storage, and cryptographic operations. By integrating TF-M into the Buildroot-generated environment, researchers can create a functional simulation of the secure state that accurately mimics real-world behaviour. This is essential for studying how vulnerabilities like buffer overflows or pointer corruption can affect secure-world processes and enable attacks like ret2ns.

- **Root Filesystem Configuration**: The root filesystem is another critical component generated by Buildroot. It provides the minimal infrastructure needed to simulate interactions between the secure and non-secure states. This filesystem enables the coexistence of secure and non-secure components within the simulation, allowing for realistic attack scenarios. For example, the root

[8]ARM. (2018). ARM TrustZone technology overview.

[9]Secure Hardware Extension (SHA). (2020). Simulating ARM Cortex-M TrustZone: Tools and techniques for real-time embedded systems.

filesystem can include libraries and binaries that facilitate state transitions, making it possible to replicate how a compromised non-secure application could exploit vulnerabilities in the secure state. This realism is vital for understanding how specific vulnerabilities manifest and how they might be exploited in embedded systems.

- **Kernel and Bootloader Setup**:Buildroot supports the generation and configuration of essential low-level software components, such as bootloaders and kernels. Bootloaders, like U-Boot, ensure proper initialization of the system and facilitate the handoff between secure and non-secure states. The kernel, configured for ARM Cortex-M processors, governs how the system manages hardware resources and state transitions. Together, these components enable accurate simulation of TrustZone-M's state transition mechanisms, including instructions like BXNS and BLXNS, which are integral to the ret2ns attack. The kernel and bootloader setup also ensures that peripherals can be accessed as they would in a real-world scenario, further enhancing the fidelity of the simulation environment.

By meticulously configuring these components, Buildroot provides a versatile and robust framework for TrustZone-M simulations. This setup not only facilitates the study of security vulnerabilities but also aids in evaluating potential mitigation strategies, contributing to the development of more secure embedded systems.

4.2.3 Challenges with Buildroot in gem5 Simulations

While Buildroot is an excellent tool for setting up the software stack required for TrustZone-M simulations, its integration with the gem5 simulator poses several practical challenges. These issues mainly stem from gem5's limited support for TrustZone-M features, such as secure/non-secure transitions, memory protection, and interrupt handling. These limitations affect the accuracy of simulations for security vulnerabilities like the ret2ns attack, making it harder to model real-world scenarios effectively.

- **Limited TrustZone-M Support in gem5**: A major challenge lies in gem5's inability to fully support TrustZone-M-specific features, particularly the transition instructions BXNS and BLXNS, which are critical for switching between secure and non-secure states. These instructions are essential for maintaining the separation of trust domains, and they play a pivotal role in attacks like ret2ns, where control flow transitions are exploited. Without native support for these instructions, gem5 struggles to replicate the conditions necessary to observe and analyse the exploitation of these transitions. This limitation makes it difficult to accurately simulate the dynamic interplay

between secure and non-secure states, hindering a comprehensive study of TrustZone-M vulnerabilities.

- **Memory Protection and MPU Limitations**: The Memory Protection Unit (MPU) is central to TrustZone-M's architecture, enforcing strict boundaries between secure and non-secure memory regions. Unfortunately, gem5 lacks full support for MPUs, complicating efforts to simulate the isolation mechanisms that prevent unauthorized access between states. This creates a gap in the ability to model how attacks like ret2ns manipulate memory regions to bypass security restrictions. The absence of accurate MPU functionality in gem5 limits its capability to simulate the memory corruption scenarios that are integral to understanding how such vulnerabilities arise and can be exploited.
- **Interrupt and Context Switching Issues**: Interrupts and exceptions are crucial in maintaining the secure/non-secure separation in TrustZone-M systems. They ensure that critical secure operations are insulated from interference, even during unexpected events. However, gem5 encounters challenges in faithfully simulating the handling of interrupts and context switches between secure and non-secure states. This becomes particularly problematic for attacks like ret2ns, where the precise timing and handling of interrupts can determine the success of the exploit. The inaccuracies in gem5's simulation of these mechanisms may lead to an incomplete or misleading understanding of how such attacks unfold in real-world systems.

In summary, while Buildroot provides a strong foundation for creating the software environment for TrustZone-M simulations, the integration with gem5 reveals significant gaps in accurately emulating key features of the architecture. These challenges, if left unaddressed, limit the ability of researchers to explore and mitigate vulnerabilities like ret2ns effectively. Improving gem5's support for TrustZone-M features such as state transitions, memory protection, and interrupt management is essential for advancing the study of ARM-based security systems.

4.3 Challenges in Simulating Trusted Firmware-M in gem5

4.3.1 Bootloader Customization and Secure Boot

The bootloader is a fundamental element in simulating the ret2ns attack, as it orchestrates the transition of the system from non-secure to secure states during initialization. Using Buildroot, researchers can customize and cross-compile bootloaders such as U-Boot, tailoring them for TrustZone-M systems. However, the integration of these bootloaders into the gem5 simulator presents distinct challenges,

particularly in configuring memory mappings and managing state transitions. Addressing these challenges is critical to ensure the simulation accurately mirrors real-world TrustZone-M operations

- **Memory Mapping**: A core function of the bootloader is to establish clear boundaries between secure and non-secure memory regions, which are essential for enforcing TrustZone-M's separation of trust domains. This process depends on the Memory Protection Unit (MPU) to define and maintain these boundaries. However, gem5 lacks full support for MPUs, making it challenging to simulate the proper isolation of memory regions. To work around this limitation, custom modifications to the bootloader are often required. These modifications involve configuring memory mappings manually or through software-level emulation to mimic the behaviour of the MPU. Such adaptations ensure the bootloader aligns with TrustZone-M's security model, even in the absence of native support within the gem5 simulator.

- **Transition Management**: Another key responsibility of the bootloader is handling secure and non-secure state transitions, ensuring a seamless initialization of the TrustZone-M architecture. These transitions rely on specific instructions, like BXNS and BLXNS, which govern the secure-to-non-secure flow. However, gem5's incomplete support for these mechanisms complicates the accurate simulation of such transitions. Customizing the bootloader often involves rewriting sections of code to approximate state-switching behaviour, ensuring that the simulation environment captures the nuances of secure and non-secure operations. These adjustments are particularly important for representing the conditions under which a ret2ns attack could exploit vulnerabilities in state transitions.

In conclusion, customizing the bootloader for TrustZone-M simulations in gem5 involves navigating significant challenges related to memory mapping and transition management. Although Buildroot simplifies the creation of tailored bootloaders like U-Boot, the limitations of gem5 demand meticulous modifications to ensure accurate representation of TrustZone-M's initialization process. These efforts are crucial for creating a reliable simulation environment capable of studying and mitigating vulnerabilities like ret2ns.

4.3.2 Kernel Challenges in Simulating Transitions

Once the system boots, the kernel assumes responsibility for managing secure/non-secure state transitions in TrustZone-M systems, overseeing essential tasks like peripheral access and interrupt management. However, replicating these operations accurately within the gem5 simulator poses several challenges. These limitations

affect the simulation's reliability and its ability to model real-world scenarios, particularly in the context of security vulnerabilities like ret2ns.

- **Interrupt Handling**: A significant challenge lies in simulating TrustZone-M's interrupt model, which distinctly separates secure and non-secure interrupts to ensure the isolation of trusted operations. In a real TrustZone-M system, this separation is critical for maintaining security boundaries, as secure interrupts must not leak information or allow unauthorized execution in the non-secure domain. Unfortunately, gem5 does not fully support this interrupt separation. This shortcoming introduces inaccuracies in simulations, as the simulator may fail to replicate the precise conditions under which secure and non-secure interrupts interact. Such inaccuracies hinder the ability to study how attacks like ret2ns exploit vulnerabilities arising from interrupt mismanagement.

- **Secure/Non-Secure Context Switches**: Proper simulation of state transitions requires gem5 to handle context switches between secure and non-secure states. However, gem5's architecture lacks robust support for TrustZone-M

- **Secure/Non-Secure Context Switches**: Another critical aspect of TrustZone-M kernel operations is the handling of secure/non-secure context switches, which are pivotal for maintaining domain separation. These switches rely on mechanisms that preserve the integrity of both secure and non-secure execution contexts during transitions. However, gem5 lacks robust support for TrustZone-M's unique context-switching mechanisms, making it challenging to simulate the nuances of these transitions. This limitation is especially problematic for replicating the ret2ns attack, as such exploits directly target vulnerabilities in the secure/non-secure divide during context switches. Without accurate simulation of these mechanisms, researchers may miss crucial insights into how these transitions can be exploited and mitigated.

- **Exception Handling**: Exception handling in TrustZone-M introduces additional complexity, as it requires the kernel to manage secure and non-secure exceptions separately to uphold system integrity. Secure world exceptions demand precise control over the execution state, ensuring that exceptions do not compromise the trusted domain. Unfortunately, gem5 struggles to simulate the TrustZone-M-specific exception models accurately. The lack of adequate support for these models affects the simulator's ability to replicate scenarios where exceptions play a pivotal role in maintaining or breaching security boundaries. This gap in functionality undermines efforts to explore how exceptions can be manipulated in attacks like ret2ns or to test potential mitigations.

While the kernel is central to managing transitions and maintaining security in TrustZone-M systems, gem5's limitations in interrupt handling, context switching, and exception management impede the accurate simulation of these operations. Addressing these challenges is essential for enabling realistic studies of vulnerabilities like ret2ns and for developing robust countermeasures to protect embedded systems.

4.4 Gem5 and TrustedFirmware-M (TF-M) Integration Challenges

4.4.1 Adapting TrustedFirmware-M for Simulation in gem5

TrustedFirmware-M (TF-M) plays a crucial role in implementing ARM's security architecture within TrustZone-M systems. It is responsible for initializing the secure world and managing critical operations, including secure boot, secure interrupts, and memory partitioning. However, adapting TF-M for use in gem5 simulations presents several technical challenges, particularly when attempting to simulate complex attacks like ret2ns.

- **Boot Sequence and Initialization**: One significant challenge lies in replicating TF-M's secure boot process, which establishes the foundation for the secure world. This process involves configuring secure memory regions, initializing secure services, and enforcing strict boundaries between secure and non-secure domains. In gem5, simulating this initialization is problematic due to discrepancies between the simulator's hardware abstraction and the actual hardware features of TrustZone-M microcontrollers. Specifically, gem5's memory management models are not fully compatible with the Memory Protection Unit (MPU) used by TF-M to implement memory segmentation. The MPU's dynamic enforcement of secure and non-secure memory access is central to TrustZone-M security, and gem5's inability to accurately model this feature creates challenges in replicating TF-M's secure world setup.
- **Secure Function Call Mechanisms**: TF-M incorporates mechanisms like the Secure Function (SF) framework, which provides controlled interfaces for non-secure applications to invoke secure functions. This framework ensures that secure services remain isolated and accessible only through predefined entry points. Simulating these secure function calls in gem5 is particularly challenging because the simulator lacks native support for TrustZone-M's privilege-level transitions and access controls. These mechanisms are essential for enforcing the isolation of secure functions and preventing unauthorized access. This limitation directly impacts the simulation of ret2ns attacks, where the exploitation of secure function return paths is central to the attack.

Without accurate simulation of these transitions, the simulation fails to capture the full scope of vulnerabilities that can arise during secure function execution.

- **Interrupt and Fault Handling**: In real TrustZone-M systems, interrupt and fault handling is critical for maintaining separation between the secure and non-secure worlds. TF-M achieves this by configuring interrupt controllers and exception handlers to route events appropriately. For example, secure interrupts must be processed in the secure world, while non-secure interrupts are directed to non-secure handlers. In gem5, however, the simulator's limited support for TrustZone-M's interrupt handling mechanisms introduces inconsistencies in how these events are managed. This gap becomes particularly problematic in scenarios where precise timing and control of interrupts are essential, such as in the study of vulnerabilities like ret2ns. Interrupt timing can significantly affect the feasibility and exploitation of such attacks, and gem5's shortcomings in this area limit its utility as a simulation platform.

Mainly, it's important here to understand that while TrustedFirmware-M provides the foundation for secure operations in TrustZone-M systems, adapting it for simulation in gem5 is fraught with challenges. The limitations in gem5's memory management models, secure function execution, and interrupt handling undermine the accuracy of these simulations, particularly for complex attack scenarios. Addressing these gaps is essential for enabling researchers to fully explore and mitigate vulnerabilities like ret2ns in TrustZone-M environments.

4.4.2 Memory Partitioning and Peripheral Access

TrustedFirmware-M (TF-M) relies heavily on the Memory Protection Unit (MPU) to create a secure boundary between memory regions, ensuring that non-secure code cannot access sensitive secure memory. This strict control of memory and state transitions is a cornerstone of TrustZone-M security. However, gem5's lack of detailed MPU support introduces significant challenges in simulating the interactions between secure and non-secure memory, particularly for studying vulnerabilities such as ret2ns.

- **Memory Segmentation**: In TF-M, the MPU is configured to divide memory into secure and non-secure regions, with access privileges tightly enforced to prevent unauthorized interactions. This segmentation is critical to maintaining system integrity, allowing secure domains to operate without interference from the non-secure side. When using gem5 for simulations, however, the absence of a realistic MPU model means that these memory protections cannot be fully enforced or accurately represented. This limitation is particularly problematic when simulating attacks like ret2ns, where bypassing memory protections is a

fundamental part of the exploit. Without proper memory segmentation, the simulation fails to reflect real-world scenarios, leading to gaps in understanding the attack's effectiveness and its implications for system security.

- **Peripheral Access**: Beyond memory segmentation, many TrustZone-M systems rely on secure peripherals, such as cryptographic modules and secure storage, which are only accessible to secure code. These peripherals are vital for performing sensitive operations like encryption or securely storing critical data. TF-M enforces strict access controls to ensure that these resources remain protected. However, gem5's limited support for simulating peripheral access control poses another hurdle. Without accurate simulation, the simulator cannot faithfully replicate how secure peripherals are isolated from non-secure domains. This limitation affects the ability to explore how vulnerabilities in peripheral access might contribute to attacks like ret2ns, potentially underestimating the impact of exploiting these security gaps.

Thus, memory partitioning and peripheral access controls are essential features of TF-M that uphold the security of TrustZone-M systems. However, the lack of robust support for these features in gem5 hampers its ability to simulate real-world security behaviours effectively. Addressing these limitations is critical for gaining a comprehensive understanding of attacks like ret2ns and developing the defenses needed to mitigate such vulnerabilities.

4.5 Future Directions for Improved Simulation of TrustZone-M in gem5

Given the limitations encountered in simulating TrustZone-M and the ret2ns attack using gem5 and Buildroot, there are several improvements that can be made to enhance the fidelity of such simulations. First, expanding gem5 to better support state transitions, like BXNS and BLXNS, would significantly improve the simulation of ret2ns attacks. This would allow for more accurate simulation of secure/non-secure context-switching. Additionally, implementing a more detailed MPU model in gem5 would help simulate memory boundaries more effectively, which is crucial for understanding how attacks exploit these vulnerabilities. Improving interrupt handling and exception management in gem5 would also be essential for more accurately simulating the separation of secure and non-secure interrupts in TrustZone-M. Lastly, refining Buildroot configurations, especially with TF-M and U-Boot, would ensure a more reliable simulation environment. These improvements would not only help in analysing more accurate attack scenarios but also provide a more robust framework for security research on ARM-based embedded systems and better simulation of ret2ns attacks.

4.5.1 Extending gem5's TrustZone-M Support

To address the current gaps in simulating TrustZone-M features, one promising research direction involves extending gem5's native support for TrustZone-M-specific functionality. The TrustZone-M architecture introduces a range of features aimed at enabling secure and efficient execution in embedded systems. However, its full potential in terms of simulation has yet to be realized in gem5. By focusing on specific extensions, researchers can enhance the accuracy and utility of gem5 as a simulation platform for security studies. Future directions for improving TrustZone-M simulation in gem5 could focus on three key areas:

- **MPU Support**:Implementing detailed Memory Protection Unit (MPU) support in gem5 would significantly improve the simulator's capability to model memory protection mechanisms as employed in TrustZone-M systems. The MPU serves as a cornerstone of TrustZone-M's security, enabling the configuration of memory regions with attributes such as read, write, or execute permissions and secure or non-secure access control. By extending gem5 to include this functionality, researchers could gain insights into the enforcement of these memory protections under various scenarios. Moreover, this enhancement would allow the simulation of sophisticated attacks like return-to-non-secure (ret2ns), which exploit the transition between secure and non-secure states to bypass memory protections. Accurate simulation of such attacks would facilitate the development of countermeasures and help evaluate their effectiveness.

- **BXNS and BLXNS Instruction Simulation**: Adding native support for TrustZone-M-specific instructions, such as BXNS (Branch with Exchange to Non-Secure) and BLXNS (Branch with Link and Exchange to Non-Secure), would improve the simulator's ability to model state transitions between the secure and non-secure worlds. These instructions are critical for facilitating transitions between execution contexts and maintaining the isolation guarantees provided by TrustZone-M. Simulating them accurately is particularly important for studying control-flow exploits. For instance, attackers often manipulate these transition instructions to alter the intended execution path, gaining unauthorized access to secure resources. Enhanced support for these instructions in gem5 would provide a platform to analyse such vulnerabilities in detail and test mitigations in a controlled environment.

- **Interrupt and Exception Handling**: Enhancing gem5's interrupt and exception handling models to account for the distinction between secure and non-secure interrupts is another vital extension. TrustZone-M introduces a dual-world model where interrupts can originate from either secure or non-secure sources, and the way these are handled is critical for maintaining system

security. Current implementations in gem5 may not fully capture the nuanced behaviour of TrustZone-M in this regard. By refining interrupt and exception handling mechanisms, researchers could simulate scenarios where secure interrupts are improperly managed, potentially leading to privilege escalation or information leakage. Such improvements would enable detailed studies of vulnerabilities like those exploited by ret2ns attacks, where mishandled interrupts or exceptions facilitate transitions into unintended states.

To conclude, extending gem5's TrustZone-M support to include MPU features, instruction simulation for state transitions, and enhanced interrupt and exception handling would address critical gaps in its current capabilities. These improvements would make gem5 a more powerful tool for researchers investigating the security implications of TrustZone-M, providing a robust platform for exploring and mitigating vulnerabilities in embedded systems.

4.5.2 Integrating Hardware-in-the-Loop (HIL) Systems

A practical way to enhance TrustZone-M simulation in gem5 is by integrating it with Hardware-in-the-Loop (HIL) systems, as described in [**Dempsey2012:HIL_simulation**]. HIL systems combine the use of real TrustZone-M hardware components with gem5's flexible simulation framework, creating a powerful hybrid model. This approach offers unique advantages by bridging the gap between simulation and physical hardware, making it possible to achieve greater accuracy and realism in security research.

- **Realistic Timing and Interrupt Behavior**: One of the biggest challenges in simulating TrustZone-M systems with gem5 is accurately capturing timing and interrupt behaviour, especially in real-time applications where even small discrepancies can lead to significant security vulnerabilities. HIL simulations address this by incorporating real hardware, which naturally exhibits precise timing and interrupt dynamics. This would make it possible to study how timing vulnerabilities and interrupt handling play a role in attacks, particularly in systems where real-time performance is critical. By offering this level of accuracy, HIL systems could greatly enhance the reliability of security simulations.
- **Validation of Software Attacks**: Combining real hardware with gem5's software simulation capabilities allow researchers to validate software-based attacks like return-to-non-secure (ret2ns) under realistic conditions. This setup ensures that simulated vulnerabilities mirror actual system behaviour, making it easier to assess how attacks would function in the real world. Additionally, researchers can use HIL systems to test proposed security mitigations, ensuring

they are effective not just in theory but in practical scenarios as well. This ability to validate both vulnerabilities and defenses in a controlled yet realistic environment is a significant benefit of HIL integration.

- **Cross-Validation of Firmware and Software**: TrustZone-M systems rely on close coordination between firmware running in the secure world and software in the non-secure world. HIL systems offer a way to study these interactions in a more detailed and practical manner. By enabling the use of real hardware alongside simulated components, researchers can better understand how complex attacks like ret2ns exploit flaws in the interaction between firmware and software. This cross-validation capability is crucial for identifying subtle weaknesses and improving both firmware and software security.

4.5.3 Accuracy of simulating the ret2ns Attack in gem5

Simulating complex security vulnerabilities, such as the Return-to-Non-Secure (ret2ns) attack, in an environment like gem5 offers significant insights into potential threats in ARM-based systems, particularly those employing TrustZone. However, to gauge the effectiveness of such simulations, it is essential to understand the extent to which gem5 can accurately represent real hardware behavior. This subsection delves into how the ret2ns attack is simulated on gem5 and evaluates the fidelity of these simulations with respect to the underlying hardware.

The Nature of the ret2ns Attack

The ret2ns attack exploits the ARM TrustZone architecture, targeting the seamless transitions between Secure and Non-secure states that occur in ARMv8-M processors. Under normal operation, these transitions are intended to enforce strict separation between secure and non-secure code, which is critical for maintaining system integrity in embedded applications. However, the ret2ns attack takes advantage of the rapid, low-overhead state transitions, which are achieved via simple function calls and returns rather than complex secure monitor calls (SMCs), as seen in the ARM Cortex-A series.
In essence, the attacker manipulates control flow by exploiting vulnerabilities in the Secure state, redirecting execution to malicious code in the Non-secure state. This seamless transition allows the attacker to gain unauthorized access to sensitive resources, potentially leading to arbitrary code execution in the Non-secure state. The ability of gem5 to simulate such intricate security vulnerabilities depends on how accurately it models ARMv8-M TrustZone's state-switching mechanism and the associated control flow between Secure and Non-secure states.

Simulating State Transitions in gem5

One of the core challenges in simulating the ret2ns attack is accurately modeling the transition between the Secure and Non-secure states. In real hardware, these transitions are controlled by mechanisms such as exception handling, interrupt handling, and Supervisor Calls (SVCs), which are designed to occur with minimal overhead. In gem5, the ARMv8-M architecture, including TrustZone features, is modeled with an abstraction that simulates these transitions through a memory-mapped approach.
However, while gem5 can replicate the high-level behavior of state transitions, the simulator may not perfectly capture the underlying timing and interactions between various hardware components involved in these transitions. The ARMv8-M architecture relies on an optimized memory management unit (MMU) and interrupt controller to facilitate state switching, but these components are abstracted in gem5 to some extent. This abstraction results in a loss of fine-grained details, such as the exact timing of interrupt handling, memory access latency, and the precise control flow involved in transitioning between the Secure and Non-secure states.
In the case of the ret2ns attack, the timing of the state transition is crucial. An attacker may attempt to exploit vulnerabilities in the Secure state by corrupting specific return addresses or function pointers, triggering a transition to the Non-secure state at a precise moment. While gem5 accurately simulates the occurrence of these transitions, it does not replicate the exact timing behavior that might occur on real hardware, where interrupt delays, pipeline flushes, and memory access times can influence the success of the attack.

Control Flow Redirection and Attack Execution

The ret2ns attack typically begins with the attacker gaining control over the Secure state program, often by manipulating a vulnerable function pointer or return address. The attacker's goal is to redirect the control flow to a Non-secure state program, thereby gaining unauthorized access to Non-secure resources. In real hardware, the success of this attack depends not only on the ability to manipulate the return address but also on the precise timing of the transition and the ability to avoid detection by security mechanisms such as memory protection units (MPUs) and execution permissions.
In gem5, the simulated execution of the attack follows a similar flow, where the Secure state program's control flow is hijacked, and execution is redirected to the Non-secure state. While gem5 correctly models the logic of this attack, the simulator cannot perfectly emulate certain hardware-level details, such as the handling of misaligned memory accesses or the behavior of hardware-based memory protection mechanisms, which could impact the attack's success in real hardware. Furthermore, gem5's abstraction of the processor's pipeline and the lack of detailed

memory access timings may result in minor discrepancies in how the attack is executed in the simulation compared to actual hardware.
The lack of precise timing in gem5 may lead to differences in the attack's outcome. For instance, real hardware may exhibit race conditions or subtle timing discrepancies that allow the attack to succeed, while these nuances are not fully replicated in the simulation. Additionally, gem5's memory model may not account for low-level optimizations, such as speculative execution, which could affect how data is loaded into the processor's cache and how quickly it becomes available for execution in the Non-secure state.

Comparison with Hardware Behavior

In terms of replicating the ret2ns attack on real hardware, gem5 offers a useful approximation but falls short in some areas due to its abstraction of low-level hardware components. The simulation is accurate in terms of the logical flow of the attack, capturing the sequence of operations such as state transitions, function call redirection, and control flow manipulation. However, the timing and hardware-specific features that could influence the success of the attack, such as interrupt handling, memory access delays, and the impact of various hardware defenses (e.g., data execution prevention, memory isolation), are not fully represented.
Real hardware provides a much richer set of details, including precise control over interrupt latency, memory access speed, and the operation of security features, which can all impact the ret2ns attack's success. For example, on real hardware, an attacker's ability to trigger a state transition at the precise moment needed to hijack control flow may be influenced by factors such as cache behavior and instruction-level parallelism. These factors are typically abstracted in gem5, which may lead to small discrepancies in the simulation's results when compared to actual hardware behavior.

Limitations of gem5 in Accurate Attack Simulation

While gem5 is a powerful tool for simulating processor architectures and security vulnerabilities, it is not without limitations. One of the primary constraints of using gem5 for the ret2ns attack simulation is its reliance on simplified hardware models. The absence of detailed timing information, especially with regard to instruction execution and memory access latency, reduces the accuracy of the attack simulation. Furthermore, gem5's memory management model, which abstracts the complexity of modern memory hierarchies, may not account for low-level hardware optimizations such as speculative execution or memory caching, which can have a significant impact on the outcome of the attack.
Additionally, the security mechanisms in real ARMv8-M hardware, such as TrustZone†-specific features for memory protection and control flow integrity, may not be fully

represented in gem5. As a result, certain attack vectors may not be as easily exploitable in the simulation as they would be on actual hardware.

4.5.4 Future Directions for HIL Integration

While integrating HIL systems with gem5 shows great promise, there are opportunities to expand this approach further. Developing standardized interfaces for connecting gem5 with various TrustZone-M hardware components could make HIL systems more accessible to researchers. Additionally, extending HIL capabilities to support other critical TrustZone-M features, such as secure boot and secure storage, could make these systems even more versatile. Automating the testing of attack scenarios within HIL setups could also improve efficiency and enable more comprehensive evaluations of proposed security strategies. In conclusion, combining gem5 with Hardware-in-the-Loop systems provides a practical and effective way to improve TrustZone-M simulations. By leveraging the strengths of both real hardware and flexible simulation, HIL systems can deliver the precision and realism needed to explore vulnerabilities, validate software attacks, and develop robust security measures. This hybrid approach has the potential to significantly advance the state of TrustZone-M research, making it an invaluable tool for the security community.

Bibliography

[1] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. Wiley, 2001 (cit. on p. 1).

[2] IEEE. «Advancements in Consumer Electronics with Embedded Systems». In: *IEEE Consumer Electronics Magazine*. 2020, pp. 1–10 (cit. on p. 1).

[3] Nicolas Navet and Francois Simonot-Lion. *Automotive Embedded Systems Handbook*. CRC Press, 2009 (cit. on p. 2).

[4] Abraham Silberschatz and Peter Galvin. *Embedded Systems in Healthcare*. Healthcare Publications, 2015 (cit. on p. 2).

[5] R.P.G. Collinson. *Avionics: Development and Implementation*. Wiley, 2003 (cit. on p. 2).

[6] Dieter Uckelmann. *Architecting the Internet of Things*. Springer, 2011 (cit. on p. 3).

[7] P. Kocher et al. «Spectre Attacks: Exploiting Speculative Execution». In: *Communications of the ACM* 62.7 (2019), pp. 93–101 (cit. on p. 3).

[8] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*. Newnes, 2014 (cit. on p. 9).

[9] ARM Ltd. *Trusted Firmware-M (TF-M) Documentation*. `https://trustedfirmware.org/projects/tf-m/documentation/`. 2023 (cit. on p. 11).

[10] Binkert, G. N.and Beckmann B.and Black, Reinhardt, S. K., Saidi, A. A.and Basu, and M. Shoaib. «The gem5 simulator». In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[11] Pier Ayoub and Clémentine Maurice. «Reproducing Spectre Attack with gem5: How To Do It Right?» In: *14th European Workshop on Systems Security (EuroSec'21)*. 2021, pp. 1–16. DOI: `10.1145/3447852.3458715`. URL: `https://inria.hal.science/hal-03215326`.

[12] Erik August. *Spectre PoC Exploit Code*. `https://github.com/ErikAugust/spectre-exploit`. 2018.

[13] G. Karsai, D. Balasubramanian, A. Ledeczi, and S. Neema. «Towards a self-adaptive and fault-tolerant cyber-physical system». In: *Proceedings of the Workshop on Cyber-Physical Systems Security and Resilience*. 2013, pp. 1–5.

[14] ARM Ltd. *Trusted Firmware-A (TF-A) Documentation*. `https://trustedfirmware.org/projects/tf-a/documentation/`. 2023.

[15] ARM Ltd. *ARM Architecture Reference Manual*. `https://developer.arm.com/documentation`. 2020.

[16] Nathan Binkert et al. «The gem5 Simulator». In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[17] ARM Ltd. *ARM TrustZone Technology Overview*. 2018.

[18] Secure Hardware Extension (SHA). *Simulating ARM Cortex-M TrustZone: Tools and Techniques for Real-Time Embedded Systems*. 2020.

[19] Buildroot. *Buildroot Documentation*. Accessed 2023.

[20] National Science Foundation. *Overview of ret2ns Attack*. Accessed 2023.

[21] Stephen W. Keckler. «A New Golden Age for Computer Architecture». In: *Communications of the ACM* (Jan. 2022). Retrieved from urlhttps://cacm.acm.org/research/a-new-golden-age-for-computer-architecture/|

[22] John L. Hennessy and David A. Patterson. «A New Golden Age for Computer Architecture». In: *Communications of the ACM* 62.2 (2019), pp. 48–60.

[23] Tal Ben-Nun and Torsten Hoefler. «Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis». In: *ACM Computing Surveys* 52.4 (2020), pp. 1–43.

[24] Inspired eLearning. *Cybersecurity Awareness Training*. `https://www.inspiredelearning.com/`. Accessed 2025.

[25] The Linux Kernel Organization. *The Linux Kernel Documentation*. `https://www.kernel.org/doc/html/latest/`. Accessed 2025. 2024.

[26] Yuval Yarom and Katrina Falkner. «Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack». In: *23rd USENIX Security Symposium*. USENIX, 2014, pp. 719–732.