Special Session: Detecting and Defending Vulnerabilities in Heterogeneous and Monolithic Systems: Current Strategies and Future Directions

Venkat Nitin Patnala[1], Sai Manoj Pudukotai Dinakarrao[1], Guru Venkataramani[2],
Jie Chen[2], Preet Derasari[2], Milos Doroslovacki[2], Fan Yao[2], Hongyu Fang[2], Meron Demissie[3],
Todd Austin[3], Lauren Biernacki[4], Saket Upadhyay[5], Arnabjyoti Kalita[5], Ashish Venkat[5]
[1]George Mason University, Fairfax, VA, USA
[2]The George Washington University, Washington, DC, USA
[3]University of Michigan, Ann Arbor, MI, USA
[4]Lafayette College, Easton, PA, USA
[5]University of Virginia, Charlottesville, VA, USA
{vpatnala,spudukot}@gmu.edu, guruv@email.gwu.edu, {mdemissi,austin}@umich.edu,
biernacl@lafayette.edu, {saket,akalita,venkat}@virginia.edu

I. Abstract

Embedded systems are evolving in complexity, leading to the emergence of multiple threats. The co-design and execution of software on the embedded systems further exacerbate the attack surface, making them more vulnerable to sophisticated attacks. As embedded systems are used in critical areas, ensuring their security is crucial. In this special session paper, primarily four major topics regarding embedded systems' security are discussed. Firstly, this paper initially explores timing channel analysis at a microarchitectural level in heterogeneous hardware to address the security challenges. It then delves into exploring software-based fuzzing techniques to detect vulnerabilities and enhance embedded system security. Additionally, the paper discusses strategies for improving security in IoT devices with a layered defense strategy known as Snowflake IoT. Finally, it examines approaches to securing large and complex monolithic systems. The challenges and opportunities for securing the embedded systems according to the scale and type of attacks.

II. Introduction

Embedded Systems are integral to modern technological ecosystems and significantly prevail in various domains such as automotive, medical devices, and the Internet of Things (IoT). Advancements in interconnection technology, scaling down of transistor sizes, fabrication methodologies, and computing architectural innovations have led to systems of scale in recent times. For instance, multi-core processors such as AMD EPYC Zen 4 [1] are embedded with up to 128 cores, providing unprecedented computational performance and throughput.

Further advancements in the heterogeneous integration and seamless working with a plethora of interfaces and resource-efficient computing paradigms have expanded the potential of embedded systems in the Internet of Things (IoT) era. IoT has exponential growth, with the connected devices reaching up to 75 billion devices [2]. Integrating low-power wide-area networks enhances connectivity for embedded devices to communicate over long distances with low power consumption.

Despite the advancements on multiple fronts in embedded systems, complex system design, integration capabilities, and large-scale have recently introduced a wide range of security concerns. In addition, as the embedded systems run software on top of hardware, the combined growth in complexity introduces a wide range of security vulnerabilities. These attacks vary both in complexity and adversarial impacts and have been encountered across the globe [3]. The increasing complexity of these attacks has made the integration of security at different levels of the embedded systems, including the hardware layer, firmware, application layer, and operating systems.

At the foundation level, the hardware layer plays a crucial role in the security of embedded systems. Detecting and defending against threats at the hardware and micro-architectural level is significant. Hardware vulnerabilities, such as side channels [4] and covert channels, can lurk behind hardware intellectual property (IP) blocks that are built by multiple hardware vendors. These vulnerabilities can be exploited to leak sensitive information through subtle variations in system behavior, making them particularly dangerous. As the cache-based timing channels [5] are one of the largest attack surfaces, we discuss some of the detection mechanisms in this work.

In order to effectively protect systems against information leakage, we need objective metrics to quantify the attacks and then tackle the damages caused by them. Covert timing channels are information leakage channels where a Trojan process intentionally modulates the timing of events on a shared system resource to illegitimately reveal data secrets to a spy process. Note that the Trojan and the spy do not communicate explicitly through send/receive or shared memory but covertly via modulating certain events. In contrast to side channels, where a process unintentionally leaks information to a spy

process, covert timing channels have an insider Trojan process (with higher privileges) that intentionally colludes with a spy process (with lower privileges) to exfiltrate the system secrets.

A fundamental strategy used by the Trojan process to achieve covert timing-based communication on shared processor hardware is modulating the timing of events by intentionally creating conflicts. We use conflict to collectively denote methods that alter either the latency of a single event or the inter-event intervals. The spy process deciphers the secrets by observing the differences in resource access times. On compute logic and buses/interconnects, the Trojan creates conflicts by introducing distinguishable contention patterns. On memory structures, the Trojan creates conflicts through repetitive patterns of intentional memory block replacements such that the spy can decipher the message bits based on the memory hit/miss latencies. This basic strategy of creating conflicts for timing modulation has been observed in numerous covert timing channel implementations [6]–[10].

Moving up the stack, firmware vulnerabilities remain a significant concern, as illustrated by incidents like the Intel x86 Processor Meltdown and Spectre [11] vulnerabilities, where attackers could exploit hardware flaws remotely. As firmware is a critical component of embedded devices, stored in ROM or non-volatile memory, its vulnerabilities can have severe consequences. There are several existing solutions, such as static analysis [12], [13], dynamic analysis [14], [15], formal verification [16], [17], and Reverse engineering [18], [19] having many limitations such as detecting runtime issues like memory leaks or race conditions, scalability, and increasing the complexity of testing. Automated fuzzing testing, which comes under dynamic analysis, involves providing randomized inputs for each iteration to detect threats or logical errors that can be exploited by a malicious attacker [20].

Automated fuzzing benefits due to its automated process for generating and inserting large sets of inputs in the firmware without manual intervention, which helps with scalability. By generating a large number of test cases, fuzzing can cover a wide input space, increasing the likelihood of uncovering edge cases that might lead to vulnerabilities. Fuzzing has emerged as a crucial tool to detect and address vulnerabilities in embedded systems to expose threats such as overflows, memory leaks, and other security flaws [21].

As aforementioned, the embedded systems are deployed in a wide scale of applications and systems, which introduces new challenges, especially in resource-constrained environments such as IoT [22]. The predominant approach to addressing security concerns revolves around patch-based security. Vulnerabilities are rectified only after their discovery, whether through proactive research efforts or reactive responses to active attacks. This leaves devices to be exposed to emerging threats for extended periods.

Despite its affordability, patch-based security represents arguably the least effective approach for ensuring the security of IoT-class devices. Vulnerabilities in IoT devices remain unaddressed until they are identified, potentially compromising the safety and security of users who rely on these devices. Moreover, the existence of an active exploit leaves every device vulnerable, as they all share the same vulnerability.

From the application perspective, techniques such as code inspection, red-team validation, and formal analysis may help identify vulnerabilities before they pose a threat to IoT users; however, none of these methods offer durable levels of protection against emergent security threats. As a result, we advocate for a departure from patch-based security toward a more durable yet affordable form of security: diversity defenses.

Building on the limitations of patch-based security for IoT devices, similar challenges are magnified in large-scale systems. This paper discusses the current strategies for detecting and defending security vulnerabilities in heterogeneous and monolithic systems, providing critical challenges and opportunities for future direction. This paper discusses three major approaches to securing these complex systems such as secure, transparent, and automated compartmentalization; field-upgradeable defenses to reduce deployment time and cost; and scalable and secure virtualization of fine-grained system resources while preventing unauthorized interference and crosstalk. These strategies are crucial for building resilient embedded systems that can withstand the evolving landscape of cyber threats, ensuring the security and reliability of heterogeneous and monolithic systems.

The rest of the paper is organized as follows. Section III discusses covert timing channels and their detection methods. Then different fuzzing techniques for embedded systems are presented in Section IV. Followed by multi layered defense strategies for IoT devices are described in Section V. Subsequently three approaches in securing large and complex monolithic systems in Section VI.

III. Timing Channels in Heterogeneous Hardware

As system-on-chips (SoCs) become dense and heterogeneous in their designs, the challenges to maintaining robust security are increasing. For example, cars are often referred to as supercomputers in motion. In such systems, the attackers may leverage their insider knowledge to compromise their security. We note that information leakage-based attacks on microcontroller units (MCUs) continue to be exposed. These attacks could manifest as side or covert channel attacks [23]. In this study, we will primarily target cache-based timing channels such as detection of covert timing channels [24].

A. Defining Covert Timing Channels

Trusted Computer System Evaluation Criteria (or TCSEC, The Orange Book) [25] defines a covert timing channel as those that would allow one process to signal information to another process by modulating its own use of system resources in such a way that the change in response time observed by the second process would provide information. Note that, between the trojan and the spy, the task of constructing a reliable covert timing channel is not very simple. Covert timing channels implemented on real systems take significant amounts of synchronization, confirmation, and transmission time, even for relatively short-length messages. As examples, (1) Okamura

et al. [7] construct a memory load-based covert channel on a real system and show that it takes 131.5 seconds just to covertly communicate 64 bits in a reliable manner achieving a bandwidth rate of 0.49 bits per second; (2) Ristenpart et al. [8] demonstrate a memory- based covert channel that achieves a bandwidth of 0.2 bits per second. This shows that the covert channels create non-negligible amounts of traffic on shared resources to accomplish their intended tasks.

TCSEC points out that a covert channel bandwidth exceeding a rate of one hundred (100) bits per second is classified as a high bandwidth channel based on the observed data transfer rates between several kinds of computer systems. In any computer system, there are a number of relatively low-bandwidth covert channels whose existence is deeply ingrained in the system design. If a bandwidth-reduction strategy to prevent covert timing channels were to be applied to all of them, it would become an impractical task. Therefore, TCSEC points out that channels with maximum bandwidths of less than 0.1 bit per second are generally not considered to be very feasible covert timing channels.

B. Detecting Covert Timing Channels

A viable strategy to detect hardware covert timing channels is by dynamically tracking conflict patterns on shared processor hardware. We design low-cost hardware support that dynamically gathers data on certain key indicator events, and software support to compute the likelihood of covert timing channels on a specific shared hardware. The first step in detecting covert timing channels is to identify the event that is behind the hardware resource contention. In our example, the event to be monitored is the memory bus lock operation. The second step is to create an Event Train, i.e., a uni-dimensional time series showing the occurrence of events. As the third step, we analyze the event train using our recurrent burst pattern detection algorithm. Our algorithm is as follows:

1. Determine the interval (Δt) for a given event train to calculate event density. Δt is the product of the inverse of the average event rate and α, an empirical constant determined using the maximum and minimum achievable covert timing channel bandwidth rates on a given shared hardware.

2. Construct the event density histogram using Δt. For each interval of Δt, the number of events is computed, and an event density histogram is constructed to subsequently estimate the probability distribution of event density. Low-density bins are to the left, and as we move right, we see the bins with higher numbers of events.

3. Detect burst patterns. From left to right in the histogram, the threshold density is the first bin, which is smaller than the preceding bin and equal to or smaller than the next bin. If there is no such bin, then the bin at which the slope of the fitted curve becomes gentle is considered as the threshold density. If the event train has burst patterns, there will be two distinct distributions (seen in Figure 1).

Unlike combinational structures, where timing modulation is performed by varying the inter-event intervals (observed as bursts and non-bursts), cache-based covert timing channels rely on the latency of events to perform timing modulation.

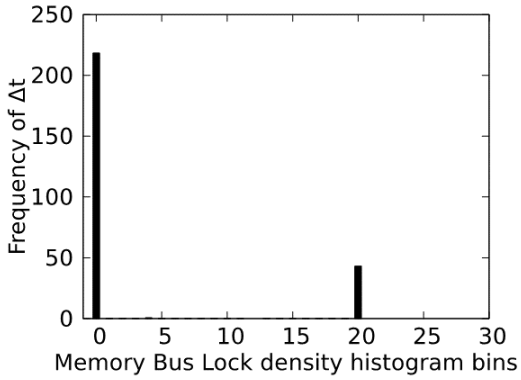


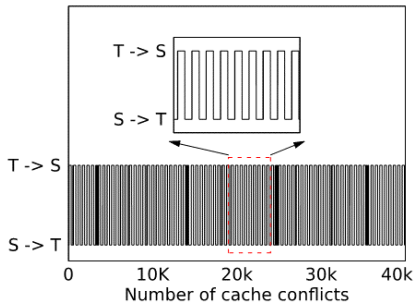
Fig. 1. Event density histogram for Memory Bus Covert Timing Channel

The trojan and the spy create a sufficient number of conflict events (cache misses) alternatively among each other lets coefficient values for a sequence of lag values. An oscillation pattern is inferred when the autocorrelation coefficient shows significant periodicity with peaks sufficiently high for certain lag values.

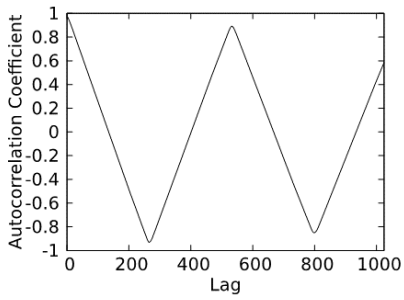
Figure 2 shows our conflict miss event train analysis. In particular, Figure 2(a) shows the event train (cache conflict misses) annotated by whether the conflicts happen due to the trojan replacing the spy's cache sets or vice versa (a legible version of the cluttered event train pattern is shown as inset figure). "T→S" denotes the Trojan (T) replacing the Spy's(S) blocks because the spy had previously displaced those same blocks owned by the trojan at that time.

Note that every ordered pair of trojan/spy contexts has unique identifiers. For example, "S→T" is assigned '0' and "T→S" is assigned "1". The autocorrelation function is computed on this conflict miss event train. Figure 2(b) shows the autocorrelogram of the event train. A total of 512 cache sets were used in G1 and G0 for transmission of "1" or "0" bit values. We observe that at a lag value of 533 (which is very close to the actual number of conflicting sets in the shared cache, 512), the autocorrelation value is highest at about 0.893. The slight offset from the actual number of conflicting sets was observed due to random conflict misses in the surrounding code, and the interference from conflict misses due to other active contexts sharing the cache.

Our framework can be extremely beneficial to the users as we transition to an era of running our applications on remote servers that host programs from many different users. Prior studies [8], [9] show how popular computing environments like cloud computing are vulnerable to covert timing channels. Static techniques to eliminate timing channels like code analyses are virtually impractical to enforce on every third-party software and binaries. Also, adopting strict system usage policies could adversely affect the overall system performance. To overcome these issues, our dynamic detection is a desirable first step before adopting damage control strategies.



(a) Event Train of conflict misses between Trojan and spy

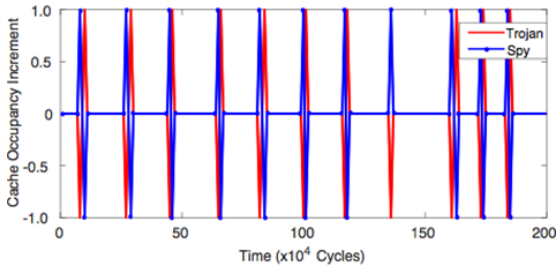


(b) autocorrelogram

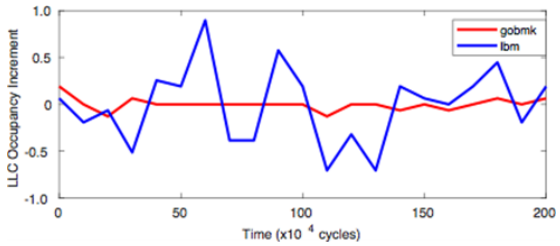
Fig. 2. L2 cache covert channel detection

C. Robust Indicators of Timing Channels

Most prior techniques use cache misses as a source of information for the detection of cache-based timing channels. However, cache misses may be inflated to mislead the detectors. Therefore, better timing channel indicators are necessary.



(a) Trojan and spy



(b) Benign apps (gobnk and lbm)

Fig. 3. Cache Occupancy pattern changes between applications

To explore the feasibility of using cache occupancy, we collect the cache occupancy traces for a set of malicious processes as well as benign applications. Figure 3(a) shows a cache occupancy pattern for a trojan and spy. The x-axis is the time series of the samples, and the y-axis is the changes in occupancy for each process. We observe increase and decrease patterns in cache occupancies for both trojan and spy. More importantly, a gain in occupancy by the trojan is mirrored by a loss of occupancy by the spy and visa versa, which is a very strong correlation pattern. This directly maps to the repetitive communicating operations for the two processes. As seen in Figure 3(b), benign workloads, which are not supposed to have such activity, do not have any such gain-loss patterns.

D. Dealing with Advanced forms of Timing channels

We studied a new vulnerability exposed by cache coherence protocol states, where the adversaries could cleverly manipulate the timing differences between accessing cache blocks in shared and exclusive coherence states and construct covert timing channels to illegitimately communicate secrets to the spy. We studied six different practical scenarios for covert timing channel construction. In contrast to prior works, we assume a broader adversary model where the trojan and spy can either exploit explicitly shared read-only physical pages (e.g., shared library code), or use memory deduplication feature to implicitly force create shared physical pages. We demonstrated how adversaries can manipulate combinations of coherence states and data placement in different caches to construct timing channels. We also explore how the adversaries could exploit multiple caches and their associated coherence states to improve the transmission bandwidth with symbols encoding multiple bits. Our experimental results on commercial systems showed that the peak transmission bandwidths of these covert timing channels can vary between 700 to 1100 Kbits/sec.

In order to close such malicious information leakage attacks possible on heterogeneously integrated MCUs, chip designers would need to consider addressing the timing gap in accessing read-only coherence states, namely shared and exclusive. To avoid performance-expensive trips to lower memory levels for shared state blocks in contrast to exclusive state blocks that can addressed in upper-level caches, appropriate notification mechanisms may be added in hardware.

IV. Fuzzing of Embedded Systems for Detecting Firmware Vulnerabilities

Fuzzing in embedded systems poses significant challenges due to the diversification of hardware and software, which includes a wide range of microcontrollers, architecture, and operating systems, each with its unique specifications and requirements. One of the critical challenges in fuzzing embedded systems is the innate nature of embedded systems with limited input/output (I/O) capabilities. This limitation impedes the fuzzer engine to detect memory crashes through sophisticated logging and debugging mechanisms [26]. Moreover, Instrumentation is also difficult in embedded systems due to the lack of open-source code. Instrumentation has a process of including additional code to monitor the execution of the program. In the context of embedded systems, closed-source code is common due to its proprietary nature. The

combination of all the factors, such as resource-constrained and closed-source code, significantly impedes the effectiveness of traditional fuzzing approaches such as AFL-fuzz and OSS-fuzz in the realm of embedded systems.

As a result, novel approaches and methodologies must be developed to overcome inherent challenges and enhance the reliability and security of embedded devices through effective fuzzing techniques. Several methodologies are proposed for fuzzing embedded systems to address all these issues. This can be broadly classified into three categories: hardware-based fuzzing, emulation-based fuzzing, and a hybrid approach.

A. Hardware-based Fuzzing

This type of fuzzing consists of testing on the hardware. It involves interfacing with the microcontrollers and peripherals of the embedded system. This approach is more accurate in finding vulnerabilities than an emulated environment. Most microcontrollers consist of a JTAG (Joint Test Action Group)/GDB (GNU Debugger) remote interface. These interfaces allow the fuzzer to set hardware breakpoints, monitor memory, and control the execution flow of the embedded software without modifying it. These parameters are crucial for obtaining coverage feedback and detecting crashes.

One notable example of hardware-based fuzzing is ARM-AFL [27], a framework for coverage-guided fuzzing on ARM-based embedded systems. It uses a lightweight heap memory corruption detector and runs the fuzzing entirely on the target device, allowing high throughput fuzzing similar to desktop systems. Another approach in hardware-based fuzzing is linker-based instrumentation, which involves modification of the linking process to insert instrumentation code. This method will trace function calls and monitor memory accesses in real-time. Harzer Roller [28] is a good example that uses this kind of instrumentation for ESP8266 microcontrollers.

The fuzzing framework for ESP32 microcontrollers [29] consists of data collection through the JTAG interface, which is sent to the fuzzer's host. Recent work [30] depends on two hardware interfaces known as tracing and debugging with three different probes as feedback to coverage-guided fuzzing. The ICS fuzz [31] uses programmable logic controller binaries for instrumentation to enable coverage-guided fuzzing for detecting crashes.

B. Emulation-based Fuzzing

It invokes the embedded software in an emulated environment, which provides precise control and observability of internal operations in manifold dimensions [32]. This method can provide detailed feedback on the system's behavior and facilitate the use of coverage-guided fuzzing. Tools like QEMU, an open-source emulator, often used in a virtual environment for the execution of a wide range of hardware architectures, separate it from physical hardware. It enables the execution of firmware in a controlled environment. HALucinator [33] and P2IM [34] are examples of emulation-based fuzzing that re-host the firmware at different levels of abstraction.

HALucinator focuses on the hardware abstraction layer. This approach relies on emulating the HAL functions, which are device-independent, making it reusable across other firmware. It enables the fuzzer to focus on software logic with hardware abstraction. This method tends to be more proficient than the physical testing environment by improving the effectiveness of fuzzing. Whereas P2IM focuses on entire hardware addresses to allow the fuzzer to learn peripheral behavior. This approach significantly reduces the manual effort of setting up the environment and improves the efficacy of fuzzing. This technique allows for scalable and efficient fuzzing of complex embedded devices.

Another notable tool in emulation-based fuzzing is known as Unicorn Engine [35], which is a lightweight, multi-architectural CPU emulator. It supports dynamic binary translation, which provides precise control over execution flow and is critical for fuzzing. Another notable framework known as Firmadyne [36] automates the process of extracting and emulating firmware images. This tool also helps find vulnerabilities without a physical device. The automation makes it valuable for large-scale testing of diverse firmware images.

C. Hybrid Approaches

This combines the robustness of both hardware-based and emulation-based fuzzing. These approaches aim to balance the fidelity of hardware testing with the flexibility of emulation. For instance, tools like Avatar2 [37], PANDA [38] use peripheral proxying to forward I/O requests from the emulator to the actual hardware, enabling a more accurate representation of the system's behavior while maintaining the benefits of an emulated environment.

FIRM-AFL is another framework that supports a hybrid approach; It is an extension of the American Fuzzy Lop (AFL). It integrates the emulation techniques with hardware feedback to guide the fuzzing process. It can refine its strategies to focus on critical areas of firmware through parameters it collected through feedback. This approach enhances the efficacy of the fuzzing process, ensuring that identified vulnerabilities are relevant to the actual hardware environment.

Another hybrid framework, concolic fuzzing [39], utilizes concolic execution to fuzz IoT firmware. It creates a symbolic execution in an emulated environment with hardware testing. The combination of low-level firmware re-hosting with fuzzing through hybrid memory-mapped I/O (MMIO) modeling in recent work HD-Fuzz [40] is a hardware dependency-aware firmware fuzzing system. It efficiently explores all firmware paths by using MMIO modeling and symbolic execution.

V. SnowFlake IoT: Ultra-low-cost Diversity Defenses

The existence of an active exploit leaves every IoT device vulnerable, as they all share the same vulnerability. Consequently, widespread penetration across the population of IoT devices is an eminent threat that compounds the risks associated with IoT devices. These population-wide IoT risks are not just imagined – in 2016, the Mirai IoT attack exploited vulnerabilities in IoT cameras to create a massive botnet [41], leading to widespread Internet denial-of-service attacks and highlighting the urgent need for enhanced IoT security.

A. Ultra-Low Cost Diversity Defenses

Diversity defenses are design- and compile-time defenses that introduce significant uncertainty into the underlying IoT device hardware and software. This uncertainty extends not only within each individual device but also across the entire IoT population, rendering each IoT device unique – a concept likened to each device being its own distinct "snowflake." Any attempts by attackers to breach the IoT network are thus complicated by the uncertainty inherent in the architecture, microarchitecture, and software of each device, necessitating costly and time-consuming reverse engineering efforts again and again before a successful exploit can be launched.

Diversity defenses have long existed in both software and hardware. A classic example from the software world is the PointGuard compiler defense against buffer-overflow attacks [42]. The PointGuard defense uses XOR-based encryption to obfuscate return address pointers on the stack. The approach was particularly effective against attacks, even emergent ones, as long as the attack needed access to code pointers on the stack. The drawback with the approach lay in its weak encryption. For example, with knowledge of a decrypted pointer, it was trivial to recover the encryption key (simply XOR the ciphertext with the known pointer value).

Our Morpheus work on hardware diversity defenses, developed in the DARPA SSITH program, made two important advances for diversity defenses [43]. First, it used strong encryption, such that having ciphertext and knowledge of the underlying pointer values provided no means to recover the key. Second, Morpheus significantly increased the entropy of encryption defenses – to 100's of bits, which eliminated the utility of probabilistic attacks. A Morpheus RISC-V was deployed and commercially red-teamed without ever being penetrated [44]. This physical design also showed that the overall cost of architecture design diversity could effectively be kept below 2% area, power, and performance.

While diversity defenses are powerful, we envision that with enough resources they can be penetrated, thus we advocate for the introduction of data sequestration capabilities [45] to protect critical information, which if exposed, could compromise the entire IoT network. Built by Agita Labs in post-SSITH commercialization, sequestered encryption (SE) ensures that protected data always remains encrypted, even during computation. Protected data is only decrypted in a small 190k-gate hardware-only functional-unit enclave [46] and never within the reach of software. With the deployment of SE in the IoT design, a single-device breach will not expose the critical data required to infiltrate the network, as this data remains encrypted and accessible only by the SE enclave.

B. Snowflake IoT

Snowflake IoT is an ongoing project at the University of Michigan that will further advance the strength and lower the cost of diversity defenses for resource-constrained IoT devices.

Snowflake's Threat Model : The Snowflake IoT threat model ensures that each IoT device is highly secure by making it extremely resource-intensive for an attacker to penetrate

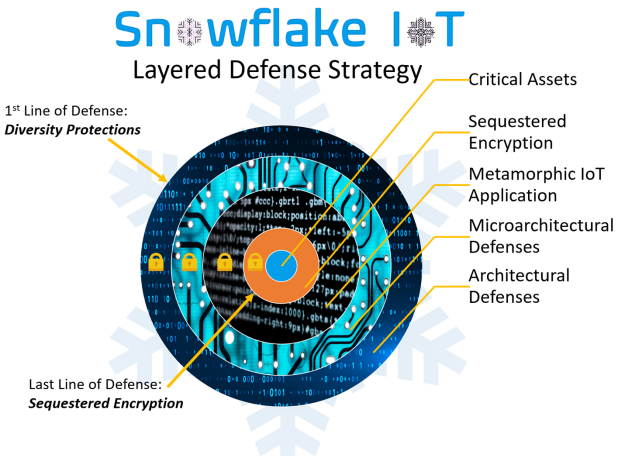


Fig. 4. Snowflake IoT's Layered Defense Strategy.

even a single device, requiring years of effort despite full access to its software and hardware specifications. The attacker can target input interfaces and monitor physical side channels but lacks knowledge of the device's internal state and the binary representation of its encrypted software. The win states for the attacker include successfully hacking a single device, which should yield no benefits for compromising additional devices due to unique per-device keys and sequestered encryption capabilities. The model prevents attackers from physically tampering with the devices and ensures that even with complete knowledge of existing vulnerabilities, they cannot gain command and control over the entire population of Snowflake IoT devices. This isolation of compromises guarantees that widespread attacks remain infeasible, preserving the security of the entire network.

The approach taken to achieve these strong defenses is to deploy diversity defenses in the *i)* architecture, *ii)* microarchitecture, and *iii)* software. All of these defenses introduce a non-trivial amount of entropy into the uncertainty of the design configuration, individualized to a particular device. Diversity between devices is also uncorrelated, taking the total degree of uncertainty for an IoT population to unprecedented levels, while keeping design costs low.

Architectural Diversity Defenses: In the Snowflake architecture, the Morpheus diversity defenses introduce randomness into the RISC-V architecture's code and code pointer representations. This randomized approach ensures that these representations are unique to each device, significantly impeding an attacker's ability to inject or manipulate code or code pointers. The efficacy of these defenses was demonstrated during the DARPA SSITH program, notably with the Morpheus secure CPU successfully resisting all red-team attacks [44].

Implementation of the Morpheus diversity defenses entails integrating light encryption into the RISC-V processor pipeline. Utilizing per-device keys, generated through onboard physically unclonable functions (PUFs), ensures that each Snowflake IoT device has a set of keys for encrypting both code and code pointers. By encrypting standard RISC-V architectural features under unique keys, the architecture achieves randomized code and code pointer implementations,

which in turn significantly increases uncertainty for potential attackers. The Snowflake design employs a 12-round Simon cipher, which provides a good balance between latency (1 cycle) and cipher strength (moderate-to-strong). Unlike the original Morpheus, the Snowflake architecture foregoes key "churn", a mechanism employed to periodically re-key code and code pointer encryption to deter ciphertext analysis. Given its cost, this capability is omitted in the Snowflake IoT design. However, the effect of churn will still be realized among different Snowflake IoT nodes since each device will possess its own distinct keys— churning in space rather than time.

Microarchitectural Diversity Defenses: The Snowflake microarchitecture is exploring multiple approaches to diversify microarchitectural characteristics [47]. A key approach is creating clones of the processor configured with boot-time PUF keys. Highly configurable versions of the processor will be designed into Snowflake IoT devices, which support a wide range of physical microarchitectural configurations (e.g., cache geometry, branch predictor organization, replacement strategies) that will be randomly configured at boot time. While the degree of entropy that is possible in the microarchitecture is more limited than the architecture and software, it works to create additional uncertainty that significantly complicates pure-microarchitectural attacks.

Software Diversity Defenses: The project's Metamorphic LLVM Compiler will work to inject semantically preserving entropy into applications, runtime software, and the operating system. In its simplest form, the metamorphic software infrastructure performs a set of semantically equivalent compile-time transformations to create many different versions of the same program – indeed, one program per installation of software on an IoT device. In traditional IoT systems, the same binary is run on each node of the system, making the system vulnerable when one node is compromised with a software vulnerability. Metamorphic software binaries will mitigate this vulnerability, adding to the Snowflake IoT diversity defenses provided by hardware-based defenses. If an attacker can reverse-engineer and discover a vulnerability on one device, they will learn very little about the software vulnerabilities of other Snowflake IoT devices in the same network.

The compiler will extend Snowflake's high-entropy hardware work to software, creating a high-entropy software toolchain capable of generating functionally equivalent C/C++ programs with vastly different characteristics programmed for intentional entropy. Although each instance of a Snowflake IoT device may use the same processor design, the functionally identical software executed will behave so differently that little can be learned about the other Snowflake IoT nodes in the system. The LLVM compiler supports high-entropy, semantics-preserving transformations. To ensure the framework's robustness, we start with a robust compiler infrastructure (i.e., LLVM), exploring the suite of readily available transformations and adding new ones that target code entropy. Compile-time transformations include those that affect the code size, composition, control flow, and data placement within the program's address space. As the project matures, we will work to incorporate optimizations into the compiler that could potentially erase any performance implications.

Fig. 5. The Snowflake IoT Demonstration Platform.

C. Putting These Ideas to the Test

To showcase the effectiveness of diversity defenses, the Snowflake IoT project is developing a physical prototype. A RISC-V based IoT device is being built to demonstrate the technique. Starting with the open-source Ibex RISC-V core and the PULPino RISC-V device platform, we are adding Snowflake IoT defenses to this system. The first two design prototypes will implement Snowflake V1, which will integrate existing Morpheus diversity defense IP. This simpler design will be faster and more reliable to deploy. It will first be deployed on an FPGA, and then later deployed in ASIC form. Later, Snowflake V2 will integrate novel hardware diversity defenses, and this platform will also be taped-out on silicon. Our demonstration platform will incorporate audio capture I/O and ML-based audio analysis to detect and classify environmental disturbances, mimicking the functionality required in fine-grained IoT ambient sensing applications. Leveraging our security model, the ambient sensing prototype benefits from layered defenses that deter single-device penetration. More significantly, the computational complexity involved in acquiring command and control of the entire network should prove computationally infeasible, further bolstering the security framework's utility in battlefield scenarios. Late in the project, we expect to engage a commercial red-team as part of our security validation.

D. Looking Ahead

The prospect that diversity defenses could improve IoT security, even for unpatched devices, while lowering security costs, is a very exciting prospect. To achieve this goal, we have to overcome the potential risks of bloating design area, power, or performance overheads. Our confidence in our defenses stems from earlier demonstrations of similar technologies that we have explored (Morpheus and SE), both of which emerged as highly secure and low-cost. These early technologies and their successes, combined with the optimizations we are building in this work, should render great potential to establish a new standard for ultra-low-cost IoT security.

VI. Securing Large Monolithic Systems: Challenges and Opportunities

Emerging embedded systems are highly heterogeneous and integrate diverse architectures, ranging from simple low-power devices to high-end accelerator-rich architectures that sport several sophisticated microarchitectural features and optimizations. The increasingly complex nature of these deployments has provided ample breeding ground for security vulnerabilities that are exploited in the wild.

Despite advances in zero-day threat analysis and mitigation, patching vulnerabilities continues to be an onerous task that involves a coordinated effort among the different vendors and stakeholders, invariably prolonging the risk of exposure. According to Google's Project Zero, a new exploit in the wild is discovered every 17 days on average, although it takes an average of 15 days across all vendors to patch a vulnerability that is being used in active attacks [48]. Moreover, legacy software modules that remain largely unmaintained or even unpatchable continue to linger in many critical codebases, as evidenced in the case of the notorious WannaCry ransomware attack [49], resulting in a non-trivial expenditure of resources and capital towards re-engineering and re-deployment.

A. Secure Compartmentalization

Most modern software and hardware systems in deployment are known to be monolithic in nature due in part to the continual demand for layering complex features amidst accelerated development life cycles. An immediate consequence of this monolithicity is that a single unchecked vulnerability within a module, when exploited, could permeate through the entire system, causing widespread damage.

Compartmentalization has often been suggested as a key strategy to break monolithicity, where each subject in the system (defined as a function or a collection thereof within a designated module/subsystem) is granted access to the bare minimum set of system resources required to carry out its pertinent task within the current execution context. The underlying enforcement mechanism ensures that the privilege set is appropriately switched out when the subject, its role, or the context changes over the course of execution. By enforcing the *principle of least privilege* [50], compartmentalization solutions have the ability to effectively limit damage due to an initial penetration to the specific part of the system containing the vulnerability. However, implementing fine-grained compartmentalization schemes for modern-day large monolithic systems in an automatic and transparent manner while also maintaining high performance is a critical challenge.

The key components of an effective compartmentalization scheme include – (a) refactoring of large monolithic systems into appropriate subject domains, (b) derivation of fine-grained access control policies for a given subject domain-object domain pair, and (C) mechanisms responsible for enforcing the derived access control policies. Most existing compartmentalization schemes [51], [52] target large monolithic software systems such as the Linux kernel and typically require manual/semi-automatic refactoring and access control policy derivation through a combination of static analysis, dynamic tracing, and domain expertise. The mechanisms used to enforce policies range from operating system mechanisms (e.g., processes), to software-based inline reference monitors, and hardware-assisted solutions [51], [53], [54]. Those leveraging hardware support typically tend to have lower performance overhead while imposing modest storage costs, although their protection granularity (i.e., page/sub-page/word/byte-level) could vary depending on the underlying hardware mechanism used for enforcement.

Despite having made these strides, several challenges still linger. First, novel compiler strategies for automatic code refactoring need to be investigated to reduce the manual effort entailed in compartmentalizing large codebases. These strategies should also incorporate compartmentalization expense modeling to make effective tradeoffs between performance and security. Second, the soundness and precision of static analysis schemes and coverage of dynamic analysis schemes need to be improved to avoid false positive and false negative access control policies. The scalability of these techniques also needs to be enhanced to support large codebases and complex privilege lattices. Third, it is critical to invest in hardware-software co-optimization strategies that will allow the software-based policy derivation schemes to better tune their approaches to the underlying hardware-based enforcement mechanism and routinely provide performance hints (e.g., prefetch hints, subject/context-switching gate call prediction hints, etc.), so as to foster the performant execution of a compartmentalized workload. Fourth, novel interchange formats need to be developed to allow for size- and time-efficient communication of metadata information to the hardware for secure and effective policy enforcement. Finally, novel strategies need to be developed to compartmentalize large monolithic hardware architectures that stack layers of intricate microarchitectural functionalities. This would allow for the quarantining of a particular hardware module infected upon initial penetration, thereby preventing the infection from seeping into the rest of the hardware.

B. Field-Upgradable Defenses

The incidence rates of new *in-the-wild* exploits have been steadily rising and the exploitation schemes have been consistently evolving – with every cycle of software patching, new variants of exploits emerge targeting new and/or more advanced features that were previously unexploited [48], [55]. For example, despite the significant progress made in curtailing stack corruption vulnerabilities, heap corruption and type confusion vulnerabilities that arise due to the lack of language-level enforcement of memory/type safety have been increasingly targeted by attackers in-the-wild [48], [55]. Similarly, logical domain-specific bugs (e.g., CVE-2021-1870 and CVE-2021-1906 that exploit design flaws due to improper error handling) have been frequent targets of remote exploits – detecting and patching logical errors is extremely difficult. As we have witnessed the industry grapple with a seemingly endless stream of zero-day vulnerabilities, it has become increasingly

clear that perhaps the two biggest challenges are time to exploit mitigation and the cost of deployment. If defenses are not deployed promptly, the window of exploitation becomes too large to control the damage. On the other hand, rising costs in verification and deployment efforts could hinder the success and longer-term sustenance of the business.

Multiple prior works have shown that low-level code instrumentation [56], [57] is an effective means to enforce security policies on-demand. Microcode-based solutions have shown further promise in enabling field-upgradeable defenses by surgically injecting security checks into running code on-demand and in the field. Notably, *context-sensitive fencing* (CSF) [58] is a microcode-level defense against multiple variants of Spectre. CSF provides a configurable framework that detects potential unauthorized accesses at the decoder and further triggers the surgical instrumentation of running code with speculation fences that are enforced at different stages of the pipeline, offering varying tradeoffs with respect to security and performance. In similar vein, CHEx86 [59], a microcode-enabled capability machine is able to secure unmodified code against temporal and spatial memory safety exploits by tracking allocation and de-allocation events, pointer arithmetic, pointer movement, and spilled pointer aliases on-the-fly, and further instrumenting an application in execution with microcoded capability operations.

However, to enable more holistic defenses that target a wider range of attack vectors, it is more critical to bolster the hardware-software contract through a security-centric stack that is decoupled from the Instruction Set Architecture (ISA), to empower software to dynamically push expressive security policies to hardware, where they can be transparently and efficiently enforced on-demand and in-the-field through novel microarchitectural mechanisms and runtime monitoring techniques, without the need for recompilation, redeployment, and frequent hardware upgrades. There are substantial benefits to such an approach. First, it promotes versatility by allowing software modules to specify their own set of security properties that are inherent to their particular problem domain (e.g., protecting privacy-sensitive data and code from side-channel inference) and/or the language in which they are written (e.g., enforcing type integrity). Second, by decoupling the security specifications from the ISA and the binary, it allows policies to be dynamically reconfigured to address new exploits in a timely fashion, without recompiling applications. Third, it allows hardware vendors to transparently adapt the underlying enforcement mechanisms with every new generation.

C. Secure Resource Virtualization

Resource virtualization is one of the key ideas underpinning modern hardware and software systems. By virtualizing physical resources into virtual pools, it promotes fair and flexible user distribution and improved overall resource utilization. Industry platforms such as VMWare's vSphere, Microsoft's Hyper-V, and OpenStack facilitate virtualization of storage, network bandwidth, and compute capacity, to name a few, and the academic literature abounds with secure system-level virtualization solutions [56], [57]. However, a key concern with sharing of physical resources is that it is vulnerable to potential side-channel inference and crosstalk where co-located attackers that compete for shared physical resources may covertly infer or influence a victim's behavior.

To facilitate the consolidation of fine-grained system-level and microarchitectural resources such as caches into virtualized pools in a secure, transparent, and scalable manner, while minimizing side-channel interference, it is important to enforce several key security properties. First, similar to compartmentalization, all accesses need to abide by the *principle of least privilege*, i.e., each entity in the system (e.g., a user or a group of users in the same trust domain) should be granted access to only those resources that have been specifically allocated to it with the minimal set of privileges required to perform their underlying task. Second, no entity in the system should be allowed to perform an operation without explicitly exercising their access rights and getting them verified prior to performing the operation. Third, sharing and revocation of resources need to be explicit and voluntary, and must to restricted to occur within a trust domain. Fourth, to ensure fairness and to prevent monopoly, it must be enforced that any given entity will not be deprived of allocating resources up to a preset minimum guarantee, while also being disallowed from exceeding its maximum allocation threshold.

However, enforcing these properties could entail high performance and/or storage overheads. First, virtualizing fine-grained system-level resources necessarily implies an additional layer of indirection which typically entails the lookup of additional software or hardware structures, thereby imposing an appropriate lookup penalty that needs to be paid upon every access, calling for novel and optimized lookup procedures where the performance cost of the lookup is minimized. Second, a substantial amount of metadata encoding access rights needs to be tracked for every user and for every virtualized resource in the system that is either temporally or spatially shared, calling for novel data structures or hardware organizations that minimize the storage overheads and scalable across a sufficiently large number of trust domains.

VII. Conclusion

In this paper, we have presented various strategies for detecting and defending vulnerabilities in heterogeneous and monolithic systems, including covert timing channel detection that leaks sensitive information behind hardware IP and various fuzzing techniques for detecting firmware vulnerabilities. This is followed by multilayered defense strategies for miniature-based IoT devices known as snowflake IoT, which provides diversified defenses in terms of architectural, microarchitectural, and software models. Finally, for enhancing the security of large monolithic systems, three key strategies can be implemented: secure compartmentalization, which isolates compromised hardware components from the rest of the system to prevent the spread of threats; the use of field-upgradable devices, which reduces costs and decreases the deployment period; and secure virtualization, which eliminates the risks of crosstalk and unauthorized interference by creating isolated virtual environments within the system.

REFERENCES

[1] K. Troester and R. Bhargava, "AMD next generation "zen 4" core and 4th Gen AMD EPYC™ 9004 server CPU," in *IEEE Hot Chips Symposium (HCS)*, 2023.

[2] M. Eceiza *et al.*, "Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems," *IEEE IoT Journal*, vol. 8, pp. 10 390–10 411, 2021.

[3] S. Sonko *et al.*, "A comprehensive review of embedded systems in autonomous vehicles: Trends, challenges, and future directions," *World J. Advanced Research and Reviews*, vol. 21, pp. 2009–2020, 2024.

[4] A. Dhavlle *et al.*, "Imitating functional operations for mitigating side-channel leakage," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, pp. 868–881, 2022.

[5] H. Fang *et al.*, "Reuse-trap: Re-purposing cache reuse distance to defend against side channel leakage," in *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[6] W.-M. Hu, "Reducing timing channels with fuzzy time," *J. Comput. Secur.*, vol. 1, p. 233–254, 1992.

[7] K. Okamura and Y. Oyama, "Load-based covert channels between xen virtual machines," in *ACM Symposium on Applied Computing*, 2010.

[8] T. Ristenpart *et al.*, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *ACM Conference on Computer and Communications Security*, 2009.

[9] Z. Wu *et al.*, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *USENIX Security Symposium*, 2012.

[10] G. Venkataramani *et al.*, "Detecting hardware covert timing channels," *IEEE Micro*, vol. 36, pp. 17–27, 2016.

[11] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018.

[12] S. L. Thomas *et al.*, "HumIDIFy: A tool for hidden functionality detection in firmware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[13] Y. David *et al.*, "Firmup: Precise static detection of common vulnerabilities in firmware," *ACM SIGPLAN*, vol. 53, pp. 392–404, 2018.

[14] A. Mera *et al.*, "DICE: Automatic emulation of dma input channels for dynamic firmware analysis," in *IEEE Symposium on S&P*, 2021.

[15] E. Gustafson *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[16] S. Ray *et al.*, "Invited: Formal verification of security critical hardware-firmware interactions in commercial socs," in *IEEE Design Automation Conference*, 2019.

[17] R. Saravanan and S. M. P. Dinakarrao, "The emergence of hardware fuzzing: A critical review of its significance," 2024. [Online]. Available: arxiv

[18] O. Shwartz *et al.*, "Reverse engineering iot devices: Effective techniques and methods," *IEEE IoT Journal*, vol. 5, pp. 4965–4976, 2018.

[19] J. Zaddach and A. Costin, "Embedded devices security and firmware reverse engineering," *Black-Hat USA*, 2013.

[20] R. Saravanan and S. M. Pudukotai Dinakarrao, "The Fuzz Odyssey: A Survey on Hardware Fuzzing Frameworks for Hardware Design Verification," in *Great Lakes Symposium on VLSI*, 2024.

[21] J. Yun *et al.*, "Fuzzing of Embedded systems: A Survey," *ACM Comput. Surv.*, vol. 55, 2022.

[22] S. Kasarapu *et al.*, "Enhancing IoT malware detection through adaptive model parallelism and resource optimization," 2024. [Online]. Available: arxiv

[23] A. Dhavlle *et al.*, "Work-in-progress: Sequence-crafter: Side-channel entropy minimization to thwart timing-based side-channel attacks," in *International Conference on (CASES)*, 2019.

[24] A. Dhavlle *et al.*, "Entropy-shield:side-channel entropy maximization for timing-based side-channel attacks," in *International Symposium on Quality Electronic Design (ISQED)*, 2020.

[25] U. S. D. of Defense, *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, 1987, vol. 83.

[26] M. Muench *et al.*, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *NDSS*, 2018.

[27] R. Fan *et al.*, "ARM-AFL: Coverage-guided fuzzing framework for arm-based iot devices," in *ACNS Workshops*, 2020.

[28] K. Bogad and M. Huber, "Harzer roller: Linker-based instrumentation for enhanced embedded security testing," in *Reversing and Offensive-oriented Trends Symposium*, 2019, pp. 1–9.

[29] M. Börsig *et al.*, "Fuzzing Framework for ESP32 Microcontrollers," in *International Workshop on Information Forensics and Security*, 2020.

[30] Z. Feng and J. Ma, "TWFuzz: Fuzzing embedded systems with three wires," in *ACM Int. Conf. on Languages, Compilers, and Tools for Embedded Systems*. Association for Computing Machinery, 2024.

[31] D. Tychalas *et al.*, "ICSFuzz: Manipulating I/Os and repurposing binary code to enable instrumented fuzzing in ICS control applications," in *USENIX Security Symposium*, 2021.

[32] M. Eisele *et al.*, "Embedded fuzzing: a review of challenges, tools, and solutions," *Cybersecurity*, vol. 5, 2022.

[33] A. A. Clements *et al.*, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *USENIX Security Symposium*, 2020.

[34] B. Feng *et al.*, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *USENIX Security Symposium*, 2020.

[35] N. A. Quynh and D. H. Vu, "Unicorn: Next generation cpu emulator framework," *BlackHat USA*, vol. 476, 2015.

[36] D. D. Chen *et al.*, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, vol. 1, 2016.

[37] M. Muench *et al.*, "Avatar 2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, 2018.

[38] B. Dolan-Gavitt *et al.*, "Repeatable reverse engineering with panda," in *Protection and Reverse Engineering Workshop*, 2015.

[39] J. Yu *et al.*, "Poster: Combining fuzzing with concolic execution for iot firmware testing," in *ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[40] J. Kim *et al.*, "Hd-fuzz: Hardware dependency-aware firmware fuzzing via hybrid mmio modeling," *Journal of Network and Computer Applications*, vol. 224, 2024.

[41] Mirai (malware). [Online]. Available: "https://en.wikipedia.org/wiki/Mirai"

[42] C. Cowan *et al.*, "Pointguard TM: Protecting Pointers from Buffer Overflow Vulnerabilities," in *USENIX Security Symposium*, 2003.

[43] M. Gallagher *et al.*, "Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[44] A. Harris *et al.*, "Morpheus II: A RISC-V security extension for protecting vulnerable software and hardware," in *IEEE International Symposium on Hardware Oriented Security and Trust*, 2021.

[45] L. Biernacki *et al.*, "Sequestered encryption: A hardware technique for comprehensive data privacy," in *IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 2022.

[46] V. B. Todd Austin and A. Kisil, "TrustForge: A Cryptographically Secure Enclave for Azure and AWS," in *HotChips 2023*, 2023.

[47] H. Fang *et al.*, "SC-K9: A self-synchronizing framework to counter micro-architectural side channels," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 11–18.

[48] B. Hawkes, "0day "in the wild"," 2019.

[49] S. Mohurle and M. Patil, "A brief study of wannacry threat: Ransomware attack," *International J. Advanced Research in Computer Science*, 2017.

[50] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, 1975.

[51] D. McKee *et al.*, "Preventing kernel hacks with hakc," in *Network and Distributed System Security Symposium*, vol. 22, 2022, pp. 1–17.

[52] N. Roessler *et al.*, "μscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts," in *International Sym. on Research in Attacks, Intrusions and Defenses*, 2021, pp. 296–311.

[53] A. Vahldiek-Oberwagner *et al.*, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *USENIX Security Sym.*, 2019.

[54] S. Amar *et al.*, "Cheriot: Rethinking security for low-cost embedded systems," Microsoft, Tech. Rep., 2023.

[55] M. Miller, "Trends and challenges in the vulnerability mitigation landscape," *WOOT*, 2019.

[56] J. Criswell *et al.*, "A virtual instruction set interface for operating system kernels," in *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2006.

[57] J. Criswell *et al.*, "Memory safety for low-level software/hardware interactions." in *USENIX Security*, 2009.

[58] M. Taram *et al.*, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ASPLOS*, 2019.

[59] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *ISCA*, 2020.