Immunization Techniques against the Side Channel Attack

Separation and Virtualization for Secure System Software

Arun Subbarao, Lynx Software Technologies, Inc

Meltdown and Spectre, two recent side channel attacks have demonstrated all too clearly how some multi-core processor-based software can be exploited, resulting in loss of confidentiality. Although it was a largely hardware design issue that forced software suppliers to provide workarounds, we contend that secure systems can be designed using separation and virtualization to isolate security components and minimize or even immunize the system from severe side channel attacks such as Meltdown and Spectre. Modern multicore processor architecture has evolved to the point where analyzing complexities and emergent behavior is a significant problem for system architects. This paper will define technical approaches to addressing these challenges.

This paper will take the audience through an explanation of side channel attacks & Meltdown and Spectre specifically. We will then discuss multi-core processor capabilities that can be used to mitigate & avoid these attacks through system software architecture & design. In effect we will show how to make multi-core system software resilient and secured by design, isolating hosted OS's and applications which become vulnerable to these attacks. We will show that these multi-core capabilities can supplant many capabilities normally assumed to need to reside in an OS or RTOS, where they are vulnerable to these attacks due to insufficient attention to least-privilege software design as they extended their services across multiple cores.

1. Introduction

Historically, commercially available operating systems & hypervisors have not been designed to provide high levels of security. Most commercially available operating systems & hypervisors contain areas of vulnerability that contribute to the weakening of cyberspace security. As systems & devices start to proliferate, the interconnectivity of these devices through the Internet creates enormous challenges for cyber-security and perhaps even creates vulnerabilities that we have yet to recognize.

This paper assesses the landscape of side channel attacks and potential immunization techniques with a review of the following topics:

- Side Channels
- Modern Processors
- Meltdown & Spectre
- Separation Kernels
- Exemplary Architecture
- Conclusions

This paper analyzes each of these topics as it relates to system design and outlines the enormous complexities associated with mitigating emergent security threats. An exemplary architecture that follows the technology principles outlined here will be capable of mitigating side channel vulnerabilities and provide a more solid foundation for designing secure systems.

2. Side Channels

Covert channels and Side channels are two sources of information leakage that cause significant security challenges in complex systems. It is important, however, to distinguish between the two.

A covert channel involves a hidden source of information leakage which is intentional. This usually implies that a trusted task (e.g.: process/thread) is deliberately leaking information in a covert manner, that another malicious entity can retrieve and use. In this scenario, the sender and receiver are not allowed to communicate through overt means, yet they leak information. It invariably leads to loss of confidentiality and requires careful analysis to analyze and eliminate.

A side channel is an unintended communication of information that is usually a side effect of the primary task. The primary, trusted task performing a trusted operation has no overt communication channel to a receiver task, nor is it intentionally leaking information. A side channel is usually an unintentional leakage of information, and the trusted task is unaware of the leakage. This too leads to loss of confidentiality but is just as complicated to analyze and mitigate.

Some of the common categories of side channels are listed below:

Timing-based

Timing-based side channels involve gathering information by observing the differences in the time taken to execute certain operations, exploiting micro-architectural changes that occur during this execution such as cache line evictions. Based on the timing parameters, one can establish a viable side channel that can extract sensitive information.

Trace-based

Trace related side channels are based on physical proximity to the victim machine, where continuous monitoring of the system while it is executing a sensitive operation can lead to differences in power draw, electromagnetic emanations etc.

Access-based

Access based side channels depend on access to the machine and exploiting shared resources between the attacker and victim. This type of attack has gotten more common as modern processor design has introduced many such shared access

mechanisms, such as caching, in the interest of performance, and at the expense of security. This type of shared access is more pronounced in the case of multi-core processor design,

This paper mainly focuses on the access-based side channels, and potential mitigations using architectural approaches.

3. Modern Processors

In the context of the types of side channels discussed, it is important to review the evolution of modern processor architecture and some of the micro-architectural innovations that lead to greater possibilities of side channels as outlined in [1].

Multi-core/Multi-processors

One of the most critical advances of modern processor designs have been the shift from uniprocessor to multiprocessor design. As the advances in uniprocessor design started to saturate in terms of performance and scalability, semiconductor vendors have been steadily moving to multi-processor & multicore designs to increase the performance of these processors, as well as allow modern system software to innovate and harness these multicore designs. This has led to the evolution of symmetric & asymmetric multiprocessor (SMP, AMP) operating systems, microkernels, hypervisors that all take advantage of the processing power afforded by these multiprocessor designs. A typical SMP system architecture is shown below.

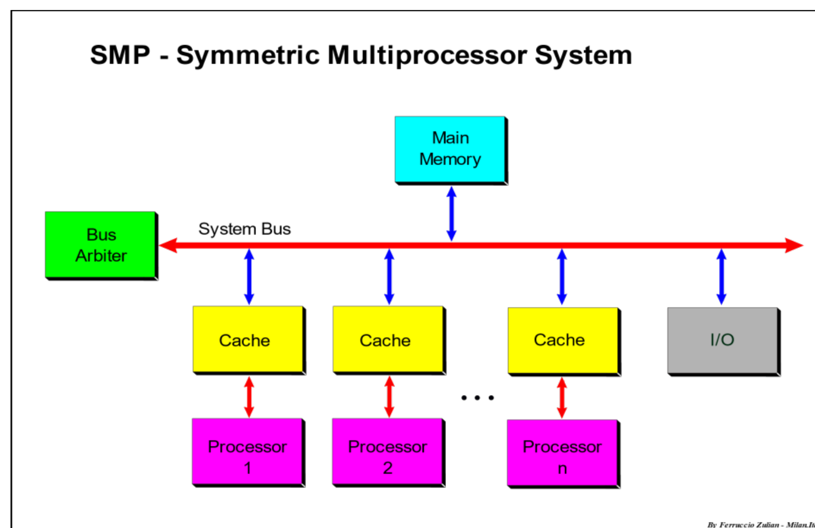

Figure 1: Symmetric Multiprocessor System[1]

1 https://en.wikipedia.org/wiki/Symmetric_multiprocessing, Ferruccio Zulian, Creative Commons Attribution

Hyper-threading

Although specific to the Intel processor family, the hyperthreading technology is worth examining for its micro-architectural approach. Hyperthreading uses two logical cores within a single physical core to increase the amount of parallelism in compute power. For instance, enabling hyperthreading on a single physical core can create two logical cores each with the ability to independently execute processes or threads. Operating systems and hypervisors that utilize hyperthreading can increase the number of parallel execution paths on these logical cores. At a micro-architectural level, hyperthreading is achieved by duplicating the execution unit for each logical processor but sharing the execution resources that includes the execution engine, caches, and system bus interface. Such shared resources at the hardware level opens a greater possibility of side channels.

Cache & Cache Hierarchy

Cache is a small memory block that is placed between the CPU and main memory (DRAM) to allow frequently accessed information to be stored for faster access. There is usually a cache hierarchy that indicates different levels of cache L1, L2, L3 which may be per-core or shared across multiple cores. One such hierarchy is shown in the diagram below which has a core-specific L1 cache, and shared L2 & L3 caches.

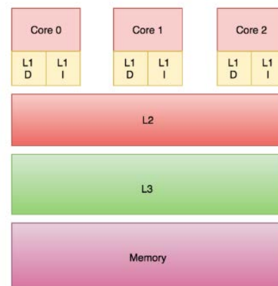

Figure 2: Multiprocessor Cache Hierarchy

Cache Coherence

Cache coherence allows multiple caches in a multiprocessor system to be synchronized with each other, so that information in one cache gets propagated to all caches to prevent stale data in each cache. There are protocols, such as snooping, that is used to keep the caches "coherent".

Branch Prediction & Speculative Execution

Modern processors have been adding capabilities to improve execution performance of both single core and multi-core processors. One such capability is the speculative execution, where the CPU aggressively fetches and executes out-of-order instructions especially at a branch point, effectively speculating on the case where the branch is taken

and when it is not. To facilitate this, various small caches are used such as the branch target buffer (BTB), or Re-order buffer (ROB), which may hold the address of the branch taken instruction or the execution contents of the speculative execution. By speculatively executing out-of-order instructions the efficiency of the execution pipeline is greatly increased. However, such speculative execution leaves lots of footprints that can be exploited as seen in the next section.

4. Meltdown & Spectre

Recently, several researchers published their findings regarding a significant, and far-reaching side channel attack on modern Intel processors [3, 4], which was later identified as being applicable to the POWER & ARM family of processors as well**.**

Meltdown

Meltdown is a software-based side channel attack using out-of-order execution in modern multiprocessors. In the Meltdown attack, a rogue process is able to read all of privileged memory that is usually in the operating system kernel, and gain access to sensitive information. The most insidious part of this hardware vulnerability is that it does not require the rogue process to have any special permissions to access the microarchitectural state of the CPU.

The attacker is exploiting a vulnerability in the processor micro-architecture that uses a combination of out-of-order execution to cause the CPU to execute instructions that are normally disallowed for the unprivileged process, and then using a cache side channel to retrieve the information that is available in the cache based on that out-of-order execution.

In a specific instance of exploiting access to kernel memory from an unprivileged user process, the Meltdown attack proceeds as follows:

- Initially a rogue process loads a kernel memory address and attempts to access this address.
- The CPU proceeds to use out-of-order execution to retrieve the contents of that kernel memory location and fills the cache.
- Subsequently the CPU recognizes that the read of the kernel memory address attempted by the rogue process is disallowed due to insufficient privilege, and the access fails as it should.
- However, the cache now has the content of the kernel memory, that can be exfiltrated using standard cache side channel exploits like Flush-Reload.
- This process can repeat for all kernel memory locations and thereby allowing a rogue process to map the entire kernel memory contents with no privilege requirement.

The root of this problem stems from a desire to get the best performance out of an OS, where you map the kernel address space into the same context as the user address space, so that extra context switches are eliminated when you make a system call for instance.

However, this architecture has resulted in widespread vulnerabilities for many OSes and hypervisors. Commercially available hypervisors are also susceptible to Meltdown since they design for the same efficiency.

Spectre

While Meltdown requires the rogue process to have no special access to the system to gain access to sensitive information, Spectre attack assumes that the attacker has some knowledge of where the sensitive information is accessed and attempts to exploit the out-of-order execution to gain access to it through a side channel.

A key microarchitectural component in this attack is the branch predictor, which based on previous execution sequences predicts that a branch will indeed be taken (or not taken as the case may be). There are a few different variants of Spectre that have been identified, which exploit the speculative execution in different ways:

Conditional branching – In this case, the rogue process mis-trains the branch predictor by first providing valid inputs to establish a pattern of the branch conditional being true/false, and then feeds it a malicious input. The branch predict assumes that the conditional will follow the previous pattern and proceeds to execute the rest of the instruction set before the conditional check is complete. By the time the check fails, the cache has already been filled with sensitive information that the attacker should not have access to and is subsequently exfiltrated.

Indirect branches – In this scenario, the attacker selects a gadget from the victim's address space and influences the victim to speculatively execute the gadget (machine code snippets in the victim's address space). The branch target buffer is trained to mis-predict the branch to a gadget's address from the indirect branch instruction. As with the previous variant, the speculative execution proceeds to execute the gadget until the permission check fails, but it is too late to stop the cache lines from being filled.

Some other variants of Spectre have also been identified, using the same methodology of exploit. It is important to distinguish that the Spectre attack is only accessing the address space that the victim is normally allowed to access during normal operation. It does not rely on any exceptions or page faults to succeed. As such, Spectre is considered different and orthogonal to the Meltdown attack.

Linux, RTOS, Microkernels, Commercial Hypervisors

The implications of Meltdown and Spectre on real-time operating systems, microkernel and commercial hypervisors are significant. Almost all of these are considered susceptible to Meltdown, which is a considerably wider attack by unprivileged processes.

This is primarily because very little attention has gone into examining side channels during the architecture and design of system software. Although the vulnerabilities that have been exploited by Meltdown and Spectre clearly stem from poor hardware security

design, the system software architecture and design have not adopted security first-principles for their implementation.

The quest for performance and efficiency at the expense of security is a root cause of such design failures [6, 7]. Here are some of the most significant design limitations in system software that result in exploitable vulnerabilities.

Mapping Kernel memory into User space

One of the design choices made by system software is to map all of kernel memory into the user space of a process, only relying on the supervisor mode privilege to prevent access from unprivileged mode. While this improves efficiency, it leaves a significant security vulnerability that can be exploited as is demonstrated by Meltdown.

Insufficient Cache handling

Most operating systems and hypervisors are not designed to monitor and allocate cache such that side channels are minimized. This is primarily due to level of dynamism that is inherent in these kernels, which makes it harder to assess which parts of the cache are active for VMs or processes. In multiprocessor systems, symmetric multiprocessing adds another level of complexity that makes it extremely complex to assess the various interference channels and their impacts on the cache. In many cases, Asymmetric multiprocessing is used to reduce the complexity of assessing potential interference channels and mitigate the threat of side channels.

Memory De-duplication

One of the key design choices made by commercial hypervisors is memory de-duplication, which is optimizes the memory used by multiple VMs and keeps a single copy until a write operation is invoked. Based on a copy-on-write approach, the memory is duplicated at the time when a write operation is invoked. While this greatly optimizes the density of VMs in a system, it creates a significant side channel that can be exploited. For instance, since the memory access times for a de-duplicated memory is longer than for a normal page it allows timing-based attacks that can leak memory information to a neighboring VM.

5. Separation Kernels

Separation kernels represent a new breed of technology that presents a design approach that is superior to other system software in the realm of mitigating side channel attacks.

Separation Kernel Technology

In 1981, John Rushby, presented a paper titled "The Design and Verification of Secure Systems" at the Eighth ACM Symposium on Operating System Principles [2]. In this paper he noted "secure systems should be conceived as distributed systems in which security is achieved partly through the physical separation of their individual components

and partly through the mediation of trusted functions performed within some of those components." This introduced the separation kernel concept as a foundation for secure systems. Rushby also proposed that virtual machines be adopted to run with the express intent of providing varying levels of component functionality with security mediation performed by the separation kernel itself. While early realization of a separation kernel entailed a more limited set of capabilities among the virtual contexts it provided, due to the limitations of the processors of the time, the idea and the supporting hardware have continued to evolve over the years. Over the last decade, the notion of separation has matured, and the processor architecture advances have resulted in capabilities within the separation kernel that were previously unavailable. With hardware offering support for virtualization through mechanisms like Intel's VT-x & VT-d, a modern separation kernel is, in effect, an operating system that can run other operating systems as "subjects". A subject is defined as a collection of resources accompanying a piece of software (like an OS) that allow it to be executed and monitored by the separation kernel. It is important to note that a subject might not necessarily be an OS at all; in fact, it could be a dedicated program that runs without an OS within a separation kernel context. Various parts of this system would be protected by the separation kernel handling low-level communication with the outside world as well as providing protected interaction between the different subjects.

A separation kernel has the following unique characteristics, each of which can offer a technique to immunize against the threat of side channel attacks:

Hard Isolation

One of they key characteristics of the separation kernel is the ability to provide hard isolation between different subjects that are executing atop the kernel. This hard isolation is typically achieved by design approaches such as time-space partitioning, where all the resources of a system are partitioned into discrete, non-overlapping subsets and assigned to exclusively to a subject. A key design principle that is fundamental to a separation kernel is that the dynamic reconfiguration of these resources at run-time is prohibited. This ensures that a resource, like memory, never gets reassigned to another subject during execution which has important implications for reducing side channels.

Fine-grained Hardware Control

Another distinguishing characteristic of the separation kernel is the ability to provide fine-grained control of hardware resources to the user. A typical separation kernel allows control of the following hardware resources:

CPU Allocation

Separation kernels allow fine-grained control of CPU allocation to subjects (VMs, bare metal applications etc.). Given the level of shared state that a hyperthreaded cores share with each other, they offer a large bandwidth side channel for exploit. A system designer can ensure that a trusted application or VM does not run adjacent to an untrusted VM, on

a hyperthreaded core for instance. One can also ensure that trusted functions and untrusted functions are scheduled on processors that are not sharing L1/L2 caches, which is a significant threat for side channels.

CPU scheduling

Fine-grained control of CPU scheduling is another facet of a separation kernel that allows the ability to allocate different levels of periodicity to VMs or tasks running on a single core to minimize overlapping execution. This would potentially prevent simultaneous execution of other processes (or Guest OS VMs) when a sensitive process is running on the same core thereby minimizing the possibilities of cache lines being evicted.

I/O devices

A separation kernel can also control the allocation and operation of I/O devices and prevent DMA bus mastering devices from accessing memory that is outside the given range. The usage of mechanisms such as IOMMU (Intel) or SMMU (ARM) ensures that the memory that I/O devices use are consistent with their allocation, and information leakage is minimized.

Timers

A separation kernel can allow the adjustment of the granularity of timers in the system. The more fine-grained the timer resolution, the more it may facilitate timing-based attacks in the system. In some cases, it may be appropriate to reduce the resolution of the timers to mitigate timing channels.

Static Instantiation & Reduced Dynamism

One of the key facets of a separation kernel is its ability to statically instantiate memory allocation and minimize dynamism in the system. The allocation of memory to VMs, and the prevention of memory re-use in the system greatly reduces the possibility of residual information leaking due to shared memory or cache line overlaps. One can prevent instantiation of new VMs in the separation kernel, so would be impossible to launch an attack where a malicious VM deliberately tries to share cache resources with a victim VM.

In many instances, shared memory between different VM environments are disallowed, or cleansed before use/reuse to minimize leakage of information due to shared resources.

Inter-Subject Communication

Following the notion of keeping communication between subjects highly regulated, the separation kernel can provide a protected, secure channel of communication between subjects using a message passing API. Message transmission is generally asynchronous

and unidirectional for security concerns. The security policy defines the authorized communication between two different subjects and can be defined as unidirectional or bi-directional.

Cache Handling

As hardware support for managing caches continues to improve, separation kernels can better utilize mechanisms such as cache allocation technology to lock cache lines to specific VMs and prevent evictions by different VMs.

Bare Metal Applications

A separation kernel can offer the ability to run sensitive and trusted functions as bare metal applications independent of an operating system. These bare metal applications can be small enough to ensure that they occupy specific cache lines can offer the ability to isolate cache footprints. These bare metal applications can also operate as software guards, that mediate communication between trusted and untrusted subjects.

6. Exemplary Architecture

An exemplary architecture of a system that utilizes various capabilities of a separation kernel to achieve a superior security posture against side channel attacks is described using the LynxSecure™ separation kernel and hypervisor.

LynxSecure

LynxSecure is a highly secure, separation kernel & hypervisor. By combining the best-of-breed capabilities of the separation kernel technology and virtualization, LynxSecure provides unmatched capabilities to run one or more guest operating systems such as Windows, Linux & RTOS using commonly available Intel or ARM platforms.

LynxSecure provides a secure virtualization environment in which multiple secure and insecure operating systems can perform simultaneously without compromising security, reliability or data [5]. LynxSecure was designed from the ground up to be small, real-time,
and security certifiable, it offers both time-space partitioning and operating-system virtualization. LynxSecure's hypervisor and virtualization technology supports multiple heterogeneous operating system environments, virtualizing the underlying hardware to guest operating systems. The use of hypervisors and virtualization technology allows an operating system and its applications to run within the environment of another kernel. LynuxWorks' Hypervisor technology allows multiple dissimilar operating systems to securely share a single physical hardware platform. LynxSecure uses its Hypervisor to create a virtualization layer that maps physical system resources to each guest operating system.

Least Privileged Design

LynxSecure utilizes a least privilege design, which ensures that each component only has the amount of privilege that is necessary for it to perform its function. This distributed privilege model ensures that no single component has global privileges that can be exploited in the system.

Page Table Isolation

LynxSecure separation kernel utilizes a separate page table for the hypervisor mode execution, and the guest VMs running on top of it. This kernel page table isolation for the separation kernel renders it immune to the Meltdown attack.

Optimized use of Hardware Protection Mechanisms

LynxSecure provides a highly optimized implementation of the separation kernel and hypervisor by utilizing advanced protection mechanisms available in multicore processor such as those in the Intel family. These key mechanisms are summarized below and form an important underpinning for the overall security of the system.

VT-x - LynxSecure enables VT-x features within the processor to manage Guest VMs, which assists transitions to LynxSecure based on modifications by Guests. Thus, privileged operations cannot bypass the hypervisor. LynxSecure performs these modifications based on security policy.

VT-d – LynxSecure utilizes VT-d extensions to establish DMA regions consistent with guest OS memory allocation. LynxSecure’s policies configure VT-d to arbitrate DMA accesses and ensure that only allowed memory regions are accessed. This ensures that the separation policy applies not only to memory but also to I/O device assignment and to data transfers between memory and DMA devices.

EPT, SMAP, SMEP – LynxSecure fully utilizes advanced technologies such as extended page tables, supervisor mode access prevention, supervisor mode execution prevention, etc. while preserving the separation and isolation between VMs.

Memory Isolation

A key element of the security of LynxSecure is the non-reuse of memory regions across guest VMs. All memory regions are assigned by LynxSecure statically based on policy and are not re-used throughout the instantiation of the system. This ensures that there can be no eavesdropping by a different VM through the re-assignment of a different VM’s memory region.

Immutable Security Policy

LynxSecure separates out security policy from the hypervisor software and includes it in a read-only manner to the hypervisor software. Thus, the security policy and the run-time image are a locked-down configuration that ensures that users cannot change the security policy.

Time-Space partitioning

LynxSecure uses a time-space partitioning technique that involves a static resource reservation & fault isolation mechanism. This mechanism involves the creation of brick-wall partitioning of memory, time and device resources, which ensures that applications can execute in an environment that is strictly partitioned, highly protected and completely isolated from other applications. Temporal (time) partitioning is achieved through a configurable, fixed cyclical time-slicing scheduler for deterministic time partitioning providing guaranteed availability of CPU time for partitions.

Bare Metal Applications

The LynxSecure application (LSA) mechanism allows bare metal applications to be created and executed on LynxSecure that provides a superior, fine-grained control of memory and CPU core allocation to trusted functions.

An exemplary architecture of LynxSecure with untrusted and trusted subjects is shown below.

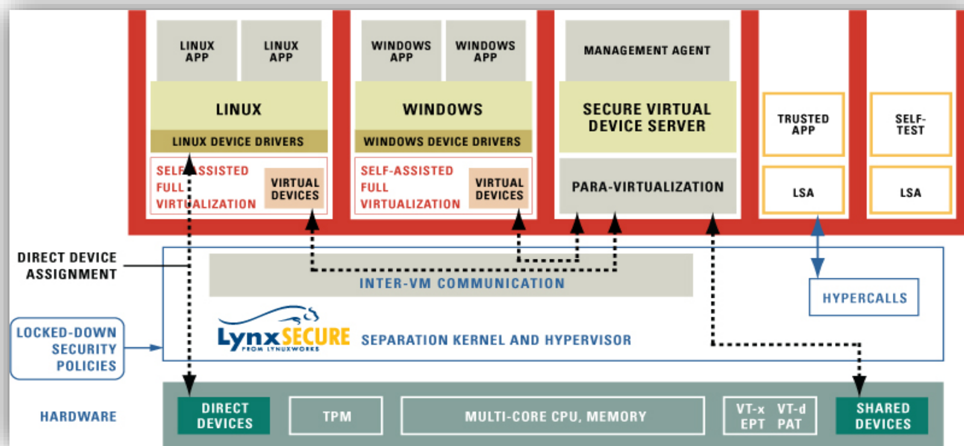

Figure 3: LynxSecure™ Separation Kernel Architecture

7. Conclusions

This paper provides an overview of the challenges involved in assessing and thwarting side channel attacks.
Hardware vulnerabilities and the complexity of modern multiprocessors have created a significant challenge for the system designer to assess emergent behavior that may lead to side channels.

The design choices of many operating systems and commercial hypervisors have exacerbated the problem of side channels due to inadequate considerations for minimizing side channel attacks.
The separation kernel technology is a least privilege design software that offers superior design control and the ability to minimize complexity, while immunizing systems against side channel attacks. The optimal use of this technology offers the best option to create a strong security posture that greatly minimizes and mitigates emerging side channel attack threats.

8. References

[1] Zhenghong Wang, Ruby B Lee , Princeton University, “Covert and Side Channels due to Processor Architecture”, 2006, 22nd Annual Computer Security Applications Conference (ACSAC'06).

[2] John Rushby, “Design and Verification of Secure Systems”, ACM Operating Systems Review Vol 15 No, 5, pp 12-21, December 1981.

[3] Moritz Lipp, Michael Schwarz, Daniel Gruss, et al, “Meltdown: Reading Kernel Memory from User Space”, USENIX security symposium 2018.

[4] Paul Kocher, Jann Horn, Anders Fogh, et al, “Spectre Attacks: Exploiting Speculative Execution”, 40th IEEE Symposium on Security and Privacy (S&P'19).

[5] Will Keegan, “Design Prevails: Protecting Critical Systems from Meltdown & Spectre”, Lynx Software Technologies Inc White Paper.

[6] Yinqian Zhang, et al, “Cross-VM Side Channels and Their Use to Extract Private Keys”, CCS’12, October 16–18, 2012, Raleigh, North Carolina, USA

[7] Jidong Xiao, et al, “Security Implications of Memory Deduplication in a Virtualized Environment”, The College of William and Mary, Williamsburg, Virginia, USA

Author

Arun Subbarao is Vice President of Engineering & Technology at Lynx Software Technologies, responsible for the development of products for the Automotive, IoT and Cyber-security markets. He has over 20 years of experience in the software industry working on security, safety, virtualization, operating systems and networking technologies. In this role, he spearheaded the development of the LynxSecure separation kernel & hypervisor product as well as other software innovations & patent filings in cyber-security.

Contact

Email: asubbarao@lynx.com
https://www.linkedin.com/in/arunsubbarao/