Improving system security using Processing-in-Memory architecture

Matthew Brown

A thesis submitted in partial completion of the degree of
Bachelor of Advanced Computing (Research & Development)(Honours)
at The Australian National University

October 2019

Except where otherwise indicated, this thesis is my own original work.

Matthew Brown
1 October 2019

To my family and friends.

Acknowledgments

I would like to thank Steve Blackburn and Adrian Herrera, for helping me stay on track throughout this thesis. Your feedback and direction has been invaluable. I appreciate all the time you have taken to share your knowledge and constructive feedback over the last year.

Abstract

Ensuring the security and privacy of computing systems and their users has become a significant concern. Consequently, efforts have been made to improve security on many different levels. These efforts include (but are not limited to) everything from the use of compile-time sanitizers to hardware-based Trusted Computing Modules, all of which are common on most modern computing systems. While there has been a large improvement in security over the years, malicious actors continue to exploit vulnerabilities. Common memory bugs such as buffer overflows, use-after-frees, and uninitialised memory remain the most common entry point for malware today.

At the same time that changes to security have occurred, the development of memory technology has also progressed. Specifically, the emergence of 3D-stacked memory, which provides significant performance improvements but requires the use of an integrated logic layer to operate. This logic layer is separately populated, providing space that allows us to add other hardware components, including general purpose processors, that exploit the locality and high bandwidth benefits of being embedded in DRAM. This technique is known as *Processing-in-Memory* (PIM).

My thesis is that PIM provides the necessary processing profile to accelerate various security applications and escape the Von Neumann bottleneck inherent in most CPUs. This thesis will examine the benefits and limitations provided by PIM, characterising security-relevant workloads that benefit from being offloaded to PIM. We will then look at how different security domains can take advantage of this technology to improve the overall security of a system. Specifically, I will provide an in depth analysis of security monitoring, examining how PIM can guarantee the integrity of core parts in an OS kernel.

I show that the PIM security monitor can capture *all* malicious changes to static regions of the OS kernel and prevent them from being nefariously modified, with minimal performance overheads. I also quantify the limiting factors of PIM with respect to the CPU cache, and demonstrate the issues that PIM has with fine-grained monitoring.

This thesis argues that PIM provides the necessary capabilities to improve system security, and should be considered a serious addition to the security toolbox of modern computing systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis explores the efficacy of using of a Processing-in-Memory (PIM) architecture to develop new security applications. My thesis is that the use of 3D stacked DRAM to implement PIM hardware architectures can be used to effectively implement existing and new security applications, to improve security and mitigate some existing attack vectors on modern computer systems.

1.1 Problem Statement

Security is a major issue for all modern computer systems. While modern computer architectures contain many layers of security and protections, they can never fully mitigate *all* attack vectors. The ubiquitous adoption of computer systems for personal, commercial, and in critical infrastructure make their security a critical aspect in our daily lives. The consequences of a security breach can have significant financial and political costs.

While the importance of having computer security is undeniable, the methods of implementing security form a wide range of techniques and solutions. Security measures can be implemented at nearly every layer of abstraction. From hardware-based memory encryption, kernel level file privileges, to application-level solutions such as antivirus software.

A problem with application and kernel-level security controls is that they are vulnerable to attacks which can modify or subvert them. For instance, a privilege escalation attack can be used to gain root access privileges. Rootkits are another common method of gaining unauthorised access to a system. Rootkit attacks are highly valuable as they are almost impossible to detect once they have occurred. Once a rootkit has been inserted into a system, the malicious user can trivially subvert or remove any other application-level security controls.

Due to these issues there has been significant research into implementing hardware-based security, from the introduction of dedicated cryptographically secure hardware like the *Trusted Platform Module* (TPM), to the development of external hardware devices to monitor malware. While hardware is generally more expensive than software, in both cost and development time, hardware can generally provide much higher assurance that the system is running correctly.

With the recent advent of 3D-stacked DRAM it is now viable to add custom hardware components to DRAM. This opens up the potential to develop security applications which are executed in memory. Previous research has identified an application for PIM as an off-chip accelerator [Ahn et al., 2016; Chi et al., 2016; Seshadri et al., 2017]. At the time of writing no research has approached the use of PIM to provide additional security features to the system. This thesis thus explores the efficacy and limitations of implementing security applications on the PIM architecture.

1.2 Scope and Contributions

I argue that the PIM architecture can be used to implement valuable security controls for the entire computer system. Modern computer systems implement many of their security controls in software. It has been shown that these security implementations alone do not provide adequate protection for sophisticated attacks such as unauthorised privilege escalation [Davi et al., 2010; Seaborn and Dullien, 2015; Pangaria et al., 2012]. The implementation of security in external hardware mitigates these risks but provides a new set of limitations. In this thesis I explore and quantify some of the key limitations of PIM, and discuss their effect on its use for different security-relevant workloads.

After understanding the limitations of PIM, I analyse a range of security domains and identify key applications which have ideal characteristics for the processing profile of PIM.

I then present a case study on three external hardware security applications. For each of these, I analyse their benefits and limitations, and discuss the ramifications of moving these to the PIM architecture.

Having identified kernel integrity monitors as an application which both fits the PIM workload profile and provides valuable security controls, I present a basic design for a kernel integrity monitor on PIM, analysing the design, its constraints, and feasibility.

1.3 Thesis Outline

Chapter 2 will explore the background of PIM and some of the existing literature relating to off-chip security applications. Chapter 3 will look deeper into the characteristics of PIM and its limitations. Using this we will identify some applications which suit the PIM architecture. Chapter 4 focuses on previous work on external hardware security solutions. I will analyse and discuss the benefits and limitations of each application. It will also investigate how these applications could be applied to PIM, and the consequences of such a change. Chapter 5 shows the use of PIM as a kernel integrity monitor, and investigates performance against existing kernel monitors. Chapter 6 will explain the future work and conclusions of this dissertation.

Chapter 2

Background and Related Work

This chapter looks at the background and existing work of PIM and hardware-based security devices. For each topic we will explore its history, existing work, and the current state-of-the-art.

2.1 Processing In Memory

The last decades have seen an explosion in the performance of CPU architectures. However, the rate of improvement has been much slower for DRAM technology [Carvalho, 2002]. While the rate of CPU improvement has slowed significantly, there still lies a large gap in performance between the CPU and memory. In the traditional Von Neumann computational model the CPU and memory are kept separate [Von Neumann, 1993]. To operate on any data it must be moved from memory, stored in DRAM, into the CPU. This movement of data is expensive both in terms of time and energy. As a consequence of this design model the CPU is forced to wait for data to traverse the slow memory bus. This is known as the Von Neumann bottleneck. Due to the divergent scaling of CPU and DRAM performance accessing memory has become an increasing bottleneck in modern architectures.

One solution to this problem is to reduce the bottleneck by locating the computation near or in memory. This concept was conceived in 1970 [Stone, 1970] but has not been feasible due to the technological limitations of adding processing logic into DRAM. However, in recent years the development of 3D stacked DRAM has reopened the possibilities for Processing-in-Memory (PIM).

3D stacked DRAM is made up from layers of memory cells [Hybrid Memory Cube Consortium, 2014]. These layers are connected vertically with the use of *through-silicon-vias* (TSVs) which allow for very high bandwidth communication. The bottom layer of the stack is a special *logic layer* and acts as a memory controller, providing the necessary logic to access the correct parts of the stack. This logic layer is sparsely used, thus gives us space to develop additional logic and hardware components which can do computation on the DRAM stack, allowing for the development of in-PIM devices. This architecture is outlined in Figure 2.1. The 3D stack of memory is broken into separate *vaults*, each with its own vault controller. Each vault can serve memory requests independently of the others, improving bandwidth and providing the potential for highly parallel computation. PIM as a high level concept does not have a specific implementation, but for the purposes of this dissertation we will define a PIM architecture to be the use 3D stacked DRAM with additional computational hardware components.

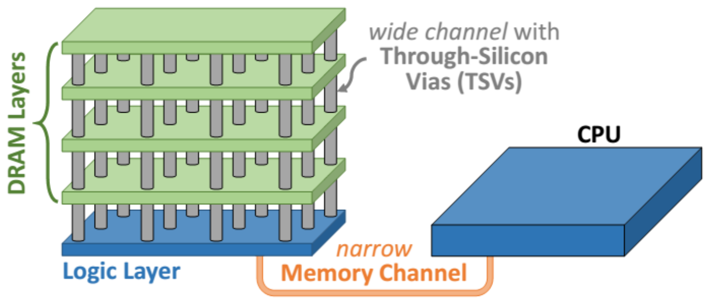

Figure 2.1: Overview of the 3D-stacked DRAM architecture [Ghose et al., 2018]

The major benefits of PIM are: its very high internal memory bandwidth; its capability for highly parallel processing; and its reduced movement of data. This makes PIM ideal for workloads that require high memory usage, without the need for significant computation, as there is limited power and silicon budget for computation within PIM. It is also interesting to note the while PIM offers massively increased internal *bandwidth*, the *latency* of memory requests is not similarly improved. This is due to the fundamental physical limitations of DRAM technology [Chang, 2017].

While this technology offers some significant improvements over conventional DRAM technologies, there are some inherent challenges associated with PIM. The fact that the PIM is separated from the CPU by the memory bus means that is cannot cheaply access any of the on-chip functionality that the CPU provides, such as memory management. The lack of access to the memory management system means that PIM cannot perform address translation, page table walks, or access the TLB. This is a significant challenge in developing a PIM device, as the translation between virtual and physical memory addresses is a critical function for most applications. The inability to access the CPU cache also provides numerous challenges for PIM. For example, the issue of coherency of data between the CPU and PIM is complicated by the possibility that the PIM may change memory state, making coherency a two-way street. This is an problem when the CPU and PIM have a shared resource, as any changes made by the CPU or PIM must be propagated and reflected in the other. To become a viable technology, solutions must be developed that address these challenges. Each of these limitations will be discussed further in Chapter 3

2.1.1 Previous work

There is a depth of knowledge on near-memory and off-chip processing. For the purposes of this dissertation we will focus on recent research into PIM architectures.

While the broad concept of PIM is not new, the work on PIM in the practical context of 3D-stacked DRAM is still in its infancy but has shown rapid progress. As the research in this field is in its early stages there are no standardised methods for how a PIM device should operate. This gives researchers and hardware architects freedom to explore many different architectures. These architectures tend to be custom made for a single purpose with specific hardware components.

Recently, there has been a wide range of research on PIM architectures for various purposes. A critical area of research is investigating methods to solve the limitations inherent in PIM. This has seen the development of LazyPIM, a cache coherency mechanism for PIM [Boroumand et al., 2017]. Hsieh et al. [2016] have also presented an implementation of an in-memory pointer chasing accelerator (IMPICA) with PIM. This work solves some of the key challenges with off-chip address translation, but incurs a significant overhead to PIM execution.

Another key issue with utilising PIM for computation is the current lack of standardisation. For PIM to be commercially viable there needs to be an ecosystem of tools including programming languages, development environments, supported operating systems, and architectures [Zhang et al., 2013]. By having no standard mechanism for communication between CPU and PIM each application must develop its own method, incurring significant development time and cost. To mitigate these costs there has been some research into developing a new layer of abstraction to communicate with PIM, through the use of PIM-enabled instructions [Ahn et al., 2015]. This abstraction allows for applications to offload workloads onto PIM without detracting from the traditional programming framework. However, these solutions require modifications to both PIM and the CPU. For initial large-scale adoption of PIM it is not feasible to require significant modification to CPU architectures, due to high costs of developing and fabricating such specialised mechanisms.

Other PIM architectures have focused on developing applications which take advantage of PIMs benefits. Such research has used PIM as an accelerator for graph processing [Ahn et al., 2016], neural network computation [Chi et al., 2016], and bulk bitwise operations [Seshadri et al., 2017]. These are areas which are traditionally limited by low memory bandwidth and number of processing cores.

Many modern applications working with large workloads need to execute bulk operations against a large set of data. The development of AMBIT as a bulk bitwise operation accelerator has demonstrated the significant advantages that PIM provides [Seshadri et al., 2017]. By performing the bitwise operations in-memory it achieves a 35-fold increase over state-of-the-art CPU systems. This technique has applications in many different fields which require the usage of bitwise operations, such as database design, DNA sequencing, and encryption algorithms.

The work of Ahn et al. [2016] shows the use of PIM for graph processing applications. This work used the HMC-based memory architecture to implement a graph-processing accelerator. The results showed that this PIM solution performed an order of magnitude better than the existing state-of-the-art while reducing energy usage by 87%.

As outlined above, even in its early stages of development, PIM has shown huge potential as a high memory bandwidth off-chip accelerator. The variety of PIM applications shows its versatility among many different disciplines such as machine-learning and HPC. Yet there has been an absence of development for the use of PIM in security. As PIM is a hardware device external to the CPU, it provides some interesting properties for its use for security. In the next section we will discuss the background, history and current literature around external hardware-based security devices.

2.2 Hardware-based Security

Security is a particularly broad field of research, and has implications in the design of every aspect and abstraction layer in a computer, from application-level security like antivirus scanning, to OS level security such as file privileges, down to hardware-based security. Hardware security solutions provide some additional advantages which cannot be replicated in software. As PIM is still a new and emerging technology, to the best of our knowledge there has been an absence of research into the use of PIM architectures as a security device; because it is a new and emerging technology. However, there have been numerous hardware-based security devices for computer systems. This section will provide a brief overview on the current research and technologies for hardware-based security, focusing on the use of external hardware solutions.

The range of hardware security solutions can be categorised as either internal or external to the CPU. Internal security hardware include solutions such as Intels SGX which applies hardware-based isolation and encryption to an application, so that its execution can be trusted [Costan and Devadas, 2016]. Since developing internal hardware on a CPU is not feasible for most researchers, the development of external hardware-based security has emerged. The primary advantage over software based security measures is that the hardware can run independent of the CPU, meaning that no level of system privilege can modify, disable, or remove the security system from running. Other benefits include the usage of hardware-based cryptography and encryption mechanisms to protect the confidentiality of the data.

Many external hardware-based security devices aim to improve security by capturing data from the system, analysing the data, and detecting any invalid, or illegal operations made by the system. This type of application describes a whole class of security programs called *monitors*. Often these monitors are looking for invalid changes to the OS data structures, which would indicate a breach of security. The implementation of a monitor has followed two key approaches, through the use of *memory snapshots* and *bus snooping*.

Snapshot-based Monitors A memory snapshot approach takes snapshots of memory at regular time intervals, analysing the data for its correctness. One of the first implementations of a security monitor, called Copilot, used this approach and implemented a kernel run time integrity monitor [Petroni Jr et al., 2004]. This method is simple to implement but can fail to catch illegal changes due to *transient attacks*. These are attacks which occur between the snapshots of the monitor, as illustrated in Figure 2.2.

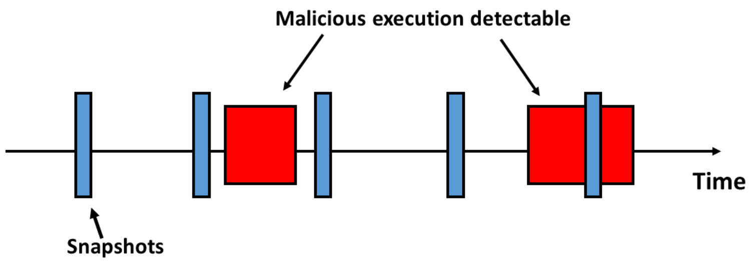

Figure 2.2: Transient attacks can occur between the snapshots

Bus Snooping-based Monitors To address the issue of transient attacks for security monitors some recent research has adopted the use of bus snooping [Moon et al., 2012; Lee et al., 2013]. This uses a more sophisticated approach by listening, or snooping, the memory bus between the CPU and memory, to analyse the data traffic in real-time and detect any invalid memory requests. Bus snooping monitors will be discussed further in Chapter 4, where we will evaluate some current implementations and the efficacy of developing such a monitor with PIM.

2.3 Summary

This chapter has provided a background of both PIM and hardware-based security, outlining some of the history and current literature in their respective areas. We have also outlined some of the security issues which can be solved by the use of hardware-based security. While previous work has shown that PIM brings some significant benefits to off-chip processing workloads there are some inherent limitations of its design. The next chapter will focus on identifying and evaluating these limitations and discussing their effect on the uses for PIM.

Chapter 3

Understanding the Limitations of PIM

The previous chapter gave a high-level overview of PIM, and the current landscape of the field. While we have discussed many of the benefits of PIM, there are some inherent limitations presented by its architecture. This chapter will analyse these limitations through experimentation, discussing the results and their implication for developing PIM applications, as well as potential solutions. As outlined in Chapter 2, the primary benefit of PIM is its potential for extremely high-bandwidth memory access. This means that PIM works best for offloading memory-intensive applications that require a relatively small amount of computation. Ideally, such applications should be designed to run in parallel among the many PIM cores. The main limitations to using PIM are the lack of access to on-chip mechanisms such as address translation, and the CPU cache. We will now explore these limitations further.

3.1 On-chip Mechanisms

Modern CPU architectures provide many different on-chip mechanisms to improve overall performance. For example, the Advanced Encryption Standard (AES) is often built into the CPU to improve security and reduce the performance overheads of encrypting/decrypting data [Gueron, 2010]. A limitation of being isolated from the CPU is the lack of low-latency access to these mechanisms. Perhaps the most important mechanism with respect to PIM is the memory management subsystem. This means that PIM does not have fast access to any high-level address translation and protection mechanisms such as the page table or Table Lookaside Buffer (TLB). While the PIM could probe the CPU with address translation requests, any performance benefit provided by PIM could easily be negated due to the long latency times associated with such probes.

As a consequence, the only information received by the PIM is the CPU's physical memory requests, with no efficient mechanism to access further information. By only seeing physical addresses the PIM loses information about how the CPU is executing a program, as program data will typically be discontiguous in physical memory. This is a problem for any system with layers of abstraction, as higher level abstractions must be translated into concrete low level constructs. In this translation we can lose critical information and context. This problem is known as the *semantic gap* [Dorai and Venkatesh, 2003]. The semantic gap means that without additional context it is difficult to generate high level understanding of the CPU and its behaviour from low level information.

Without fast access to the on-chip mechanisms the use cases and applications for PIM are reduced. To solve these problems, the PIM must either implement its own mechanisms, or develop applications that are not affected by this limitation. Duplicating the CPU's Memory Management Unit (MMU) in PIM would not be feasible for a number of reasons. First among them is the need for additional coherency mechanisms between the CPU and PIM MMU, which is a nontrivial task and adds significant complexity to the system. It would also undermine the PIM's advantages by increasing bus traffic to maintain coherency.

As discussed in Chapter 2, there has been some research into solving this issue. The work of Hsieh et al. [2016] demonstrated the use of PIM for an In-Memory Pointer Chasing Accelerator (IMPICA), which implements its own address engine to perform address translation *outside* of the CPU. This works by allocating linked data structures in contiguous regions of virtual memory, and using custom region-based page tables optimised for pointer chasing. In this way, given the first pointer it can determine the physical addresses of all further linked structures. This solution performs well for the specific task of pointer chasing. However, it does not scale beyond this use case and does not solve the issue of address translation for all memory. By allocating all data in the manner presented in IMPICA, we would remove all of the benefits of the virtual address space, regressing to a physical memory model.

The address translation problem is still an open area of research for PIM architectures. Current solutions may work for a specific subset of workloads and applications, but do not solve the fundamental problem.

3.2 CPU Cache

The CPU cache provides another key challenge in developing PIM applications. The cache is a data store which offers access speeds orders of magnitude greater than DRAM. The trade off is that caches are limited in capacity due to die size and cost constraints. The cache is used to store frequently used pieces of data and instructions to increase the performance of the system. To get data back to memory the cache must implement a write policy, commonly either write-back, or write-through. A write-back policy is most common for CPU caches, writing first to the cache, and only writing back to memory when the data is evicted from the cache. This improves performance and reduces bus traffic, but also introduces the time delay between changing a value and observing the change in memory. The write-through policy will synchronously write to both the cache and memory. This is a simple policy that mitigates the complexity of a write-back policy at the cost of performance.

The primary challenge is maintaining and propagating the changes made to data that is in both the CPU cache and PIM. This is a key concept called *cache coherency*. The lack of native cache coherency mechanisms between the main CPU and PIM is one of the most significant disadvantages of PIM. In the context of running PIM workloads, the CPU cache becomes a problem when the main CPU and PIM are reading and writing to the same memory location. For some applications this may not be an issue because the CPU offloads the execution to PIM to work independently. However, security applications such as integrity monitors may be accessing and analysing data concurrently with the main CPUs execution. Previous work has shown an implementation for a cache-coherent PIM, but it still requires significant overheads for some applications [Boroumand et al., 2017].

The following work investigate the issues that the CPU cache presents when PIM does not have a cache coherency mechanism. The caching mechanisms of the CPU leads to three core issues for the correct execution of PIM: the loss of information; temporal lag of information; and reduced granularity of information.

These problems undermine the efficacy of security applications such as security monitors. These are often used to mitigate memory violations like buffer overflows, use-after-free, and other common exploits. Each cache issue will be discussed in depth in the following sections in the context of developing a PIM security monitor.

3.2.1 Loss of Information

CPU caches typically employ a *least-recently-used* (LRU) eviction policy for the last level (LL) cache, thus, in principle data which is used often by the CPU could stay in the cache indefinitely. This data can be written and modified by the CPU an arbitrary number of times. When a byte in the cache is overwritten, all knowledge of the previous state is lost. While the data is in the CPU cache, the PIM has no visibility to the overwritten values, meaning that the information is lost. For a PIM security monitor this results in the loss of potentially critical information about security breaches.

As part of my research I developed a tool to quantify the scope of this issue. I added functionality to the *Cachegrind* [Valgrind Developers, 2009] cache simulator which is part of the Valgrind [Nethercote and Seward, 2003] suite of tools. This allowed me to measure the number of overwrites that occur to the same piece of data during the execution of a program. This was tested against the SPEC CPU 2006 benchmarks, which provides a range of compute and memory intensive applications with varying cache behaviours [Spradling, 2007]. We chose this suite of benchmarks in the absence of standardised security-focused workloads.

Figure 3.1 shows the percentage of bytes that are overwritten during the execution of each benchmark. These values were computed by finding the total number of bytes that were overwritten in the cache against the total number of modified bytes which were flushed from cache for each benchmark. From these results we can observe a wide variation between the different benchmarks, with the percentage of bytes overwritten ranging between 0.02 and 204 percent of the bytes written to the cache. The highest three results are all over 100%, suggesting that a significant proportion of modified bytes were overwritten more than once. For other applications we see a very small amount of information loss, with 7 out of 15 benchmarks showing a loss of less than 10%. Table 3.1 shows the median and average loss of information, with 15% and 47% respectively. Because the average value is heavily skewed by the top three results, we take median value as a more representative figure.

This data does not capture the distribution of these overwrites. For instance, an application that overwrites only one byte many times would achieve a similar percentage score to an application which overwrites many bytes only once. However, this does not change the validity of this analysis as this data is still lost and unreachable for PIM.

One factor that could influence these results is the total execution time of the program, and the overall number of bytes written to the cache. Figure 3.2 shows the total time and number of bytes written to cache for each benchmark. The *time* in this context increments on every read and write to cache. The only result that would suggest a correlation is the *464.h264href* benchmark. While this has a particularly high total time, it has a proportionally small number of writes to the cache, meaning that the longer execution time is not performing a significantly more number of writes. We can conclude that there is no significant correlation between the loss of information and program execution time.

From this data we can conclude that the amount of lost information is highly dependant on the specific application behaviour. The lack of cache coherence between the CPU and the PIM leads to lost information, which is uncontrollable by PIM. The loss of information from the cache is a significant problem for the PIM, and the development of PIM applications. While for some experiments there was a minimal amount lost information, the scope of this loss is not observable to the PIM. For some applications, such as security monitors, the loss of any information could lead to a violation of security being missed. The use of a write-through cache policy would mitigate this issue, but would cause a significant performance reduction and increased bus traffic, making it an unreasonable solution. For these reasons, the loss of information caused by the CPU cache appears to be a significant limitation for PIM architectures.

3.2.2 Temporal Lag of Information

The temporal lag of information propagating from the CPU down to the PIM is the most intuitive limitation for PIM architectures. This is caused by the use of a write-back cache policy. The CPU cache will store a local copy of data until it is evicted. A commonly used cache eviction heuristic is the *least-recently-used* method. This means that only the data that is least used is written back to memory. The problem is that frequently used data can stay in the cache until the program execution finishes, which for some programs such as OS kernels could be the lifetime of the system.

To quantify the significance of this issue I added functionality to the *Cachegrind* cache simulator to track the amount of time each *cache line* is held in the cache for. Similar to the previous experiments, we use the SPEC CPU 2006 benchmarks to analyse the lifespan of cache lines for various computationally and memory intensive

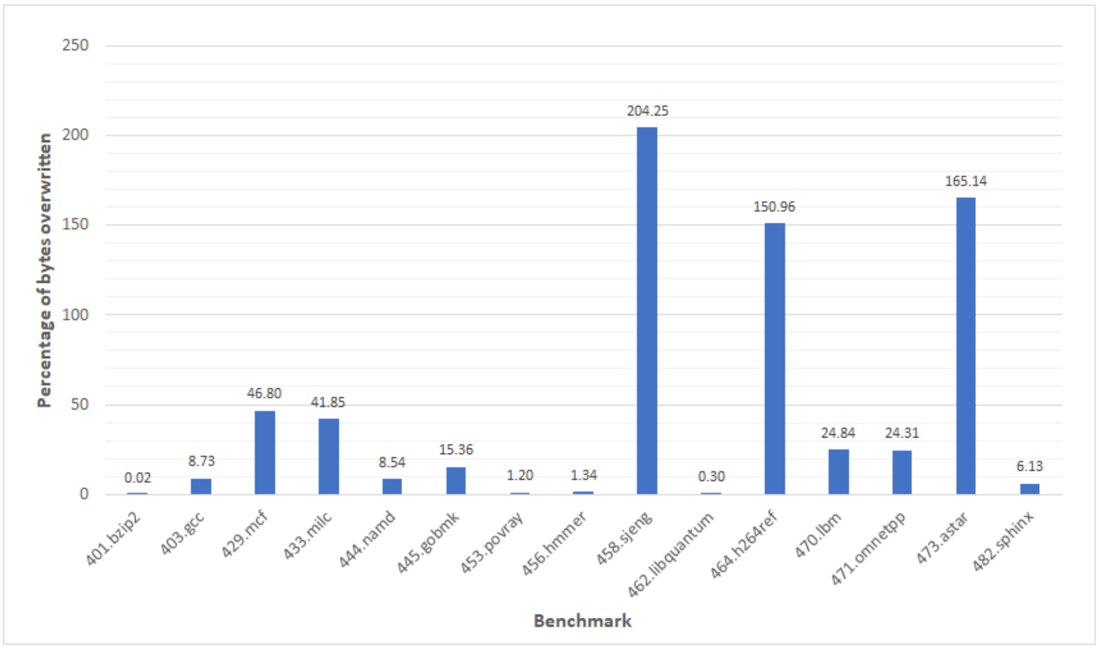

Figure 3.1: Proportion of bytes lost to overwrites in the CPU cache is highly application specific.

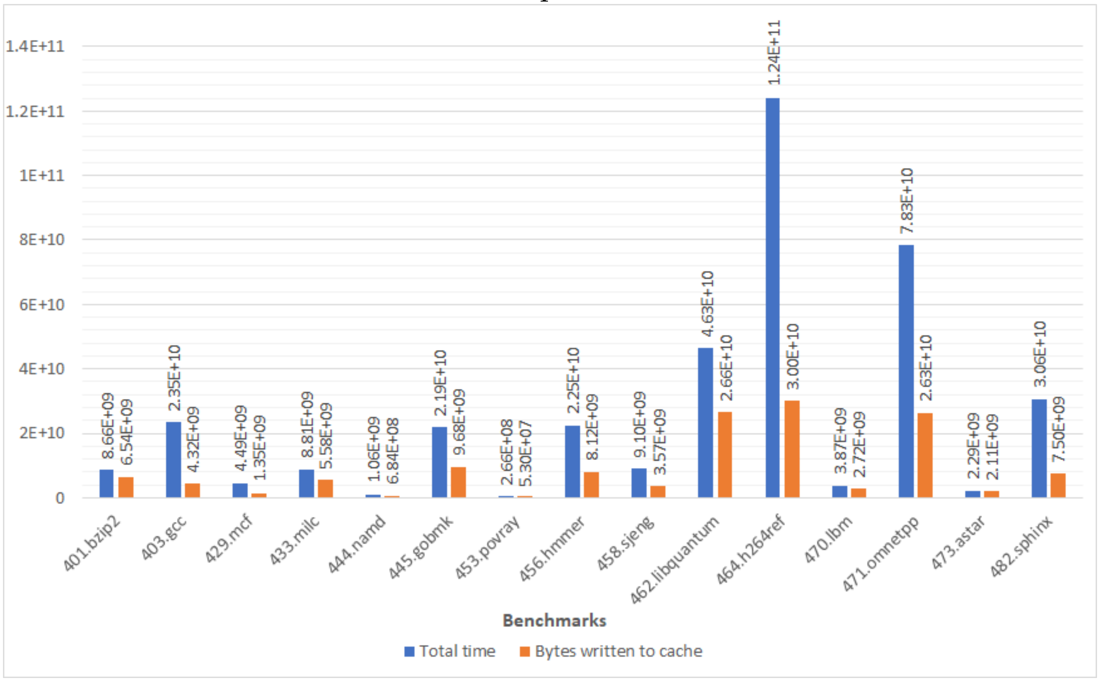

Figure 3.2: The benchmarks show a wide variation in total cache events and bytes written.

workloads. The data was collected by measuring the time taken for each cache line to be evicted. Due to the large number of cache lines, the data was collected as a histogram, bucketing the data on a $\log_2$ scale. Figure 3.3 shows a high level view of the normalised cumulative frequency of the lifespan of cache lines. It is important to note that this data is plotted using a logarithmic scale. As such, each step in the lifespan axis increases by two. The consequence of using a logarithmic scale is the reduction of fine-grained analysis, as the window for each cache line lifespan increases exponentially, but allows us to observe the high level lifespans and distribution of cache lines.

Using this data we can extract some interesting information about how the lifespan of cache lines is distributed. From Figure 3.3 we observe that most benchmarks follow a similar pattern. A few of these experiments show a large step in the percentage of cache lines. This would indicate a large quantity of data being held in cache for a similarly long amount of time. From this data we can compute that on average, the longest held cache lines were in cache for 111 (or $2^{6.8}$) times the median cache lines lifespan.

To gain a better understanding of these results, we can plot these experiments on a linear scale. Figure 3.4 shows a representative subset of experiments plotted on a linear scale. It is clear from this that the majority of cache lines have a relatively short lifespan. We can also observe that the last 20% of cache lines survive in cache for a significantly longer period of time, as shown by the long tails. This is to be expected, as programs will often only need to read or use a cache line once. The longer living cache lines are those which the CPU uses often. This could indicate a commonly used constant, or updated working variable that could stay in cache for the majority of the runtime.

The results show that the majority of cache lines used are stored in cache for a relatively short proportion of the total execution time. While most cache lines were evicted quickly, the few that remained stayed significantly longer. There is a chance that an area of memory PIM requires is being held indefinitely by the CPU, causing the PIM to miss security violations due to the use of stale data. For some application this is a significant limitation. Using the example of a PIM security monitor; the CPU could violate a security control, but PIM might not find it until the end of the programs execution. Taking this further, the CPU could intentionally keep the data in cache to stop it from being evicted. This is also coupled to the previous issue of information loss, both of which are consequences of using a write-back cache policy. This is because the longer the data sits in memory, the more likely it is to be modified, incurring a loss of information for PIM. The use of a write-through policy would similarly mitigate this issue, but would not be a reasonable solution due to the high reduction in performance. For time-critical security-related applications, the issue of temporal lag may be a significant limitation.

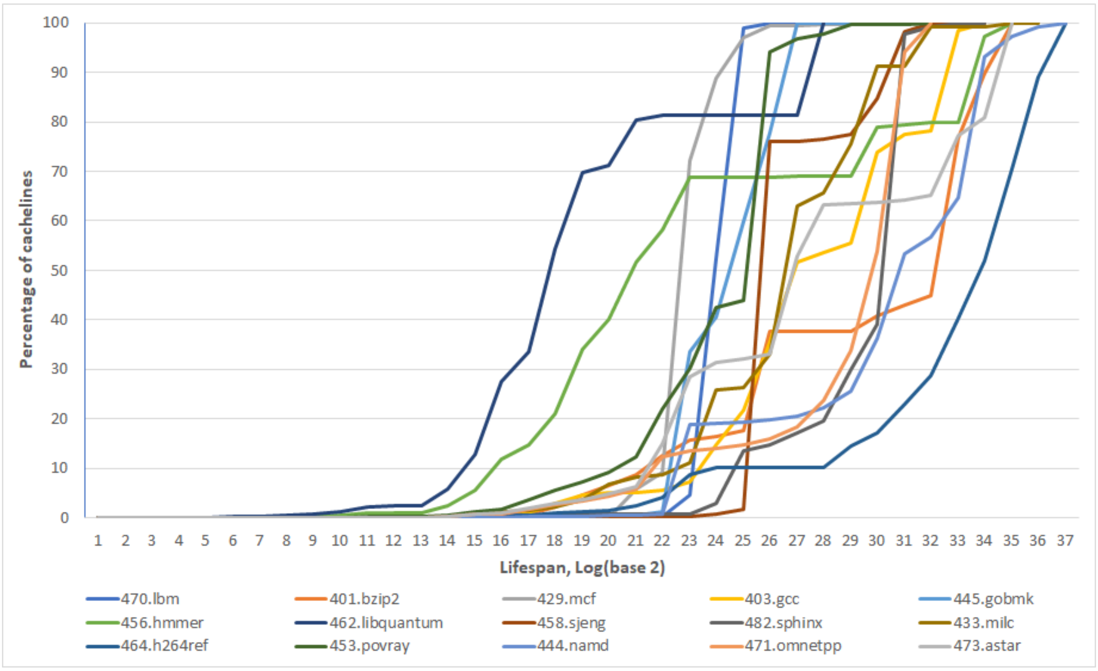

Figure 3.3: Normalised cumulative frequency of cache line lifespans.

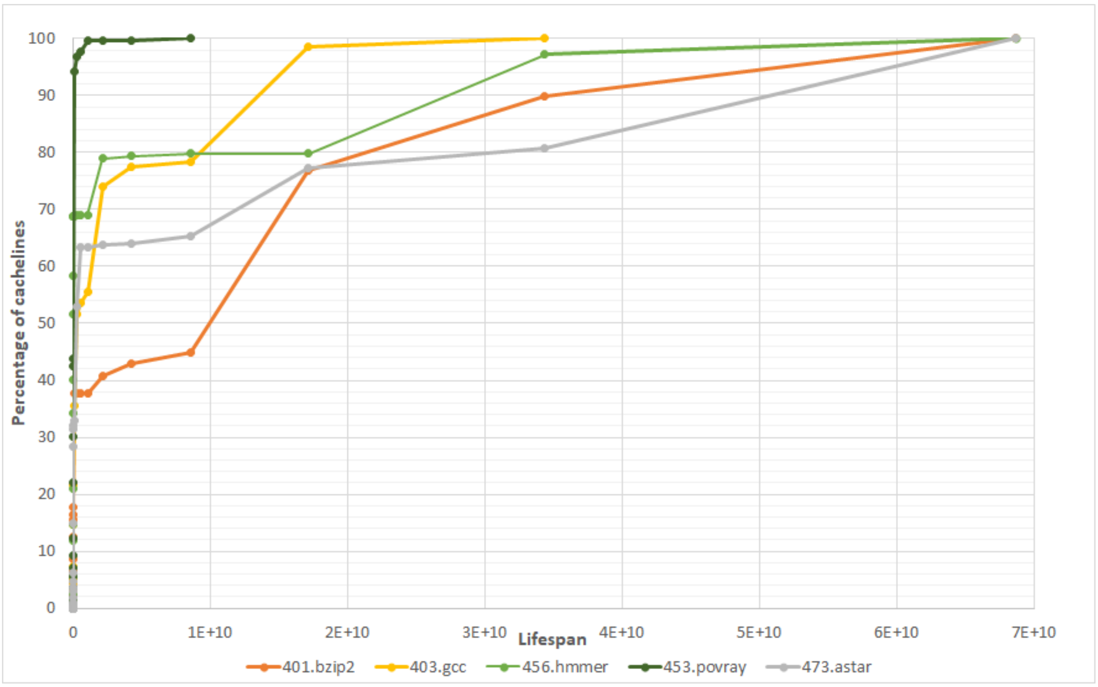

Figure 3.4: A linear scale shows that most cache lines have a short lifespan.

3.2.3 Reduced Granularity of Information

The final issue is more nuanced than the others described previously. First, we need to understand how the CPU interacts with the cache. When the CPU requests a byte of data from memory, the memory controller loads in a whole cache line. The size of the cache line is dependent on the Last Level (LL) cache line size. The cache line is the minimum size that the cache can read in, or write out to memory. Once the data is in the cache, the CPU can access and modify each byte. When the data is no longer required, the cache will evict the entire cache line back to memory.

The reduced granularity of information comes from the fact that the cache is working at a lower resolution: at the cache line level instead of the byte level. This means that the PIM can only observe changes to the entire cache line when it gets evicted from the cache. The only information we have about modifications to the cache line is from the *dirty bit*, which tells us if any data in the cache line has been modified. Any further information about which bytes have been modified cannot be captured without additional hardware. It is feasible for PIM to implement a mechanism to check the incoming cache line with the existing data in memory to identify the modified values. However, even then, the full extent of the changes cannot be captured due to the loss of information problem which we previously outlined, and could reduce performance due to latency. In this case, a byte could be modified from its initial value, used for some amount of execution, and then changed back to its initial form. The PIM would not be able to catch these edge cases. For applications such as a security integrity monitor this issue could mean that malicious code may be able to modify and break critical parts of the program of system without detection.

3.3 Offloading to PIM

Another issue comes with offloading a process from the CPU to PIM. Early in this research we investigated the potential for offloading security application like *Address-Sanitizer* (ASan) to PIM. ASan is compile-time and run-time tool which instruments code with additional checks and ensures that the code doesn't violate any memory bugs; such as buffer overflows, dangling pointers etc.. This is a costly operation which slows performance, and as such is not used widely in production environments. Offloading the ASan operations to PIM would provide significant value to developers. However, when investigating ASan's behaviour we find that it adds a small number of sanity checks for each allocation and write to data. By offloading this to PIM it would generate a large number of small workloads. Any potential gain from PIM would be lost due to the huge overheads associated with transferring data between the CPU and PIM, as well as the synchronisation required.

This issue is separate from the caching and on-chip mechanisms, as some applications are not well suited to be offloaded to PIM. Offloading a process to PIM can perform particularly well if it can run independently from any other process. The addition of dependencies and coupling it to the CPU runtime creates a far more complex model for using PIM and can reduce the benefits of PIM. Doing small batch workloads on PIM adds additional traffic onto the memory bus and can reduce performance. Ideally the workloads are larger and have high memory usage and can be executed independently of the main CPU.

Work by Ahn et al. [2015] has presented the concept of *PIM Enabled Instructions* (PEIs), which are special instructions constructed for operations that are optimised for PIM. This allows any workload compiled with these instructions to utilise PIM. The runtime also calculates whether offloading the operation will be faster than keeping it in the CPU due to the overheads. This solution provides a good base for utilising PIM. However, it does require the addition of multiple mechanisms to the CPU, which is not feasible in the short term for commercial grade CPU architectures outside of simulations.

3.4 PIM Isolation

The concept of isolation between the CPU and PIM can either be beneficial or a significant limitation, depending on the application. For many PIM applications, the isolation from the CPU causes many problems, largely due to the lack of exposure to on-chip mechanisms and caches. Communication between the CPU and the PIM could allow for more complex tasks to be offloaded to and executed by PIM. This is useful for many applications which require input from the CPU. However, the ability to modify the execution of the PIM from the CPU presents some serious security concerns. Specifically, that malicious agents could interact with the PIM to modify or subvert its execution. For security-related applications this provides a critical attack vector which could eliminate any additional security features provided by PIM. This is particularly important in protecting against privileged attacks at the kernel level, as we can no longer trust the host OS to operate correctly. Removing the ability to modify and change the PIM's firmware and execution would mitigate this risk, and further isolate the two as independent devices. This does limit the efficacy of some applications, as this would remove the ability to offload arbitrary workloads.

Many PIM applications do this already, as the firmware is baked into the hardware, and cannot be updated or changed by the CPU. This makes such PIM devices an Application Specific Integrated Circuit (ASIC), which detracts from the versatility and value of these devices. To compensate for this, the PIM chips could be designed to be updated manually by reprogramming the chip on the physical device. Of course, the security of this approach depends on the assumption that malicious actors do not have physical access to the hardware.

Currently, many security solutions running on the CPU, such as traditional security monitors, can be subverted by malware to avoid detection. The isolation of the PIM guarantees that no malware can modify its behaviour, meaning that we can trust the execution of the PIM. While the execution cannot be changed, any inputs from the CPU must be trusted to be correct. For instance, if the PIM has implemented an antivirus scanner and the CPU can update the signature list; a malicious actor could remove all signatures from the scanner, rendering it useless. To mitigate corruption from invalid CPU inputs we can either develop applications which do not require inputs; or program all relevant information into the PIM. In the previous example we could program the antivirus signature list into the PIM directly to remove the need for CPU inputs. Chapter 4 will further discuss this idea, and the problems which arise from this isolation, such as updating.

3.5 Hybrid System

So far we have focused on offloading entire workloads from the CPU to PIM. While this can provide significant performance benefits, there are also many unavoidable limitations. Primarily, as it restricts the set of workloads significantly: only those which match its strict processing profile would be viable. An interesting idea is the use of the PIM in conjunction with traditional CPU-based security applications. This would see a CPU and a PIM application working cooperatively, thus mitigating caching issues. In this way, the CPU program could monitor the CPU cache, with the PIM monitoring memory. The CPU could then send minimal relevant information to the PIM to validate. Implementing such a system would be challenging, as we must develop two separate but cooperating programs, but could resolve some of the inherent issues. This helps mitigate some of the fundamental limitations of PIM while maintaining a high degree of security. Unfortunately, this does not improve system performance. However, it could be used to guarantee that the system is running correctly. The CPU is still vulnerable to existing attacks and exploits, but as the PIM is isolated it could still maintain a base level of security. This is different to the PEIs discussed in Section 3.3 as these are two distinct programs which are working together. It would also require no further modification to the CPU architecture, unlike the PEI implementation.

3.6 Summary

In this chapter we have identified and discussed the limitations and constraints of using the PIM architecture in its current state. For many of these limitations there are no simple, efficient or scalable solutions. Therefore, it is important to identify and characterise the areas and applications which will not be compatible, or benefit from PIMs capabilities. Chapter 4 will explore potential security applications for PIM architectures with respect to these limitations, and discuss the efficacy of implementing existing security controls on PIM.

Benchmark	Total time (cache events)	Total bytes written to cache (bytes)	Total bytes modified and flushed from cache (bytes)	Total bytes overwritten in cache (bytes)	Percentage of bytes lost to overwrites, %
401.bzip2	8,658,630,335	6,543,864,970	22,942,008	5,376	0.02
403.gcc	23,477,770,185	4,321,028,809	10,433,271	910,553	8.73
429.mcf	4,488,991,933	1,350,333,749	21,467,282	10,047,155	46.8
433.milc	8,805,812,053	5,581,804,491	36,807,828	15,405,040	41.85
444.namd	1,061,372,079	684,229,735	8,986,726	767,636	8.54
445.gobmk	21,931,480,982	9,680,516,263	518,686,435	79,689,628	15.37
453.povray	265,918,057	53,039,888	514,188	6,176	1.2
456.hmmer	22,490,253,431	8,122,935,050	22,196,680	298,085	1.34
458.sjeng	9,104,134,874	3,565,323,523	9,738,861	19,891,721	204.25
462.libquantum	46,316,560,168	26,624,390,706	135,003,271	411,248	0.3
464.h264ref	124,129,492,128	30,014,687,349	202,998,306	306,448,842	150.96
470.lbm	3,865,266,709	2,717,811,080	13,287,730	3,301,074	24.84
471.omnetpp	78,276,730,453	26,342,921,128	164,920,237	40,088,114	24.31
473.astar	2,290,206,666	2,107,295,205	7,429,016	12,268,347	165.14
482.sphinx	30,605,902,655	7,499,752,854	32,442,300	1,988,950	6.13
Average	25,717,901,514	9,013,995,653	80,523,609	32,768,530	46.65
Median	9,104,134,874	5,581,804,491	22,196,680	3,301,074	15.36

Table 3.1: Cache event data from benchmarks with average and median values.

Chapter 4

PIM Security Applications

The previous chapter looked in depth at the benefits and limitations of the PIM architecture. This chapter will identify and discuss a range of security-related areas which are suited for PIM architectures. We will evaluate three applications which use hardware to implement additional security features. These applications implement security integrity monitoring systems using different technologies and mechanisms. I will evaluate all three applications in-depth, analysing the benefits and limitations of each. We will then explore the potential of moving these applications to PIM.

4.1 PIM Workload Profile

As discussed in Chapter 3, PIM architectures are suited to workloads which require high memory usage, with relatively low computation, that can be run in parallel to utilise all PIM cores. We have identified some of the limitations for PIM workloads, which include: high latency access to CPU on-chip mechanisms; no cache coherence mechanisms; and no cache visibility for applications dependent on CPU execution. Consequently, PIM applications which can run independently from the CPU are the best candidates for implementation.

4.2 Security Problems

There are many security applications which could employ PIM. However, many of these would not be feasible due to the limitations previously outlined. Many security solutions are implemented at the OS kernel level, such as file privileges, network security, namespaces, and cryptography. While additional security mechanisms could be implemented with a PIM at the kernel level it would limit the PIM's versatility. To encourage the adoption of a PIM, it should work without the addition of any special mechanisms on the CPU. As such, the focus of security problem space is on applications which work outside of the kernel. The following sections outline different security areas which could be addresses through the use of a PIM.

4.2.1 String Matching

String matching on large regions of data is a simple process which is constrained by the speed of reading memory into the CPU. In the security world, string matching is an important technique, often used for applications such as deep packet inspection and antivirus (AV) scanning. An AV scanner commonly uses a signature-based check. This searches all of the data in memory and on disk against a list of known virus signatures, stored in the AV database. The signature-based search is the most primitive form of AV checking, but still makes up for a significant part of modern AV systems. One of the limitations of this system is that it can only find viruses that are known in the database. Viruses are often purposefully obfuscated and manipulated in an attempt to avoid signature detection. More modern AV systems also use advanced behavioural and heuristic approaches to detect any suspicious activity on a system. The AV signature check is equivalent to a substring search problem, as we are searching large text regions (memory and disk data) for a smaller string (the virus signature). This application fits the PIM processing profile as it is computationally simple and is a memory-bandwidth bound problem which can run independent of the CPU. Signature checking can also exploit the PIM's parallelism by searching all memory vaults simultaneously.

Developing an AV signature checking PIM architecture would provide some security benefits. The PIM could continually analyse memory for potential viruses, reducing the compute workload for the CPU and catching viruses faster than traditional approaches. To reduce its overhead, the application could run passively in the background, only running when there is low activity. For example, the application may run if the PIM detects that a memory vault has a low or empty memory request queue. This would significantly reduce the AV scanner's compute time. The most significant issue with this solution is the need for AVs to regularly update their signature databases, ensuring they can detect new malware. The method of updating the database on a PIM would depend on its implementation. If the PIM allows for modification from the CPU, then the CPU could simply send the updated information across the memory bus to the PIM. Alternatively, if the CPU has restricted access to the PIM then it would have to be reprogrammed with the new database. This would be an annoyance for the user, as AV databases are updated frequently. The trade-off between security and usability is harder to justify for this usecase, and may cause users to not update, or not use the system.

Another point to consider is the necessity for developing an accelerated AV scanner. Unlike other security controls, AV scanning is not a time-critical operation. A system may only be scanned once a day or week. Because this operation is not time-critical and is run infrequently, the benefit and value of using a PIM is reduced. While the PIM could provide improvements to conventional AV scanning, the value of these improvements may not be enough to justify the cost of current hardware.

4.2.2 Integrity Monitoring

Integrity monitoring is a security application that aims to prevent malicious changes and operations by monitoring the execution of the system. These monitors have a long history and have been used to ensure the integrity of critical parts of the system, such as program code, data structures, and files. The history of monitoring technologies as well as some current solutions were discussed in Section 2.2. In practice, integrity monitoring works by ensuring a set of invariants are maintained throughout the system's execution. This is a computationally and memory-intensive task, and can significantly reduce the performance of applications. Such monitors are either implemented in the hypervisor layer, which sacrifices performance, or by external hardware, which can be expensive and difficult to maintain. Therefore, by offloading this process to a PIM we can reduce the performance overheads without the introduction of additional external hardware, which improves the efficacy of using integrity monitors; in turn improving the security of the system.

Implementing integrity monitors: As previously outlined, there are a couple of approaches to implementing an integrity monitor. Modern integrity monitors are either implemented in the hypervisor layer or as external hardware. Early development for integrity monitors used in-VM approaches which ran the monitor in user space, as a normal application [Forrest et al., 1996; Kim and Spafford, 1994; Abadi et al., 2005]. This was quickly found to be insufficient as malicious applications could gain unauthorised privileges and subvert the monitor. To combat this, new monitors were moved to a higher level of privilege, specifically, the hypervisor [Jiang and Wang, 2007; Dolan-Gavitt et al., 2011; Hofmann et al., 2011].

Implementing the monitor at the hypervisor level means that it has a higher privilege than the kernel or applications. In theory, this ensures that no malicious applications can gain access to the hypervisor layer to modify or remove the monitor from operating. However, hypervisors themselves have been shown to contain software vulnerabilities which can be exploited, thus negating the effectiveness of integrity monitors [MITRE, 2019]. Consequently, the use of external hardware has been adopted to mitigate the risk of malware interfering with the monitor. This creates a clear separation between the monitoring system and the OS, making it far more difficult to exploit. There have been many implementations of monitors using external hardware to provide a secure platform that runs independently of the main CPU [Bauman et al., 2015].

Given that integrity monitoring is a memory-intensive task that has been shown to work on external hardware devices, a PIM appears to be a suitable fit for this application. As outlined in Section 2.2, external hardware-based monitors are commonly implemented by taking snapshots of memory at regular intervals. Some recent approaches have used bus-snooping to capture incoming memory requests to monitor [Moon et al., 2015; Lee et al., 2013]. Utilising the PIM for this application would allow for either approach to be viable, as it can see all incoming requests, and has high-bandwidth access to memory. The consequence of implementing some types of monitors on a PIM is that we face the caching limitations outlined in Chapter 3. This is due to the lag and loss issues stemming from the use of write-back caching policy. This could be solved by allocating the monitored structures into a write-through region of memory, but as previously discussed this would cause significant performance degradation for frequently used memory regions. However, this is an issue common to all external hardware-based integrity monitors. The development of the Kernel Integrity Monitor (KIM) mitigates many of these limitations.

Kernel Integrity Monitoring (KIM) KIM is a specialised implementation of an integrity monitor. Instead of monitoring an application, a KIM focuses on maintaining the integrity of the OS kernel. This works by monitoring changes to kernel code, data structures and control flow, to ensure that the kernel is not maliciously or incorrectly modified.

Operating systems have dedicated security mechanisms to stop changes to kernel code and static regions. For example, the page table has a read-only bit which stops any process from writing to the page. However, some types of malware (e.g., rootkits) are able to gain kernel privileges and hide from the system. These types of malware can modify any part of the system, including changing the page table read-only bit. Most KIMs aim to protect systems from sophisticated malware attacks such as rootkits, which can subvert and compromise any security mechanism implemented in kernel or user space. Rootkits are types of malware which can be deployed to gain root privileges and modify the system while remaining hidden.

For its use on a PIM, a kernel integrity monitor offers a small advantage for solving the caching issues. As many parts of the kernel are read-only static objects, the use of a write-through cache region would have little effect on the system performance. By allocating all immutable kernel objects in this region, the monitor can catch any violations immediately. Extending this for mutable kernel objects makes it more complex as the monitor may incur significant performance loss, and must verify that any changes are correct. Section 4.4 will discuss this problem in depth, looking at how a monitor for mutable kernel objects could work with the PIM architecture.

The following sections focus on three different KIMs which are implemented using external hardware. These applications will be discussed and analysed, looking at their benefits, limitations, and the efficacy of implementing a similar system on PIM.

4.3 Case Study 1: HyperCheck – A Hardware-Assisted Integrity Monitor

HyperCheck presents an external hardware device which is used to remotely assure kernel integrity [Wang et al., 2010]. This paper uses a modified PCI Network Interface Card (NIC) device to take a snapshot of the CPU and memory state, before sending it to a remote machine to be analysed. The remote machine compares the given data against the initialised state for any malicious changes. By offloading the analysis to a dedicated machine, it saves in compute time, as well as allowing it to scale for a network of machines. This system works at the BIOS level, allowing it to protect the hypervisor, kernel, and applications above it. HyperCheck makes use of the CPU *System Management Mode* (SMM) which creates a snapshot of the current CPU state. This feature is available on all commodity x86 CPU architectures. When SMM is used the processor saves its entire state to the *System Management RAM* (SMRAM), which cannot be accessed from standard CPU modes.

4.3.1 Original Design

The system is comprised of three parts: the memory acquisition; CPU register checking; and the analysis module. In its implementation, the analysis is done by a remote monitoring machine, while the acquisition and register checking is done by the target machine. Figure 4.1 shows a high level view of the HyperCheck design.

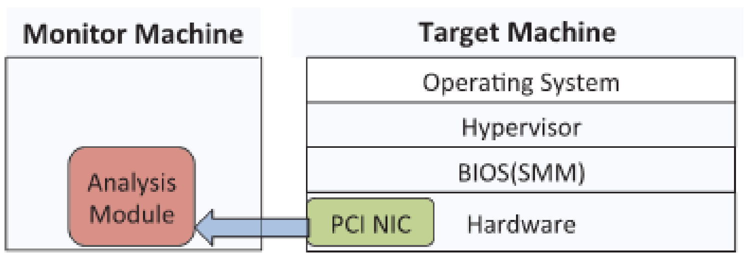

Figure 4.1: HyperCheck's design uses a remote machine for memory analysis [Wang et al., 2010].

The analysis module is used to check the current memory state of the kernel against its initial, trusted state. If a difference is found then an alert is made, which the administrator can decide is either a valid or malicious change. The register checking is used to ensure that critical CPU registers have not been maliciously tampered with. Specifically, the *Interrupt Descriptor Table Register* (IDTR) and *CR3* registers are checked. The IDTR is a register that points to the *Interrupt Descriptor Table* (IDT); a data structure which determines the response for interrupts and exceptions. Modifying the IDTR a common method for malware to exploit a system to run malicious interrupts. HyperCheck regularly verifies the IDTR is still valid, as it should not change after its initialisation. The CR3 register points to the page table, and is used for address translation. Periodically checking these registers for changes prevents some potential attacks which the monitor can otherwise fail to detect [Jang et al., 2014].

A key limitation of this system is in its memory analysis. Many mutable kernel objects may change throughout the system life, any changes that occur to these mutable objects will be reported by the HyperCheck system, relying on the outside administrator to determine the correctness of the modification. This allows for a degree of human error, where the administrator may mistakenly accept a modified object as valid.

The overhead associated with HyperCheck is largely dependant on the sampling rate. Sampling at one second intervals incurs an 11% performance overhead. By using faster sampling rates the overhead increases substantially. For some malware a one second interval may not be sufficient to detect the changes, as the malware may have completely hidden with no trace in that time. As such, improving the efficiency of the system would lead to increased sampling rates, and overall improving performance and security of HyperCheck. Another source of performance and energy overhead in the system is by sending large amounts of memory over the network for remote analysis. For each sample, the system collects the data from the specified memory regions, and then send it to the remote machine, spending approximately 90 million cycles to transmit the data. This transmission accounts for the majority of the systems execution time. The following section will discuss the possibility for using the PIM architecture to implement HyperCheck. The advantages and limitations of implementing the various components of HyperCheck in PIM will now be discussed.

4.3.2 HyperCheck on the PIM

Using the HyperCheck system on the PIM architecture would require some fundamental changes to its design. However, this would improve its performance and detection rates while reducing the system complexity. The PIM could accelerate this application by removing the need for a remote monitoring machine and running the analysis on the PIM. The PIM could be designed to perform all three functions: memory acquisition; memory analysis; and CPU checking. This would remove the need to transmit the data to a remote machine, which the system spends the vast majority of its CPU cycles executing. By doing the analysis on the same machine it also removes the additional latency and attack surface of transferring the data to a remote machine. Without the use of encryption, the data could give away critical information about the system and how it is set up, which is valuable for potential attackers. Even with encryption, the existence of the communication mechanism to a remote machine provides a large attack surface for malicious agents to exploit.

The HyperCheck system uses a snapshot-based approach as its method of memory acquisition. The use of a PIM for memory acquisition would significantly improve performance, because the PIM has much higher bandwidth to memory. This would allow the PIM to acquire memory faster, enabling faster sampling rates. However, by using this approach the system is still vulnerable to transient attacks, in which the malware is only visible between snapshots, and is not detectable. While the PIM could use the snapshot-based approach, recent research has showed improved efficiency from the use of bus snooping-based monitors [Moon et al., 2012; Lee et al., 2013]. As such, the PIM could implement its memory analysis by snooping all incoming traffic for the monitored regions. This would also capture the changes immediately, removing the potential for transient attacks. The PIM architecture could also facilitate the reporting of errors to an external machine. Similar technologies have shown the use of an embedded serial bus controller to send messages to external machines [Liu et al., 2013]. Figure 4.2 shows a potential architecture for a PIM implementing the snapshot-based HyperCheck system. This shows the components for a single PIM core.

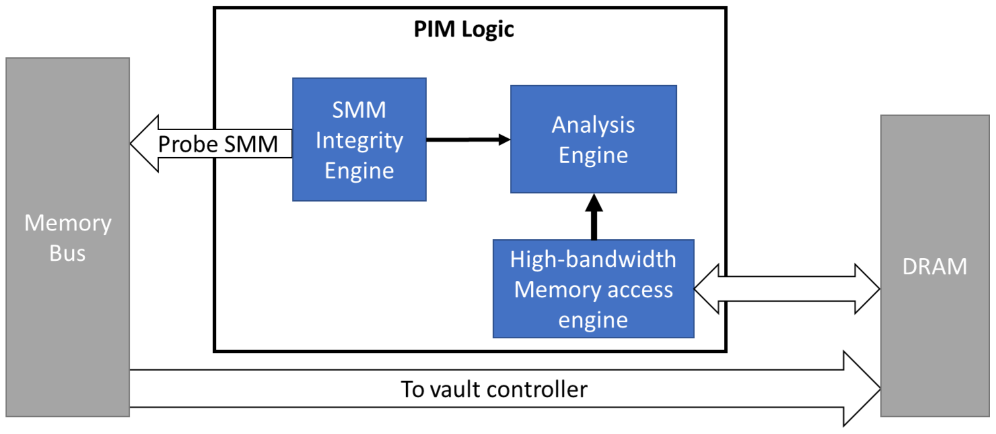

Figure 4.2: The PIM can implement the necessary components for the snapshot-based HyperCheck integrity monitor.

To address the CPU register checking, the PIM could similarly use SMM to regularly probe the CPU to gain visibility of its critical registers. This would be a slow process for both the original HyperCheck and the PIM as it must access the on-chip mechanism. However, the SMM does provide valuable integrity information, and would ensure the PIM version can catch the same integrity issues as the original implementation. Another possibility is the PIM to use other mechanisms like the *Trusted Platform Module* (TPM) to execute a similar register integrity check, with the advantage that TPM can inherently be trusted [Trusted Computing Group, 2014].

Both the original and the PIM implementation suffer from the same caching problems. As discussed in Chapter 3, the CPU cache causes the lag and loss of information for external hardware devices. This issue is not addressed in HyperCheck, and is assumed that all modifications will eventually propagate to memory. To solve this issue it is possible to create write-through regions of memory. By allocating our kernel in this region, any modifications in the CPU will be immediately propagated to memory. While this would improve security, this will have a large overhead for regions which are modified frequently.

Additionally, both systems would only be capable of efficiently monitoring static kernel objects: dynamic objects are harder to monitor. The allocation of dynamic kernel objects is substantially more complex, and requires sophisticated mechanisms which are difficult for external hardware devices to maintain due to the cache coherence problems.

The PIM architecture provides the necessary properties to implement and improve the HyperCheck system. The key benefits of using a PIM for this application is its highly parallel computation, high bandwidth access to memory, and its fast access to incoming memory requests. Using a PIM, the HyperCheck analysis could spread its workload across many PIM processing modules, reducing the analysis time and enabling faster detection of malware.

4.4 Case Study 2: KI-Mon – Monitoring Mutable Kernel Objects

KI-Mon is an external hardware device developed to monitor *mutable* kernel objects [Lee et al., 2013]. This extends previous works, such as Viligare [Moon et al., 2012] which presented a solution for monitoring *immutable* kernel objects. The KI-Mon platform is the first practical application of a monitor for mutable kernel objects. The methods used to implement a monitor for immutable kernel objects is simple, as we can assume that any write to these regions is malicious without further computation or verification. We also know where these regions are because the kernel is loaded into the same address space at boot time. Therefore, a monitor for immutable objects only needs a list of the corresponding addresses to function.

For monitoring mutable objects we must detect the writes to monitored regions and then verify that the change is valid. This task is more complex than immutable region monitoring. KI-Mon presents a hardware solution which can efficiently monitor mutable regions by comparing the updated value against a whitelist of known *good* values. Determining which values are valid is beyond the scope of this work; so we must assume that there is a finite list of known legitimate values.

4.4.1 Original Design

KI-Mon consists of both a hardware and software platform. The hardware platform consists of a *Vault Table Management Unit* (VTMU), the KI-Mon processor, *Direct Memory Access* (DMA) module, and an *Address Translation Engine*. The core mechanisms which enables the efficiency of KI-Mon are implemented in the VTMU. The VTMU is responsible for bus snooping and filtering of memory traffic before passing it onto the main processor using a FIFO queue. This reduces the workload and problem size and allows KI-Mon to only verify data which is in the monitored regions. The DMA module is used to take snapshots of parts of the host memory. This allows the platform to read regions such as the page table, which in conjunction with the Address Translation Engine, allows it to perform address translation. Figure 4.3 shows the design of this architecture, and the flow of information in the system.

The main KI-Mon processor is used to run the software platform. This software layer, named KI-Veri, verifies and enforces *MonitoringRules* which are used to program the invariants for the monitored regions. KI-Veri processes the events it receives from the VTMU, and detects any malicious events with respect to the defined MonitoringRules. Creating the MonitoringRules is a manual process, and must be set up for each individual kernel. Rules are created for mutable objects which are known to have been manipulated by existing rootkit software. A limitation of this approach is that the rules do not cover all possible attack vector, and is susceptible to human error, by not covering all possible entry points for the malware. For instance, if a new type of rootkit is developed and uses a new mechanism to gain access there may not be a MonitoringRule to detect it. While this is unlikely, it is still a potential point of failure.

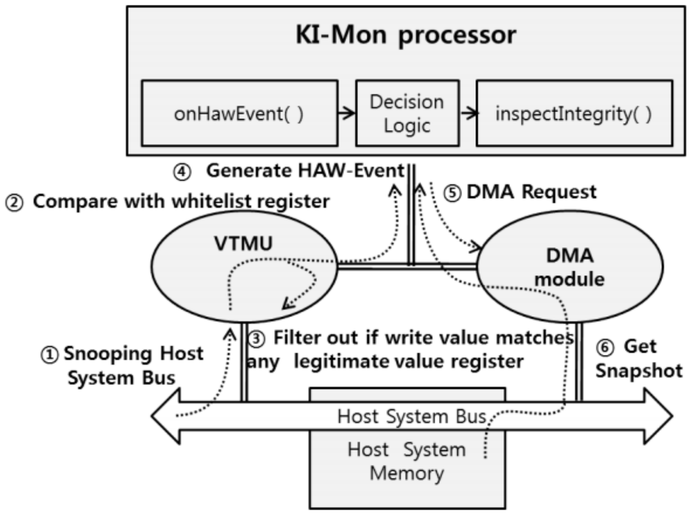

Figure 4.3: KI-Mon utilises the separate VTMU component to capture writes to memory [Lee et al., 2013].

In its implementation the VTMU was placed between the L1 and L2 caches. This allows it to intercept the modified data quickly, as the L1 cache commonly uses a write-through policy. This mitigates the caching issues we have identified for PIM applications. Unfortunately, this is not possible for most CPU architectures, as high performance processors keep all layers of cache on-chip to optimise performance. As such, KI-Mon will be able to detect malicious modifications faster than a PIM implementation.

The next section will detail the implications of using the PIM architecture to implement KI-Mon. Both the advantages and limitations will be discussed, evaluating the efficacy of using the KI-Mon system on a PIM.

4.4.2 KI-Mon on the PIM

For many areas of this applications design, the use of a PIM could improve its performance significantly. This is largely due to the PIM's capability for highly parallel computation, meaning it could evaluate and verify mutable objects concurrently at a larger scale. Another advantage of having many PIM cores is the increased number of registers, which the VTMU uses to store its address data for the monitored regions, which would allow for a greater number of monitoring rules to be enforced.

KI-Mon uses a DMA module to access host memory and take a snapshots of regions which are stored in KI-Mon's private memory. The PIM simplifies this system by removing the need for DMA and private memory of the monitor. Instead, the PIM can efficiently access memory directly. However, the primary use of the DMA module was to allow for address translation engine. Previous work has found that implementing address translation in a PIM is not viable due to the prohibitive cost and complexity it requires [Ghose et al., 2018]. To solve this we can look at the previous work of Hsieh et al. [2016] which we discussed in Section 3.1. However, as we are capturing these changes at the PIM and not in the CPU the addresses would have been translated into the physical address space already. This essentially removes the need for the address translation engine. While this simplifies the system substantially, it creates other critical limitations.

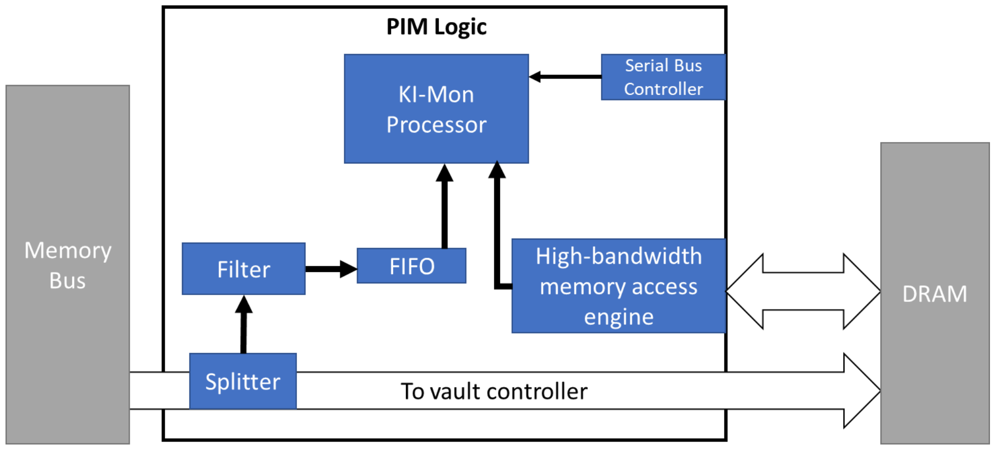

Figure 4.4: The PIM enables KI-Mon to detect malicious changes, but is limited to observing data evicted from cache.

One such limitation with implementing KI-Mon on a PIM is the problems associated with CPU caching. As demonstrated in Chapter 3 the CPU cache can cause a large amount of information loss and lag. While the PIM will eventually see the changes made to the mutable object, it is impossible to determine if there existed previous lost values which were invalid. Without a sufficient method of quickly communicating the modified data to the PIM this limitation would ruin the efficacy of this application for the PIM. One potential solution is to place all kernel objects in a write-through memory region. This is a good solution for immutable regions as there should be no modifications to those regions. However, mutable regions may be accessed and modified many times during the kernels runtime, whereby a write-through cache would cause large performance overheads.

Another approach could see the use of the PIM only as the KI-Mon processor; keeping the VTMU as a separate hardware mechanism. If it is feasible to connect the VTMU between the L1 and L2 cache it would avoid the caching issues identified for the use of an entirely PIM implementation. In doing so, it would require the modification of the CPU architecture to implement the VTMU modules. This is expensive and not feasible to implement for commodity hardware. It also limits the usage of this system to a small subset of architectures which support these features, such as ARM Cortex [ARM, 2019].

Implementing KI-Mon on the PIM would require some significant modifications to the original design. The PIM can offer improved performance and parallelism from the many PIM cores, and improved control of memory, but the costs associated from the PIM's loss and temporal lag of information detract from its efficacy and value. Figure 4.4 illustrates a basic PIM architecture required to implement the KI-Mon system. This solution makes use of a serial bus connection to allow updates to the system. If mechanisms are developed to solve the caching issues then the KI-Mon system could prove to be a valuable application for the PIM architecture.

4.5 Case Study 3: Kernel Integrity with Programmable DRAM

The work by Liu et al. [2013] presents a kernel integrity monitor built into the control module for Fully Buffered (FB)-DIMM memory. This work is similar to PIM as it implements processing off-chip near physical memory. FB-DIMM DRAM was developed for server-grade machine to improve scalability of memory by chaining DRAM together [Ganesh et al., 2007]. This reduces the number of data lines required on the memory bus, as conventionally each DRAM module needs its own set of data lines. To control and direct the memory requests to the correct device FB-DIMM introduced the Advanced Memory Buffer (AMB). The AMB is a small chip which acts as a router for memory requests. This paper introduces a kernel integrity monitor called *MGuard*.

4.5.1 Original Design

MGuard is implemented by extending the AMB logic on FB-DIMM DRAM to incorporate filtering and verification mechanisms. This additional hardware intercepts the incoming traffic, filters it for the relevant packets then analyses the data for modifications to known immutable memory regions. If it detects that an immutable area has been modified then it reports to a remote administrative machine. The computation for MGuard is run on a lightweight RISC processor, which is part of the extended-AMB. This is a general purpose processor and allows MGuard to be updated and customised depending on the specific kernel, hypervisor or new knowledge of potential attack areas. Another benefit of using a RISC core is its small die space and low energy consumption, incurring only a 3.5% energy overhead. This can be achieved as the RISC core stays in idle mode most of the time, waking periodically to run the integrity checks on thee queue of the captured system memory modifications. To utilise the RISC processor a serial bus controller was also implemented, in order to update, and program the operation of MGuard. This allows an extra level of flexibility, and enables the program to be developed and fine-tuned over time without having to fabricate new hardware.

MGuard uses an approach similar to the bus-snooping monitors which we have previously discussed, by copying all incoming write requests for the system to analyse. This allows MGuard to continuously monitor the accesses to physical DRAM from the entire execution of the system. The benefit of this approach is that it does not introduce any performance overheads for FB-DIMM as all computation is done off the critical path. Figure 4.5 shows a detailed design architecture for MGuard.

The system does have a few limitations, that may cause it to miss some types of kernel integrity violations. First, MGuard can only verify the immutable kernel

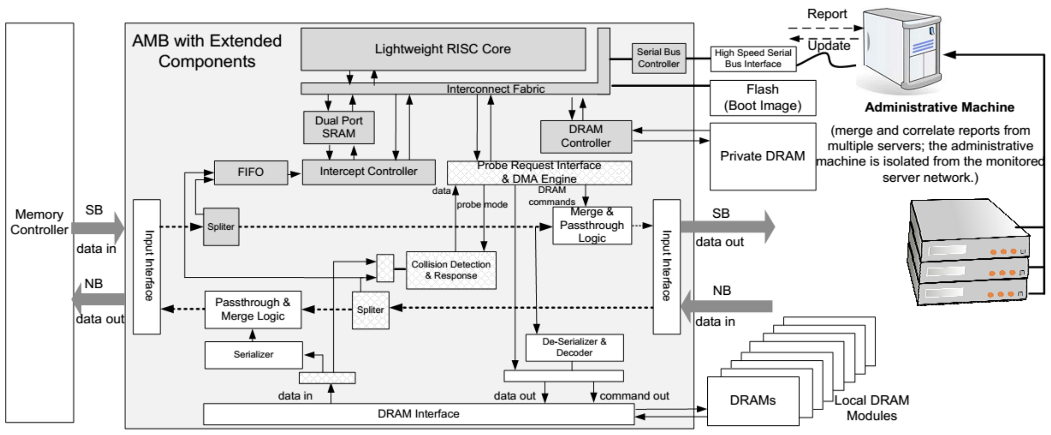

Figure 4.5: The MGuard paper illustrates the additional hardware required for monitoring [Liu et al., 2013].

regions, meaning that any mutable kernel object is not checked. If malware were to modify these mutable objects, the monitor will not be able to detect it. Second, the CPU caching provides another key limitation to the system. MGuard does not provide any mechanisms to check the CPU cache, and is susceptible to the loss and lag of cache data, as discussed in Chapter 3. The authors note that MGuard could force the CPU to flush its caches periodically, however, this would cause some significant reduce the CPU performance.

While this research shows the effectiveness of MGuard, the use of FB-DIMM limits its value. FB-DIMM is not widely used, and is no longer being developed, as the use of DDRX DRAM technologies has become widely adopted. The future work in this paper suggest its use with DDR4 DRAM technology, however, no research has investigated this problem. However, we can repurpose the concepts developed in this paper to evaluate its efficacy in a PIM architecture. The following will look into the advantages and limitations implementing the MGuard system on the PIM architecture.

4.5.2 MGuard on the PIM

The design of MGuard is similar to the PIM architecture, due to its near-memory processing capabilities. Designing a PIM for MGuard is simpler than the previous case studies because of this similarity. The PIM can provide all the necessary properties and components to fully implement the MGuard system in a method similar to the FB-DIMM work. The primary advantage of using a PIM for MGuard is the improvement in value and practical usage of the system, as FB-DIMM is no longer available for modern systems, replaced today by DDR4 DRAM.

The PIM would have the same limitations as FB-DIMM in terms of CPU caching. Both of these suffer from the loss of information and temporal lag due to the CPU cache. As such, the monitor would only be able to detect changes that affect physical memory.

For computation, the PIM could also implement lightweight RISC processors. Current die space on 3D-stacked memory allows for the addition of a lightweight RISC processor for each memory vault. This would allow for highly parallel computation for the monitoring, reducing the analysis time. The ability to program and update the PIM cores would also be necessary for MGuard to be feasible on the PIM. Implementing a serial bus controller onto a PIM has not been explored in the existing literature. Assuming that the serial bus controller can fit on the die, there will need to be further design choices in how the cores will communicate. For instance, there could be a master-slave hierarchy, where the master PIM core communicates with the many slaves to distribute the firmware and workloads, or each core could read from a shared boot image and run completely independently. This will be discussed further in Chapter 5. Figure 4.6 demonstrates an example architecture of how MGuard could be implemented on the PIM. The use of a shared boot image among all PIM cores allows the process to be updated, while still ensuring the work is spread among many processors.

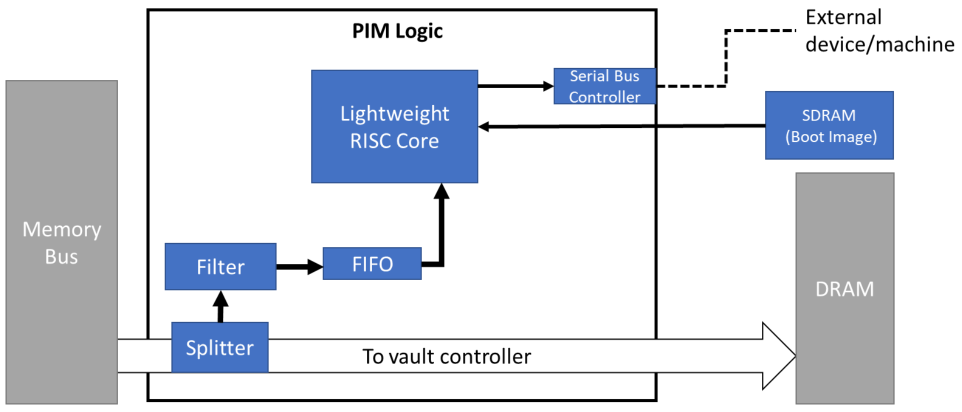

Figure 4.6: The PIM implementation of MGuard incorporates the same key components into the newer technology.

MGuard appears to be an ideal application for the PIM. Its use with FB-DIMM for near-memory computation allows for minimal design changes in a potential implementation on a PIM. The PIM can provide the system with some performance and scaling improvements due to its parallel design. These case studies will now be evaluated, discussing the benefits and limitations with each applications use of the PIM.

4.6 Evaluation

The previous three case studies show widely-different implementations for a Kernel Integrity Monitor using external hardware. For each of these applications we have discussed and analysed their benefits and limitations, looking at how each application could utilise the PIM architecture to improve performance, both in terms of security and computation.

Each of the case studies suffer from similar limitations of CPU caching and slow access to on-chip mechanisms. These align with the findings in Chapter 3. Case study 1 is the only monitor to check the CPU registers to ensure that no malicious changes are made to critical CPU registers, and to detect any attempts to subvert the monitor. Case study 2 is a monitor to verify mutable kernel objects. In its implementation it resolves the caching issue by placing the VTMU between the L1 and L2 cache. This is not feasible for the CPU architectures which typically use integrity monitoring. Case study 3 is the simplest to move to a PIM application. It shares many similarities in its design, as MGuard is implemented as a near-memory application.

For case studies 1 and 2, the use of a PIM can improve the effectiveness and performance of the monitoring. The PIM's high bandwidth access to memory offers a huge performance boost, as these systems are frequently checking against physical memory. For case study 3, the PIM can improve the processing through its parallel architecture. It can also provide value to MGuard by being implemented on widely used DRAM memory technology. The added hardware mechanisms should not impede the memory speed. As seen in case study 3, MGuard was able to intercept and analyse incoming traffic without interfering with the core memory request paths. The PIM could implement these mechanisms in a similar approach, to ensure that the performance of memory is not affected.

In case studies 2 and 3 the example PIM architecture shares many of the same components with each other. The main differences are in the software layer. Where the KI-Mon is focused on the integrity of mutable kernel objects, MGuard focuses primarily on the immutable objects. As these architectures are similar to each other, it could be possible to merge the functionality of MGuard and KI-Mon to enable the monitoring of both mutable and immutable kernel objects. This will be explored further in Chapter 5.

A potential limitation with the PIM is its interactions with other devices. In the case studies we have seen applications use a remote machine to report to. This requires additional hardware components to allow for this communication. No existing work has investigated the use of a PIM to communicate with a remote machine. However, using the MGuard system as a template, it is feasible to implement a serial bus controller, with a serial interface to the remote machine. This is dependant on the die space required for these components. This is an area which should be explored by future work, to determine what components are feasible to implement in the PIM, and how to securely report violations without creating additional security flaws in the system.

4.7 Summary

This chapter has identified multiple security areas where PIM can provide value and improve upon existing solutions and technologies. In addition, I have further analysed three case studies for implementing a KIM in hardware. I have shown that with modifications, these applications have the necessary characteristics be implemented in PIM. Chapter 5 will extend this idea by designing and evaluating a PIM device to monitor the integrity of an OS kernel.

Chapter 5

Kernel Integrity Monitor on the PIM

The previous chapter has demonstrated various security-related applications in which a PIM architecture could accelerate existing solutions. In particular, we have focused on existing technologies for *Kernel Integrity Monitors* (KIM), investigating the ways in which PIM could be utilised to improve existing monitoring systems. This chapter extends these ideas by proposing a design for a PIM architecture, called *PIMGuard*, that implements a KIM capable of monitoring all static kernel objects. The proposed design will be analysed and evaluated for its viability against the limitations discussed in Chapter 3, and compared to existing monitoring solutions.

5.1 Background and Overview

The use and implementation of KIMs has been extensively covered in this dissertation: particularly in Chapters 2 and 4. In short, a KIM is used to ensure the integrity of critical OS kernel regions in memory. These memory regions encompass the kernel code and data structures, and can be monitored due to the observation that almost all kernel code is static after initialisation. Most monitors focus on immutable memory regions [Moon et al., 2012; Wang et al., 2010; Petroni Jr et al., 2004]. These are objects which should not change throughout the kernel's execution. More recently, there has been research to investigate the use of monitors for mutable kernel objects, which require additional verification [Lee et al., 2013]. The KIM's aim is to improve security by detecting malware, such as rootkits, which make malicious changes to these important kernel structures. PIMGuard aims to replace traditional hypervisor monitoring solutions, such as Window's PatchGuard [Microsoft, 2006] and the Linux Kernel Runtime Guard (LKRG) [Openwall, 2019]. This can be done because the PIM is isolated from the CPU, acting as an external observer, meaning that it cannot be modified or subverted by the CPU.

5.1.1 Threat Model

The proposed PIM architecture, *PIMGuard*, is effective against attackers who have gained unauthorised, privileged access to the system through the use of a rootkit or similar exploits. PIMGuard aims to detect malicious modifications to the immutable kernel regions, as well as detecting potential malicious changes to mutable object.

In Linux, immutable objects include the kernel code and system call table. Mutable objects are regions of the kernel which change throughout its execution which can be exploited by rootkits. For example on Linux, the Virtual File System (VFS), and Loadable Kernel Modules (LKM) are mechanisms commonly exploited by rootkits [Bunten, 2004]. By monitoring the changes to these regions it is possible to detect malicious changes outside of normal operation.

The monitor should be capable of detecting all malicious modifications that reach physical memory. Attacks which utilise the CPU cache to hide modifications from the PIM is an important aspect to address, but is out of scope and left for future research. This system does not aim to detect attacks which exploit critical CPU registers, such as *Interrupt Descriptor Table* (IDT)-hook rootkits [Adamyse, 2002]. Sophisticated IDT-hook rootkits have been shown to subvert detection from security monitors by creating a malicious copy of the IDT and changing the IDT register to the malicious copy address [Jang et al., 2014]. These attacks are out of scope for PIMGuard and are left for future research.

In this work we assume that an attacker does *not* have physical access to the machine. Attacks from insiders with full access to the machine hardware is outside to the scope of this work.

While PIMGuard may not catch all forms of malicious activity on a machine, it is another layer of defence in protecting the system. No single tool or technique can solve all security issues. The PIM monitor adds another layer of assurance and defence against a malicious agent, in line with the defence-in-depth model of computer security [Ahmad et al., 2014].

The monitor is composed of both the hardware and software platforms. The hardware is the PIM logic, which provides the capabilities to run computation in memory. In addition, the software layer is the program running on the PIM, enforcing the monitoring rules.

5.2 Hardware Platform

In Chapter 4 we observed that the PIM architecture for case study 2 and 3 were particularly similar. This similarity can be exploited to develop a system which combines the functionality of both monitors; allowing for both immutable and mutable objects to be monitored in the same system. At the time of writing there is no published monitoring applications which combine these methods.

To facilitate the PIM architecture we are using Hybrid Memory Cube (HMC) memory. As discussed in Chapter 2, HMC uses the 3D-stacked DRAM technology, controlled through the use of a logic layer. Figure 5.1 shows how the HMC architec-

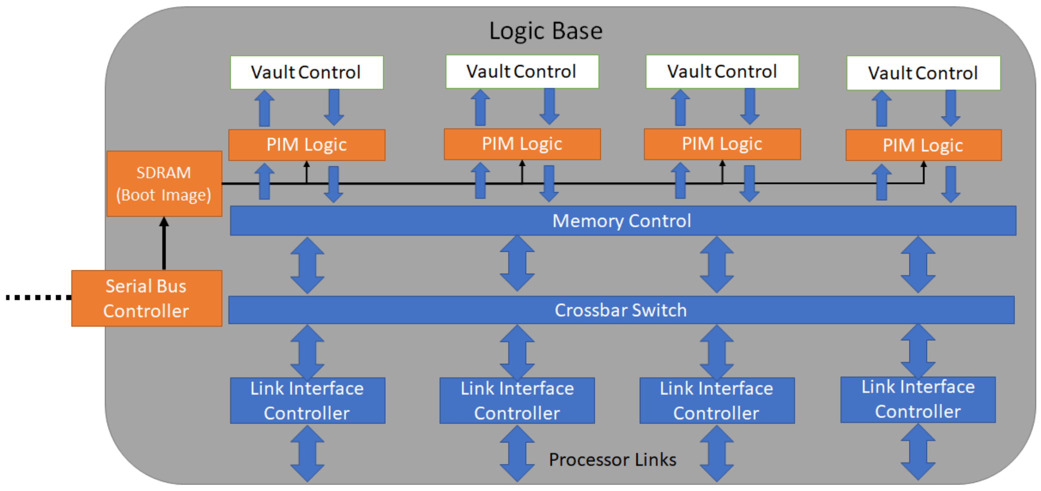

Figure 5.1: The logic layer on HMC memory can be modified to include the additional hardware required for the PIM.

ture could be modified to allow for the PIM architecture. In this architecture we have a single PIM *core* for each vault controller. For the current HMC specification [Hybrid Memory Cube Consortium, 2014], the logic layer consists of sixteen vaults. The existing logic layer is very sparsely populated, and could readily accommodate additional hardware components. With current microprocessor technology it is possible to add a lightweight general purpose RISC processing core for each memory vault [Ghose, 2019]. The RISC core is the primary hardware mechanism for the PIM computation. However, additional components are required to enable the processor to function efficiently.

The firmware for each PIM core is stored in a central SDRAM chip. This stores the PIM application code and region data. Having the firmware in a central location allows us to update the application for all PIM core at once. This is far more efficient than having individual copies for each core, saving die space and energy usage. The firmware can be written to via the serial bus, meaning that the chip would need to be physically accessed to be modified. This serial bus is completely isolated from the CPU and memory bus, ensuring that malware is incapable of modifying the PIM's firmware. The serial bus controller is used by the PIM core in the eventuality of a security violation. When this occurs the PIM core can send a report of the violation to a remote machine over the serial connection.

Figure 5.2 shows the hardware architecture within the PIM Logic. As mentioned, the main processing is done by a lightweight RISC core. The bus-snooping approach is enabled by the *splitter* and *static region filter* mechanisms. The splitter first copies all incoming memory traffic. The static region filter then only forwards on the write requests which are in the static kernel regions. This works because the kernel location can be known ahead of execution time, and allows only kernel memory writes to be processed further. Further work could potentially merge these into a single

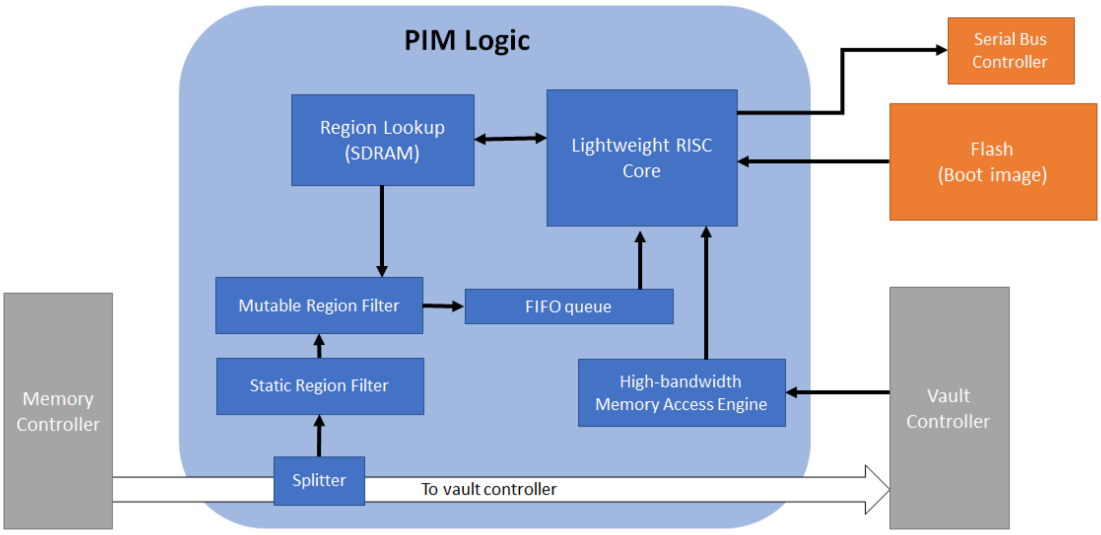

Figure 5.2: Design of PIM Logic for PIMGuard, an in-memory Kernel Integrity Monitor

component, to copy and filter in the same step.

These requests then pass through the *Mutable Region Filter*. This filter is used to determine whether the given request is for a mutable or immutable memory region. Depending on the number of mutable monitoring regions, the filter can keep the address data in registers, allowing for fast checks on the memory requests. This is similar to the design of the KI-Mon filter [Lee et al., 2013]. However, in the KI-Mon filter, the number of regions monitored is restricted by the number of registers available. In PIMGuard we can allow for an arbitrary number of mutable regions. For a large number of mutable regions, the filter can use the data stored in the region lookup to determine whether the memory is a mutable region. Because the monitor is monitoring all static kernel regions, if the memory is not found in the mutable region table it must be an immutable region. After the mutable region filter has analysed a packet, it adds a flag to indicate to the processor whether it is a mutable or immutable region. All packets which make it through the filters are placed in a First-In First-Out (FIFO) queue. This acts as a buffer for the RISC core, and allows for bursts of traffic to be captured. Further testing would be required to determine the size of this queue. Once packets are in the FIFO queue, the RISC core can start its analysis for security violations. The software running on the processor is used to verify the integrity of both mutable and immutable kernel objects against the provided region data stored in the *Region Lookup*.

The *high-bandwidth memory access engine* is used to retrieve data from physical memory, for use in the immutable region computation. During its initialisation, the monitor must use the memory access engine to read and store a copy of the monitored regions, to use as the baseline to check for modifications. This data is stored in the Region Lookup SDRAM. This is required because of the reduced granularity of information which is observed by the PIM, (as discussed in Section 3.2.3). The consequence of this issue is that, using the incoming data, we are unable to determine whether the immutable region has been modified, because the only information received is the cache line and dirty bit. To determine whether the immutable region has been changed we must check against its previous value. The previous value is stored in the Region Lookup instead of reading it from physical DRAM, since the write request may have already changed its value because we are analysing a copy of the request.

The *Region Lookup* is used as persistent storage for the monitored region information. This contains four data structures: the immutable region table; copies of immutable region data; the mutable region table; and the mutable region whitelist table. The immutable and mutable region tables both contain the start and end addresses for each monitored region. The mutable region whitelist is slightly more complex, and must store all of the correct values for each mutable monitored region. The size of these tables, specifically the whitelist, is currently unknown, pending further testing.

5.3 Software Platform

While the hardware architecture enables the PIM to capture memory write requests, the software layer must implement the verification mechanisms used by the main processor. The software platform must allow for the creation of monitoring rules for both mutable and immutable regions. The boot image used by the RISC core contains both the firmware and monitoring region's data. This allows the monitored regions and rules to be easily updated. Both the mutable and immutable region tables are implemented as a look-up table, with the start and end addresses for each region. The monitoring of mutable regions also requires a whitelist of correct values for each region. To add monitoring rules for the PIM core requires modifying the firmware data. This data stores the information about the mutable and immutable memory regions as lookup tables. By adding an entry to the corresponding lookup table, it will automatically become a new monitoring rule. Tools can be developed to support the creation of these data structures.

To find the correct values for the mutable regions the system can run in a *capture* mode. In this mode, the system captures all of the values for the specified mutable regions, recording them in the whitelist, which can be sent back to a remote machine for further analysis. This significantly reduces the overhead in finding correct values, however, we must ensure the system is running correctly and has not been compromised during this mode. Because this mode may not capture all possible value, additional values can be added in manually.

When first booted, the PIM core must copy the boot image and region data from the SDRAM and into the region lookup store. Additionally, for each immutable region, the processor must use the memory access engine to retrieve a copy of the original values. During the regular execution of the PIM core, the processor will take packets from the FIFO queue, check the flag to detect whether the region is mutable or immutable, and analyses accordingly. For the immutable case, the processor can use the immutable look-up table to ensure that this region should not be modified. Once found, it can be compared against the original value, also stored in the region lookup. If the region is found to be different, the processor will report the violation via the serial bus.

For the mutable case, the processor must verify that the modification is valid. This requires the processor to check the corresponding whitelist entry from the address lookup. If the modified value is not found on the whitelist, then the processor can send an alert via the serial bus.

The reporting mechanism must differentiate between strict security violations and potential breaches. This is because for mutable objects, the kernel may make a valid modification that is not in the whitelist of correct values. In this case, it is unknown whether a breach has occurred, but must alert the admin of the potential security breach. On the other hand, any modification to the immutable regions can be treated as a security breach. When reporting, the software can include the physical memory address of the object to help pinpoint the entry point of the breach.

5.4 Design analysis

Currently, there is no standardised design for implementing a PIM architecture. Many researchers have proposed architectures purpose-built for a single task. Previously shown in Section 4.5 was the capability for lightweight processors to be placed in memory. This architecture allowed for the processors to be updated through the use of a serial link. Similarly, this technology could be used in PIM architectures, as shown in PIMGuard. As explained in Section 5.2, the logic layer on current 3D-stacked DRAM contains enough free die space to implement a lightweight RISC core for each vault. The use of a general purpose processor improves the value of the architecture, as it allows it to be updated or re-purposed for different applications. The development and adoption of a general purpose PIM core would dramatically improve the efficacy of the PIM, and would lead to a larger eco-system of PIM applications. To enable the architecture to be programmable it must contain a boot image which can be modified. The serial bus controller allows for the boot image to be modified by an external device or machine.

The following sections will analyse the design of PIMGuard, discussing the design choices, and limitations imposed by the architecture.

5.4.1 Reporting Security Breaches

When the application detects a violation of the kernel integrity it must respond. There are a few ways in which the monitor can take action. It can report it to a remote administrative machine, or it can report it to the CPU. An extreme, but effective response to such violation of security would be to force the system to crash. This could be done by refusing to answer any new memory requests from the CPU, stalling all applications, and eventually causing the system to crash. However, this is definitely not an ideal solution as the system may be running critical infrastructure, and it would also leave no trace log for what occurred to the system.

Using a remote machine to report the security violations is the simplest solution, and allows for a network of machines to be running and monitored from a remote machine. However, as mentioned in Section 4.3 the use of networking with a remote machine opens up a far larger attack surface. This choice can be justified, as very few modern computing systems are completely isolated, and would usually already have a network interface card communicating to other machines.

The main issue is that when a violation is detected, the CPU cannot be trusted, as it may already be compromised. Reporting the violation to the CPU is problematic, because malware with knowledge of this system could choose to ignore or remove the report from the PIM.

5.4.2 Limitations

The PIMGuard design is not free from limitations. As described in Chapter 3, the issues of caching and on-chip mechanisms cannot be avoided. For PIMGuard, the CPU cache provides the most problems. The loss of information provides a challenge, because potentially malicious modifications may be missed. Additionally, the temporal lag of information can reduce the detection time, between the malware execution and its detection. To mitigate both of these challenges it is possible to allocate these areas to a region with a write-through cache policy, allowing all modifications to be passed to the PIM immediately. However, the performance costs of this must be evaluated.

Other limitations include: the efficient distribution of workload for parallel computation; and protection from CPU register attacks. The parallelism in the proposed PIM architecture could have limitations depending on the workload. In the proposed design, the workload is automatically distributed among the PIM cores, depending on the physical memory address of the memory request. This could cause asymmetry in the distribution of workloads, as the PIM cores only receive the memory requests which are going to its corresponding vault controller, as demonstrated in Figure 5.1. As such, it is possible that some vault do not contain any monitored regions, making these PIM cores redundant. The worst case scenario is having all monitored regions located in the same vault. This would reduce the parallel computing to just the single PIM core. Future research should analyse this problem to determine the scale of this issue for PIMGuard, and further develop PIM architectures to fairly and efficiently spread the workload.

A recent security flaw has been discovered for the use of external integrity monitors. Researchers outlined an Address Translation Redirection Attack (ATRA), which takes advantage of an underlying assumption made by the monitor, about the integrity of the page table [Jang et al., 2014]. This vulnerability used the CPU to create its own dummy page table and give the monitor the dummy table location. This allowed the CPU to modify the real page table, and the kernel without detection. In addition, side-channel attacks which exploit characteristics of internal CPU mechanisms, such as Meltdown [Lipp et al., 2018] and Spectre [Kocher et al., 2019] cannot be caught by the PIMGuard system. Such attacks are deemed outside the scope of this work.

A fundamental limitation of this type of monitor is that it cannot monitor dynamically allocated kernel regions. To monitor these regions the CPU must communicate with the PIM to tell it where the dynamic kernel regions are in physical memory. However, if a rootkit has compromised the system it could stop the CPU from sending these messages, or mislead the monitor by giving it false information. Executing such an attack is non-trivial and has not been shown to subvert existing hardware-based monitors.

5.5 Future Work

There is a wide range of work which can follow on from this architecture. First, the efficacy of this design must be tested. This can be done by using a system simulator. Implementing PIMGuard in a full system simulator allows us to measure the effectiveness of PIMGuard as a monitor, and the overheads associated with its execution, in terms of computation and energy. As the PIM's usage in 3D-stacked memory is still in its infancy, the limitations in terms of compute performance and energy consumption are still unknown. This is an important area to consider when looking to use the PIM, as some tasks may be too computationally complex for the PIM to efficiently achieve. For the PIMGuard application to be a viable security tool it most not impose significant performance overheards to the system.

An improved design for PIMGuard is also an area of future work. There may be areas which can be streamlined in the computation. For instance, merging the splitter and static region filter into a single component may improve efficiency and throughput of the monitor.

PIMGuard could also incorporate the CPU register checking mechanism found in HyperCheck [Wang et al., 2010]. This would help protect against attacks such as the Address Translation Redirection Attack (ATRA) [Jang et al., 2014]. ATRA exploits an underlying assumption that the address translation mechanisms have not been violated. Such exploits could subvert the monitor by copying the kernel memory regions to a new region, and exploiting the page table pointer to the address of a new maliciously crafted page table.

5.6 Summary

This chapter has proposed and evaluated PIMGuard, a PIM architecture which implements a Kernel Integrity Monitor. The design builds on existing monitoring solutions which use external hardware to operate. I have shown how the PIM architecture can be used to monitor both mutable and immutable kernel regions in the same application. While the design is not free from limitations, it covers a wider range of kernel regions than existing technologies, and demonstrates a practical security application for the PIM.

Chapter 6

Conclusion

Security is a critical part of modern computing systems. The emergence of 3D-stacked DRAM has revived the development of Processing-In-Memory (PIM) architectures. In its current state, the PIM architecture provides full flexibility for testing and implementing new architectures and designs; because PIM research is in its infancy and lacks standardisation. In this dissertation I have identified and evaluated the efficacy of using PIM in security-related applications.

Computer security covers a wide range of applications. For a security-related application to perform effectively on the PIM, the application must not be restricted by the inherent limitations of the architecture. Many of these limitations come from the CPU's on-chip mechanisms, such as caching and address translation. For many applications the CPU cache can ruin the PIM's feasibility. This is evident from the significant amount of information loss (median loss of 15%), and temporal lag observed by the PIM.

With respect to these limitations, I argue that signature-based antivirus scanning and kernel integrity monitoring are ideal security applications to utilise the full potential of a PIM. To test this claim, three case studies of external hardware-based kernel integrity monitors were evaluated. For each study, the PIM provided key advantages over the existing method; with no additional limitations compared to the existing approach.

To demonstrate the efficacy of developing security-related applications on the PIM, PIMGuard is presented. PIMGuard uses the PIM architecture to implement integrity monitoring for both mutable and immutable regions of the OS kernel. This design fully utilises the high-bandwidth and parallel processing benefits of the PIM. PIMGuard extends the current literature by being the first external hardware device to monitor both mutable and immutable kernel regions. With further testing, this architecture could prove to be a significant improvement over traditional hypervisor-based security monitors, and could provide better performance and security assurance than existing hardware-based implementations. Such designs demonstrate that the PIM offers the necessary properties and processing profile to implement various new and existing security applications.

6.1 Future Work

The open-ended nature of identifying and analysing the efficacy of security applications for the PIM means that there is a substantial amount of future work that can follow on from the work presented here. Most importantly, the proposed design, PIMGuard must be experimentally tested. This can be done through the use of system simulators. Using a simulator, we can analyse and quantify the performance difference between monitors in terms of computational and energy overheads, as well as test the effectiveness of the proposed design against a range of rootkits.

The use of multi-VM systems has exploded over the last decade with the adoption of cloud computing infrastructure. Cloud computing systems run many VMs on a single system. The ability to monitor the integrity of these VMs would provide huge value for both cloud providers and consumers. Implementing this with current PIM technology has many technical challenges. Specifically, it would require closer control and communication between the hypervisor and the PIM to determine which regions to monitor. This is outside the scope of PIMGuard, but is an interesting monitoring challenge for future research.

Taking a broader view of the PIM architecture, the applications which fit the PIM's current properties are very particular. For the PIM architecture to be widely adopted, the key limitations need to addressed. Identifying and developing solutions for the address translation and cache coherency problems would greatly improve the PIM's versatility and value for consumers. Although previous work has looked into these areas, the solutions are not scalable at this stage due to the need for major modifications to CPUs. Finding efficient solutions to these key limitations would open the door to many more security applications being feasible on the PIM; further improving the efficacy of security-related applications on the PIM.

Bibliography

Abadi, M.; Budiu, M.; Erlingsson, U.; and Ligatti, J., 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05 (Alexandria, VA, USA, 2005), 340–353. ACM, New York, NY, USA. doi: 10.1145/1102120.1102165. http://doi.acm.org/10.1145/1102120.1102165. (cited on page 23)

Adamyse, K., 2002. Handling interrupt descriptor table for fun and profit. phrack 59. (cited on page 38)

Ahmad, A.; Maynard, S. B.; and Park, S., 2014. Information security strategies: Towards an organizational multi-strategy perspective. *Journal of Intelligent Manufacturing*, 25, 2 (2014), 357–370. (cited on page 38)

Ahn, J.; Hong, S.; Yoo, S.; Mutlu, O.; and Choi, K., 2016. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43, 3 (2016), 105–117. (cited on pages 2 and 5)

Ahn, J.; Yoo, S.; Mutlu, O.; and Choi, K., 2015. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 336–348. IEEE. (cited on pages 5 and 17)

ARM, 2019. Arm v7-m architecture. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html. (cited on page 30)

Bauman, E.; Ayoade, G.; and Lin, Z., 2015. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, 48, 1 (2015), 10. (cited on page 23)

Boroumand, A.; Ghose, S.; Patel, M.; Hassan, H.; Lucia, B.; Hsieh, K.; Malladi, K. T.; Zheng, H.; and Mutlu, O., 2017. Lazypim: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters*, 16, 1 (2017), 46–50. (cited on pages 5 and 11)

Bunten, A., 2004. Unix and linux based rootkits techniques and countermeasures. In *16th Annual First Conference on Computer Security Incident Handling, Budapest*. (cited on page 38)

Carvalho, C., 2002. The gap between processor and memory speeds. *Proc. of IEEE International Conference on Control and Automation*, (2002). (cited on page 3)

Chang, K. K., 2017. Understanding and improving the latency of dram-based memory systems. *arXiv preprint arXiv:1712.08304*, (2017). (cited on page 4)

Chi, P.; Li, S.; Xu, C.; Zhang, T.; Zhao, J.; Liu, Y.; Wang, Y.; and Xie, Y., 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, vol. 44, 27–39. IEEE Press. (cited on pages 2 and 5)

Costan, V. and Devadas, S., 2016. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016, 086 (2016), 1–118. (cited on page 6)

Davi, L.; Dmitrienko, A.; Sadeghi, A.-R.; and Winandy, M., 2010. Privilege escalation attacks on android. In *international conference on Information security*, 346–360. Springer. (cited on page 2)

Dolan-Gavitt, B.; Leek, T.; Zhivich, M.; Giffin, J.; and Lee, W., 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *2011 IEEE Symposium on Security and Privacy*, 297–312. doi:10.1109/SP.2011.11. (cited on page 23)

Dorai, C. and Venkatesh, S., 2003. Bridging the semantic gap with computational media aesthetics. *IEEE multimedia*, 10, 2 (2003), 15–17. (cited on page 10)

Forrest, S.; Hofmeyr, S. A.; Somayaji, A.; and Longstaff, T. A., 1996. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, 120–128. IEEE. (cited on page 23)

Ganesh, B.; Jaleel, A.; Wang, D.; and Jacob, B., 2007. Fully-buffered dimm memory architectures: Understanding mechanisms, overheads and scaling. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 109–120. IEEE. (cited on page 31)

Ghose, S., 2019. personal communication. (cited on page 39)

Ghose, S.; Hsieh, K.; Boroumand, A.; Ausavarungnirun, R.; and Mutlu, O., 2018. Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions. *arXiv preprint arXiv:1802.00320*, (2018). (cited on pages xiii, 4, and 29)

Gueron, S., 2010. Intel® advanced encryption standard (aes) new instructions set. *Intel Corporation*, (2010). (cited on page 9)

Hofmann, O. S.; Dunn, A. M.; Kim, S.; Roy, I.; and Witchel, E., 2011. Ensuring operating system kernel integrity with osck. In *ACM SIGARCH Computer Architecture News*, vol. 39, 279–290. ACM. (cited on page 23)

Hsieh, K.; Khan, S.; Vijaykumar, N.; Chang, K. K.; Boroumand, A.; Ghose, S.; and Mutlu, O., 2016. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 25–32. IEEE. (cited on pages 5, 10, and 30)

Hybrid Memory Cube Consortium, 2014. *Hybrid Memory Cube Speicifcation 2.1*. (cited on pages 3 and 39)

Jang, D.; Lee, H.; Kim, M.; Kim, D.; Kim, D.; and Kang, B. B., 2014. Atra: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 167–178. ACM. (cited on pages 25, 38, 43, and 44)

Jiang, X. and Wang, X., 2007. "out-of-the-box" monitoring of vm-based high-interaction honeypots. In *Recent Advances in Intrusion Detection*, 198–218. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 23)

Kim, G. H. and Spafford, E. H., 1994. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 18–29. ACM. (cited on page 23)

Kocher, P.; Horn, J.; Fogh, A.; ; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; Schwarz, M.; and Yarom, Y., 2019. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*. (cited on page 44)

Lee, H.; Moon, H.; Jang, D.; Kim, K.; Lee, J.; Paek, Y.; and Kang, B. B., 2013. Kimon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 511–526. (cited on pages xiii, 7, 23, 26, 28, 29, 37, and 40)

Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; Yarom, Y.; and Hamburg, M., 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*. (cited on page 44)

Liu, Z.; Lee, J.; Zeng, J.; Wen, Y.; Lin, Z.; and Shi, W., 2013. *Cpu transparent protection of os kernel and hypervisor integrity with programmable dram*, vol. 41. ACM. (cited on pages xiii, 26, 31, and 32)

Microsoft, 2006. An introduction to kernel patch protection. https://blogs.msdn.microsoft.com/windowsvistasecurity/2006/08/12/an-introduction-to-kernel-patch-protection/. (cited on page 37)

MITRE, 2019. Vmware: Security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-252/Vmware.html. (cited on page 23)

Moon, H.; Lee, H.; Heo, I.; Kim, K.; Paek, Y.; and Kang, B. B., 2015. Detecting and preventing kernel rootkit attacks with bus snooping. *IEEE Transactions on Dependable and Secure Computing*, 14, 2 (2015), 145–157. (cited on page 23)

Moon, H.; Lee, H.; Lee, J.; Kim, K.; Paek, Y.; and Kang, B. B., 2012. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM conference* *on Computer and communications security*, 28–37. ACM. (cited on pages 7, 26, 28, and 37)

Nethercote, N. and Seward, J., 2003. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89, 2 (2003), 44–66. (cited on page 11)

Openwall, 2019. Lkrg - linux kernel runtime guard. https://www.openwall.com/lkrg/. (cited on page 37)

Pangaria, M.; Shrivastava, V.; and Soni, P., 2012. Compromising windows 8 with metasploitâĂŹs exploit. *IOSR Journal of Computer Engineering (IOSRJCE)*, 5, 6 (2012), 01–04. (cited on page 2)

Petroni Jr, N. L.; Fraser, T.; Molina, J.; and Arbaugh, W. A., 2004. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 179–194. San Diego, USA. (cited on pages 6 and 37)

Seaborn, M. and Dullien, T., 2015. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15 (2015). (cited on page 2)

Seshadri, V.; Lee, D.; Mullins, T.; Hassan, H.; Boroumand, A.; Kim, J.; Kozuch, M. A.; Mutlu, O.; Gibbons, P. B.; and Mowry, T. C., 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 273–287. ACM. (cited on pages 2 and 5)

Spradling, C. D., 2007. Spec cpu2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 35, 1 (2007), 130–134. (cited on page 11)

Stone, H. S., 1970. A logic-in-memory computer. *IEEE Transactions on Computers*, 100, 1 (1970), 73–78. (cited on page 3)

Trusted Computing Group, 2014. Trusted platform module library: Architecture. https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.07-2014-03-13.pdf. (cited on page 27)

Valgrind Developers, 2009. Cachegrind: a cache and branch-prediction profiler. http://valgrind.org/docs/manual/cg-manual.html. (cited on page 11)

Von Neumann, J., 1993. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15, 4 (1993), 27–75. (cited on page 3)

Wang, J.; Stavrou, A.; and Ghosh, A., 2010. Hypercheck: A hardware-assisted integrity monitor. In *International Workshop on Recent Advances in Intrusion Detection*, 158–177. Springer. (cited on pages xiii, 24, 25, 37, and 44)

Zhang, D. P.; Jayasena, N.; Lyashevsky, A.; Greathouse, J.; Meswani, M.; Nutter, M.; and Ignatowski, M., 2013. A new perspective on processing-in-memory architecture design. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 7. ACM. (cited on page 5)