A 3D-Stacked Architecture for Secure Memory Acquisition

Guilherme Cox, Zi Yan, Abhishek Bhattacharjee, Vinod Ganapathy
Department of Computer Science, Rutgers University
{guilherme.cox, zi.yan, abhib, vinodg}@cs.rutgers.edu

Rutgers University Computer Science Technical Report DCS-TR-724, May 2016

ABSTRACT

Many security and forensic analyses rely on the ability to fetch memory snapshots from a target machine. To date, the security community has relied on virtualization, external hardware or trusted hardware to obtain such snapshots. We show that these prior techniques either sacrifice snapshot consistency or impose a performance penalty on applications executing atop the target. We present a new snapshot acquisition system based on emerging 3D-stacked architectures that offers snapshot consistency without impacting the performance of the target's applications. We have implemented our system in a hardware simulation infrastructure and report on our evaluation with several data intensive workloads.

1. INTRODUCTION

A number of important security and forensics applications rely on the ability to fetch memory snapshots from a target machine. Most techniques to diagnose a target machine for rootkit infection, for instance, work by obtaining memory pages containing the target's operating system (OS) code and data, and checking whether they satisfy certain invariants [8, 15, 19, 20, 34, 60–62, 67]. This paper focuses on mechanisms to obtain snapshots from a target machine. Ideally, a memory snapshot acquisition mechanism should satisfy three properties:

① **Tamper resistance.** The target's OS may be compromised with malware that actively tries to evade detection. The snapshot acquisition mechanism must resist malicious attempts by an infected target OS to tamper with its operation.

② **Snapshot consistency.** A consistent snapshot is one that faithfully mirrors the memory state of the machine at a given instant in time. Consistency is important for client applications that analyze the snapshot. Without consistency, different portions of the snapshot may represent different points in time during the execution of the machine, making it difficult to assign semantics to the snapshot.

③ **Performance isolation.** Snapshot acquisition must only minimally impact the performance of other applications that may be executing on the target machine.

The security community has converged on three broad classes of techniques for memory snapshot acquisition, namely *virtualization-based*, *trusted hardware-based* and *external hardware-based* techniques, but as we argue below, none of these solutions is able to simultaneously achieve all three properties (see Figure 1).

In virtualization-based techniques (pioneered by Garfinkel and Rosenblum [27]), the target is a virtual machine (VM) running atop a trusted hypervisor. The hypervisor has the privileges to inspect the virtual memory and CPU state of VMs, and can therefore obtain a snapshot of the target. This approach has the obvious benefit of isolating the target VM from the snapshot acquisition mechanism, which is implemented within the hypervisor. However, it imposes a tradeoff between consistency and performance-isolation. To ensure consistency, the hypervisor can pause the target VM, thereby preventing the target from modifying the VM's CPU and memory state during snapshot acquisition. But this consistency comes at the cost of preventing applications within the target from executing during snapshot acquisition. The hypervisor could instead allow the target VM to execute concurrently with memory acquisition, but this would compromise consistency of the snapshot. Besides this tradeoff, the use of virtualization has two additional disadvantages. First, it requires the target system to run as a VM. This restricts the scope of memory acquisition only to environments where the target satisfies this assumption, *i.e.,* server-class systems and cloud platforms. Second, it requires a substantial software TCB—the entire hypervisor. Production-quality hypervisors have in excess of 100K lines of code and a history of exploitable vulnerabilities [21–25, 45], which can jeopardize the isolation guarantees that they provide.

Hardware-based techniques reduce the software TCB and are applicable to any target system that has the necessary hardware support. Methods that use trusted hardware rely on the hardware architecture's ability to isolate the snapshot acquisition system from the rest of the target. For example, the ARM TrustZone [1, 5, 28, 73] partitions the execution of the system into two "worlds." The target runs in a deprivileged world, without access to the snapshot acquisition system, which runs in a privileged world with full access to the target. However, because the processor can only be in one world at any given time, this system offers the same snapshot-consistency versus performance-isolation tradeoff as virtualized solutions.

External hardware-based techniques use a physically isolated hardware module, such as a PCI-based co-processor, on the target system and perform snapshot acquisition using remote DMA (*e.g.,* [8, 14, 48, 49, 56, 60, 62]). These techniques offer

Property→ Method↓	① *Tamper resistance*	② *Snapshot consistency*	③ *Performance isolation*
Virtualization	✓⋆	②✓ ⇒ ③ ✗ and ③✓ ⇒ ② ✗	
Trusted hardware	✓	②✓ ⇒ ③ ✗ and ③✓ ⇒ ② ✗	
External hardware	✓⧫	✗	✓
This paper (3D-stacks)	✓	✓	✓

Figure 1: Design space of memory acquisition systems. *(⋆) Virtualized systems provide tamper-resistance assuming that the hypervisor is trusted; however, attacks on hypervisors violate this assumption [21–25, 45]. (⧫) While external hardware-based techniques were originally thought to be tamper-resistant, a number of attacks have allowed malicious OSes to evade detection by hiding state from the external hardware mechanism [41, 47, 65].*

performance-isolation by design—the co-processor executes in parallel with the CPU of the target system and therefore fetches snapshots without pausing the target. However, this very feature also compromises consistency because memory pages in a single snapshot may represent the state of the system at different points in time. Finally, researchers have demonstrated attacks using which snapshot acquisition can be subverted despite physical isolation of the co-processor. Rutkowska [65] designed an attack (circa 2007) that leveraged the fact that the co-processor relies on the PCI bus controller to correctly route its remote DMA requests to the target's physical memory. Rutkowska's attack, originally designed for certain AMD-based chipsets, leveraged properties of the chipset to reroute requests thereby effectively hiding portions of the target's physical memory from the snapshot acquisition mechanism. This attack has become easier to implement on modern chipsets, which ship with IOMMUs. Just as benign OSes can leverage IOMMUs to protect themselves from malicious DMA (*e.g.,* [53]), a malicious target OS can simply reprogram the IOMMU to reroute DMA requests away from physical memory regions that it wants to hide from the snapshot acquisition mechanism. Other researchers have discussed address-translation attacks that leverage the inability of co-processors to view the CPU's page-table base register [41, 47]. This can be used to bootstrap malicious virtual-to-physical address translations requested by the acquisition mechanism and hide memory contents from the snapshot.

Contributions. The primary contribution of this paper is a novel hardware-based snapshot acquisition mechanism that achieves all three properties. To build the mechanism, we leverage the growing trend toward *3D-stacked* memory [13, 32, 50, 51] (also called die-stacked memory or high-bandwidth memory). This trend is motivated by processor vendors' quest to leverage extra transistors on chip to improve memory system performance. One approach that has gained some traction is to use these transistors to build fast memory on-chip with CPU components, stacked atop the processor die. 3D-stacked memory complements traditional DRAM, but offers higher bandwidth (4×-8×) because accesses to it do not go to off-chip DRAM via the memory bus [42, 43, 46, 59]. Because this memory is located on-chip, the processor vendor can also add circuitry for *near-memory processing* [2, 3, 35], which implements compute logic atop the 3D-stacked memory.

Our snapshot acquisition mechanism has a hardware-based TCB implemented on-chip, stacked on the processor die and with support in the memory controller to capture and digitally sign pages in the 3D-stacked memory. The resulting snapshot captures the memory and CPU state of the machine faithfully, and any attempts by a malicious target OS to corrupt the state of the snapshot can be detected during snapshot analysis. The snapshot is consistent in that it mirrors the CPU and memory state at the instant when snapshot acquisition was initiated. Moreover, consistency does not come at the cost of performance-isolation. Snapshot acquisition proceeds concurrently with applications executing atop the processor. We have implemented our mechanism atop a novel real-system emulation framework for 3D-stacked memory. Our experimental evaluation with data intensive workloads demonstrates that our mechanism offers performance isolation during snapshot acquisition.

2. OVERVIEW AND THREAT MODEL

We begin by presenting a high-level overview of our snapshot acquisition system. We defer a detailed overview of its design to the next section and focus instead on describing the hardware TCB, software support, and the threat model.

Figure 2 shows the components of our system. Our system relies on a combination of novel but low-overhead hardware TCB on the target machine, and ancillary untrusted software components. Our system's objective is to acquire a complete snapshot of the off-chip DRAM memory at a given instant in time, when the analyst requests a snapshot. Our use of 3D-stacked memory and near-memory processing lies at the heart of the snapshot process. Since it is on the same package as the processor die and memory controller, 3D-stacked memory and near-memory processing logic placed on the stacked die are part of the hardware TCB. We use the 3D-stacked memory as a hardware cache of the off-chip DRAM (though our approach generalizes to other uses of 3D-stacking too), copying in all the physical pages from off-chip DRAM and digitally signing them one by one; the process of signing is performed by readily implementable near-memory cryptographic logic, as implemented in many trusted hardware chips including TPMs.

While we rely exclusively on the hardware TCB to copy pages into the snapshot, we do rely on untrusted software to trigger and marshal snapshot creation. That is, an analyst triggers snapshot acquisition via a user-space application (the snapshot process) that invokes a supporting kernel-mode driver via a system call. In response, the kernel driver communicates to the hardware that a snapshot has been requested. The kernel driver and the hardware then work in tandem to obtain the snapshot.

Hardware Components. Our system requires architectural changes to memory controllers, processors and the integration of 3D-stacked memory with near-memory processing on-package. Changes to the processor and memory controller are merely intended to facilitate control of die stacking-enabled snapshot acquisition.

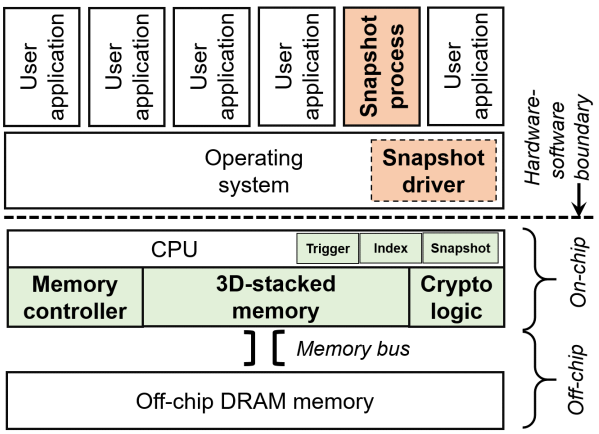


Figure 2: Architecture of our snapshot acquisition system. *The system contains all the components shown in bold font. However, only the hardware components are trusted. The software components are untrusted.*

① *Memory controller enhancements.* To manage the 3D-stacked memory, we rely on memory controller support. The architecture community is actively exploring different ways to architect memory controllers to manage a diverse range of 3D-stacked memory designs. In general, the designs range from using 3D-stacked memory as a cache of the off-chip DRAM memory, *i.e.,* the physical address space is equal to the off-chip DRAM capacity [52, 63] to using it as an extension of the physical address space, *i.e.,* the physical address space is the sum of the off-chip DRAM and 3D stacked memory capacities [17]. Further, architects are studying the granularity of data movement between the stacked and off-chip memory devices, ranging from moving data in pages [59] to cache lines [43, 52, 63].

Our snapshot acquisition mechanism can be built on top of any such architectures as long as the 3D-stacked memory is integrated on-package and is hence part of the TCB. Without loss of generality, we pick a specific design point for this study—using 3D-stacked memory as a fully-associative cache of off-chip DRAM, with page-level movement—demonstrating the effectiveness of our approach. However, we intend studying the myriad of 3D-stacked memory design options in future work.

To manage this stacked memory, our memory controller ensures that every read or write access to a off-chip DRAM memory address (from user-mode or kernel-mode) first brings that page to the 3D-stacked memory area. Note that user-mode applications and the kernel generate virtual addresses, but our memory controller brings the physical page from off-chip DRAM to the 3D-stack using paging hardware (as is usual). Because the memory controller treats 3D-stacked memory as a hardware-managed cache, this memory is not addressable from user- or kernel-mode. The main novelty in the design of the memory controller is its management of 3D-stacked memory to balance snapshot consistency and performance isolation. We present its design in the next section.

② *Processor enhancements.* We add three special-purpose registers to serve as the hardware/software interface. This adds minimal overhead to the existing register file of modern processors, typically consisting of several tens of architecturally-visible and several hundreds of physical registers. The software components use the *trigger register* to initiate snapshot acquisition by writing a non-zero value in it. The hardware then brings the page frame referenced by the *index register* into the 3D-stacked memory area, signs and includes it in the snapshot. The *snapshot register* stores a pointer to a buffer region where the copy of the page and its digital signature are temporarily stored before being committed to an external medium such as persistent storage.

③ *3D-stacked memory and near-memory processing.* As detailed, the final piece of our system is the on-package 3D-stacked memory itself. This consists of conventional 3D stacked memory, connected via conventional through-silicon via (TSV) links [52] from the processor die to support high bandwidth data movement between the processor and 3D stack. Furthermore, we integrate hardware logic on the 3D-stacked memory to hash and sign memory pages [16]. Thus, we assume that the processor is endowed with a public/private key pair. Digital signatures protect the integrity of the snapshot even from an adversary with complete control of the target's software, *e.g.,* via a rootkit-infected OS.

Software Components. The hardware components provide the basic ability to copy and digitally-sign individual pages of memory fetched to the 3D-stack. The software components are used to initiate and shepherd snapshot acquisition atop the primitive interface exposed by the hardware. However, as we will see, it is not necessary to trust these software components.

The hardware/software interaction happens within a kernel-mode snapshot driver. The snapshot driver has two main responsibilities. First, it instructs the memory controller to iteratively fetch off-chip DRAM pages to the 3D-stack to be copied and signed. The snapshot driver is designed to ensure that the snapshot contains all pages of off-chip DRAM memory. Second, as each page of memory is copied and signed, the snapshot driver writes it out to a suitable medium, *e.g.,* persistent storage, a network connection, or a diagnostic serial port, depending upon the method chosen by a forensic analyst. We also include a user-mode process to serve as a front-end/UI for the forensic analyst. This process invokes the snapshot driver via a system call.

Our design departs from prior snapshot acquisition systems in an important way. Prior systems based on virtualization, trusted hardware and external hardware *do not involve the target's software stack* during snapshot acquisition. In contrast, the snapshot driver and the user-mode process described above are important components of our system. Despite their importance, these software components are *not part of our TCB*. This is because the software components merely initiate and shepherd snapshot acquisition. The hardware components are responsible for actually obtaining and protecting the integrity of memory pages included in the snapshot. Any attempts by the software components to subvert the integrity of pages in the snapshot can be detected because the snapshot is digitally signed by the hardware. The digital signature also attests a nonce supplied by the forensic analyst and physical page frame numbers. This allows the forensic analyst to ensure freshness of the snapshot and that all physical pages from off-chip DRAM are included in the snapshot.

It can be argued that the snapshot driver can collude with a malicious kernel to clean up a rootkit infection before creating a snapshot. The resulting snapshot will appear clean to a forensic analyst. We counter this argument with two observations. First, we observe that if snapshot acquisition indeed triggers kernel cleanup, then it has a desirable side-effect of also eliminating the malware infection. Second, we note that if the kernel arranges for the cleanup to be temporary—*i.e.,* that it reintroduces the infection after the snapshot has been obtained—then it will most likely leave a footprint, either in the memory snapshot or in a snapshot of the disk image from which the OS boots. It is then possible for a forensic analyst to detect the infection, either by analyzing the memory snapshot or the disk snapshot.

Threat Model. The goal of our system is to ensure reliable snapshot acquisition from a target machine. Our threat model admits a powerful attacker who actively tries to subvert snapshot acquisition from the target. The attacker is assumed to have complete control over the target's software components. However, the on-chip components, namely, the CPU, the memory controller, die-stacked memory and the cryptographic logic described earlier are assumed to be trusted. Physical attacks on the off-chip hardware components, *e.g.,* those that modify contents of off-chip DRAM pages via electromagnetic methods or modify the contents of pages as they transit the memory bus to the on-chip components, are out of scope.

We aim to *detect* all attempts by the attacker to subvert snapshot acquisition. Specifically, we aim to ensure *integrity* of the snapshot, and therefore provide forensic analysts the ability to detect an attacker's attempt to corrupt or suppress parts of the snapshot. An attacker may also attempt to launch a denial of service attack, *e.g.,* using a malicious snapshot driver in the target, that prevents the snapshot from being output. However, such denial of service attacks are readily detectable by the forensic analyst. We assume that the content of the snapshot itself is not confidential, and can therefore be disclosed to the attacker.

It is important to note that our focus is on *mechanism—i.e.,* we wish to obtain a memory snapshot from the target machine in a reliable and secure fashion. To detect any attempts by the attacker to subvert snapshot acquisition, we require the forensic analyst to run a few basic integrity checks on the obtained snapshot. These integrity checks, which are described in Section 3.4, verify digital signatures in the snapshot, and ensure freshness and completeness of the snapshot. Once the forensic analyst establishes the integrity of the snapshot, he can then analyze the snapshot using various domain-specific *policies* to determine if the target is infected with malware. In this paper, we limit ourselves to a description of the mechanism and the checks needed to establish snapshot integrity. There is a large body of research on algorithms and policies used to analyze memory snapshots (*e.g.,* [8, 15, 19, 20, 26, 34, 60, 61]). Our work is orthogonal to this body of prior research, and snapshots acquired using our system can be analyzed using any one of the methods developed in prior work.

3. SYSTEM DESIGN

This section presents the design of the hardware and software components in detail. We begin with a discussion of snapshot consistency, which our system offers.

3.1 Snapshot Consistency

A key feature of our system is its ability to ensure *snapshot consistency.* A snapshot of a target machine consists of a copy of all the pages in the off-chip physical DRAM memory of the target and digital signatures to attest the integrity of the snapshot. The snapshot also contains the values of CPU registers, which play an important role in downstream forensic analysis of the acquired snapshot. The snapshot is consistent if it reflects the state of the target machine at a given instant in time.

Consistency is an important property for forensic applications that analyze snapshots. Without a consistent snapshot, different memory pages in the snapshot will represent the state of the target machine at different points in time, thereby causing the forensic analysis to be imprecise. For example, consider a forensic analysis that detects rootkits by checking whether kernel data structures satisfy certain invariants, *e.g.,* that function pointers only point to valid function targets [61]. Such a forensic analysis operates on on the snapshot by identifying pointers in kernel data structures, recursively traversing these pointers to identify more data structures in the snapshot, and checking invariants when it finds function pointers in the data structures. Consider a situation where a page F_1 contains a pointer to an object allocated on page F_2. If F_1 and F_2 are copied to the snapshot representing different states of the target machine, the forensic analysis can encounter a number of illogical situations, as Figure 3 depicts. Such inconsistencies can also be used to hide malicious code and data modifications in memory [41]. Prior research projects [8, 61] encountered such situations in the analysis of inconsistent snapshots, and had to resort to unsound heuristics to remedy the problem. A consistent snapshot will capture the state of the target's memory pages at either T or at $T + \delta$, thereby allowing the forensic analysis to traverse data structures in memory without the above problems.

In contemporary snapshot acquisition systems, consistency comes at the cost of performance or vice versa. A target machine may be in the process of executing several user applications when a memory snapshot is requested. These user applications

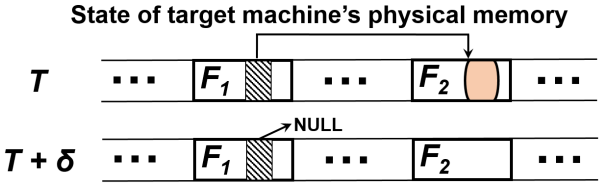


Figure 3: An example showing the importance of snapshot consistency. *This figure shows the memory state of a target machine at two points in time: at T, when the pointer in F_1 points to the object in F_2, and at $T + \delta$, when the object in F_2 has been freed, and the pointer in F_1 is set to* NULL*. If the snapshot contains a copy of F_1 at time T, and F_2 at time $T + \delta$, then the pointer in the snapshot refers to a non-existent object. The forensic analysis will simply interpret the memory referenced by the pointer as an object, which may lead to unsoundness downstream in the analysis. If the snapshot contains F_1 at $T + \delta$ and F_2 at T, the forensic analysis will fail to account for the object.*

and the kernel actively modify memory. To ensure consistency, prior systems have resorted to pausing the target, either using the hypervisor in virtualized systems or using a world-switch in systems that rely on trusted hardware. This prevents the user applications on the target machine from making progress for the duration of snapshot acquisition, and is disruptive, especially in server settings and when snapshot acquisition is frequent. When pausing the target is not an option, *e.g.,* in snapshot acquisition systems based on external hardware, the snapshots are not guaranteed to be consistent.

Our system acquires consistent memory snapshots without pausing the target machine. Snapshot acquisition proceeds in parallel with user applications and kernel execution that can actively modify memory. Our hardware design ensures that the acquired memory snapshot reflects the state of the target machine at the instant when the snapshot was requested.

While our system ensures that an acquired snapshot will faithfully mirror the state of the machine at a given time instant, we do not attempt to specify what that time instant should be. In particular, we note that while snapshot consistency is a *necessary* property for client forensic analysis tools, it is not *sufficient*, *i.e.,* not every consistent snapshot is ideal from the perspective of client forensic analyses. For example, consider a consistent snapshot acquired when the kernel is in the midst of creating a new process. The kernel may have created a structure to represent the new process but may not have finished adding it to the process list, resulting in a snapshot where the process list is not well-formed.

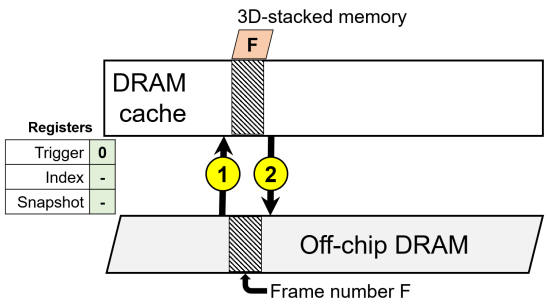
To address this problem, prior work has suggested collecting snapshots when the target machine is in *quiescence* [34], *i.e.,* a state of the machine when kernel data structures are likely to be well-formed. Quiescence is a domain-specific property that depends on which data structures are relevant for the forensic analysis and what it means for them to be well-formed. Because our focus is on mechanism, rather than any specific forensic analysis policies, we only discuss how to achieve snapshot consistency. We do not attempt to define *when* a snapshot acquisition must be initiated and instead leave that task to domain-specific forensic analyses. With our system, the forensic analysis may initiate snapshot acquisition at time instant t when the system is quiescent. Even if the target enters a non-quiescent state as the snapshot is being acquired, *e.g.,* because of concurrent kernel activity initiated by other user applications, the snapshot will reflect the quiescent state of the target at time t.

3.2 Hardware Design

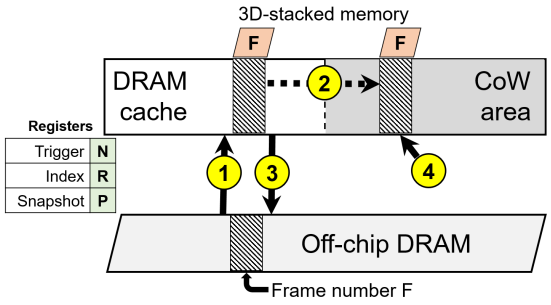
The hardware components form the trusted core of our system, and are key to enabling consistent snapshot acquisition without pausing the target system. The bulk of the hardware design is in the logic of the memory controller, which is the on-chip component that mediates *all* accesses to DRAM memory. These include CPU accesses to off-chip and 3D stacked memory, which are directed to the memory controller via the MMU, as well as accesses from devices (*e.g.,* using DMA) via the IOMMU.

During regular operation, *i.e.,* when snapshot acquisition is not in progress, the memory controller configures the 3D-stacked memory region to serve as a cache of off-chip memory (*DRAM cache* in Figure 4(a)). The 3D-stacked memory therefore effectively acts as the last level cache before off-chip DRAM. While any cache configuration is supported by our snapshot acquisition scheme, we model each 3D stacked memory cache line at the granularity of a baseline memory page size of 4KB (as is the case for x86 and ARM architectures). Every read or write access to an address brings the entire memory page containing that address from off-chip DRAM into 3D-stacked memory if it is not already present there. Our implementation treats 3D-stacked memory as a fully-associative cache, although we can readily accommodate direct-mapped or set-associative cache designs as well. Each cache line, as is standard, has a tag to identify the frame number of the page cached in that line and additional bits to denote usage information (*e.g.,* validity of the cache line, recency of cache line usage). When a new page must be brought into an already-full cache, the memory controller selects and evicts a victim page using standard page-replacement [18].

The hardware enters snapshot acquisition mode when a non-zero value is written into the trigger register. The memory controller then reconfigures the 3D-stacked memory region by splitting it into two portions (Figure 4(b)). The first portion continues to serve as a cache of off-chip DRAM memory. Since only this portion of 3D-stacked memory is available for caching, the memory controller tries to fit all the pages that were previously cached during regular operation into the available space in the cache. If all pages cannot be cached, the memory controller selects and evicts victims to off-chip DRAM using the page-replacement policy. The second portion of 3D-stacked memory serves as a copy-on-write (CoW) area. As explained below, the CoW area allows user applications and the kernel to modify memory concurrently with snapshot acquisition, while saving pages that have not yet been included in the snapshot.



4(a) ***During regular operation****, the memory controller configures the on-chip 3D-stacked memory as a cache of off-chip DRAM pages. This memory serves as a new level in the cache hierarchy, e.g., between the L3 cache and DRAM on traditional machines.* ① *Any access by the CPU to a DRAM page brings the page to the 3D-stacked memory region. Each entry in the 3D-stack has cache-tags associated with it, denoting the frame number of the DRAM page that is cached (**F**) and additional bits to store information pertaining to recent use of the page.* ② *Pages are evicted from the 3D-stacked memory region when it reaches its capacity using the standard second-chance (CLOCK) algorithm [18].*



4(b) ***During snapshot acquisition****, the memory controller splits 3D-stacked memory into two portions, one a DRAM cache and the second a copy-on-write (CoW) area.* ① *The first portion continues to serve as a cache of off-chip DRAM pages.* ② *If the CPU access is a write and the page has not yet been snapshot (i.e., $\boldsymbol{F} \geq \boldsymbol{R}$), the memory controller copies that page into the CoW area.* ③ *The page may be evicted from the 3D-stack when the DRAM cache reaches it capacity, as in regular operation.* ④ *The copy of the page in the CoW area remains there until it has been included in the snapshot (i.e., $\boldsymbol{F} < \boldsymbol{R}$), after which it can be overwritten with other pages as they are brought into the CoW area.*

Figure 4: Layout of the on-chip 3D-stacked memory (a) during regular operation, and (b) when snapshot acquisition is in progress. The figures also show the state of the trigger, index, and snapshot registers in both modes.

We chose a simple 50-50% split of the 3D-stacked memory for the DRAM cache and the CoW area. This choice is not fundamental, and alternative designs can explore different fractions of 3D-stacked memory allocated to the CoW area. The split can even be adaptive, with the CoW area growing in size as more pages are added to it. All these alternatives are compatible with our overall hardware design. In fact, despite its relative simplicity, this partitioning strategy offers good performance (Section 5); we intend exploring the benefits more sophisticated partitioning in future work.

Recall that a snapshot contains a copy of all pages in off-chip memory. The hardware copies one page of off-chip memory at a time into the snapshot together with its digital signature, guided by the software components (Section 3.3) to sequentially iterate over all page frames in off-chip DRAM. As this iteration proceeds, other applications and the kernel may concurrently modify memory pages that have not yet been included in the snapshot. If the memory controller sees a write to a memory page that the hardware has not yet copied and signed, the memory controller creates a copy of the original page in the CoW area, and lets the write operation proceed in the DRAM cache area. A page frame is copied *at most once* into the CoW area, and this happens only if the page has to be modified by other applications before it has been copied into the snapshot. The *index register* stores the frame number (R in Figure 4(b)) of the page that is currently being processed by the hardware for inclusion in the snapshot. The memory controller copies a frame F from the DRAM cache to the CoW area when it has to write to that frame and $F \geq R$, indicating that the software-driven sequential iteration over memory pages has not yet reached frame F. If $F < R$, then the frame has already been included in the snapshot, and can be modified without copying it to the CoW area.

The software components indicate to the hardware that a new page is to be included in the snapshot by writing the frame number R of that page into the index register. The memory controller first checks whether this page frame is in the CoW area. If it exists, the hardware signs and copies the page from the CoW area into the snapshot. The memory controller can then reuse the space occupied by this page in the CoW area to copy other pages from the DRAM cache. If the page frame is not in the CoW area, the memory controller checks to see if it already exists in the DRAM cache. If not, it brings the page from off-chip DRAM into the DRAM cache. The hardware then signs and copies the page into the snapshot from the DRAM cache area.

The memory controller's use of the CoW area helps achieve snapshot consistency. Snapshot acquisition starts at the instant when the trigger register contains a non-zero value. According to our definition of consistency, the snapshot must contain a copy of off-chip DRAM memory as of that instant. The hardware keeps track of the progress of snapshot acquisition using the index register. The hardware and software components also interact using the index register (see Section 3.3). If any memory page that is not yet in the snapshot is to be modified, the memory controller creates a copy of the original page in the CoW area. This ensures that concurrently-executing applications can make progress, while still maintaining the original copies of memory pages for a consistent snapshot. Snapshot acquisition can proceed in parallel with other user applications as long as there is space in the CoW area to accommodate copies of pages that are not yet in the snapshot. The hardware pauses other user applications during snapshot acquisition only when the CoW area becomes full. In this case, the hardware can resume these applications when space is available in the CoW area, *i.e.,* when a page from there is copied to the snapshot.

```
1  void snapshot_driver (ulong nonce) {
2    char *plocal;
3
4    /* Initialize snapshot acquisition */
5    %index_reg &= 0x0;
6    plocal = kmalloc(SNAPSHOT_ENTRY_SIZE);
7    %snapshot_reg = virt_to_phys(plocal);
8    %trigger_reg = nonce;
9
10   /* Iterate over pages, creating snapshot entries */
11   while (%index_reg < NUM_FRAMES*PAGE_SIZE) {
12     while (%index_reg & 0x1 == 0);
13     write_out(plocal, SNAPSHOT_ENTRY_SIZE);
14     %index_reg &= 0xFFF...FFE;
15     %index_reg += PAGE_SIZE;
16   }
17   %trigger_reg &= 0x0;
18   write_out(plocal, SNAPSHOT_ENTRY_SIZE);
19   kfree (plocal);
20 }
```

Figure 5: Pseudocode of the kernel-mode snapshot driver.

The snapshot itself is simply a sequential list of entries (see Figure 6). Each entry has three components: ① a frame number, F (64-12=52 bits); ② the contents of that page frame in off-chip DRAM, C (4KB); ③ a digital signature of $\mathsf{Hash}(F||N||C)$, where N is the value stored in the trigger register. N is supplied by the forensic analysis obtaining the snapshot, and serves as a nonce to ensure freshness of the snapshot. We use SHA-256 as our hash function, which outputs a 32-byte hash value. The size of the digital signature depends on the key length used by the hardware. For instance, a 1024-bit RSA key would produce a 86-byte signature for a 32-byte hash value with OAEP padding. The hardware directly reads F, C and N from the on-chip components, computes the hash and digital signature using its private key, and stores this entry in the physical page frame referenced by the *snapshot register* (P in Figure 4(b)). The software components then externalize each entry, either by writing it to the disk or the network, and instruct the hardware to compute the next entry in the snapshot.

Snapshot acquisition ends when the trigger register is set back to zero. When the value of the trigger register changes back to zero, the hardware copies the values of all machine registers into the page frame referenced by the snapshot register and signs it. This entry is recorded as the last entry in the snapshot. This feature helps record the CPU register state of the target and include it in the snapshot.

3.3 Software Design

The kernel-mode snapshot driver coordinates with the hardware to obtain the snapshot. It is not trusted and any attempts by the driver to corrupt the snapshot can be detected by the checks described in Section 3.4. Figure 5 shows the pseudocode of the snapshot driver. A user-mode process invokes the driver with a random nonce via a system call.

The first job of the snapshot driver is to initialize the index, snapshot and trigger registers for snapshot acquisition (lines 5–8). It starts on line 5 by setting the index register to the first frame of off-chip DRAM memory.[1] Next, it allocates a buffer (henceforth called the `plocal` buffer) that is the size of one snapshot entry. This buffer serves as the temporary storage area in which the hardware stores entries of the snapshot before they are committed to an external medium. It then obtains and stores stores the physical address of `plocal` in the snapshot register, *i.e.,* the snapshot register contains the physical address translation of the virtual address `plocal`. The hardware uses this physical address to store computed snapshot entries into the `plocal` buffer. Pages allocated using `kmalloc` cannot be moved, thereby ensuring that the buffer is in the same location for the duration of the snapshot driver's execution. (If the page moves, *e.g.,* because of a malicious implementation of `kmalloc`, the snapshot will not pass the checks in Section 3.4). The driver then sets the trigger register to the random nonce on line 8, indicating to the hardware that it can begin snapshot acquisition.

In the loop on lines 11–16, the snapshot driver iterates over all off-chip DRAM page frames. Each iteration of the loop body processes one page frame. The hardware begins processing the first page of DRAM as soon as trigger register is set on line 8, and storing the snapshot entry for this page in the physical memory location referenced by the snapshot register, *i.e.,* the `plocal` buffer. On line 12, the driver waits for the hardware to complete this operation. The hardware informs the driver that the `plocal` buffer is ready with data by setting the least-significant bit of the index register. The driver then commits the contents of this buffer to an external medium, such as persistent storage, a network interface, or a diagnostic serial port. We denote this operation using `write_out` on line 13. The driver then resets the least-significant bit on line 14 before iterating to the next page. Note that the time taken to execute this loop depends on the number of page frames in off-chip DRAM and the speed of the external storage medium. Other applications can concurrently modify memory as this loop executes, but the resulting snapshot will still be consistent.

On line 17, the driver resets the trigger register. As previously discussed, when this happens, the hardware copies and signs all machine registers and stores them in the `plocal` buffer. On line 18, the driver records this as the last entry in the snapshot, thereby resulting in a snapshot that has the structure shown in Figure 6.

[1] Even though the index register stores a full physical address, the hardware only uses the most-significant bits of the address that store the page frame number, akin to standard paging hardware.

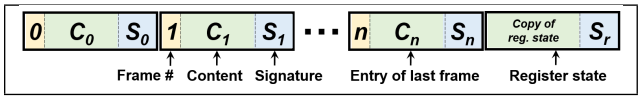


Figure 6: Structure of a memory snapshot produced by the snapshot driver in Figure 5.

3.4 Snapshot Integrity and Security Analysis

Our system's goal is reliable snapshot acquisition from a target machine. Therefore, in our security analysis below, we focus on attacks that compromise snapshot acquisition. Such attacks are defined to be those that manipulate the snapshot so that it does not faithfully represent the memory state of the machine as of line 8 in Figure 5. The target's software, including its OS itself, may be compromised, but we do not consider these to be attacks on the snapshot acquisition system. Indeed, one of the goals of snapshot acquisition is to perform forensic analysis, which determines if the target's software is compromised.

As discussed in the threat model, we aim to *detect* attacks on snapshot acquisition. Thus, we require a forensic analyst to run two basic checks on the snapshot. If the checks succeed, then we can guarantee that snapshot acquisition proceeded free of attack:

① Completeness. The snapshot should contain one entry for each page frame in off-chip DRAM. This criterion ensures that malicious software on the target (*e.g.,* a malicious snapshot driver) cannot suppress memory pages from being included in the snapshot. Each snapshot entry is created by the hardware, by directly reading the frame number and page contents from 3D-stacked memory, thereby ensuring that these entities are correctly recorded in the entry. Thus, if the snapshot has one entry per frame of off-chip DRAM, the snapshot completely captures the memory state of the target machine.

② Per-entry Integrity. The digital signature in each entry must correspond to the frame number and the contents in that entry. Since the digital signature is computed by the hardware, this check catches any attempts by a malicious target to change snapshot entries after they have been computed by the hardware. Recall that the signature also incorporates the nonce supplied via the trigger register. The nonce ensures freshness of the snapshot.

If the snapshot passes these two checks, the memory controller's CoW feature guarantees that the snapshot is a copy of off-chip DRAM state as of the execution of line 8 in Figure 5. The memory controller mediates all writes to DRAM, whether from the MMU or the IOMMU, thereby ensuring that even if memory is modified, the snapshot consistency is not compromised. A malicious kernel may try to interfere with the execution of the snapshot driver by tampering with the execution of the loop in lines 11–16 of Figure 5. It could do so in a number of ways, all of which are detectable using the checks above. For example, if it modifies the value of `index_reg`, some memory pages will not appear in the snapshot and the Completeness criterion will be violated. If it instead tampers with the contents of the `plocal` buffer, the Per-entry Integrity condition will be violated. It could instead modify the virtual address `plocal` or the corresponding physical address stored in `snapshot_reg`. If the modified virtual address and/or physical address do not map to each other, the `write_out` in line 13 will not record the values of some snapshot entries, thereby violating Completeness. If it modifies the value of `trigger_reg`, it will either change the nonce or stop snapshot acquisition (if the new value is 0). In either case, either Per-entry Integrity or Completeness is violated. In summary, any malicious attempts by the kernel to modify the values of the index, snapshot or trigger registers between lines 8–16 will be detectable. Via the same argument, if a malicious kernel induces a race condition during register initialization on lines 5–8, the resulting snapshot will violate Completeness or Per-entry Integrity.

The snapshot driver relies on two kernel functions, `kmalloc` and `virt_to_phys`, on lines 6–7. However, we do not have to trust these functions. If `kmalloc` fails to allocate a page, snapshots cannot be obtained from the target, resulting in a detectable denial-of-service attack. If the pages allocated by `kmalloc` are remapped during execution or `virt_to_phys` does not provide the correct virtual to physical mapping for the allocated space, the snapshot will contain missing entries, thereby violating Completeness.

Although a snapshot that satisfies Completeness and Per-entry Integrity faithfully represents the memory state of the target, we do not offer a similar guarantee for the state of CPU registers recorded in the snapshot. A malicious kernel could modify CPU registers (such as `cr3`, the page-table base register on x86 machines) between lines 8–18 of Figure 5. This is possible because the design of the memory controller only ensures the consistency of memory state and not register state. At first glance, it may appear that a possible workaround for this problem is to capture the target's register state on line 8. However, even if this were hypothetically possible, a malicious kernel could simply have modified the registers even before the execution of line 8. Thus, the situation is no worse even if we record the register state in line 18 instead of line 8. Indeed, a malicious kernel could have modified the CPU registers *even before the snapshot driver begins execution.* Hence, we do not consider this scenario to be an attack on memory snapshot acquisition *per se.*

3.5 Address-Translation Attacks and Defense

Given the discussion above, it is natural to ask why we even record the register state in the snapshot. The answer is that register state, in particular the value of the page-table base register (PTBR) is important for downstream analysis of the snapshot. Many forensic analyses, for instance, work by recursively traversing pointer values that appear in memory pages [8, 15, 20, 60–62, 67]. These pointers are virtual addresses but the snapshot contains physical page frames. Thus, the forensic analysis translates pointers into physical addresses by consulting the page table, which it locates in the snapshot using the PTBR.

Prior work has developed *address-translation redirection attacks* (ATRA) attacks, in which a malicious kernel modifies the PTBR [41, 47] or even individual page-table entries [41] to defeat forensic analysis of the snapshot. The main idea behind these

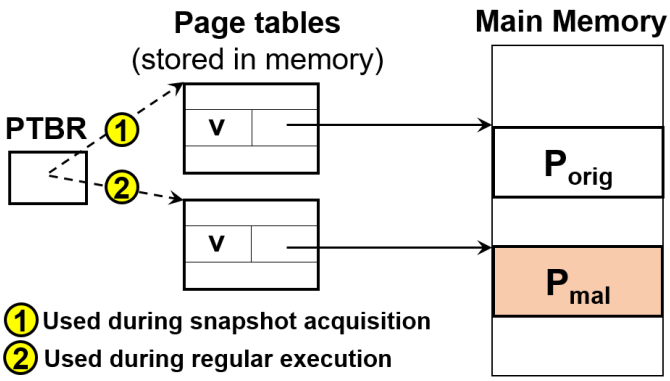


Figure 7: An address-translation redirection attack [41, 47]. *Our hardware-based defense works by checking the integrity of page table entries corresponding to kernel code and data for each PTBR change.*

attacks is to hide the presence of malicious code and data modifications by making them unreachable from the page tables linked from the PTBR. For example, consider a situation where an attacker maliciously modifies a kernel data structure, such as the system-call table or the process list. The attacker's goal is to get kernel execution to use the modified data structure but to hide it from forensic analysis. To do so, the attacker creates two copies of the page containing the data structure, as shown in Figure 7. P_{orig} contains the unmodified data structure, while P_{mal} contains the modified data structure. The attacker then creates then two copies of the page table, one with virtual addresses mapping to P_{orig} and the second mapping them to P_{mal}. He switches between these page tables by changing the PTBR. During regular execution, he uses the page-table that resolves mappings to P_{mal}. During snapshot acquisition, he changes the PTBR to point to the page table that resolves mappings to P_{orig}. Even though both P_{orig} and P_{mal} and both page tables appear in a complete snapshot of the target's memory, the forensic analysis will never analyze P_{mal} because the data in it is unreachable via recursive traversal of pointers from the kernel's entrypoints. Jang *et al.* [41] demonstrate variants of this attack that work by modifying the PTBR, the page directory, or individual entries in multi-level page tables. This leads to a WYSINWYX[2] scenario where the content that the forensic analysis sees in the snapshot is not what actually executes on the system. External hardware-based snapshot acquisition systems are especially vulnerable to ATRA attacks because they have no visibility of CPU registers such as the PTBR.

We describe a simple hardware-based defense that protects kernel code and data from ATRA attacks. Our defense is based on three observations that we empirically find to hold on modern commodity OSes: ① the page table for every process created on the system has mappings for physical memory pages that store kernel code and data; ② the kernel is mapped to the same virtual address range of every process; and ③ physical page frames storing kernel code and data are not swapped to disk. Together, these observations imply that *the virtual to physical mappings for kernel code and data must be the same in the page tables of all processes on the target.*

Our hardware-based defense enforces this invariant whenever the PTBR is changed to point to a new page table. During boot time, the hardware computes the expected values of page table mappings for kernel code and data, and stores it in an internal buffer, inaccessible from software. For each change of the PTBR, the hardware checks the page table to ensure that the kernel mappings match those that it has precomputed. The hardware also stores a hash of these mappings to the last entry in the snapshot on lines 17–18 of Figure 5 (with register state). Kernel page allocations/deallocations can add/delete mappings in the page table. However, these mappings must still be the same across all processes and the hardware enforces this invariant. This defense works even with multi-level page tables. The hardware only needs to check the integrity of the kernel's mappings in the top-level page table (the page global directory), which the PTBR directly references. Lower-level page tables are recorded in the snapshot, and any malicious entries in these page tables can be detected by applying the same invariant above during forensic analysis of the snapshot [41].

Note that this defense does *not* ensure that kernel code and data are free from rootkit-based attacks. An attacker could launch *transient* attacks on individual process page tables, between consecutive changes of the PTBR. For example, the attacker could exploit a buffer overflow in a system call to run code with kernel-mode privileges. The attacker could even place malicious code on unmapped physical memory pages, and modify the page table of the process via a kernel buffer overflow to include mappings for these memory pages. These attacks will be successful because in our defense, the hardware only checks the PTBR during context switches.

However, this defense does complement our system for snapshot acquisition. It ensures that the PTBR recorded in the snapshot points to a page table with the same mappings that were recorded when the kernel first booted on the target. The analyst verifies this by checking that the kernel's mappings in that page table correspond to the hash of the entries stored by the hardware in the last snapshot entry. It also ensures that this invariant holds every time the PTBR is modified on a context-switch. Thus, it avoids the WYSINWYX scenario illustrated in Figure 7 from being used in a *persistent* kernel attack; any WYSINWYX scenarios must be transient. The forensic analyst obtains the guarantee that the page table mappings in the snapshot are those that will be loaded into the address space of a process during each context switch. This allows forensic

[2] "What you see is not what you execute," an abbreviation originally introduced for use in another context [7].

① **Canneal**	Multi-threaded benchmark from PARSEC [9]
② **Memcached**	In-memory key-value store [55]
③ **Graph500**	Graph-processing benchmark [30]
④ **Mcf**	Single-threaded benchmark from SPEC 2006 [71]

Figure 8: Description of benchmark user applications.

analysis of the snapshot to correctly reason about kernel code and data structures as they appear in a process address space following a context switch. This defense also illustrates the power of an on-chip approach, with ready access to observe changes to the PTBR.

4. EXPERIMENTAL METHODOLOGY

Evaluation Infrastructure. As 3D-stacked memories on-package with processors are in their infancy, the architecture community has almost exclusively focused on simulation-based studies to study their design. Our work however requires careful full-system analysis (including the OS and driver stack), and an assessment of how our snapshot mechanism affects long-running, real-world applications with realistic memory footprints. We have therefore gone beyond simulation and created a novel modeling infrastructure that takes an existing hardware platform, and through memory contention, creates two different speeds of DRAM. This is partly inspired by recent work [59], but extends it by enhancing a stock Linux distribution to page between these DRAM devices.

Our infrastructure consists of a two-socket Xeon E5-2450 processor, equipped with a total of 32GB of memory, running Debian-sid with Linux kernel 4.4.0. The system supports 8 cores per socket, each two-way hyperthreaded, for a total of 16 logical cores per socket. Each socket has two DDR3 DRAM memory channels. To emulate 3D-stacking, we do the following. We dedicate the first socket for execution of our user applications, our kernel-level snapshot driver, and our user-level snapshot process. This first socket hosts our "fast" or 3D-stacked memory. The second socket is dedicated to host our "slow" or off-chip DRAM. Further, the cores on the second socket are calibrated with contention (using the memory contention benchmark `memhog` [59]) such that the emulated 3D-stacked memory or DRAM cache is 4.5× faster compared to the emulated off-chip DRAM (or in reality, the slow memory has been slowed down by 4.5×). This provides a similar memory bandwidth performance ratio of a 51.2GBps off-chip memory system compared to a 256GBps of 3D-stacked memory, consistent with the expected performance ratios of real-world 3D-stacking [52, 59].

In tandem, we modify the Linux kernel to support paging between our emulated fast and slow memory. While we begin our implementation by modifying existing `libnuma` [59] patches, we have modified the kernel to page using LRU policies between the memory devices on the two sockets. This paging support emulates what would otherwise be purely hardware paging via our enhanced memory controller; we model the timing aspects of paging to faithfully reproduce the performance that our actual system with an enhanced memory controller would sustain.

We emulate the overhead of marshaling snapshots and signatures to an external medium by introducing artificial delays in our user-level snapshot process. We vary the delay based on several emulated external media, ranging from fast network connections to (relatively) slower SSD devices.

Hardware Estimates. We use well-established tools from the architecture community to assess the area, energy, and access of our proposed hardware for the processor, the memory controller, the near-memory logic for the SHA-256 hash and signatures, and for the comparator logic pertaining to the address-translation attacks and defense. Specifically, we use CACTI [57], a well-known tool for modeling SRAM and DRAM memories, and Aladdin [70], a pre-RTL, power-performance accelerator modeling framework to assess the overheads of our hashing, signing, and comparator logic.

Workloads. Finally, we study the performance implications of our approach by quantifying snapshot overheads on several big-memory applications. These workloads are known to be sensitive to 3D-stack memory design, and hence represent a "worst-case" performance scenario for our approach; that is, by establishing that memory acquisition only minimally degrades the performance of these applications, we can establish that our approach would be even lower-overhead for less memory-intensive applications.

Figure 8 shows our benchmarks, ranging from single- and multi-threaded benchmarks to modern big-memory workloads such as key/value stores and graph processing workloads. All benchmarks are configured to have memory footprints in the range of 12-14GB, so that they exceed the capacity of the maximum 3D-stacked memory we emulate (8GB) while also fitting in the bigger off-chip DRAM (16GB). To realistically gauge the performance implications of our approach, we run each application several times.

5. EVALUATION

Our evaluation answers four questions:

(**Q1**) What is the impact of snapshot acquisition on the performance of other user applications executing on the target machine?

(**Q2**) How quickly can we obtain snapshots from the target?

(**Q3**) What are the hardware overheads from the additional memory controller logic, processor registers, and near-memory processing for SHA-256 hashing and cryptograhic signatures?

(**Q4**) What is the performance impact of our ATRA defense?

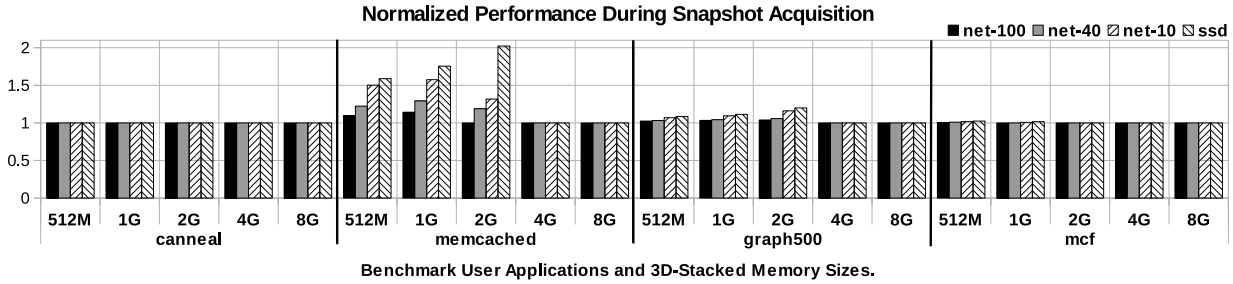


Figure 9: Performance impact of snapshot acquisition. *This chart reports the observed performance of user applications executing on the target during snapshot acquisition, normalized against their observed performance during regular execution, i.e., no snapshot acquisition. For each of the four benchmarks, we report the performance for various sizes of 3D-stacked memory (= 2×CoW area size), and for different methods via which the* `write_out` *in Figure 5 writes out the snapshot.*

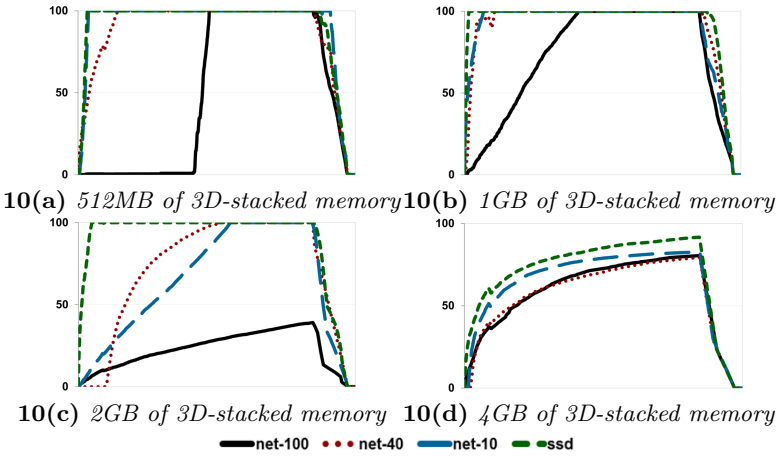


Figure 10: CoW area utilization over time for memcached. *The y-axis shows the percentage of the CoW area utilized to store page frames that have not yet been included in the snapshot as the memcached application executes on the target, while the x-axis denotes time. We measured CoW utilization for every 1024 snapshot entries recorded by the system. The four charts show CoW utilization trends for various sizes of 3D-stacked memory (= 2×CoW area size). Snapshot acquisition does not impact memcached performance when CoW utilization is below 100%.*

To answer **Q1**, we measured each benchmark's performance as we concurrently collected a snapshot from the target. We then normalized this number against the baseline performance of the benchmark when no snapshots are collected from the target. All the numbers reported in this section are averaged over five executions of each benchmark.

Figure 9 shows the result of this experiment for five sizes of 3D-stacked memory (512MB, 1GB, 2GB, 4GB and 8GB); half of this memory is used by the CoW area during snapshot acquisition. We considered four methods to externalize snapshots: a NIC with a throughput of 100Gbps, one with 40Gbps, one with 10Gbps, and a solid-state storage disk (SSD) with sequential write throughput of 900MBps. Larger CoW areas offer more room to store pages that have not yet been included in the snapshot. Faster methods to externalize snapshot entries allow the CoW area to drain quicker. Some of the configuration points that we discuss are not yet in wide commercial use. For example, the AMD Radeon R9, a top-of-the-line chipset series (GPUs) supports only up to 4GB of 3D-stacked memory. Similarly, 40Gbps and 100Gbps NICs, while currently available, are expensive and not yet in widespread use. We still consider these data points to illustrate the impact of technologies that will see widespread adoption in the near future.

This chart shows that the performance of canneal and mcf is *unaffected* during snapshot acquisition, even in the configuration that uses the smallest amount of 3D-stacked memory and the slowest method to externalize snapshots (SSDs). Snapshot acquisition does reduce the performance of memcached and graph500 for 3D-stacked memory sizes under 2GB in size. The highest overhead that we observed was for memcached when we equipped the target with 2GB of 3D-stacked memory and used SSDs to externalize snapshots—in this case, the observed throughput of memcached was cut in half. However, the performance of both memcached and graph500 was equal to their baseline versions when the target was equipped with more 3D-stacked memory. Since upcoming 3D stacked memory capacities are expected to comfortably span 4-8GB [59], we continue to expect good performance.

As discussed in Section 3, the performance of a benchmark application suffers during snapshot acquisition only if the CoW area fills to capacity. In this case, the benchmark is unable to make progress until some pages from the CoW area are copied

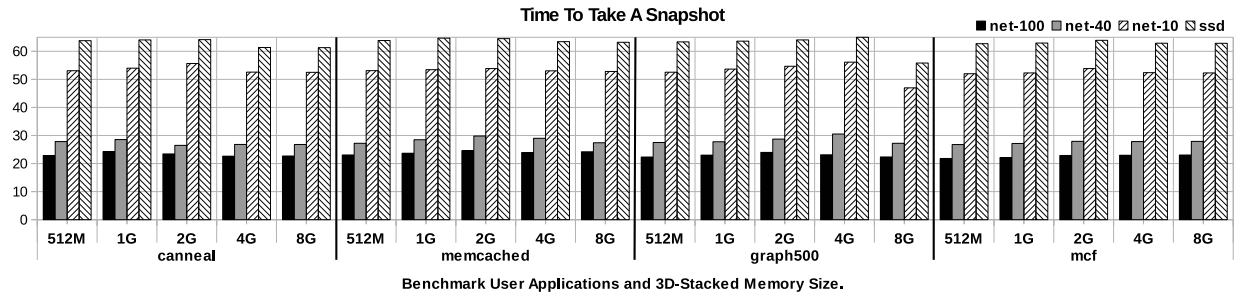


Figure 11: Time to acquire a snapshot from a machine with 32GB of off-chip DRAM. *We report the observed time (in seconds) as various user applications execute concurrently during snapshot acquisition for various sizes of 3D-stacked memory and different methods to write out the snapshot.*

to the snapshot, thereby creating space for more pages and allowing the benchmark to resume. Figure 10 illustrates this fact, and explains the performance of memcached. We elide charts for other benchmarks because they illustrate the same trend. Figure 10 shows the fraction of the CoW area utilized over time during the execution of memcached. The fraction of time for which the CoW area is at 100% directly corresponds to the observed performance of memcached. When CoW utilization is below 100%, as is the case in Figure 10(d) or with the 100Gbps NIC in Figure 10(c), the performance of memcached is unaffected.

Note that Figure 9 shows that the performance overheads for memcached and graph500 *increase* as 3D-stacked memory increases from 512MB to 2GB, whereas one would expect overheads to reduce when the target is equipped with more 3D-stacked memory. This is because of two reasons. First, as more 3D-stacked memory is available, even the baseline performance of the benchmark, *i.e.,* without snapshot acquisition, improves. Second, during snapshot acquisition, the amount of 3D-stacked memory available for use as a DRAM cache is halved (Figure 4), resulting in slower benchmark performance. These two reasons, combined with the fact that Figure 9 shows *normalized* performance, *i.e.,* a ratio against the baseline, explain the trend.

To answer **Q2**, we measured the total time taken to obtain a complete snapshot of the 32GB of installed DRAM on our target system. We did this experiment with the same setup as before, with user application benchmarks executing during snapshot acquisition. Figure 11 shows the results of this experiment. The main takeaway from this chart is that the method used to externalize the snapshot is the only variable that impacts snapshot acquisition time. Writing out a full snapshot over the network with a 100Gbps NIC takes under 25 seconds in all cases, while with our slowest medium, a 900MBps SSD, it takes about 65 seconds.

To answer **Q3**, we use Aladdin to assess the area, timing, and energy overheads of our hardware enhancements. For example, we find that the three additional registers present roughly 1% more area per core, and negligible changes in the access time and energy of the register file. We then use CACTI and Aladdin to model the hardware block used to realize the finite-state machine that marshals data movement from the off-chip DRAM to the 3D-stacked memory, one page at a time. Our hardware enhancements adds roughly 2% more area to a state-of-the-art on-chip demand-first FR-FCFS scheduling [58, 64], with negligible changes to the area and cycle times. We similarly quantify the area overheads for the SHA-256 hash and the hash signature logic. We find that the total area on the 3D-stack for this logic takes up less than 1% of a 512MB 3D stacked memory. Given its small area, its operation also occurs within the cycle time of a 3D-stacked memory access.

Finally, we answer **Q4** using CACTI to model the overheads of the per-core comparator logic. While we model these overheads for an x86 system, the approach and insights apply generally to other architectures with similar forward-mapped page tables (*e.g.,* ARM). Our comparators check 2KB of the page global directory in our four-level page table. We leverage the following insight—Our comparators need to complete their operation in the same amount of time necessary for a TLB flush operation, which occurs on a PTBR switch. Linux kernel developers periodically quantify TLB flush time using microbenchmarks and currently believe this to take 100ns. In comparison, our comparator network takes 6ns to perform the page table check; since this is in parallel with the TLB flush, it imposes no additional overhead. Further, it requires merely 4% of the area of an L1 cache and imposes negligible additional energy as it is active on only PTBR switches.

6. RELATED WORK

Memory Acquisition Technologies. As Section 1 discusses, there is much prior work on remote memory acquisition. In contrast to virtualization-based solutions (starting with the work of Garfinkel and Rosenblum [27] and numerous other papers over the past decade that we do not cite here), we offer a hardware TCB and performance isolation for the target's applications. However, even virtualized solutions can adapt our CoW-based scheme to obtain performance isolation. In contrast to solutions that use trusted hardware [1, 5, 28, 73] or the processor's system management mode (SMM) [6, 66, 78], we do not require the processor to enter a special mode to obtain snapshots, thereby offering performance isolation. And in contrast to external hardware-based solutions [4, 14, 40, 48, 49, 56, 62], our system offers both snapshot consistency and better security because of the ability to inspect register state, thereby preventing WYSINWYX scenarios (Section 3.5). Aside from these, there are other

mechanisms to fetch memory snapshots for the purpose of debugging (*e.g.,* [29, 33, 44, 72, 74]). Because their focus isn't forensic analysis, these systems do not assume an adversarial target OS.

3D Architectures for Security. There are two broad strands of work on using 3D manufacturing technology for security. The first strand of work focuses on leveraging 3D-stacking to implement myriad security features such as monitoring program execution, access control and cryptography [36–38, 54, 75–77]. This strand of work observes that 3D-stacking allows processor vendors to decouple core CPU logic from "add-ons," such as security, thereby improving their chances of deployment. Our work follows this strand in that we also leverage additional circuitry on the 3D-stack to implement the logic needed for memory acquisition. Unlike prior work, which focused solely on additional processing logic integrated using 3D-stacking, our focus is also on 3D-stacked memory, which is beginning to see deployment in commercial processors. While our work also uses the 3D-stack to integrate additional cryptographic logic and modify the memory controller, we do so to enable near-data processing on the contents of 3D-stacked memory.

The second strand of work focuses on using 3D manufacturing technology to detect malicious logic inserted into the processor. The threat model here is that an outsourced chip manufacturer can insert Trojan-horse logic into the hardware. This strand of work suggests various methods to combat this threat using 3D-stacked manufacturing. For example, one method is to divide the implementation of a circuit across multiple layers in the stack, each manufactured by a separate agent, thereby obfuscating the functionality of individual layers [39, 68]. Another method is to add logic into 3D-stacked layers to monitor the execution of the processor and detect the effects of maliciously-inserted logic [10–12]. We view this strand of work as being orthogonal to ours.

Finally, although not directly related to 3D-stacking, there is prior work on near-data processing to enable security applications (*e.g.,* authenticated data structures [31]) and on modifying memory controllers to implement a variety of security features [69, 79].

7. CONCLUSION

Processor vendors are increasingly beginning to integrate memory and processing logic on-chip using 3D-stacked manufacturing technology. We have presented a novel application of this technology to secure memory acquisition. Our system has a hardware TCB, and allows forensic analysts to collect consistent memory snapshots from a target machine while offering performance isolation for applications executing on the target. We also demonstrate the benefits of a 3D-stacked approach to security by building a novel defense against address-translation attacks. Our experimental evaluation on a number of data intensive workloads shows the benefit of our approach.

Acknowledgments. We thank Liviu Iftode for his comments that helped sharpen the arguments in the paper. This work was supported in part by NSF grants 1253700, 1337147, and 1441724.

REFERENCES

[1] ARM security technology – Building a secure system using TrustZone technology, 2009. ARM Technical Whitepaper. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[3] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[4] The InfiniBand Trade Association. The InfiniBand^{TM} architecture specification. http://www.infinibandta.org.

[5] A. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across Worlds: Real-time kernel protection from the ARM TrustZone secure world. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[6] A. Azab, P. Ning, Z. Wang, X. Jiang, and X. Zhang. HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[7] G. Balakrishnan and T. Reps. *ACM Transactions on Programming Languages and Systems*, 32(8), August 2010.

[8] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5), 2011.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. 2008.

[10] M. Bilzor. 3D execution monitor (3D-EM): Using 3D circuits to detect hardware malicious inclusions in general purpose processors. In *6th International Conference on Information Warfare and Security*, 2011.

[11] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Security checkers: Detecting processor malicious inclusions at runtime. In *IEEE International Symposium on Hardware-oriented Security and Trust*, 2011.

[12] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage. In *IEEE International Symposium on Hardware-oriented Security and Trust*, 2012.

[13] B. Black, M. Annavaram, E. Brekelbaum, J. DeVale, L. Jiang, G. Loh, D. McCauley, P. Morrow, D. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. P. Shen, and C. Webb. Die stacking 3D microarchitecture. In *International Symposium on Microarchitecture (MICRO)*, 2006.

[14] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote repair of operating system state using backdoors. In *International Conference on Autonomic Computing (ICAC)*, 2004.

[15] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[16] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis. Improving SHA-2 hardware implementations. In *IACR International Cryptology Conference (CRYPTO)*, 2006.

[17] C.-C. Chou, A. Jaleel, and M. K. Qureshi. CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *International Symposium on Microarchitecture (MICRO)*, 2012.

[18] F. J. Corbato. A paging experiment with the Multics system, May 1968. MIT Project MAC Report MAC-M-384.

[19] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *International Conference on Software Engineering (ICSE)*, 2016.

[20] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security Symposium*, 2012.

[21] CVE-2007-4993. Xen guest root escapes to dom0 via pygrub.

[22] CVE-2007-5497. Integer overflows in libext2fs in e2fsprogs.

[23] CVE-2008-0923. Directory traversal vulnerability in the shared folders feature for VMWare.

[24] CVE-2008-1943. Buffer overflow in the backend of XenSource Xen paravirtualized frame buffer.

[25] CVE-2008-2100. VMWare buffer overflows in VIX API let local users execute arbitrary code in host OS.

[26] Q. Feng, A. Prakash, H. Yin, and Z. Lin. MACE: High-coverage and robust memory analysis for commodity operating systems. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[27] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, 2003.

[28] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing kernel code integrity on the TrustZone architecture. In *IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[29] Google. Using DDMS for debugging. http://developer.android.com/tools/debugging/ddms.html.

[30] Graph500. http://www.graph500.org.

[31] A. Gundu, A. S. Ardestani, M. Shevgoor, and R. Balasubramonian. A case for near data security. In *3rd Workshop on Near Data Processing*, 2014.

[32] M. Healy, K. Athikulwongse, R. Goel, M. Hossain, D. H. Kim, Y. Lee, D. Lewis, T. Lin, C. Liu, M. Jung, B. Ouellette, M. Pathak, H. Sane, G. Shen, D. H. Woo, X. Zhao, G. Loh, H. Lee, and S. Lim. Design and analysis of 3D-MAPS: A many-core 3D processor with stacked memory. In *IEEE Custom Integrated Circuits Conference (CICC)*, 2010.

[33] A. P. Heriyanto. Procedures and tools for acquisition and analysis of volatile memory on Android smartphones. In *11th Australian Digital Forensics Conference*, 2013.

[34] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[35] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. Keckler. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[36] T. Huffmire, T. Levin, M. Bilzor, C. Irvine, J. Valamehr, M. Tiwari, and T. Sherwood. Hardware trust implications of 3-D integration. In *Workshop on Embedded Systems Security*, 2010.

[37] T. Huffmire, T. Levin, C. Irvine, R. Kastner, and T. Sherwood. 3-D extensions for trustworthy systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2011.

[38] T. Huffmire, J. Valamehr, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. Trustworthy system security through 3-D integrated hardware. In *International Workshop on Hardware-oriented Security and Trust*, 2008.

[39] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara. Securing computer hardware using 3D integrated circuit technology and split manufacturing for obfuscation. In *USENIX Security Symposium*, 2013.

[40] Mellanox Technologies Inc. Introduction to InfiniBand, September 2014. http://www.mellanox.com/blog/2014/09/introduction-to-infiniband.

[41] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. Kang. ATRA: Address translation redirection attack against hardware-based external monitors. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[42] N. Jerger, A. Kannan, Z. Li, and G. Loh. NoC architectures for silicon interposer systems. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[43] D. Jevdjic, G. Loh, C. Kaynak, and B. Falsafi. Unison cache: A scalable and effective die-stacked DRAM cache. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[44] Joint Test Action Group (JTAG). 1149.1-2013 - IEEE Standard for test access port and boundary-scan architecture, 2013. http://standards.ieee.org/findstds/standard/1149.1-2013.html.

[45] K. Kortchinsky. Hacking 3D (and breaking out of VMWare). In *BlackHat USA*, 2009.

[46] A. Kannan, N. Jerger, and G. Loh. Enabling interposer-based disintegration of multi-core processors. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[47] Y. Kinebuchi, S. Butt, V. Ganapathy, L. Iftode, and T. Nakajima. Monitoring system integrity using limited local memory. *IEEE Transactions on Information Forensics and Security*, 8(7), 2013.

[48] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. Kang. KI-Mon: A hardware-assisted event-triggered monitoring platform for mutable kernel objects. In *USENIX Security Symposium*, 2013.

[49] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi. CPU-transparent protection of OS kernel and hypervisor integrity with programmable DRAM. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[50] G. Loh. 3D-stacked memory architectures for multi-core processors. In *International Symposium on Computer Architecture (ISCA)*, 2008.

[51] G. Loh. Extending the effectiveness of 3D-stacked DRAM caches with an adaptive multi-queue policy. In *International Symposium on Microarchitecture (MICRO)*, 2009.

[52] G. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[53] A. Markuze, A. Morrison, and D. Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[54] D. Megas, K. Pizolato, T. Levin, and T. Huffmire. A 3D data transformation processor. In *Workshop on Embedded Systems Security*, 2012.

[55] Memcached. https://memcached.org.

[56] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. Kang. Vigilare: Toward a snoop-based kernel integrity monitor. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[57] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[58] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[59] M. Oskin and G. Loh. A software-managed approach to die-stacked DRAM. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[60] N. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX Security Symposium*, 2006.

[61] N. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[62] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot: A coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[63] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *International Symposium on Microarchitecture (MICRO)*, 2012.

[64] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. Memory access scheduling. In *International Symposium on Computer Architecture (ISCA)*, 2000.

[65] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition, part I: AMD case. In *Blackhat Conf.*, 2007.

[66] J. Rutokowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. In *Blackhat Briefings USA*, 2008.

[67] K. Saur, M. Hicks, and J. S. Foster. C-Strider: Type-aware heap traversal for C. *Software, Practice, and Experience*, May 2015.

[68] Tezzaron Semiconductors. 3D-ICs and integrated circuit security, 2008. http://www.tezzaron.com/media/3D-ICs_and_Integrated_Circuit_Security.pdf.

[69] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[70] Y. Shao, B. Reagen, G-Y. Wei, and D. Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling design space exploration of customized architectures. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[71] Spec. https://www.spec.org/cpu2006/.

[72] A. Stevenson. Boot into recovery mode for rooted and un-rooted Android devices. http://androidflagship.com/605-enter-recovery-mode-rooted-un-rooted-android.

[73] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. TrustDump: Reliable memory acquisition on smartphones. In *European Symposium on Research in Computer Security (ESORICS)*, 2014.

[74] J. Sylve, A. Case, L. Marziale, and G. G. Richard. Acquisition and analysis of volatile memory from Android smartphones. *Digital Investigation*, 8(3-4), 2012.

[75] J. Valamehr, T. Huffmire, C. Irvine, R. Kastner, C. Koc, T. Levin, and T. Sherwood. A qualitative security analysis of a new class of 3-D integrated crypto co-processors. In *Cryptography and Security: From Theory to Applications, LNCS volume 6805*, 2012.

[76] J. Valamehr, M. Tiwari, T. Sherwood, R. Kastner, T. Huffmire, C. Irvine, and T. Levin. Harware assistance for trustworthy systems through 3-D integration. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[77] J. Valamehr, M. Tiwari, T. Sherwood, R. Kastner, T. Huffmire, C. Irvine, and T. Levin. A 3-D split manufacturing approach to trustworthy system development. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 32(4), April 2013.

[78] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A hardware-assisted integrity monitor. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2010.

[79] Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing channel protection for a shared memory controller. In *IEEE International Conference on High-performance Computer Architecture (HPCA)*, 2014.