Secure, Consistent, and High-Performance Memory Snapshotting

Guilherme Cox
Rutgers University

Zi Yan
Rutgers University

Abhishek Bhattacharjee
Rutgers University

Vinod Ganapathy
Indian Institute of Science

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. These techniques either sacrifice snapshot consistency or degrade the performance of applications executing atop the target. We present SnipSnap, a new snapshot acquisition system based on on-package DRAM technologies that offers snapshot consistency without excessively hurting the performance of the target's applications. We realize SnipSnap and evaluate its benefits using careful hardware emulation and software simulation, and report our results.

CCS CONCEPTS

• **Security and privacy → Tamper-proof and tamper-resistant designs**; *Trusted computing*; *Virtualization and security*;

KEYWORDS

Cloud security; forensics; hardware security; malware and unwanted software

ACM Reference Format:
Guilherme Cox, Zi Yan, Abhishek Bhattacharjee, and Vinod Ganapathy. 2018. Secure, Consistent, and High-Performance Memory Snapshotting. In *CODASPY '18: Eighth ACM Conference on Data and Application Security and Privacy, March 19–21, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3176258.3176325

1 INTRODUCTION

The notion of acquiring memory snapshots is one of ubiquitous importance to computer systems. Memory snapshots have been used for tasks such as virtual machine migration and backups [4, 19, 21, 23, 31, 34, 39, 45, 63, 71, 94] as well as forensics [18, 81], which is the subject of this paper. In particular, memory snapshot analysis is *the* method of choice used by forensic analyses that determine whether a target machine's operating system (OS) code and data are infected by malicious rootkits [10, 17, 24, 25, 43, 72–74, 80]. Such forensic methods have seen wide deployment. For example, Komoku [72, 74] (now owned by Microsoft) uses analysis of memory snapshots in its forensic analysis, and runs on over 500 million hosts [8]. Similarly, Google's open source framework, Rekall Forensics [2], is used to monitor its datacenters [68]. Fundamentally, all these techniques depend on secure and fast memory snapshot acquisition. Ideally, a memory snapshot acquisition mechanism should satisfy three properties:

① **Tamper resistance.** The target's OS may be compromised with malware that actively evades 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 target machine at a given instant in time. Consistency is important for forensic tools that analyze the snapshot. Without consistency, different portions of the snapshot may represent different points in time during the execution of the target, 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. Unfortunately, none of these solutions achieve all three properties (see Figure 1).

With virtualization-based techniques (pioneered by Garfinkel and Rosenblum [35]), the target is a virtual machine (VM) running atop a trusted hypervisor. The hypervisor has the privileges to inspect the memory and CPU state of VMs, and can therefore obtain a snapshot of the target. This approach has the benefit of isolating the target VM from the snapshot acquisition mechanism, which is implemented within the hypervisor. However, virtualization-based techniques:

- *impose a tradeoff between consistency and performance-isolation.* To obtain a consistent snapshot, 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, which is disruptive if snapshots are frequently required, *e.g.,* when a cloud provider wants to monitor the health of the cloud platform in a continuous manner. The hypervisor could instead allow the target VM to execute concurrently with memory acquisition, but this compromises snapshot consistency.
- *require a substantial software trusted computing base (TCB).* The entire hypervisor is part of the TCB. Production-quality hypervisors have more than 100K lines of code and a history of bugs [26–30, 55, 79] that can jeopardize isolation.
- *are not applicable to container-based cloud platforms.* Virtualization-based techniques are applicable only in settings where the target is 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 that use virtualization. An increasing number of cloud providers are beginning to deploy lightweight client isolation mechanisms, such as those based on containers (*e.g.,* Docker [1]). Containers provide isolation by enhancing the OS. On container-based systems, obtaining a full-system snapshot would require trusting the OS, and therefore placing it in the TCB. However, doing so defeats the purpose of snapshot acquisition if the goal is to monitor the OS itself for rootkit infection.

CODASPY '18, March 19–21, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5632-9/18/03...$15.00
https://doi.org/10.1145/3176258.3176325

Property→ **Method↓**	① *Tamper resistance*	② *Snapshot consistency*	③ *Performance isolation*
Virtualization	✓	Tradeoff: ② ✓ ⇔ ③ ✗	
Trusted hardware	✓	Tradeoff: ② ✓ ⇔ ③ ✗	
External hardware	✗	✗	✓
SnipSnap	✓	✓	✓

Figure 1: Design tradeoffs in snapshot acquisition.

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, ARM TrustZone [5, 9, 36, 85] partitions the processor's execution mode so that the target runs in a deprivileged world ("Normal world"), without access to the snapshot acquisition system, which runs in a privileged world ("Secure 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. The situation is more complicated on a multi-processor TrustZone-based system, because the ARM specification allows individual processor cores to independently transition between the privileged and deprivileged worlds [5, §3.3.5]. Thus, from the perspective of snapshot consistency, care has to be taken to ensure that when snapshot acquisition is in progress on one processor core, all the other cores are paused and do not make concurrent updates to memory. This task is impossible to accomplish without some support from the OS to pause other cores. Trusting the OS to accomplish this task defeats the purpose of snapshot acquisition if the goal is to monitor the OS itself.

External hardware-based techniques use a physically isolated hardware module, such as a PCI-based co-processor (*e.g.,* as used by Komoku [8]), on the target system and perform snapshot acquisition using remote DMA (*e.g.,* [10, 16, 50, 58, 59, 65, 67, 72, 74]). These techniques offer 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. Further, a malicious target OS can easily subvert snapshot acquisition despite physical isolation of the co-processor [78]. Co-processors rely on the target OS to set up DMA. On modern chipsets with IOMMUs, a malicious target OS can simply program the IOMMU to reroute DMA requests away from physical memory regions that it wants to hide from the co-processor (*e.g.,* pages that store malicious code and data). Researchers have also discussed address-translation attacks that leverage the inability of co-processors to view the CPU's page-table base register (PTBR) [51, 56]. These attacks enable malicious virtual-to-physical address translations, which effectively hide memory contents in the snapshot from forensic analysis tools.

Contributions. We propose and realize **Secure and Nimble In-Package Snapshotting** or SnipSnap, a hardware-based memory snapshot acquisition mechanism that achieves all three properties. SnipSnap frees snapshotting from the shackles of the consistency-performance tradeoff by leveraging two related hardware trends—the emergence of high-bandwidth DRAM placed on the same package as the CPU [15, 41, 60, 61], and the resurgence of near-memory processing [6, 7, 44]. Specifically, processor vendors have embraced technologies like embedded on-package DRAM in products including IBM's Power 7 processor, Intel's Haswell, Broadwell, and Skylake processors, and even in mobile platforms like Apple's iPhone [32]. More recently, higher bandwidth on-package DRAM has been implemented on Intel's Knight's Landing chip, while emerging 3D and 2.5D die-stacked DRAM is expected to be widely adopted [60]. On-package DRAM has in turn prompted flurry of research on near-memory processing techniques that place logic close to these DRAM technologies. Consequently, near-memory processing logic for machine learning, graph processing, and general-purpose processing has been proposed [6, 7, 44] for better system performance and energy.

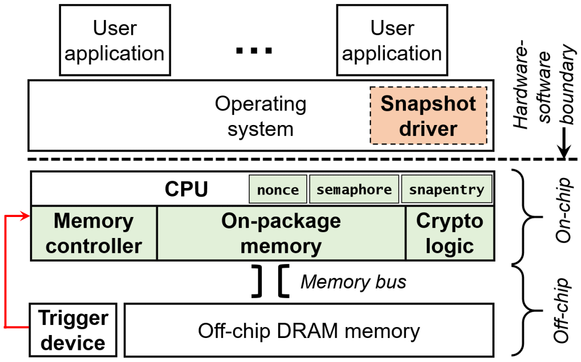


Figure 2: Architecture of SnipSnap. Only the on-chip hardware components are in the TCB.

SnipSnap leverages these hardware trends to realize fast and effective memory snapshotting. SnipSnap leverages on-package DRAM by realizing a fully hardware-based TCB. With modest hardware modifications that increase chip area by under 1%, SnipSnap captures and digitally signs pages in the on-package DRAM. 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. Because SnipSnap's TCB consists only of the hardware, it can be used on target machines running a variety of software stacks, *e.g.,* traditional systems (OS atop bare-metal), virtualized systems, and container-based systems. We identify *consistency* as an important property of memory snapshots and present SnipSnap's memory controller that offers both consistency and performance isolation. We implement SnipSnap using real-system hardware emulation and detailed software simulation atop state-of-the-art implementations of on-package die-stacked DRAM (*e.g.,* UNISON cache [52]). We vary on-package die-stacked DRAM from 512MB to 8GB capacities. We find that SnipSnap offers 4-25× performance improvements while also ensuring consistency. Finally, we verify SnipSnap's consistency guarantees using TLA+ [57].

In summary, SnipSnap securely obtains consistent snapshots while offering performance-isolation using non-exotic hardware that is already being implemented by chip vendors. This makes SnipSnap a powerful and general approach for snapshot acquisition, with applications to memory forensics and beyond.

2 OVERVIEW AND THREAT MODEL

SnipSnap allows a forensic analyst to acquire a complete snapshot of a target machine's off-chip DRAM memory. SnipSnap's mechanisms are implemented in a hardware TCB and an untrusted snapshot driver in the target's OS. The hardware TCB consists of on-package DRAM, simple near-memory processing logic, and requires modest modification of the on-chip memory controller and CPU register file. In concert, these components operate as described below.

A forensic analyst initiates snapshot acquisition by triggering the hardware to enter *snapshot mode*. Subsequently, the memory controller iteratively brings each physical page frame from off-chip DRAM to the on-package DRAM. SnipSnap's on-chip near-memory processing logic creates a copy of the page and computes a cryptographic digest of the page. The untrusted snapshot driver in the target OS then commits the snapshot entry to an external medium, such as persistent storage, the network, or a diagnostic serial port. The hardware exits snapshot mode after the near-memory processing logic has iterated over all page frames of the target's off-chip DRAM. A well-formed memory snapshot from SnipSnap contains one snapshot entry per page frame and one entry with CPU register state and a cryptographic digest. Figure 2 shows the components of SnipSnap:

① The *trigger device* is an external mechanism that initiates snapshot acquisition. When activated, the trigger device toggles the hardware into snapshot mode. It also informs the target's OS that the hardware has entered snapshot mode.

② The *memory controller* brings pages from off-chip DRAM into on-package DRAM to be copied into the snapshot when the hardware is in snapshot mode (as discussed above). The memory controller maintains internal hardware state to sequentially iterate over all off-chip DRAM page frames. The main novelty in SnipSnap's memory controller is a copy-on-write feature that allows snapshot acquisition to proceed without pausing the target.

③ The *near-memory processing logic* implements cryptographic functionality for hash and digital-signature computation in on-package DRAM [20]. As we show, such near-memory processing is readily implemented atop, for example, die-stacked memory [60]. As such, we assume that the hardware is endowed with a public/private key pair (as are TPMs—trusted platform modules). Digital signatures protect the integrity of the snapshot even from an adversary with complete control of the target's software stack.

④ The *snapshot driver*, SnipSnap's only software component, is implemented within the target's OS. Its sole responsibility is to copy snapshot entries created by the hardware to a suitable external medium.

⑤ The *hardware/software interface* facilitates communication between the snapshot driver and the hardware components. This interface consists of three special-purpose registers and adds minimal overhead to the existing register file of modern processors, which typically consists of several tens of architecturally-visible and hundreds of physical registers.

Threat Model. Our threat model is that of an attacker who controls the target's software stack and tries to subvert snapshot acquisition. The attacker may try to corrupt the snapshot, return stale snapshot entries, or suppress parts of the snapshot. A snapshot produced by SnipSnap must therefore contain sufficient information to allow a forensic analyst to verify integrity, freshness, and completeness of the snapshot. We assume that the on-chip hardware components described above are trusted and are part of the TCB. We exclude physical attacks on off-chip hardware components, *e.g.,* those that modify contents of pages either in off-chip DRAM via electromagnetic methods, or as they transit the memory bus.

SnipSnap's snapshot driver executes within the target OS, which may be controlled by the attacker. We will show that despite this, a corrupt snapshot driver cannot compromise snapshot integrity, freshness, or completeness. At worst, the attacker can misuse his control of the snapshot driver to prevent snapshot entries (or the entire snapshot) from being written out to the external medium. However, the forensic analyst can readily detect such denial of service attacks because the resulting snapshot will be incomplete. Once the forensic analyst obtains a snapshot, he can analyze it using methods described in prior work (*e.g.,* [10, 17, 24, 25, 33, 43, 72, 73]) to determine if the target is infected with malware.

SnipSnap's main goal is secure, consistent, and fast memory snapshot acquisition. Forensic analysts can perform offline analyses on these snapshots, *e.g.,* to check the integrity of the OS kernel or to detect traces of malware activity. While analysts can use SnipSnap to request snapshots for offline analysis as often as they desire, it is not a tool to perform continuous, event-based monitoring of the target machine. To our knowledge, state of the art forensic tools to detect advanced persistent threats (*e.g.,* [8, 10, 17, 24, 25, 43, 72–74, 80]) rely on offline analysis of memory snapshots.

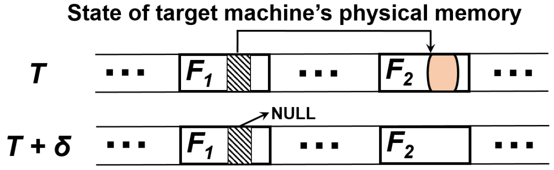


Figure 3: Example showing need for snapshot consistency. *Depicted above is the memory state of a target machine at two points in time, T and T+δ. At T, a pointer in F_1 points to an object in F_2. At T+δ, the object has been freed and the pointer set to* null. *Without consistency, the snapshot could contain a copy of F_1 at time T and F_2 at time T+δ (or vice-versa), causing problems for forensic analysis.*

3 DESIGN OF SNIPSNAP

We now present SnipSnap's design, beginning with a discussion of snapshot consistency.

3.1 Snapshot Consistency

A snapshot of a target machine is *consistent* if it reflects the state of the target machine's off-chip DRAM memory pages and CPU registers at a given instant in time. Consistency is an important property for forensic applications that analyze snapshots. Without consistency, different memory pages in the snapshot represent the state of the target at different points in time, 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 [73]. Such forensic analysis operates 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. If a page F_1 of memory contains a pointer to an object allocated on a page F_2, and the snapshot acquisition system captures F_1 and F_2 in different states of the target, then the forensic analysis can encounter a number of illogical situations (Figure 3). Such inconsistencies can also be used to hide malicious code and data modifications in memory [51]. Prior work [10, 73] 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+δ, thereby allowing the forensic analysis to traverse data structures in memory without the above problems.

As discussed in Section 1, prior systems have achieved snapshot consistency at the cost of performance isolation, or vice versa. SnipSnap acquires consistent memory snapshots without pausing the target machine in the common case. Snapshot acquisition proceeds in parallel with user applications and kernel execution that can actively modify memory. SnipSnap's hardware design ensures that the acquired memory snapshot reflects the state of the target machine at the instant when the hardware entered snapshot mode.

Consistency versus Quiescence. While SnipSnap ensures that an acquired snapshot faithfully mirrors the state of the machine at a given time instant, we do not specify what that time instant should be. Specifically, 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.

In response, prior work suggests collecting snapshots when the target machine is in *quiescence* [43], *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. SnipSnap only guarantees consistency, and relies on the forensic analyst to trigger snapshot acquisition at an instant when the system is quiescent. Because SnipSnap guarantees consistency, even if the target enters a non-quiescent state after snapshot acquisition has been triggered, *e.g.,* due to concurrent kernel activity initiated by user applications, the snapshot will reflect state of the target at the beginning of the snapshot acquisition. Triggering snapshot acquisition when the system is in *non-quiescent* state may require a forensic analyst to retake the snapshot.

3.2 Triggering Snapshot Acquisition

An analyst requests a snapshot using SnipSnap's trigger device. This device accomplishes three tasks: ① it toggles the hardware TCB into snapshot mode; ② it informs the target's OS that the hardware is in snapshot mode; and ③ it allows the analyst to pass a random nonce that is incorporated into the cryptographic digest of the snapshot.

Task ① requires direct hardware-to-hardware communication between the trigger device and the hardware TCB that is transparent to, and therefore cannot be compromised by, the target OS. Commodity systems offer many options to implement such communication, and SnipSnap can adapt any of them. For example, we could connect a physical device to the programmable interrupt controller, and have it deliver a non-maskable interrupt to the processor when it is activated. Upon receipt of this interrupt, the hardware TCB examines the IRQ to determine its origin, and switches to snapshot mode. Since this triggering mechanism piggybacks on the standard pin-to-bus interface, we find that implementing it requires less than 1% additional area on the hardware TCB.

Task ② is to inform the OS, so that it can start executing the snapshot driver. This task can be accomplished by raising an interrupt. The target OS invokes the snapshot driver from the interrupt handler.

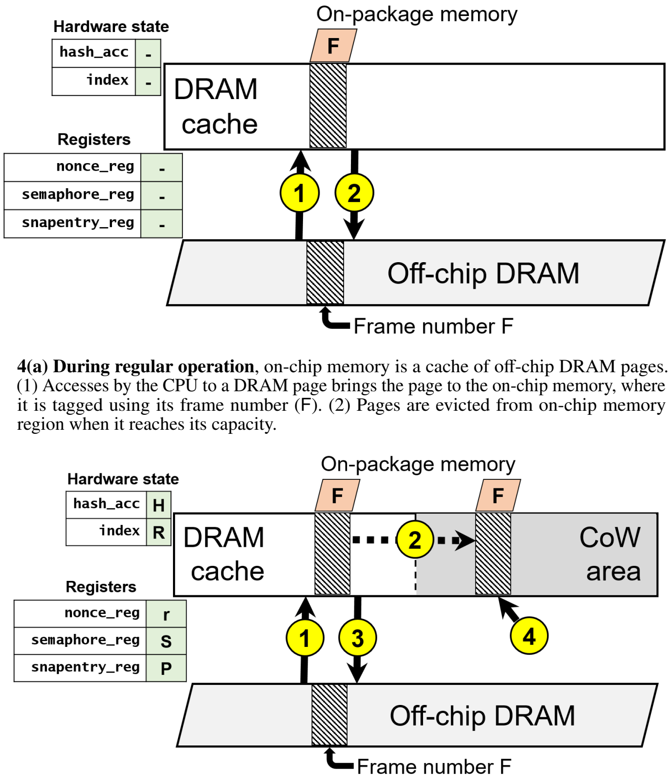
To accomplish task ③, we assume that the trigger device is equipped with device memory that is readable from the target OS. The analyst writes the nonce to device memory, and the OS reads it from there, *e.g.,* after mounting the device as `/dev/trigger_device`.

3.3 DRAM and Memory Controller Design

SnipSnap relies on on-package DRAM for secure and consistent snapshots. Today, research groups are actively studying how best to organize on-package DRAM. Research questions focus on whether on-package DRAM should be organized as a hardware cache of the off-chip DRAM *i.e.,* the physical address space is equal to the off-chip DRAM capacity [52, 62, 77], or should extend the physical address space instead, *i.e.,* the physical address space is the sum of the off-chip DRAM and on-chip memory capacities [22, 93]. While SnipSnap can be implemented on any of these designs, we focus on die-stacked DRAM caches as they have been widely studied and are expected to represent initial commercial implementations [52, 53, 62, 77].

DRAM caches can be designed in several ways. They can be used to cache data in units of cache lines like conventional L1-LLCs [52, 62, 77]. Unfortunately, the fine granularity of cache lines results in large volumes of tag metadata stored in either SRAM or DRAM caches themselves [52, 53, 62, 77]. Thus, architects generally prefer to organize DRAM caches at page-level granularity. While SnipSnap can be built using any DRAM cache data granularity, we focus on such page-level data caching approaches.

Overall, as a hardware-managed cache, the DRAM cache is not directly addressable from user- or kernel-mode. Further, all DRAM references are mediated by an on-chip memory controller, which is responsible for relaying the access to on-package or off-chip DRAM. That is, CPU memory references are first directed to per-core MMUs before being routed to the memory controller, while device memory references (*e.g.,* using DMA) are directed to the IOMMU before being routed to the memory controller.



4(a) During regular operation, on-chip memory is a cache of off-chip DRAM pages. (1) Accesses by the CPU to a DRAM page brings the page to the on-chip memory, where it is tagged using its frame number (F). (2) Pages are evicted from on-chip memory region when it reaches its capacity.

4(b) In snapshot mode, on-chip memory is split in two. (1) The DRAM cache works as in Figure 4(a). (2) If there is a write to a page that has not yet been snapshot (*i.e.,* $F \geq R$), it is copied into the CoW area. (3) The page may be evicted if the DRAM cache reaches capacity. (4) The CoW area copy of the page remains until it has been included in the snapshot (*i.e.,* $F < R$), after which it is overwritten with other pages that enter the CoW area. In snapshot mode H and R are initialized to 0.

Figure 4: Layout of on-chip memory.

Regular Operation. When snapshot acquisition is not in progress, SnipSnap's on-package memory acts as a hardware DRAM cache, before off-chip DRAM (see Figure 4(a)). The DRAM cache stores data in the unit of pages, and maintains tags, as is standard, to identify the frame number of the page cached and additional bits to denote usage information, like valid and replacement policy bits. When a new page must be brought into an already-full cache, the memory controller evicts a victim using standard replacement policies.

Snapshot Mode. When the trigger device signals the hardware to enter snapshot mode, several hardware operations occur. First, the hardware captures the CPU register state of the machine (across all cores). Second, all CPUs are paused, their pipelines are drained, their cache contents flushed (if CPUs use write-back caches), and their load-store queues and write-back buffers drained. These steps ensure that all dirty cache line contents are updated in main memory before snapshot acquisition begins. Third, SnipSnap's memory controller reconfigures the organization of on-package DRAM to ensure that a consistent snapshot of memory is captured. It must track any modifications to memory pages that are not yet included in the snapshot and keep a copy of the original page till it has been copied to the snapshot.

To achieve this goal, the memory controller splits the on-package DRAM 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 on-package DRAM is available for caching in snapshot mode, the memory controller tries to fit in it all the pages that were previously cached during regular operation into the available space. If all pages cannot be cached, the memory controller selects and evicts victims to off-chip DRAM. The second portion of die-stacked memory serves as a copy-on-write (CoW) area. 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. We study several ways to partition on-package DRAM into the CoW and DRAM cache areas in Section 6.

Recall that a snapshot contains a copy of all pages in off-chip DRAM memory. However, the hardware creates a snapshot entry one page of memory at a time. It works in tandem with the snapshot driver to write this snapshot entry to an external medium and then iterates to the next page of memory until all pages are written out to the snapshot. As this iteration proceeds, other applications and the kernel may concurrently modify memory pages that have not yet been included in the snapshot. If SnipSnap's memory controller sees a write to a memory page that the hardware has not yet copied, 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 memory controller maintains internal hardware state in the form of an *index* that stores the frame number (R in Figure 4(b)) of the page that is currently being processed for inclusion in the snapshot. The hardware initializes the index to 0 when it enters snapshot mode. The memory controller uses the index as follows. It copies a frame F from the DRAM cache to the CoW area when it has to write to that frame and $\mathsf{F} \geq \mathsf{R}$, indicating that the hardware has not yet iterated to frame F to create a snapshot entry for it. If $\mathsf{F} < \mathsf{R}$, then it means that the frame has already been included in the snapshot, and can be modified without copying it to the CoW area. SnipSnap requires that page frames be copied into the snapshot sequentially in ascending order by frame number.

To create a new snapshot entry for a page frame, the memory controller first checks whether this page frame is in the CoW area. If it exists, the hardware proceeds to create a snapshot entry using that copy of the page. The memory controller can then reuse the space occupied by this page in the CoW area. 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, from where the hardware creates a snapshot entry for that page. It places the newly-created entry in a physical page frame referenced by the *snapshot entry register* (`snapentry_reg` in Figure 4), and informs the snapshot driver using the *semaphore register* (`semaphore_reg` in Figure 4). The driver then writes out the entry to a suitable external medium and informs the hardware, which increments the index and iterates to the next page frame.

The hardware exits snapshot mode when the index has iterated over all the frames of off-chip DRAM. At this point, the hardware creates a snapshot entry containing the CPU register state (captured on entry into snapshot mode), and appends it as the last entry of the snapshot. We leverage die-stacked logic to capture and record register state. SnipSnap's approach is inspired by prior work on introspective die-stacked logic [69], where hardware analysis logic built on die-stacked layers uses probes or "stubs" on the CPUs to introspect on dynamic type analysis, data flight recorders, *etc*. Similarly, we design hardware support to capture register state, using: ① stubs that allow the contents of the register file to be latched into the logic on the die-stack; and ② logic on the die-stack that copies the contents of register files into the last snapshot entry.

The memory controller's use of CoW ensures that concurrent applications can make progress, while still maintaining the original copies of memory pages for a consistent snapshot. The hardware pauses a user application during snapshot acquisition only when the CoW area fills to capacity and when that application attempts to write to a page that the hardware has not yet included in the snapshot. 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.

Our implementation of SnipSnap has important design implications on recently-proposed DRAM caches. Research has shown that DRAM caches generally perform most efficiently when they use page-sized allocation units to reduce tag array size requirements [52, 53]. However, they also employ memory usage predictors (e.g., footprint predictors [52, 53]) to fetch only the relevant 64B blocks from a page, thereby efficiently using scarce off-chip bandwidth by not fetching blocks that will not be used. This means the following for SnipSnap. During regular operation, SnipSnap continues to employ page-based DRAM caches with standard footprint prediction. However, to simplify our design, SnipSnap does not use footprint prediction during snapshot mode and moves entire pages of data with their constituent cache lines in both the CoW and DRAM cache partitions. Naturally, this does degrade performance of applications running simultaneously with snapshotting; however, our results (see Section 6) show that performance improvements versus current snapshotting techniques remain high.

3.4 Near-Memory Processing Logic

Near-memory processing logic implements cryptographic functionality to create the snapshot. On a target machine with N frames of off-chip DRAM memory, the snapshot itself contains N+1 entries. The first N entries store, in order, the contents of page frames 0 to N-1 of memory (thus, an individual snapshot entry is 4KB). The last entry of the snapshot stores the CPU register state and a cryptographic digest that allows a forensic analyst to determine the integrity, freshness and completeness of the snapshot.

The near-memory processing logic maintains an internal *hash accumulator* that is initialized to zero when the hardware enters snapshot mode. It updates the hash accumulator as the memory controller iterates over memory pages, recording them in the snapshot. Suppose that we denote the value of the hash accumulator using $\mathsf{H_{idx}}$, where idx denotes the current value of the memory controller's index (thus, $\mathsf{H_0}$ = 0). When the memory controller creates a snapshot entry for page frame numbered idx, the near-memory processing logic updates the value of the hash accumulator to $\mathsf{H_{idx+1}}$=$\mathsf{Hash(idx \,||\, r \,||\, H_{idx} \,||\, C_{idx})}$. Here:

① The value idx is the hardware's index. It records the frame number of the page that included in the snapshot;

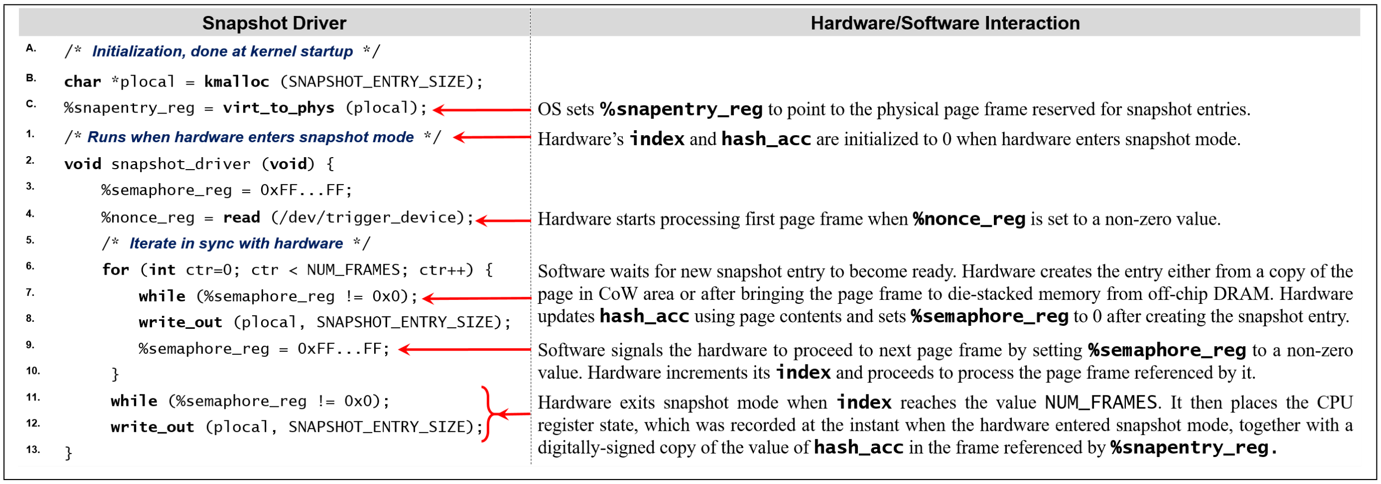
② The value r denotes a random nonce supplied by the forensic analyst using the trigger device and stored in the on-chip *nonce register* (`nonce_reg` in Figure 4(b)). The use of the nonce ensures freshness of the snapshot;

③ $\mathsf{H_{idx}}$ denotes the current value of the hash accumulator;

④ $\mathsf{C_{idx}}$ denotes the actual contents of page frame idx.

All these values are readily available on-chip.

When the memory controller finishes iterating over all N memory page frames, the value $\mathsf{H_N}$ in the hash accumulator in effect denotes



Snapshot Driver

```
A.  /* Initialization, done at kernel startup */
B.  char *plocal = kmalloc (SNAPSHOT_ENTRY_SIZE);
C.  %snapentry_reg = virt_to_phys (plocal);
1.  /* Runs when hardware enters snapshot mode */
2.  void snapshot_driver (void) {
3.      %semaphore_reg = 0xFF...FF;
4.      %nonce_reg = read (/dev/trigger_device);
5.      /* Iterate in sync with hardware */
6.      for (int ctr=0; ctr < NUM_FRAMES; ctr++) {
7.          while (%semaphore_reg != 0x0);
8.          write_out (plocal, SNAPSHOT_ENTRY_SIZE);
9.          %semaphore_reg = 0xFF...FF;
10.     }
11.     while (%semaphore_reg != 0x0);
12.     write_out (plocal, SNAPSHOT_ENTRY_SIZE);
13. }
```

Hardware/Software Interaction

OS sets `%snapentry_reg` to point to the physical page frame reserved for snapshot entries.

Hardware's `index` and `hash_acc` are initialized to 0 when hardware enters snapshot mode.

Hardware starts processing first page frame when `%nonce_reg` is set to a non-zero value.

Software waits for new snapshot entry to become ready. Hardware creates the entry either from a copy of the page in CoW area or after bringing the page frame to die-stacked memory from off-chip DRAM. Hardware updates `hash_acc` using page contents and sets `%semaphore_reg` to 0 after creating the snapshot entry.

Software signals the hardware to proceed to next page frame by setting `%semaphore_reg` to a non-zero value. Hardware increments its `index` and proceeds to process the page frame referenced by it.

Hardware exits snapshot mode when `index` reaches the value NUM_FRAMES. It then places the CPU register state, which was recorded at the instant when the hardware entered snapshot mode, together with a digitally-signed copy of the value of `hash_acc` in the frame referenced by `%snapentry_reg`.

Figure 5: Pseudocode of the snapshot driver and the corresponding hardware/software interaction.

the value of a *hash chain* computed cumulatively over all off-chip DRAM memory pages. The final snapshot entry enlists the values of CPU registers as recorded by the hardware when it entered snapshot mode—let us denote the CPU register state using C_{reg}. The near-memory logic updates the hash accumulator one final time to create H_{N+1}=Hash(N || r || H_N || C_{reg}). It digitally signs H_{N+1} using the hardware's private key, and records the digital signature in the last entry of the snapshot. This digital signature assists with the verification of snapshot integrity (Section 4). 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.

3.5 Snapshot Driver and HW/SW Interface

The hardware relies on the target's OS to externalize the snapshot entries that it creates. We rely on software support for this task because it simplifies hardware design, and also provides the forensic analyst with considerable flexibility in choosing the external medium to which the snapshot must be committed. Although we rely on the target OS for this critical task, we do not need to trust the OS and even a malicious OS cannot corrupt the snapshot created by the hardware.

The hardware and the software interact via an interface consisting of three registers (nonce, snapshot entry and semaphore registers), which were referenced earlier. Figure 5 shows the software component of SnipSnap and the hardware/software interaction. SnipSnap's software component consists of initialization code that executes at kernel startup (lines A–C) and a snapshot driver that is invoked when the hardware enters snapshot mode (lines 1–13). The implementation of the snapshot driver in the target OS depends on the trigger device and executes as a kernel thread. For example, if the trigger device raises an interrupt to notify the target OS that the hardware has switched to snapshot mode, the snapshot driver can be implemented within the corresponding interrupt handler. If the trigger device instead uses ACPI events for notification, the snapshot driver can be implemented as an ACPI event handler.

In the initialization code, SnipSnap allocates a buffer (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 the physical address translation of `plocal` in `snapentry_reg`, The hardware uses this physical address to store computed snapshot entries into the `plocal` buffer and the snapshot driver writes it out. Pages allocated using `kmalloc` cannot be moved, 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`, or if `virt_to_phys` returns an incorrect virtual to physical translation, the snapshot will appear corrupted to the forensic analyst.

When hardware enters snapshot mode, it initializes its internal index and hash accumulator, captures CPU register state, and invokes SnipSnap's snapshot driver. The goal of the snapshot driver is to work in tandem with the hardware to create and externalize one snapshot entry at a time. The snapshot driver and the hardware coordinate using the semaphore register, which the driver first initializes to a non-zero value on line 3. It then reads the nonce value that the forensic analyst supplies via the trigger device. Writing this non-zero value into `nonce_reg` on line 4 activates the near-memory processing logic, which creates a snapshot entry for the page frame referenced by the hardware's internal index.

In the loop on lines 6–10, the snapshot driver iterates over all page frames in tandem with the hardware. Each iteration of the loop body processes one page frame. The hardware begins processing the first page of DRAM as soon as line 4 sets `nonce_reg`, and stores the snapshot entry for this page in the `plocal` buffer. On line 7, 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 `semaphore_reg` to 0. The driver then commits the contents of this buffer to an external medium, denoted using `write_out` on line 8. The driver then sets `semaphore_reg` to a non-zero value on line 9, indicating to the hardware that it can increment its index and iterate to the next page for snapshot entry creation. 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.

When the loop completes execution, the hardware would have iterated through all DRAM page frames and exited snapshot mode. When it exits, it writes out the CPU register state captured during snapshot mode-entry and the digitally-signed value of the hash accumulator to the `plocal` buffer, which the snapshot driver can then output on line 12.

3.6 Formal Verification

We used TLA+ [57] to formally verify that SnipSnap produces consistent snapshots. To do so, we created a system model that mimics

SnipSnap's memory controller in snapshot mode and during regular operation. Our TLA+ system model can be instantiated for various configurations, such as memory sizes, cache sizes, and cache associativities. We encoded consistency as a safety property by checking that the state of the on-package and off-chip DRAM at the instant when the system switches to snapshot mode will be recorded in the snapshot at the end of acquisition. We verified that our system model satisfies this property using the TLA+ model checker. Our TLA+ model of SnipSnap is open source [3].

4 SECURITY ANALYSIS

When a forensic analyst receives a snapshot acquired by SnipSnap, he establishes its integrity, freshness, and completeness. In this section, we describe how these properties can be established, and show how SnipSnap is robust to attempts by a malicious target OS to subvert them.

① Integrity. An infected target OS may attempt to corrupt snapshot entries to hide traces of malicious activity from the forensic analyst. To ensure that the integrity of the snapshot has not been corrupted, an analyst can check the digital signature of the hash accumulator stored in the last snapshot entry. The analyst performs this check by essentially mimicking the operation of SnipSnap's memory controller and near-memory processing logic, *i.e.,* iterating over the snapshot entries in order to recreate the value of the hash accumulator, and verify its digital signature using the hardware's public key. Since the hash accumulator is stored and updated by the hardware TCB, which also computes its digital signature, a malicious target cannot change snapshot entries after they have been computed by the hardware.

② Freshness. The forensic analyst supplies a random nonce via the trigger device when he requests a snapshot. SnipSnap's hardware TCB incorporates this nonce into the hash accumulator computation for each memory page frame, thereby ensuring freshness. Note that SnipSnap uses the untrusted snapshot driver to transfer the nonce from trigger device memory into the hardware's nonce register (line 4 of Figure 5). A malicious target OS cannot cheat in this step, because the nonce is incorporated into the hardware TCB's computation of the hash accumulator.

③ Completeness. The snapshot should contain one entry for each page frame in off-chip DRAM and one additional entry storing CPU register state. This criterion ensures that a malicious target OS 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 die-stacked memory, thereby ensuring that these entities are correctly recorded in the entry.

Our attack analysis focuses on how a malicious target OS can subvert snapshot acquisition. A forensic analyst uses the trigger device to initiate snapshot acquisition by toggling the hardware TCB into snapshot mode. The trigger device communicates directly with SnipSnap's hardware TCB using hardware-to-hardware communication, transparent to the target's OS, and therefore cannot be subverted by a malicious OS. The hardware then notifies the OS that it is in snapshot mode, expecting the snapshot driver to be invoked.

A malicious target OS may attempt to "clean up" traces of infection before it jumps to the snapshot driver's code so that the resulting snapshot appears clean during forensic analysis. However, once the hardware is in snapshot mode, SnipSnap's memory controller, which mediates all writes to DRAM, uses the CoW area to track modifications to memory pages. Even if the target's OS attempts to overwrite the contents of a malicious page, the original contents of the page are saved in the CoW area to be included in the snapshot. Thus, any attempts by the target OS to hide its malicious activities after the hardware enters snapshot mode are futile. Of course, the target OS could refuse to execute the snapshot driver, which will prevent the snapshot from being written out to an external medium. Such a denial of service attack is therefore readily detectable.

A malicious OS may try to interfere with the execution of the initialization code in lines A–C of Figure 5. The initialization code relies on the correct operation of `kmalloc` and `virt_to_phys`. 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 `write_out` operation on line 8 will write out incorrect entries that fail the Integrity check.

Once the snapshot driver starts execution, a malicious target OS can attempt to interfere with its execution. If it copies a stale or incorrect value of the nonce into `nonce_reg` from trigger device memory on line 4, the snapshot will violate the Freshness criterion. It could attempt to bypass or short-circuit the execution of the loop on lines 5–10. The purpose of the loop is to synchronize the operation of the snapshot driver with the internal index maintained by SnipSnap's memory controller. If the OS short-circuits the loop or elides the `write_out` on line 8 for certain pages, the resulting snapshot will be missing entries, thereby violating Completeness. Attempts by the target OS to modify the virtual address of `plocal` or the value of `snapshot_reg` during the execution of the snapshot driver will trigger a violation of Integrity for the same reasons that attacks on the initialization code triggers an Integrity violation.

Finally, a malicious target could try to hide traces of infection by creating a synthetic snapshot that glues together individual entries (with benign content in their memory pages) from snapshots collected at different times. However, such a synthetic snapshot will fail the Integrity check since the hash chain computed over such entries will not match the digitally-signed value in the last snapshot entry.

The last entry records the values of all CPU registers at the instant when the hardware entered snapshot mode. For forensic analysis, the most useful value in this record is that of the page-table base register (PTBR). As previously discussed, forensic analysis of the snapshot often involves recursive traversal of pointer values that appear in memory pages [10, 17, 25, 72–74, 80]. 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. External hardware-based systems [10, 16, 58, 59, 67, 72, 74] cannot view the processor's CPU registers. Therefore, they depend on the untrusted target OS to report the value of the PTBR. Unfortunately, this results in address-translation redirection attacks [51, 56]. The target OS can create a synthetic page table that contains fraudulent virtual-to-physical mappings and return a PTBR referencing this page table. The synthetic page table exists for the sole purpose of defeating forensic analysis by making malicious content unreachable via page-table translations—it is not used by the target OS during execution. SnipSnap can observe and record CPU register state accurately when the hardware enters snapshot mode and is not vulnerable to such attacks. It captures the PTBR pointing to the page table that is in use when the hardware enters snapshot mode.

5 EXPERIMENTAL METHODOLOGY

5.1 Evaluation Infrastructure

We use a two-step approach to quantify SnipSnap's benefits. In the first step, we perform evaluations on long-running applications with full-system and OS effects. Since this is infeasible with software simulation, we develop hardware emulation infrastructure similar to recent work [70] to achieve this. This infrastructure takes an existing hardware

① **Canneal**	Simulated annealing from PARSEC [11]
② **Dedup**	Storage deduplication from PARSEC [11]
③ **Memcached**	In-memory key-value store [66]
④ **Graph500**	Graph-processing benchmark [38]
⑤ **Mcf**	Memory-intensive benchmark/SPEC 2006 [83]
⑥ **Cifar10**	Image recognition from TensorFlow [87]
⑦ **Mnist**	Computer vision from TensorFlow [87]

Figure 6: Description of benchmark user applications.

platform, and through memory contention, creates two different speeds of DRAM. Specifically, we use a two-socket Xeon E5-2450 processor, with a total of 32GB of memory, running Debian-sid with Linux kernel 4.4.0. There are 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 our DRAM cache, 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 on-package memory. The second socket hosts our "slow" or off-chip DRAM. The cores on the second socket are used to create memory contention (using the memory contention benchmark `memhog`, like prior work [75, 76]) such that the emulated die-stacked memory or DRAM cache is 4.5× faster compared to the emulated off-chip DRAM. This provides a similar memory bandwidth performance ratio of a 51.2GBps off-chip memory system compared to a 256GBps of die-stacked memory, consistent with the expected performance ratios of real-world die-stacking [62, 70]. We modify Linux kernel to page between the emulated fast and slow memory, using the `libnuma` patches. We model the timing aspects of paging to faithfully reproduce the performance that SnipSnap's memory controller would sustain. Since our setup models CPUs with write-back caches, we include the latencies necessary for cache, load-store queue, and write buffer flushes on snapshot acquisition. Finally, we emulate the overhead of marshaling to external media by introducing artificial delays. We vary delay based on several emulated external media, from fast network connections to slower SSDs.

While our emulator includes full-system effects and full benchmark runs, it precludes us from modeling SnipSnap's effectiveness atop recently-proposed (and hence not available commercially) DRAM cache designs. Therefore, we also perform careful software simulation of the state-of-art UNISON DRAM cache [52], building SnipSnap atop it. Like the original UNISON cache paper, we assume a 4-way set-associative DRAM cache with 4KB pages, a 144KB footprint history table, and an accurate way predictor. Like recent work [93], we use an in-house simulator and drive it with 50 billion memory reference traces collected on a real system. We model a 16-core CMP and with ARM A15-style out-of-order CPUs, 32KB private L1 caches, and 16MB shared L2 cache. We study die-stacked DRAM with 4 channels, and 8 banks/rank with 16KB row buffers, and 128-bit bus width, like prior work [53]. Further, we model 16-64GB off-chip DRAM, with 8 banks/rank and 16KB row buffers. Finally, we use the same DRAM timing parameters as as the original UNISON cache paper [52].

5.2 Workloads

We study the performance implications of SnipSnap by quantifying snapshot overheads on several memory-intensive applications. We evaluate such workloads since these are the likeliest to face performance degradation due to snapshot acquisition. Even in this "worst-case," we show SnipSnap does not excessively hurt performance.

Figure 6 shows our single- and multi-threaded workloads. All benchmarks are configured to have memory footprints in the range of 12-14GB, which exceeds the maximum size of die-stacked memory we emulate (8GB). To achieve large memory footprints, we upgrade the inputs for some workloads with smaller defaults (*e.g.,* Canneal, Dedup, and Mcf), so that their memory usage increases. We set up memcached with a snapshot of articles from the entire Wikipedia database, with over 10 million entries. Articles are roughly 2.8KB on average, but also exhibit high object size variance.

6 EVALUATION

We now evaluate the benefits of SnipSnap. We first quantify performance, and then discuss its hardware overheads.

6.1 Performance Impact on Target Applications

A drawback of current snapshotting mechanisms is that they *must* pause the execution of applications executing on the target to ensure consistency. SnipSnap does not suffer from this drawback. Figures 7 and 8 quantify these benefits. We plot the slowdown in runtime (lower is better) with benchmark averages, minima, and maxima, as we vary on-package DRAM capacity. We separate performance based on how we externalize snapshots: NICs with 100Gbps, 40Gbps, and 10Gbps throughput, and a solid-state storage disk (SSD) with sequential write throughput of 900MBps. Larger on-package DRAM (and hence, 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 high-end chipset series supports only up to 4GB of on-package DRAM. Similarly, 40Gbps and 100Gbps NICs are expensive and not yet in wide use.

Figure 7 shows results collected on our hardware emulator, assuming that 50% of on-package DRAM is devoted to the CoW area during snapshot mode. We vary the size on-package DRAM from 512MB to 8GB, and assume 16GB off-chip DRAM. Further, our hardware emulator assumes that on-package DRAM is implemented as a page-level fully-associative cache. We show the performance slowdown due to idealized current snapshotting mechanisms, as we take 1 and 10 snapshots. By idealized, we mean approaches like virtualization-based or TrustZone-style snapshotting which require pausing applications on the target to achieve consistency, but which assume unrealizable zero-overhead transition times to TrustZone mode or zero-overhead virtualization. Despite idealization, current approaches perform poorly. Even with only one snapshot, runtime increaseas by 1.2-2.4× using SSDs. SnipSnap fares much better, outperforming the idealized baseline by 1.2-2.2×, depending on the externalization medium and on-package DRAM size. Snapshotting more frequently (*i.e.,* 10 snapshots) further improves performance by 10.5-22×. Naturally, the more frequent the snapshotting, the more SnipSnap's benefits, though our benefits are significant even with a single snapshot.

Similarly, Figure 8 quantifies SnipSnap's performance improvements versus current snapshotting, assuming a baseline with state-of-the-art UNISON cache implementations of on-package DRAM [52], as UNISON cache sizes are varied from 512MB to 8GB. Some key differences between UNISON cache and our fully-associative hardware emulated DRAM cache is that UNISON cache also predicts 64B blocks within pages that should be moved on a DRAM cache miss, and also is implemented as 4-way set associative (as per the original paper). Nevertheless, Figure 8 (collected assuming SSDs as the externalizing medium) shows that SnipSnap outperforms idealized versions of current snapshotting mechanisms by as much as 22×, and by as much as 3× when just a single snapshot is taken.

SnipSnap's performance also scales far better than idealized versions of current snapshotting with increasing off-chip DRAM capacities. Figure 9 compares the performance slowdown due to one snapshot, as off-chip DRAM varies from 16GB to 64GB. These results are collected using UNISON cache (8GB in normal operation, 4GB in snapshot mode, with 4GB CoW), and assuming SSDs. Consider idealized versions of current snapshotting approaches – runtime increases from 3× with 16GB off-chip DRAM to as high as 5.3× with

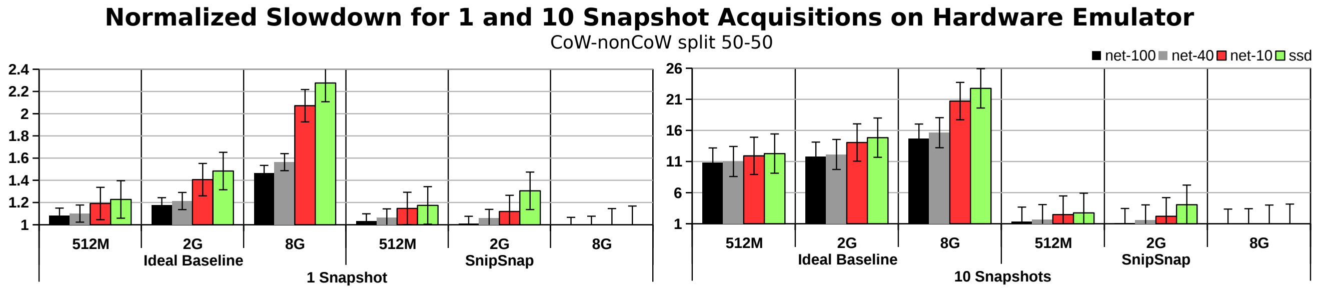


Figure 7: Performance impact of snapshot acquisition from hardware emulator studies. *Slowdown caused by modern snapshot mechanisms that also assure consistency, and compare against SnipSnap. We plot results for 1 and 10 snapshots separately (note the different y axes), showing averages, minima, and maxima amongst benchmark runtimes. X-axis shows the amount of on-package memory available on the emulated system. SnipSnap provides 1.2-22× performance improvements against current approaches.*

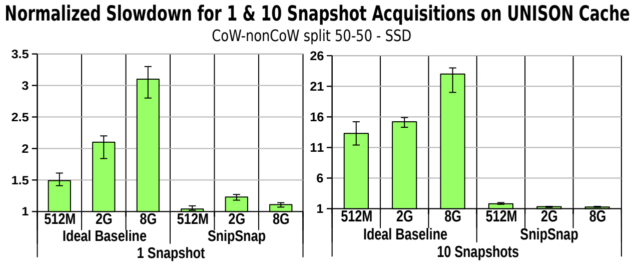


Figure 8: Performance impact of snapshot acquisition from simulator studies with UNISON cache [52]. *SnipSnap outperforms idealized versions of current snapshotting approaches by as much as 22× (graphs show benchmark averages, maxima, and minima).*

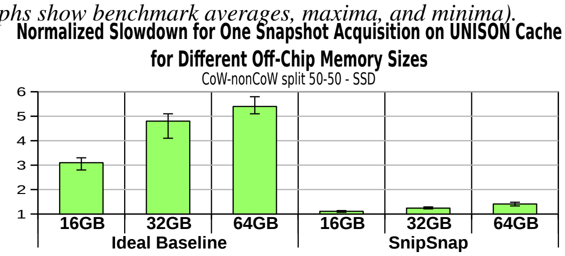


Figure 9: Average performance with varying off-chip DRAM size. *Bigger off-chip DRAM takes longer to snapshot, so SnipSnap becomes even more advantageous over current idealized approaches. These results assume UNISON cache with 8GB, split 50:50 in CoW:non-CoW mode during snapshot acquisition and SSDs, taking just one snapshot.*

64GB of memory, when taking just a single snapshot. More snapshots further exacerbate this slowdown. While SnipSnap also suffers slowdown with larger off-chip DRAM, it still vastly outperforms current approaches by as much as 5× at 64GB of off-chip DRAM.

So far, we have shown application slowdown comparisons of SnipSnap versus current approaches. Figure 10 focuses, instead, on per-benchmark runtime slowdown using SnipSnap, when varying the size of on-package DRAM and the externalizing medium. Results show that most benchmarks, despite being data-intensive, remain unaffected by SnipSnap's snapshot acquisition. The primary exceptions to this are memcached, cfar, and mnist, though their slowdowns vastly outperform current approaches (see Figures 7 and 8).

6.2 CoW Analysis

As discussed in Section 3, benchmark runtime suffers during snapshot acquisition only if the CoW area fills to capacity. When this happens, the benchmark stalls until some pages from the CoW area are copied to the snapshot. Figure 11 illustrates this fact, and explains the performance of memcached. Figure 11 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 11(b) the performance of memcached is unaffected.

Next, Figure 12 quantifies the performance impact of varying the percentage of die-stacked memory devoted to the CoW area. We vary the split from 50-50% to 25-75% and 75-25% for CoW-nonCoW portions, for various externalization techniques. We present the average results across all workloads for various total die-stacked memory sizes (individual benchmarks follow these average trends). Figure 12 shows that performance remains strong across all configurations, even when the percentage of DRAM cache devoted to CoW is low, which potentially leads to more stalls in the system. Furthermore, low CoW only degrades performance at smaller DRAM cache sizes of 512MB, which are smaller than DRAM cache sizes expected in upcoming systems.

Finally, note that the set-associativity of the DRAM cache devoted to the CoW region influences SnipSnap's performance. Specifically, consider designs like UNISON cache [52] (and prior work like Footprint cache [53]), which use 4-way set-associative (and 32-way set-associative) page-based DRAM caches. In these situations, if an entire set of the DRAM cache becomes full (even if other sets are not), applications executing on the target must pause until pages from that set are written to the external medium (i.e., SSD, network, etc.). Even in the worst case (all the application's data maps to a single set so the CoW region always stalls application execution and writing pages to the external medium takes as long as the entire snapshot time) this is *no worse* that idealized versions of current approaches. However, we find that this scenario does not occur in practice. Figure 13 quantifies SnipSnap's performance versus an ideal baseline for one snapshot, as off-chip DRAM capacity is varied from 16GB to 64GB, on-chip DRAM capacity is varied from 512MB to 8GB, and associativity is varied between 2-way and 4-way. Larger DRAM caches and higher associativity improve SnipSnap's performance, but even when we hamper UNISON cache to be 512MB and 2-way set-associative, it outperforms idealized current approaches by ~2×. More frequent snapshots further increase this number.

Beyond these studies, we also considered quantifying SnipSnap's performance on a direct-mapped UNISON cache. However, as pointed out by prior work, the conflict misses induced by direct-mapping in baseline designs without snapshotting are so high, that no practical page-based DRAM cache design is direct-mapped [52, 53]. Therefore, we begin our analysis with 2-way set-associative DRAM caches, showing that SnipSnap consistently outperforms alternatives.

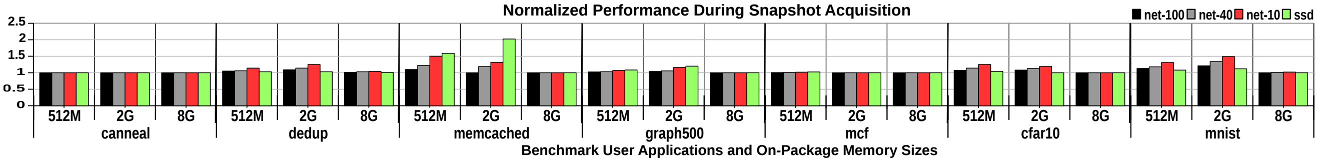
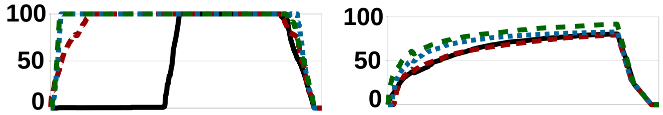


Figure 10: Performance impact of snapshot acquisition. *This chart reports the observed performance of user applications executing on the target during snapshot acquisition, normalized against their observed performance during regular execution, i.e., no snapshot acquisition. For each of the seven benchmarks, we report the performance for various sizes of die-stacked memory (50% of which is the CoW area), and for different methods via which the* `write_out` *in Figure 5 writes out the snapshot.*



11(a) *512MB of on-chip memory* **11(b)** *4GB of on-chip memory*

Figure 11: CoW area utilization over time for memcached. *Y-axis shows CoW area percentage used to store page frames that have not yet been included in the snapshot. X-axis denotes execution progress. We measured CoW utilization for every 1024 snapshot entries recorded. The two charts show CoW utilization trends for various sizes of die-stacked memory and for different methods to write out the snapshot:* ▬**net-100** ••• **net-40** ▬**net-10** ▬ ▬**ssd**. *Snapshot acquisition does not impact memcached performance when CoW utilization is below 100%.*

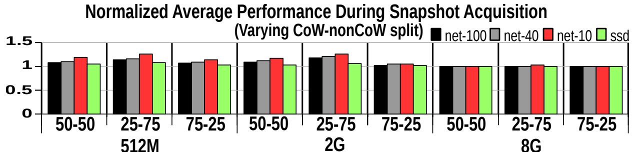


Figure 12: Performance impact of snapshot acquisition for different CoW-Cache partitions. *Y-axis shows average performance impact of all benchmarks to take a snapshot, varying CoW-nonCoW partition for different cache sizes. X-axis shows different total sizes of die-stacked memory and various ways in which to partition die-stacked memory for CoW (50%, 25% and 75% for CoW).*

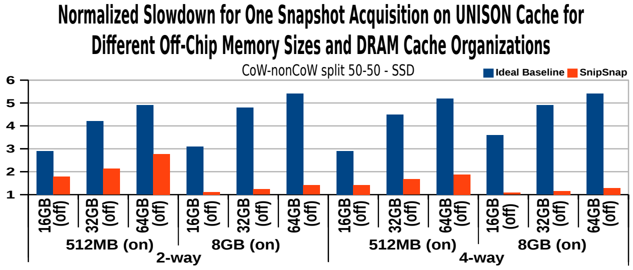


Figure 13: Performance as size and set-associativty of UNISON cache changes. *Lower UNISON cache size and set-associativity increases the chances that a set in the CoW region fills up and pauses execution of applications on the target. Results are shown using SSDs, varying off-chip DRAM capacity from 16GB to 64GB, UNISON cache size from 512MB to 8GB, and set-associativity from 2 to 4 way.*

7 RELATED WORK

As Section 1 discusses, there is much prior work on remote memory acquisition based on virtualization, trusted hardware and external hardware. Figure 1 characterizes the difference between SnipSnap and this prior work. Aside from these, there are other mechanisms to fetch memory snapshots for the purpose of debugging (*e.g.,* [37, 42, 54, 84, 86]). Because their focus isn't forensic analysis, these systems do not assume an adversarial target OS.

Prior work has leveraged die-stacking to implement myriad security features such as monitoring program execution, access control and cryptography [46–48, 64, 69, 89–91]. This work observes that die-stacking allows processor vendors to decouple core CPU logic from "add-ons," such as security, thereby improving their chances of deployment. Our work also leverages additional circuitry on the die-stack to implement the logic needed for memory acquisition. Unlike prior work, which focused solely on additional processing logic integrated using die-stacking, our focus is also on die-stacked memory, which is beginning to see deployment in commercial processors. While SnipSnap also uses the die-stack to integrate additional cryptographic logic and modify the memory controller, it does so to enable near-data processing on the contents of die-stacked memory.

Prior work has also used die-stacked manufacturing technology to detect malicious logic inserted into the processor. The threat model is that of an outsourced chip manufacturer who can insert Trojan-horse logic into the hardware. This work suggests various methods to combat this threat using die-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 [49, 88]. Another method is to add logic into die-stacked layers to monitor the execution of the processor for maliciously-inserted logic [12–14].

There is prior work on near-data processing to enable security applications [40] and modifying memory controllers to implement a variety of security features [82, 92]. There is also work on using programmable DRAM [59] to monitor systems for OS and hypervisor integrity violations. Unlike SnipSnap, which focuses on fetching a complete snapshot of DRAM, and must hence consider snapshot consistency, this work only focuses on analysis of specific memory pages, *e.g.,* those that contain specific kernel data structures. It also cannot access CPU register state, making it vulnerable to address-translation attacks [51, 56].

8 CONCLUSION

Vendors are beginning to integrate memory and processing logic on-chip using on-package DRAM manufacturing technology. We have presented SnipSnap, an application of this technology to secure memory acquisition. SnipSnap 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. Our experimental evaluation on a number of data intensive workloads shows the benefit of our approach.

Dedication and Acknowledgments. We would like to dedicate this paper to the memory of our friend, colleague and mentor, Professor Liviu Iftode (1959-2017). This work was funded in part by NSF grants 1337147, 1319755, 1441724, and 1420815.

REFERENCES

[1] [n. d.]. Docker – Build, Ship and Run Any App, Anywhere. ([n. d.]). https://www.docker.com/.

[2] [n. d.]. Rekall Forensics – We can remember it for you wholesale! ([n. d.]). http://www.rekall-forensic.com/.

[3] [n. d.]. TLA+ model of SnipSnap. ([n. d.]). http://bit.ly/2mOCY23.

[4] [n. d.]. Volatility – An advanced memory forensics framework. ([n. d.]). https://github.com/volatilityfoundation/volatility.

[5] 2009. 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.

[6] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *International Symposium on Computer Architecture (ISCA)*.

[7] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. 2015. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *International Symposium on Computer Architecture (ISCA)*.

[8] William Arbaugh. [n. d.]. Komoku. In https://www.cs.umd.edu/~waa/UMD/Home.html.

[9] A. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *ACM Conference on Computer and Communications Security (CCS)*.

[10] A. Baliga, V. Ganapathy, and L. Iftode. 2011. Detecting Kernel-level Rootkits using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing* 8, 5 (2011).

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Parallel Architectures and Compilation Techniques (PACT)*.

[12] M. Bilzor. 2011. 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*.

[13] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. 2011. Security Checkers: Detecting Processor Malicious Inclusions at Runtime. In *IEEE International Symposium on Hardware-oriented Security and Trust*.

[14] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. 2012. 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*.

[15] 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. 2006. Die Stacking 3D Microarchitecture. In *International Symposium on Microarchitecture (MICRO)*.

[16] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. 2004. Remote Repair of Operating System State Using Backdoors. In *International Conference on Autonomic Computing (ICAC)*.

[17] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. 2009. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *ACM Conference on Computer and Communications Security (CCS)*.

[18] Andrew Case and Golden G. Richard. 2017. Memory forensics: The path forward. *Digital Investigation* 20 (2017), 23 – 33. https://doi.org/10.1016/j.diin.2016.12.004 Special Issue on Volatile Memory Analysis.

[19] Michael Chan, Heiner Litz, and David R. Cheriton. 2013. Rethinking Network Stack Design with Memory Snapshots. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, USA, 27–27. http://dl.acm.org/citation.cfm?id=2490483.2490510

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

[21] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. 2012. HICAMP: Architectural Support for Efficient Concurrency-safe Shared Structured Data Access. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 287–300. https://doi.org/10.1145/2150976.2151007

[22] C.-C. Chou, A. Jaleel, and M. K. Qureshi. 2012. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *International Symposium on Microarchitecture (MICRO)*.

[23] Lei Cui, Tianyu Wo, Bo Li, Jianxin Li, Bin Shi, and Jinpeng Huai. 2015. PARS: A Page-Aware Replication System for Efficiently Storing Virtual Machine Snapshots. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. ACM, New York, NY, USA, 215–228. https://doi.org/10.1145/2731186.2731190

[24] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. 2016. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *International Conference on Software Engineering (ICSE)*.

[25] W. Cui, M. Peinado, Z. Xu, and E. Chan. 2012. Tracking Rootkit Footprints with a Practical Memory Analysis System. In *USENIX Security Symposium*.

[26] CVE-2007-4993. [n. d.]. Xen guest root escapes to dom0 via pygrub. ([n. d.]).

[27] CVE-2007-5497. [n. d.]. Integer overflows in libext2fs in e2fsprogs. ([n. d.]).

[28] CVE-2008-0923. [n. d.]. Directory traversal vulnerability in the Shared Folders feature for VMWare. ([n. d.]).

[29] CVE-2008-1943. [n. d.]. Buffer overflow in the backend of XenSource Xen ParaVirtualized Frame Buffer. ([n. d.]).

[30] CVE-2008-2100. [n. d.]. VMWare buffer overflows in VIX API let local users execute arbitrary code in host OS. ([n. d.]).

[31] Bernhard Egger, Erik Gustafsson, Changyeon Jo, and Jeongseok Son. 2015. Efficiently Restoring Virtual Machines. *International Journal of Parallel Programming* 43, 3 (2015), 421–439. https://doi.org/10.1007/s10766-013-0295-0

[32] Wikipedia entry. [n. d.]. eDRAM. In https://en.wikipedia.wiki/EDRAM.

[33] Q. Feng, A. Prakash, H. Yin, and Z. Lin. 2014. MACE: High-Coverage and Robust Memory Analysis for Commodity Operating Systems. In *Annual Computer Security Applications Conference (ACSAC)*.

[34] H. Fujita, N. Dun, Z. A. Rubenstein, and A. A. Chien. 2015. Log-Structured Global Array for Efficient Multi-Version Snapshots. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 281–291. https://doi.org/10.1109/CCGrid.2015.80

[35] T. Garfinkel and M. Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network and Distributed System Security Symposium (NDSS)*.

[36] X. Ge, H. Vijayakumar, and T. Jaeger. 2014. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *IEEE Mobile Security Technologies Workshop (MoST)*.

[37] Google. [n. d.]. Using DDMS for debugging. ([n. d.]). http://developer.android.com/tools/debugging/ddms.html.

[38] Graph500. [n. d.]. http://www.graph500.org.

[39] Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. 2013. *Hypervisor Memory Forensics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 21–40. https://doi.org/10.1007/978-3-642-41284-4_2

[40] A. Gundu, A. S. Ardestani, M. Shevgoor, and R. Balasubramonian. 2014. A Case for Near Data Security. In *3rd Workshop on Near Data Processing*.

[41] 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. 2010. Design and Analysis of 3D-MAPS: A Many-Core 3D Processor with Stacked Memory. In *IEEE Custom Integrated Circuits Conference (CICC)*.

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

[43] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. 2011. Ensuring Operating System Kernel Integrity with OSck. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[44] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. Keckler. 2015. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *International Symposium on Computer Architecture (ISCA)*.

[45] Y. Huang, R. Yang, L. Cui, T. Wo, C. Hu, and B. Li. 2014. VMCSnap: Taking Snapshots of Virtual Machine Cluster with Memory Deduplication. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. 314–319. https://doi.org/10.1109/SOSE.2014.45

[46] T. Huffmire, T. Levin, M. Bilzor, C. Irvine, J. Valamehr, M. Tiwari, and T. Sherwood. 2010. Hardware Trust Implications of 3-D Integration. In *Workshop on Embedded Systems Security*.

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

[48] T. Huffmire, J. Valamehr, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine. 2008. Trustworthy System Security through 3-D Integrated Hardware. In *International Workshop on Hardware-oriented Security and Trust*.

[49] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara. 2013. Securing Computer Hardware using 3D Integrated Circuit Technology and Split Manufacturing for Obfuscation. In *USENIX Security Symposium*.

[50] InfiniBand. [n. d.]. The InfiniBand Trade Association—The InfiniBand^{TM} Architecture Specification. ([n. d.]). http://www.infinibandta.org.

[51] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. Kang. 2014. ATRA: Address Translation Redirection attack against Hardware-based External Monitors. In *ACM Conference on Computer and Communications Security (CCS)*.

[52] D. Jevdjic, G. Loh, C. Kaynak, and B. Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *International Symposium on Microarchitecture (MICRO)*.

[53] D. Jevdjic, S. Volos, and B. Falsafi. 2013. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *International Symposium on Computer Architecture (ISCA)*.

[54] Joint Test Action Group (JTAG). 2013. 1149.1-2013 - IEEE Standard for Test Access Port and Boundary-scan Architecture. (2013). http://standards.ieee.org/findstds/standard/1149.1-2013.html.

[55] K. Kortchinsky. 2009. Hacking 3D (and Breaking out of VMWare). In *BlackHat USA*.

[56] Y. Kinebuchi, S. Butt, V. Ganapathy, L. Iftode, and T. Nakajima. 2013. Monitoring System Integrity using Limited Local Memory. *IEEE Transactions on Information Forensics and Security* 8, 7 (2013).

[57] L. Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education.

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

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

[60] G. Loh. 2008. 3D-Stacked Memory Architectures for Multi-Core Processors. In *International Symposium on Computer Architecture (ISCA)*.

[61] G. Loh. 2009. Extending the Effectiveness of 3D-Stacked DRAM Caches with an Adaptive Multi-Queue Policy. In *International Symposium on Microarchitecture (MICRO)*.

[62] G. Loh and M. D. Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches. In *International Symposium on Microarchitecture (MICRO)*.

[63] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. 2014. XvMotion: Unified Virtual Machine Migration over Long Distance. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 97–108. http://dl.acm.org/citation.cfm?id=2643634.2643645

[64] D. Megas, K. Pizolato, T. Levin, and T. Huffmire. 2012. A 3D Data Transformation Processor. In *Workshop on Embedded Systems Security*.

[65] Mellanox Technologies. 2014. Introduction to InfiniBand. (September 2014). http://www.mellanox.com/blog/2014/09/introduction-to-infiniband.

[66] Memcached. [n. d.]. https://memcached.org.

[67] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. Kang. 2012. Vigilare: Toward a Snoop-based Kernel Integrity Monitor. In *ACM Conference on Computer and Communications Security (CCS)*.

[68] Andreas Moser and Michael I. Cohen. 2013. Hunting in the enterprise: Forensic triage and incident response. *Digital Investigation* 10, 2 (2013), 89 – 98. https://doi.org/10.1016/j.diin.2013.03.003 Triage in Digital Forensics.

[69] S. Mysore, B. Agrawal, N. Srivastava, S-C. Lin, K. Banerjee, and T. Sherwood. 2016. Introspective 3D Chips. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[70] M. Oskin and G. Loh. 2015. A Software-managed Approach to Die-Stacked DRAM. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[71] Eunbyung Park, Bernhard Egger, and Jaejin Lee. 2011. Fast and Space-efficient Virtual Machine Checkpointing. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '11)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/1952682.1952694

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

[73] N. Petroni and M. Hicks. 2007. Automated Detection of Persistent Kernel Control-flow Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.

[74] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. 2004. Copilot: A Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*.

[75] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *International Symposium on Microarchitecture (MICRO)*.

[76] B. Pham, J. Vesely, G. Loh, and A. Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways?. In *International Symposium on Microarchitecture (MICRO)*.

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

[78] J. Rutkowska. 2007. Beyond the CPU: Defeating Hardware based RAM Acquisition, part I: AMD case. In *Blackhat Conf.*

[79] J. Rutkowska and R. Wojtczuk. 2008. Preventing and detecting Xen hypervisor subversions. In *Blackhat Briefings USA*.

[80] K. Saur, M. Hicks, and J. S. Foster. 2015. C-Strider: Type-aware Heap Traversal for C. *Software, Practice, and Experience* (May 2015).

[81] Bradley Schatz and Michael Cohen. 2017. Advances in volatile memory forensics. *Digital Investigation* 20 (2017), 1. https://doi.org/10.1016/j.diin.2017.02.008 Special Issue on Volatile Memory Analysis.

[82] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari. 2015. Avoiding Information Leakage in the Memory Controller with Fixed Service Policies. In *International Symposium on Microarchitecture (MICRO)*.

[83] Spec. [n. d.]. https://www.spec.org/cpu2006/.

[84] A. Stevenson. [n. d.]. Boot into Recovery Mode for Rooted and Un-rooted Android devices. ([n. d.]). http://androidflagship.com/605-enter-recovery-mode-rooted-un-rooted-android.

[85] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. 2014. TrustDump: Reliable Memory Acquisition on Smartphones. In *European Symposium on Research in Computer Security (ESORICS)*.

[86] J. Sylve, A. Case, L. Marziale, and G. G. Richard. 2012. Acquisition and analysis of Volatile Memory from Android Smartphones. *Digital Investigation* 8, 3-4 (2012).

[87] TensorFlow. [n. d.]. https://www.tensorflow.org.

[88] Tezzaron Semiconductors. 2008. 3D-ICs and Integrated Circuit Security. (2008). http://www.tezzaron.com/media/3D-ICs_and_Integrated_Circuit_Security.pdf.

[89] J. Valamehr, T. Huffmire, C. Irvine, R. Kastner, C. Koc, T. Levin, and T. Sherwood. 2012. 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*.

[90] J. Valamehr, M. Tiwari, T. Sherwood, R. Kastner, T. Huffmire, C. Irvine, and T. Levin. 2010. Harware Assistance for Trustworthy Systems through 3-D Integration. In *Annual Computer Security Applications Conference (ACSAC)*.

[91] J. Valamehr, M. Tiwari, T. Sherwood, R. Kastner, T. Huffmire, C. Irvine, and T. Levin. 2013. 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).

[92] Y. Wang, A. Ferraiuolo, and G. E. Suh. 2014. Timing Channel Protection for a Shared Memory Controller. In *IEEE International Conference on High-performance Computer Architecture (HPCA)*.

[93] Zi Yan, Jan Vesely, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware Translation Coherence for Virtualized Systems. In *International Symposium on Computer Architecture (ISCA)*.

[94] Ruijin Zhou and Tao Li. 2013. Leveraging Phase Change Memory to Achieve Efficient Virtual Machine Execution. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/2451512.2451547

URLs in references were last accessed January 7, 2018