Secure Page Fusion with VUsion

https://www.vusec.net/projects/VUsion

Marco Oliverio
Vrije Universiteit Amsterdam
Università della Calabria

Kaveh Razavi
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

ABSTRACT

To reduce memory pressure, modern operating systems and hypervisors such as Linux/KVM deploy page-level memory fusion to merge physical memory pages with the same content (i.e., *page fusion*). A write to a fused memory page triggers a copy-on-write event that unmerges the page to preserve correct semantics. While page fusion is crucial in saving memory in production, recent work shows significant security weaknesses in its current implementations. Attackers can abuse timing side channels on the unmerge operation to leak sensitive data such as randomized pointers. Additionally, they can exploit the predictability of the merge operation to massage physical memory for reliable Rowhammer attacks. In this paper, we present VUsion, a secure page fusion system. VUsion can stop all the existing and even new classes of attack, where attackers leak information by side-channeling the merge operation or massage physical memory via predictable memory reuse patterns. To mitigate information disclosure attacks, we ensure attackers can no longer distinguish between fused and non-fused pages. To mitigate memory massaging attacks, we ensure fused pages are always allocated from a high-entropy pool. Despite its secure design, our comprehensive evaluation shows that VUsion retains most of the memory saving benefits of traditional memory fusion with negligible performance overhead while maintaining compatibility with other advanced memory management features.

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**;

KEYWORDS

Memory Management, Page Fusion, Side channels, Rowhammer

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132781

1 INTRODUCTION

On modern systems, different processes [13] or co-hosted virtualized environments [9, 10, 16] tend to have many memory pages that store similar data. To remedy this situation, operating systems and hypervisors perform *page fusion* by periodically scanning memory to find pages with the same content and merging them by keeping a single read-only copy. Writing to a shared page from either party triggers an unmerge by performing copy-on-write into a private copy. Given its memory saving benefits, page fusion has been applied to several classes of real-world systems, ranging from co-hosted virtual machines (VM) [9] and containers [23] in the cloud to desktop [13] and mobile [17] systems.

Unfortunately, existing page fusion systems are insecure. As we shall see, attackers can abuse the *unmerge* operation to build side-channel attacks to leak sensitive data, or the *merge* operation to construct reliable and deterministic Rowhammer attacks that flip bits in vulnerable memory modules. As a result, vendors either disable page fusion by default (e.g., Microsoft [2] or VMWare [41]) or sacrifice security in favor of capacity (e.g., Intel Clear Containers [23]). In contrast, we show that it is not only possible to fuse pages securely, but even retain its memory saving benefits and compatibility without sacrificing performance. This is possible due to a key observation: benefits of page fusion mostly come from idle pages in the system. Hence, we can apply heavy-weight mechanisms to secure page fusion on idle pages, while preserving performance by not fusing pages in the working set.

Attacking page fusion There are two classes of attacks that abuse page fusion: information disclosure and physical memory massaging [37] (i.e., preparing the state of physical memory for corrupting target data using a DRAM bit flip [27]). Known information disclosure attacks [11, 13, 25, 34, 39, 43] abuse the slow unmerge to detect whether another copy of a certain memory page exists in the system. Some attacks

even use advanced versions of the side channel to break address space layout randomization (ASLR) in the browser from JavaScript [13] or across VMs [11]. The recent Flip Feng Shui attack [37], on the other hand, abuses the predictability of the merge operation to control where sensitive data is stored in physical memory. In cloud environments, Flip Feng Shui allows an attacker VM to compromise a victim VM by performing a Rowhammer attack on (fused) RSA public keys in memory.

With a careful security analysis of current implementations of page fusion, we realize that the attack surface is much larger than previously assumed, raising even more challenges to build secure page fusion systems. On the information disclosure side, it is possible to detect merge events by observing changes in the physical memory or in the virtual memory translation data structures, using a variety of side channels on shared system resources (e.g., last-level cache). This allows an attacker to disclose sensitive data *by just reading from memory*. On the Flip Feng Shui side, it is possible to abuse the predictability of memory reuse to perform reliable Rowhammer attacks. This allows an attacker to mount Flip Feng Shui *even when merged pages are backed by new page allocations*. We show that while the implementation of page fusion in Windows is protected against plain Flip Feng Shui since new pages are allocated during merge, the attackers can overcome this hurdle by exploiting memory reuse patterns for fused pages.

Secure page fusion Our security analysis helps us derive a number of design principles that, once respected, can protect against all existing and future attacks on page fusion. To stop an attacker from disclosing sensitive information, a secure page fusion system should enforce the same behavior for all pages in the system, whether they are merged or not. We call this principle Same Behavior (**SB**). Every time we are making a decision on whether to merge a page or not, we remove all accesses to that page. The next access to this page, regardless of its merging status, results in a page fault. To stop Flip Feng Shui, we should always allocate a random physical page for backing the page that is a candidate for merging. We call this principle Randomized Allocation (**RA**).

While **RA** can be implemented with negligible overhead, **SB** can be expensive in terms of performance due to the increased number of page faults and can reduce fusion benefits since memory pages need to be unmerged even when read. Fortunately, neither is a problem in practice: a simple working set estimation can restrict page fusion to idle pages and dramatically reduce the additional number of page faults. This strategy has a small impact on memory savings, since, as we show, most benefits of page fusion come from idle pages. As a result, VUsion, our secure page fusion system built on top of the Linux kernel, provides similar benefits in terms of saving memory with minimal performance overhead (e.g., 2.7% on SPEC CPU2006 and 0.4% on memcached) compared to the default insecure implementation in the Linux kernel. We further address the non-trivial challenge of keeping VUsion compatible with popular memory management features of the Linux kernel such as transparent huge pages (THPs) [8, 12, 18, 29, 42].

Contributions We make the following contributions:

- The first study of page fusion in recent Windows operating systems, which reveals a different design than the widely studied Kernel Same-page Merging (KSM) in Linux (§2).
- The first study of previously known attack vectors on page fusion complemented by new attack vectors, which we use to draw up principles for secure page fusion (§4 and §5).
- The design and implementation of VUsion, a Linux-based secure page fusion system which follows these principles (§6 and §7).
- A comprehensive evaluation of security, performance and memory savings of VUsion when compared to KSM. Our results demonstrate that VUsion's design improves the security of KSM and only marginally reduces memory savings while preserving performance (§9).

2 PAGE FUSION

Page fusion is often used in situations where it is not possible to directly share memory that originates from the same content. For example, while it is possible to share pages for libraries across different processes *inside* a VM, the same is not possible *across* VMs. Hence, to find memory pages with the same content, a page fusion system should periodically scan the memory.

Once pages with duplicate content are found, only one copy is kept, and all the page-table entries (PTEs) of the sharing parties are updated to point to this copy without the write permission bit (i.e., they are *fused*). The duplicates can now be returned to the system. At any point in time, one party may decide to write to this (now) fused page, resulting in a page fault. To preserve the correct behavior, the system handles this situation with copy-on-write: a new page is allocated and filled with the content from the shared copy before updating the PTE of the writing party to this new page with the write permission bit so that the write can continue.

We now study how these steps are implemented in practice using two popular implementations in the Linux and Windows operating systems.

2.1 Linux Kernel Same-page Merging

The Linux kernel fuses memory pages in its KSM subsystem. KSM is opt-in and user processes that want page fusion should inform KSM via an `madvise` system call. The Kernel Virtual Machine (KVM) is a prime user of KSM and co-hosted virtualized environments in the cloud are an important target.

Scanning. Whenever a process registers a chunk of its virtual memory to KSM, KSM first locates all virtual memory areas (VMAs) associated to this chunk. In Linux, VMAs are contiguous areas of virtual memory and the (virtual) memory pages that belong to the same VMA share certain properties such as permissions. After finding the VMAs, KSM adds them to a list containing all candidate VMAs for fusion. KSM scans this list in a round-robin fashion. Every T milliseconds, the KSM thread wakes up and scans N virtual memory pages (belonging to one or more VMAs). T and N are configurable, for example on the Linux kernel version 4.10.10, the default values are $N = 100$ and $T = 20$, i.e., 5000 pages are scanned per second. The pages are merged in-line with the scan.

Merging. To detect sharing opportunities, KSM uses two red-black trees, called *stable* and *unstable*, that use the contents of the pages to balance themselves. The stable tree contains fused pages that are made read-only and are (often) shared between multiple parties. The pages in the unstable tree, however, are not protected against writes and their contents may change. Hence, the tree is not always perfectly balanced, but given that every page insertion and deletion triggers rebalancing, the tree mostly maintains its balance [9].

Figure 1 shows how KSM finds duplicate pages in the registered VMAs. For each page that is scanned, it is first checked whether another page with the same content already exists in the stable tree. If that is the case, then the page is merged by updating the PTE of the page that is being scanned to point to the page in the stable tree without the write permission bit and releasing the page back to the system (Figure 1-A). If that is not the case, KSM matches the page against the unstable tree. If a match is found, it makes the page read-only and puts it in the stable tree. It also removes the match from the unstable tree and makes it point to the stable tree page, similar to the previous case (Figure 1-B). Finally, if KSM finds no match in either tree, it puts the page in the unstable tree (Figure 1-C). Note that merging implies that the physical memory of one of the pages is used for backing the shared copy between various (distrusting) parties.

Unmerging. The pages in the stable tree are reference-counted (very much like any page in Linux). As soon as one party writes to a write-protected page, a copy-on-write event triggers, as mentioned earlier. This copy-on-write event drops the reference count by one. Only if the count reaches zero, the system removes the original page from the stable tree, which means that as long as there is even a single user, that page remains in the stable tree.

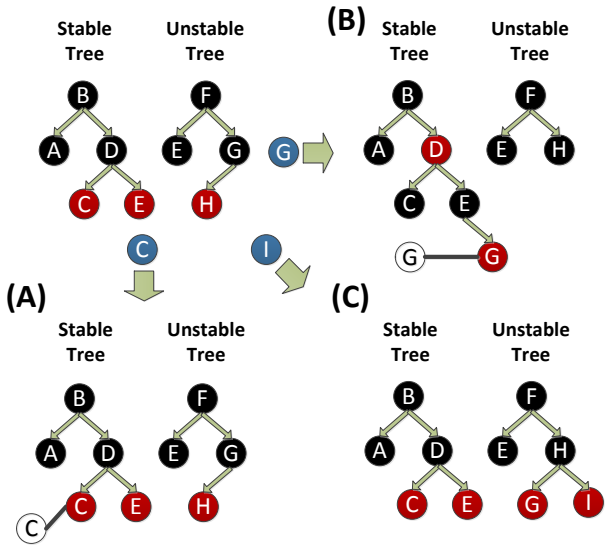

Figure 1: Modifications to the KSM red-black trees during merging. Blue circles are the pages that are being considered for fusion.

2.2 Windows Page Fusion

Microsoft turned page fusion on by default for Windows 8.1 and later releases. Duplicate pages generated by related processes in the absence of *fork* semantics are a primary target of page fusion on Windows systems. We have reverse engineered parts of the Windows kernel to gain insight into the mechanisms that Windows Page Fusion (WPF) uses for this purpose. Note that Microsoft recently disabled WPF after the Dedup Est Machina attack [13], at the expense of memory wastage. However, it is still important to study a different (insecure) design other than KSM to derive the necessary key principles for secure memory fusion.

Scanning. Compared to its Linux counterpart, WPF has no opt-in mechanism that allows user-space applications to register which memory pages can be merged. Instead, it scans all anonymous physical memory pages every 15 minutes and tries to merge as many pages as possible. WPF stores the metadata about the already merged pages in multiple AVL trees that have the same functionality as the KSM's stable tree. During each scan, WPF computes the hash of every physical page that is a candidate for merging in a list that is sorted by the hash value.

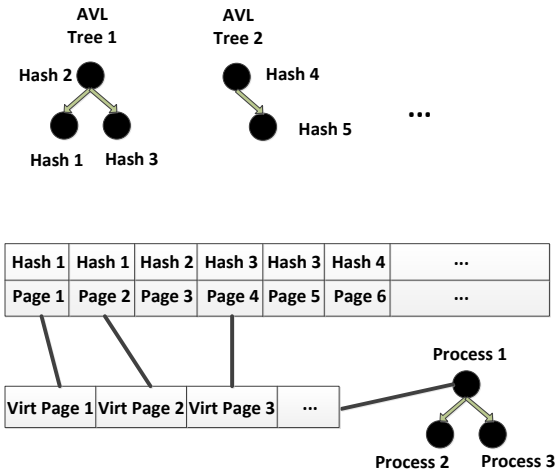

Figure 2: WPF's sorted list of hashes and processes.

Merging. With the sorted list of hashes, it is now time for WPF to start merging physical pages. For each physical page, through a reverse mapping, WPF determines the process that owns the page. Each process will have a list of these physical pages, sorted by their corresponding virtual addresses. These processes are inserted into a tree that is sorted by their memory management struct pointer (i.e., `_EPROCESS->Vm`). Figure 2 visualizes the relation between these structures.

WPF then performs the actual merging on a per-process basis. For each process, it first examines the content of each physical page in the process's list. If a page with the same content has been previously fused, then WPF will find it in one of the AVL trees. WPF then updates the corresponding PTE to point to the page in the AVL tree and returns the physical page back to the system. For pages that have not been previously fused, WPF checks whether there are pages with the same content. In case there are some, WPF adds a copy to one of the AVL trees, updates the relevant PTEs and returns the duplicate pages back to the system.

An important difference between KSM and WPF is that WPF allocates *new* physical pages for insertion into the AVL trees (i.e, the physical pages of the sharing parties are not used to back the fused page). Specifically, it allocates new pages using a specialized linear allocator to improve performance by not contending for the system-wide page allocator. This allocator scans the physical address space from the end and tries to reserve as many pages as necessary. If the allocator finds a page that is currently in use, it tries to steal this page from the owner. As a result, the allocated memory will be mostly contiguous, starting from the end of the physical address space. While this mechanism leads to some uncertainty in the selected page during a merge operation, we later show this design is still vulnerable to Flip Feng Shui.

Unmerging. Unmerging is performed similar to KSM using copy-on-write.

3 THREAT MODEL

We assume a strong threat model in line with prior work in the area [11, 13, 37], where an attacker can (1) directly interact with the page fusion system by crafting memory pages with her chosen contents, and (2) trigger bit flips using the Rowhammer vulnerability or other potentially exploitable reliability issues [15, 26]. In case page fusion is applied inside the OS, the attacker can remotely create memory pages with arbitrary contents in a malicious JavaScript application and in the case that page fusion is applied in a cloud setting, the attacker executing in a malicious VM can directly create arbitrary memory pages. The attacker pursues one of the following two goals:

- *Information disclosure*: abuse timing side channels induced by page fusion to detect fused pages in the system and disclose secrets.
- *Flip Feng Shui*: abuse physical memory massaging primitives induced by page fusion to land a target of interest into a chosen vulnerable physical page and mount Flip Feng Shui attacks.

Next, we describe how these attacks can be mounted in practice.

4 KNOWN ATTACK VECTORS

We now describe how an attacker can exploit known page fusion issues in order to achieve the goals mentioned in our threat model.

4.1 Information Disclosure

Existing information disclosure attacks based on page fusion are *unmerge-based*. They exploit timing side channels introduced by unmerge (or copy-on-write) events in traditional page fusion systems.

Writing to a merged page will trigger a copy-on-write event which is measurably slower than a normal write. The attacker can use this timing difference as a side channel to tell whether a page exists in the victim. In the past, researchers used this side channel to fingerprint applications, libraries, operating systems and to build covert channels [25, 34, 39, 43].

The CAIN [11] attack brute-forces pointers of other VMs randomized by Address Space Layout Randomization (ASLR) [35] by creating many guesses for possible pointers and checking which guess gets shared with the victim VM. Brute-forcing high-entropy data in this way requires a large amount of memory and becomes noisy in a Web browser.

Dedup Est Machina [13] shows that it is possible to leak high-entropy secrets in the browser by controlling the alignment of the secret, partially overwriting the secret with known data, or engaging in a birthday attack. The first two techniques allow for leaking only a part of the secret in the first fusion pass. Once a part of the secret is known, it is possible to leak another part in a subsequent fusion pass. The birthday attack relies on generating many secrets in the target process (in this case a JavaScript runtime) to increase the chance of guessing one of them.

These techniques show that page fusion can act as a weird machine [14] in the hands of an attacker able to time *unmerge* events for reading secret information from a victim process or VM.

4.2 Flip Feng Shui

Existing Flip Feng Shui attacks based on page fusion are *merge-based*. They exploit physical memory massaging capabilities introduced by merge events to land sensitive information in a vulnerable memory page and trigger hardware bit flips to corrupt it. Existing attacks of this kind are based on the DRAM Rowhammer hardware vulnerability. We now briefly provide some background information on how Flip Feng Shui triggers memory corruption using Rowhammer. We refer interested readers to the original Flip Feng Shui article for more information [37] .

DRAM architecture. Memory is internally organized in *rows*. Depending on the DRAM architecture, each row can span a number of pages. The rows themselves consist of memory cells. Each cell is made out of a capacitor and a transistor. The transistor controls whether the cell is being read or written into and the capacitor stores the actual one bit of data. Capacitors lose charge over time and if enough charge is lost, the stored data will be lost. To avoid this, the memory chip periodically refreshes the cells in each row.

Every time the CPU needs to read data from memory, the memory chip selects the corresponding row and loads it into the *row buffer*. The row buffer acts as a cache for rows, so the CPU can keep reading from it as long as the requested addresses belong to the same row. Similar to refreshing, writing the row from the row buffer back (before opening another row) recharges the capacitors.

Rowhammer. Kim et al. [27] noticed that if two rows (i.e., aggressor rows) are activated in succession many times within a refresh interval (e.g., 64 ms), some bits that are in adjacent rows (i.e., victim rows) to these aggressor rows will start to flip. This is because a small charge leaks from cells in the victim rows every time aggressor rows are loaded into the row buffer and if this happens fast enough between refresh intervals, enough charge will leak from some cells in the victim rows. This causes the value of the high bit to switch to a low bit which is observed as a bit flip from the perspective of the CPU. Reading from memory in this fashion with the aim of triggering bit flips has been dubbed as Rowhammer. A variant of Rowhammer where the aggressor rows are one row apart, called double-sided Rowhammer, is known to trigger more bit flips reliably in the victim row that is in the middle.

Exploitation. Being able to change memory without having control over it has strong security implications. Recent studies show that it is possible to abuse these bit flips to escalate privilege by flipping bits in the PTEs [38, 40] or escape the JavaScript sandbox by flipping bits in data pointers [13]. The Flip Feng Shui attack [37] shows that it is possible to reliably circumvent the strong hardware-enforced virtual machine abstraction using Rowhammer and the merge operation of page fusion.

To mount a Flip Feng Shui attack, the attacker VM first finds memory cells that are susceptible to exploitable Rowhammer bit flips in the physical memory that backs her VM. At this point, the attacker needs to force the system to store the sensitive data of a victim VM (e.g., cryptographic keys) on the physical page with the exploitable bit flip. Page fusion makes this step extremely easy: in the case of KSM, for example, the physical memory of one the sharing parties is chosen to back the merged page. Hence, if attackers want to corrupt a page in the victim, all they need to do is write that page content on one of their own pages that is vulnerable to Rowhammer bit flips. If KSM chooses the attacker's physical page, then the victim page gets corrupted when the attacker triggers Rowhammer. The last step essentially breaks the copy-on-write semantics necessary for safe and correct behavior of page fusion.

Flip Feng Shui shows that page fusion can act as a *physical memory massaging* primitive in the hands of attackers, making it possible for them to control which physical memory pages should be used to back sensitive data through *merge events*. In turn, this allows them to reliably corrupt sensitive information from a victim process or VM.

5 NEW ATTACK VECTORS

In this section, we describe two new classes of advanced attack vectors against page fusion. The first class targets *merge* events in page fusion to mount information disclosure attacks. The second class targets *reuse* properties of page fusion to mount Flip Feng Shui attacks. Along with the attack vectors detailed earlier, they help us derive design principles for secure page fusion that we adhered to in the implementation of VUsion.

5.1 Information Disclosure

Traditional page fusion is characterized by well-defined *merge* and *unmerge* events. Existing attacks exploit copy-on-write

Attack	Issue	Abused mechanism	Attacker operation(s)	Mitigation
Copy-on-write [11, 13]	Slow write	Unmerge	Write	**SB**
Page color (new)	Physical address changes	Merge	Read or fetch	**SB**
Page sharing (new)	Sharing changes	Merge	Read or fetch	**SB**
Translation (new)	Translation changes	Merge	MMU ops	**SB**
Flip Feng Shui [37]	Predictable merge	Merge	Memory massaging	**RA**
Reuse-based Flip Feng Shui (new)	Predictable reuse	Reuse	Memory massaging	**RA**

Table 1: Summary of attacks against page fusion and how design principles used in VUsion mitigate them.

side channels associated with *unmerge* events, but we now show an attacker can also exploit several timing side channels associated with *merge* events to detect fused pages without writing to these pages.

Page color changes. Page colors refer to how physical pages map on cache sets such as those of the last-level cache (LLC). For example, the Intel Xeon E3-1240 v5 processor used in our evaluation partitions its 8 MB LLC into 8192 cache sets of 16 cache lines of 64 bytes each, and each 4 KB page covers 64 cache lines in 64 cache sets. If the first cache lines of two distinct physical pages share the same cache set, the other 63 cache lines are also guaranteed to land on the same cache sets as well. This allows us to color different physical memory pages in the system only based on their first cache line. For example, our Intel Xeon E3-1240 v5 processor has $8192/64 = 128$ different page colors.

To color a page, we first build eviction sets for all possible colors in the system. An eviction set is a sequence of memory addresses that all map to the same cache set and "covers" the cache set completely [32]. Thus, by accessing all the addresses in the eviction set, we clear out all other data from the corresponding cache set. Using a `PRIME+PROBE` attack [30, 33], it is now possible to determine the current color of a page. The attacker first primes a certain color C by accessing its eviction set. After that, the attacker reads from the target page. During the probe phase, if accessing the eviction set for C is slow, it means that the target page is of color C. By waiting for a page fusion pass to occur, the attacker can detect whether a target page has been fused if its color is no longer C. This attack assumes that a new page is allocated to back the shared copy (e.g., WPF) and is successful if the new page has a different color. That is $P_{success} = \frac{C_t - 1}{C_t}$ where C_t is the total number of colors. For example, in our testbed, $P_{success} = \frac{128-1}{128} = 0.99$. Our implementation of this attack can find eviction sets for all colors in a few minutes and detect changes in the color after a fusion pass.

Page sharing changes. It is tempting to think that if we always randomize the physical location of a page considered for fusion (regardless of a merge), then allowing reads stops information disclosure while conserving performance and benefits of page fusion. Unfortunately, this design is still insecure since attackers can detect whether pages are fused over shared resources. An attack over the LLC is similar in spirit to a 1-bit version of `FLUSH+RELOAD` [44]. We first flush the target page from the cache by either executing a cache flush instruction or accessing a cache eviction set (i.e., `FLUSH`). Next, we make the victim access the secret page for which we want to check whether it is merged. Finally, we access the target page again and measure how long it takes (i.e., `RELOAD`). If the access is slow, it means that the data was not in the cache which implies that the victim did not access the page. If, on the other hand, the access is fast, it means that the victim has accessed the exact same physical location as the attack, suggesting that a merge event has occurred as a result of page fusion.

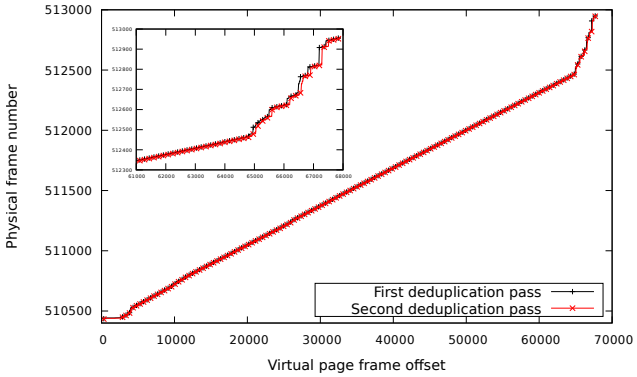

Figure 3: Ideal physical memory massaging in WPF by exploiting the deterministic behavior of page allocation during page fusion passes.

Translation changes. Finally, it is also possible for an attacker to detect a merge event indirectly, by observing changes in the behavior of pages that are physically adjacent to the target page. For example, KSM breaks a huge page when merging a 4 KB page inside of it. This means that the other adjacent pages that make up the huge page now require an additional page table look up for the last translation level. As our recent AnC attack shows [19], attackers can easily observe the additional page table lookup in the LLC and detect a merge event of the target page.

5.2 Flip Feng Shui

At the physical memory management level, page fusion systems either *merge* two duplicate physical pages into one (and discard the other), as done on Linux, or allocate a third physical page by *reusing* memory from a dedicated pool of pages to improve performance. Existing attacks exploit physical memory massaging capabilities associated with the *merge* behavior, but we now show that an attacker can also perform memory massaging by exploiting *reuse* behavior.

Our first attempt at reproducing Flip Feng Shui on top of WPF failed. This was due to the fact that Windows allocates a new page when updating its stable AVL trees as discussed in §2.2. This means that memory massaging is no longer reliable. At this point, we started looking into memory reuse patterns during page fusion. Interestingly, we found that the *reuse* behavior of WPF provides much better predictability than that of the standard system physical page allocator, encouraging fusion-based memory massaging rather than a system-wide attack (e.g., [40]).

Given that WPF merges all possible candidates in one go, it knows how many (new) physical pages are necessary for backing fused pages. We found that, as a performance optimization, WPF calls the `MiAllocatePagesForMdl` routine with the number of physical pages it needs. Reverse engineering this routine showed that it tries to allocate contiguous physical pages from the end of memory, but allows for holes if physical pages cannot be reclaimed.

This has an interesting implication for physical memory massaging: we can get close to perfect memory reuse if we can directly use memory that is backing fused pages for mounting the attack. Furthermore, Flip Feng Shui relies on huge pages for double-sided Rowhammer but we do not always have access to huge pages on Windows (e.g., in the browser). As fused pages are, due to how `MiAllocatePagesForMdl` works, mostly contiguous we can use them as an alternative to perform a reliable double-sided Rowhammer. Hence, we can follow these steps for templating with WPF:

(1) Allocate a large number of memory pages. (2) Write pair-wise duplicates into these pages. (3) Wait for WPF to merge each pair. (4) Execute Rowhammer on the fused pages for finding flips.

Note that we control the order of these pages in physical memory based on the hash of their contents as discussed in §2.2, necessary for performing double-sided Rowhammer. Once we find some exploitable bit flips, we trigger copy-on-write to release all the pages back to the system. In this stage, we write security-sensitive data on a large number of pages, such that every page is duplicated extacly once. After another fusion pass, we complete the attack by triggering Rowhammer again to corrupt the security-sensitive data, very much like the original Flip Feng Shui.

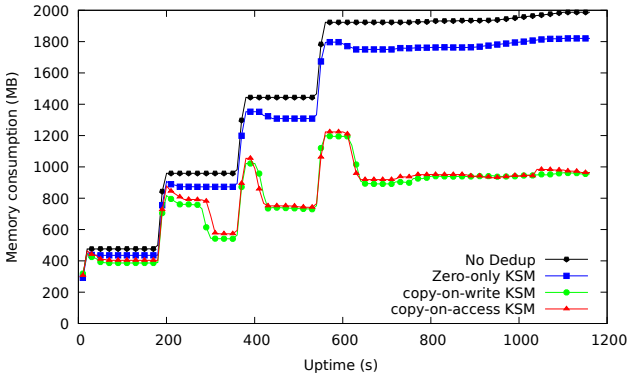

Figure 4: Comparing the effect of copy-on-access with copy-on-write on fusion rates.

Figure 3 shows a near-perfect physical memory reuse between the two page fusion passes with WPF. Note the physical memory pages at the end of available memory to the guest (2 GB or 524,288 physical pages). This shows that we can perform Flip Feng Shui even when new pages are allocated to back merged pages. One potential source of unreliability is the fact that the order in which processes are selected for page fusion is not known to the attacker as described earlier in §2.2. Further, the content hash of the target pages influence the order of physical memory allocation. We resolve both problems by allocating a large number of pair-wise duplicated targets (similar to [13]) to minimize the impact of other processes in the system.

5.3 Summary

Table 1 summarizes the attacks that we described in this paper with the underlying issues that permit information leakage or physical memory massaging. Our new merge-based information disclosure attacks show that just reading from fused pages is enough to leak information, obviating even the need to rely on copy-on-write events. Further, it is possible to observe changes in the virtual memory translation to detect page fusion indirectly without accessing the target page at all. Finally, our reuse-based attack shows that it is possible to perform Flip Feng Shui even when new allocations are used to back fused pages.

We implemented all these attacks on page fusion to verify their practicality. We further performed a deeper analysis of the attack surface of page fusion involving other side channels (e.g., the TLB and the DRAM row buffer) which we omit due to space limitation. Our conclusion is that the design principles that we describe next protect against all these attacks.

6 DESIGN PRINCIPLES

To stop all the potential attacks on page fusion, including the ones we described in §4 and §5, we follow two main design principles. The first principle stops information disclosure and the second principle stops Flip Feng Shui.

6.1 Stopping Information Disclosure

To protect against information disclosure attacks, we should stop the ability of an attacker to detect whether memory pages are fused via either merge or unmerge events. We can achieve this by ensuring that every page that an attacker tests always behaves the same, whether it is merged or not. We call this principle Same Behavior (**SB**) and discuss how we can enforce it.

Merge-based attacks. It should be clear to the reader by now that sharing memory between mutually distrusting parties is not safe even if shared pages are write-protected. Hence, to preserve security, no memory pages should be shared between distrusting parties. However, not sharing memory pages across security boundaries means that we need to disable page fusion. To resolve this dichotomy, we propose sharing pages with the same contents that are not accessed by either party. We do this by removing all access permissions to pages that are shared. Any access to a shared page will cause a trap and lead to an explicit copy of the page contents to a new page in a copy-on-access fashion. We call this mechanism share xor fetch or, in short, **S⊕F**. **S⊕F** ensures sharing *only* pages that are not accessed or prefetched (countering implicit accesses via the `prefetch` x86 instruction [20]). This design converges to two sets of pages: pages in the working set that are not fused and idle pages that are candidates for fusion.

S⊕F potentially reduces fusion rates as pages which are (continuously) read or executed can no longer be shared. To investigate this, we modify KSM to unmerge on any page fault and compare it with the baseline which only unmerges on writes. Figure 4 shows the result of the experiment when starting four VMs with five minutes between launch times, each running an Apache server. After the page fusion process stabilizes, we can see that copy-on-access only marginally reduces fusion rates (1%). The reason for this is the fact that most of the fusion benefits come from idle pages in the system (e.g., the page cache) as we will later show in our evaluation. We also experimented with merging only zero pages to mitigate information disclosure as proposed before [13]. Compared with when page fusion is off, zero-pages account for only 16% of all duplicate pages, as shown in Figure 4. This is insufficient in cloud settings and further justifies our efforts in securing full (vs. zero-page only) page fusion. We also note that zero-page fusion is not by itself secure against Flip Feng Shui. We will further experiment with the effects of **S⊕F** on fusion rates and performance later in our evaluation.

Take-away: **S⊕F** prevents attacks that disclose information based on detecting a merge event because while the attacker can check whether a page is a candidate for fusion, she cannot infer whether it is actually fused. We now discuss how we prevent information disclosure through unmerged-based attacks.

Unmerge-based attacks. While **S⊕F** protects page fusion against merge-based attacks, attackers can still use the difference in the behavior of merged and unmerged pages to detect unmerge events by, for example, measuring if accessing a page generates a lengthy page fault. To resolve this problem, all memory pages should behave the same, whether they are merged or not—enforcing **SB**.

A simple way to enforce this is by a mechanism which we call Fake Merging (**FM**). **FM** ensures that pages that are not merged behave the same as pages that are. For this purpose, fake merging removes all access permissions and performs copy-on-access for non-shared pages as well. Again, **FM** sounds prohibitively expensive, but we will show that in realistic settings the performance penalty is negligible. The main reason is that page fusion systems take a long time to scan memory in order not to interfere with the main computation and, as a result, the performance penalty is amortized over long periods of time. Furthermore, we can apply working set estimation to reduce the number of additional page faults caused by **FM** without compromising security. Our working set-based optimization, discussed in §7.2, exploits the intuition that **SB** naturally converges to performing fusion on cold (and highly fusable) pages in the system.

Take-away: **FM** prevents the popular copy-on-write side channel since all first accesses to attacker-controlled pages result in copy-on-access until the next fusion pass.

6.2 Flip Feng Shui attacks

To defend against Flip Feng Shui-like attacks, we need to eliminate the attacker's capability to control how the page fusion system chooses the physical page that backs the merged pages. We enforce this by making sure that the page fusion system does not use physical memory in a predictable manner. We achieve this by properly randomizing page allocations on merge events. We call this principle Randomized Allocation (**RA**).

To protect against merge-based attacks [37], we need to always allocate a new page on a merge event. Unfortunately, as we showed in our new Flip Feng Shui attack against Windows, simply allocating a new page to back a shared page is not enough to prevent an attacker's ability to massage the physical memory. The reason is that efficient physical memory allocators often promote predictable reuse to reduce overhead. Hence, **RA** should be enforced either by randomizing the

system-wide page allocator or at the page fusion system itself. Since randomizing the system-wide page allocator has non-trivial performance and usability implications due to the inherent fragmentation, we opt for enforcing **RA** at the page fusion system directly.

Take-away: **RA** prevents Flip Feng Shui-like attacks since attackers cannot predict the physical page that may back a fused page.

6.3 Discussion

Enforcing **SB** has interesting implications for page fusion. Pages in the working set will no longer fuse with other (idle) pages. This design basically partitions the candidate memory into two sets: the working set which is not considered for page fusion and the idle set which may or may not get merged. It becomes impossible for the attacker to tell whether pages in the idle set are merged since we remove all access/fetch permissions to these pages. Without access/fetch permissions, these pages cannot be stored on shared resources in the system such as the LLC or the DRAM row buffer, mitigating the new attacks we discussed in §5.1.

Enforcing **RA** will stop Flip Feng Shui and its new reuse-based variant which we discussed in §5.2) by randomizing allocations that back fused pages. In Table 1, we also detail which principle stops which classes of attack.

7 IMPLEMENTATION

We now discuss how VUsion adheres to **SB** and **RA**. We implemented VUsion on top of the Linux kernel version 4.10.0-rc6 and reused most of KSM's original implementation and kernel tracing functionality. Our patch changes only 846 lines of code in the Linux kernel. This suggests that the proposed modifications are realistic and that VUsion contributes a practical page fusion system for production use. We assume 4 KB pages in this section and expand our implementation to huge pages in the next section.

7.1 Enforcing the Design Principles

Share XOR Fetch. To implement **S⊕F**, we need to intercept all accesses to a given page. Modifying the present bit in the PTEs to trigger a page fault on access is one possibility, but it requires intrusive changes to the Linux kernel as this bit is used for tracking memory pages in many places in Linux. Instead, we opted to use the reserved bits. According to the Intel and AMD manuals [6, 24], if the reserved bits are set, the processor generates a page fault on access regardless of the permission bits in the corresponding PTE. In the page fault handler, we check these bits and perform copy-on-access if they are set. To prevent `prefetch`-based attacks, in turn, we also set the *Caching Disabled* bit in the PTEs.

Fake Merging. To enforce **SB**, we also set the reserved bits in PTEs of non-shared pages and perform copy-on-access similarly to pages that are shared. We ensure that both shared and non-shared pages follow the same code paths to avoid opening other timing channels that an attacker could use to detect merge events. More specifically, we make three different design decisions compared to KSM.

(i) KSM uses the unstable tree to avoid write-protecting the pages that are being considered for fusion. This opens up a side channel since pages that become merged are immediately write-protected compared to pages that remain in the unstable tree. Fortunately, to enforce **SB**, we need to remove access permission from *any* page that is being considered for fusion. Hence, VUsion simply does not require an unstable tree and is hence protected from this side channel. (ii) Without additional care, pages that are merged take shorter during copy-on-access than pages that are fake merged. This is due to the fact that pages which are fake merged can be freed in the page fault handler, as references to the page drop to zero. This entails an expensive interaction with the buddy allocator. To counter this, we perform deferred free by queueing these pages and freeing them in the background. Note that deferred frees are already common practice in the kernel with the advent of RCU lock. The real merge also queues a dummy request to ensure the execution of the same instructions for both merge and fake merge. (iii) An advanced attacker could perform a page coloring attack on the page fault handler during copy-on-access to infer the color of the source page. If this is done across multiple scans, the attacker can infer a merge with high probability if the color of the source page does not change. We hence select new (random) physical pages to back pages that are merged or faked merged during each scan. Thanks to our working set-based optimization detailed below, these extra costs are only incurred for cold pages with little impact on performance.

Randomized Allocation. We reserve 128 MB of physical memory in a cache to add 15 bits of entropy to physical memory allocations performed by VUsion during both merging and unmerging. Under these conditions, an attacker seeking to abuse page fusion in a Flip Feng Shui attack can only mount a probabilistic (and thus unreliable) attack, where a specific vulnerable template is controllably reused by the allocator with a probability of only 2^{-15}, providing much more entropy than the fairly predictable standard page allocator [40].

7.2 Working Set Estimation

A naive version of VUsion would assume that all the pages in the system are idle in each fusion round and the pages that are actually in the working set will trigger a page fault regardless of whether they were merged. This implementation, while

secure, results in a considerable number of page faults when removing access to pages in the working set.

To address this problem, VUsion estimates the working set of registered memory and does not consider it for fusion. Doing so allows it to significantly reduce the number of page faults, improving performance. To this end, we use the idle page tracking facility in the Linux kernel [1]. During a scan, we check if the page has not been accessed for a period that can be controlled in VUsion. If that is the case, then VUsion considers the page for fusion.

Note that working set estimation does not reveal any information to the attacker except that the page is being considered for fusion (but may or may not be actually fused) since it is idle. In other words, any side channel attack can only leak already known information: which of the attacker's own pages are in the working set.

8 TRANSPARENT HUGE PAGES

Huge pages are becoming increasingly important for modern systems as the working sets of applications increase in size. With many workloads, virtualization also greatly benefits from huge pages, owing to the higher cost of a TLB miss in the guest VM [8, 12, 18, 29, 42]. `khugepaged` is a Linux kernel daemon that runs in the background and transparently collapses consecutive physical pages into huge pages (THPs). Conversely, KSM breaks up THPs again whenever there is a sharing opportunity in them. Unfortunately, as discussed in §5, this opens up new side channels to detect merge events. Here, we discuss a secure implementation of THPs for our design, making VUsion deployable in practice. Our design for THPs follows that of Ingens [29] while addressing security issues such as translation attacks.

8.1 Handling Idle and Active Pages

Since the difference between huge and normal pages can be used to detect a merge event, we have to ensure that pages that are being considered for fusion are either all huge or normal to enforce **SB**. Since VUsion considers only idle pages for fusion, the size of the pages does not affect performance. Hence, we should opt for maximizing fusion rate. Since even a single byte difference makes it impossible to merge pages, sharing opportunities will be greater for normal pages. As a result, every time we consider a THP for fusion, we first break it up into normal pages. As mentioned earlier, the only information this provides to the attacker is that this THP is idle and a candidate for fusion. Since all pages considered for fusion are now small, we stop the attacks based on the difference in translation of huge and normal pages.

The pages in the working set are not candidates for fusion, but they are important for performance. It is possible that a huge page becomes partially idle and partially active. This creates a performance versus capacity trade-off. On x86-64, for example, there are 512 pages in a huge page. At any point in time, any number K of these pages could be active. If we consider the huge page active if $K \geq n$ (at least n active 4 KB pages), then $n = 1$ provides the best possible performance (conserving huge pages) while bigger values of n will provide more fusion opportunity, increasing available capacity. VUsion provides support for both high performance (a la Ingens [29]) or maximum fusion rate (a la KSM) while preserving security. Recent work shows how one can optimize n dynamically depending on the workload [21].

8.2 Securing khugepaged

As mentioned, we need to collapse normal pages that become active into huge pages to improve performance. Fortunately the background `khugepaged` thread performs this for free. However, we must be careful to prevent it from collapsing (fake) merged pages back into huge pages. Otherwise, an attacker can perform the translation attack using pages that are next to the target page.

To this end, we again use the idle page tracking mechanism available in the Linux kernel. If one (or more) of 512 pages that can potentially form a huge page is active, we will (fake) unmerge the other 511 pages if any of them is (fake) merged. As part of transforming 512 contiguous virtual pages into a huge virtual page, khugepaged will copy the contents of these pages into 512 contiguous physical pages. This is safe because all of these pages are first (fake) unmerged. This way, `khugepaged` preserves the **SB** semantics by securely collapsing pages in the working set into huge pages.

To summarize, VUsion enforces **SB** on huge pages by breaking them before (fake) merging them, and khugepaged only collapses pages if their surrounding pages are first (fake) unmerged before forming a huge page with them.

9 EVALUATION

We evaluate three aspects of VUsion compared to original Linux/KSM: (1) Does VUsion stop all the attacks reviewed in §4 and §5? (2) How does the performance compare? (3) How do the fusion rates compare?

Benchmarks. We run synthetic microbenchmarks to evaluate the security of VUsion. To evaluate performance and memory saving, we use various benchmarks that stress different parts of the system. To fully stress the memory subsystem, we use `Stream` [3], a synthetic memory bandwith benchmark. We further use SPEC CPU2006 and PARSEC as general-purpose memory-intensive benchmarks, Postmark as a file system benchmark, and Apache (httpd), memcached and Redis as server benchmarks. Unless otherwise specified, the benchmarks run with four VMs configured with a virtual core and 2 GB of RAM. One of the VMs runs the benchmark while

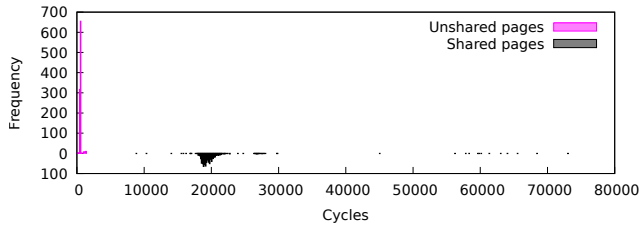

Figure 5: Freq. dist. of timing 1,000 writes in KSM.

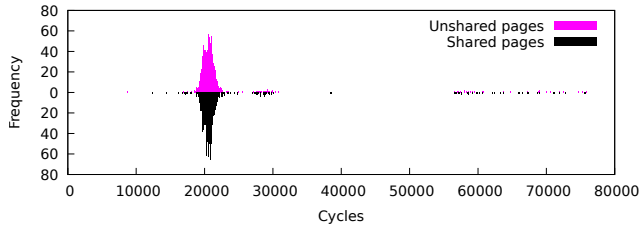

Figure 6: Freq. dist. of timing 1,000 reads in VUsion.

others provide load for page fusion. We also test a diverse set of VMs to observe the effects on fusion rates.

Testbed. We use a 4-core Intel Xeon E3-1240 v5 processor running at 3.5 GHz with 24 GB of DDR4 memory as our evaluation testbed. We further experimented with a dual processor Xeon E5-2650 v2 system with 32 GB of memory and obtained similar results which we do not include for brevity. we configure both VUsion and KSM with KSM default values (i.e., T = 20 ms and N = 100 pages). We run server benchmarks using a client machine over a 1 Gbps network. We configure the VMs' virtual NICs using virtio with vhost enhancements for high-performance I/O.

9.1 Security

We show that VUsion enforces **SB** and **RA** by timing reads and writes to pages that are shared or unshared, and checking whether the page allocations are truly random.

Enforcing **SB**. Figure 5 shows the frequency distribution of timing 1,000 writes in KSM after a fusion pass. The two distinct peaks for shared and unshared pages show the copy-on-write side channel present in KSM. Figure 6 shows the results of 1,000 reads in VUsion. In the case of VUsion, there is no visible difference between shared and unshared pages since both cases trigger copy-on-access events. The results for writes are similar to reads. To gain more confidence, we perform a Kolmogorov-Smirnov test to see whether the timing events for merged and unmerged pages follow the same distribution in VUsion. The calculated p-value is high (0.36) which means that we do not reject the hypothesis that these events are from the same distribution.

These tests show that VUsion conforms to the **SB** principle that we described in §6. We note that our analysis can guarantee correctness, but not the complete absence of other (arbitrary) side channels. While recent work shows promising results for formal constant-time verification [7], doing so is difficult in the context of the Linux kernel but an interesting direction for future work. Further, our experiments with the `prefetch` instruction confirm that setting the "Caching Disabled" bit in the PTEs of (fake) merged pages stops the recently reported side channel [20] where pages can be *prefetched* into the cache without access permissions.

	copy MB/s	scale MB/s	add MB/s	triad MB/s
No Dedup	11109	10690	12463	12342
KSM	11035	10644	12431	12291
VUsion	11019	10695	12423	12280
VUsion THP	11022	10646	12441	12271

Table 2: Performance of the Stream benchmark.

Enforcing **RA**. We record the offsets of pages chosen for merge and fake merge in VUsion when executing two VMs. We then perform a Kolmogorov-Smirnov goodness of fit test against the uniform distribution. The calculated p-value is high (0.44) which means that the test does not reject our hypothesis that physical page allocations in VUsion follow the uniform distribution.

9.2 Performance

To gain a complete understanding of possible performance issues when enforcing **S⊕F**, we first quantify the nature of common merge events. Table 3 shows the type of merged pages in one of the four VMs. Interestingly, most possibilities for fusion come from idle pages in the system (i.e., buddy) allocator and the page cache. To experiment with the former, we evaluate VUsion with the `Stream` microbenchmark as well as the memory-intensive SPEC and PARSEC benchmarks. To experiment with the latter, we evaluate VUsion with the file system-intensive Postmark benchmark. We also complement our analysis with a server application benchmark, important to evaluate the impact on common cloud workloads. Our results are as follows.

Stream. Table 2 shows the available memory bandwidth when running Stream in a VM with/without KSM and in VUsion with/without THP enhancements. In all cases, the additional overhead introduced by KSM or VUsion is below 1%. This is due to the fact that the default scanning rate is slow (5000 pages/second) and the few additional page faults only marginally affect the memory bandwidth.

SPEC CPU2006. Figure 7 shows the performance overhead of KSM and VUsion with/without THP enhancements compared to when page fusion is turned off over the general-purpose SPEC CPU2006 benchmark suite. Considering the

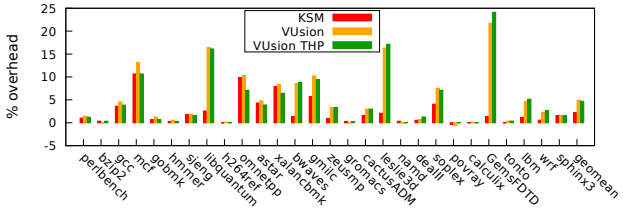

Figure 7: Performance overhead on SPEC CPU2006.

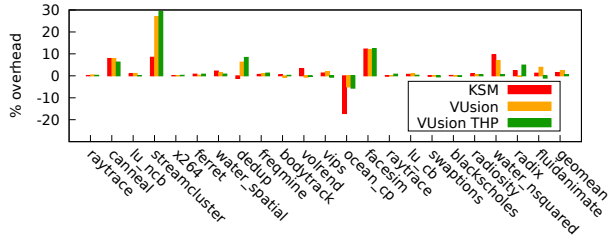

Figure 8: Performance overhead on PARSEC.

geometric mean, KSM adds 2.2% overhead to the baseline. VUsion adds another 2.7% (overall 4.9%) and enabling huge pages adds 2.4% (overall 4.6%). Most of the benchmarks are insensitive to the additional page faults caused by enforcing **S**⊕**F**. Similar to `Stream`, the additional page faults are bounded by the number of pages that become active over a page fusion period (i.e., a few hundred seconds). Hence, we conclude that VUsion enforces **SB** and **RA** with minimal performance overhead in a general-purpose workload.

PARSEC. To study the behavior of the system under concurrent workloads, we experiment with PARSEC. We increase the number of virtual cores to four to increase possible parallelism in our test VM. Figure 8 shows the outcome. `fmm` and `barnes` require more than 8 GB of RAM to execute and the `netapps` category hangs in our unmodified setting and hence we excluded. KSM adds 1.7% of performance overhead on top of the baseline. Consideraig the geometric mean, VUsion slightly degrades KSM performance by 0.5% (overall 2.2%) while VUsion's THP enhancements improve KSM's performance by 1.4% (overall 0.8%). These results further prove that VUsion introduces low overhead and can even improve KSM's performance.

Postmark. Table 4 shows the number of transactions per second in Postmark, a benchmark emulating a mailserver that heavily interacts with the file system. KSM degrades performance by 1.5% while VUsion degrades performance by 2.9%. VUsion with THP enhancements slightly improves the performance relative to KSM (0.2% improvement over baseline). These results suggest that VUsion can secure page fusion without performance penalty in workloads that benefit from it the most.

	page cache (%)	buddy (%)	kernel (%)	rest (%)
KSM	51.8	38.4	6.9	2.9
VUsion	51.2	38.6	6.6	3.6
VUsion THP	50.4	32.8	6.3	10.5

Table 3: Contribution of different page types to page fusion.

	mean (tx/s)	min (tx/s)	max (tx/s)
No dedup	3237.3	3191	3289
KSM	3221.7	3215	3232
VUsion	3178.7	3154	3191
VUsion THP	3246.3	3222	3285

Table 4: Performance of the Postmark benchmark.

	kreq/s (rel.)	lat. 75%	lat. 90%	lat. 99%
No Dedup	22.03 (100%)	1.34 ms	1.95 ms	4.49 ms
KSM	18.42 (83.6%)	1.59 ms	2.34 ms	5.87 ms
VUsion	18.28 (82.3%)	1.64 ms	2.47 ms	6.51 ms
VUsion THP	21.18 (96.1%)	1.37 ms	2.03 ms	5.55 ms

Table 5: Performance of the Apache server.

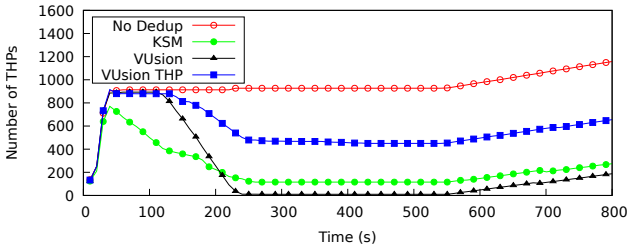

Figure 9: Conserving THPs with the Apache benchmark.

Apache. We use Apache 2.4.18 with the default prefork module and wrk [5] to generate load on the server at remote CPU saturation using 20 concurrent connections and 10 threads for a duration of 500 seconds. Table 5 shows the throughput and various latency percentiles that we achieve under different configurations. In the case of throughput, for Apache, KSM incurs 20.0% of overhead on the baseline while VUsion adds a marginal 0.4% overhead. THP enhancements in VUsion improve the performance relative to KSM by 12.7%. Latency follows a similar trend: VUsion provides similar performance to KSM while the THP enhancements in VUsion improve the latency relative to KSM.

Figure 9 shows the number of huge pages during runtime of the Apache benchmark. As expected, the number of huge pages is higher in VUsion with THP compared to KSM. More importantly, these huge pages are part of the working set,

	Redis (kreq/s)	Memcached (kreq/s)
No dedup	175.30 (100%)	167.5 (100%)
KSM	155.66 (88.8%)	163.97 (97.9%)
VUsion	155.09 (88.4%)	155.11 (92.6%)
VUsion THP	163.8 (93.4%)	163.87 (97.8%)

Table 6: Throughput of Redis and Memcached.

	Redis SET (ms)			Memcached SET (ms)		
Percentile	90.0	99.0	99.9	90.0	99.0	99.9
No Dedup	1.6	2.4	4.9	1.7	2.5	3.5
KSM	1.7	2.8	6.7	1.8	3.2	6.3
VUsion	1.8	3.0	7.3	2.0	3.6	6.3
VUsion THP	1.6	2.8	7.0	1.8	2.9	4.7

	Redis GET (ms)			Memcached GET (ms)		
Percentile	90.0	99.0	99.9	90.0	99.0	99.9
No Dedup	1.6	2.4	5.0	1.7	2.5	3.5
KSM	1.7	2.8	7.7	1.8	3.2	6.2
VUsion	1.8	3.0	7.0	2.0	3.6	6.2
VUsion THP	1.6	2.7	6.7	1.8	2.9	4.7

Table 7: Latency of Redis and Memcached.

improving performance as reported in Table 5. During the runtime of the benchmark, the VM allocates more memory with demand paging. Initially the allocations are backed by huge pages, but VUsion without THP enhancements breaks them down when considering them for fusion.

Key-value stores. To experiment with server applications that have a large memory footprint, we experiment with Redis (version 3.0.6) and Memcached (version 1.4.25), two popular key-value stores. We use memtier_benchmark [4] with the default configuration for generating load: using 4 threads, 50 clients, a set/get ratio of 1:10, and 32-byte objects from a 10 million key space.

Table 6 shows the throughput of Redis and Memcached. Redis follows a similar trend as Apache discussed earlier: KSM and VUsion provide similar throughput and the THP enhancements in VUsion improve overall performance. The throughput of Memcached with VUsion is more severely impacted (5.3% worse than KSM), but the THP enhancements bring the throughput of VUsion close to KSM. Table 7 shows the latency of GET and SET requests in both key-value stores. We observe similar trends again: the latency with VUsion is marginally impacted compared with KSM and the THP enhancements in VUsion improve the results, even at the tail but with the exception of SET requests (4.5% overhead compared to KSM).

9.3 Fusion Rates

To study fusion rates with VUsion, we showcase three scenarios. The first scenario shows fusion rates of idle VMs and how quickly VUsion fuses memory compared to KSM.

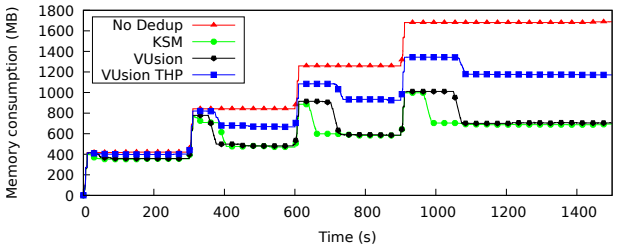

Figure 10: Memory consumption of idle VMs.

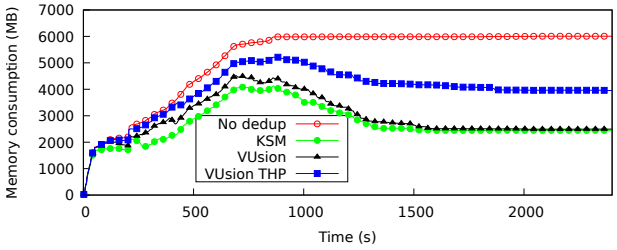

Figure 11: Memory consumption of different VMs.

The second scenario shows the scalability of VUsion, namely when starting many different VMs of different types. The third scenario shows the memory consumption during our Apache server benchmark.

Idle VMs. Figure 10 shows the total memory consumption of four VMs started 5 minutes after each other under different systems. This shows that, in an idle setting (expected on many cloud hosts), VUsion's fusions rates converges to that of KSM despite the conservative **S⊕F** policy. VUsion, however, takes longer to merge pages when compared to KSM. As we discussed in §7.1, KSM merges pages as soon as it finds a match in its stable tree. VUsion, however, waits one scan round before (fake) merging to enforce **SB**. Further, waiting one round allows VUsion to reduce page faults by focusing on pages that are not in the working set.

Diverse VMs. To understand the effects of VUsion on fusion rate in a more realistic setting, we experiment with 44 VM images from our DAS4 cloud deployment registered by various teams. These images include various Linux distributions and software stacks. We start 16 VMs at the same time using randomly selected VM images and report the consumed memory over time in Figure 11. The results are similar to our synthetic test; VUsion achieves a similar fusion rate compared to KSM. VUsion with THP enhancements conserves huge pages that are in the working set while reducing fusion rate by 61%. These results show that VUsion provides its users with security with a trade-off between page fusion and performance.

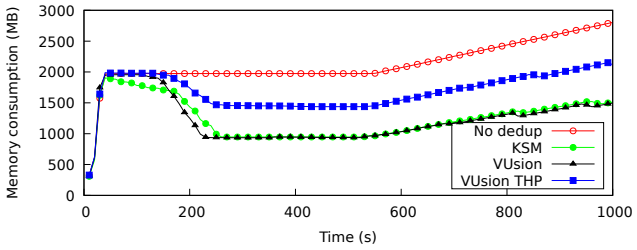

Figure 12: Memory consumption of the Apache benchmark.

Apache. Figure 12 shows the memory consumption during the Apache benchmark which we reported on earlier. We start four VMs together and start the benchmark on one of them after 360 seconds. Again VUsion achieves a similar fusion rate compared to KSM while similarly degrading fusion rate when conserving huge pages to improve performance. We also notice memory consumption increasing during the benchmark period for all cases. This is due to Apache's self-balancing strategy, which gradually expands the number of worker processes to serve many parallel requests to improve its throughput.

10 RELATED WORK

10.1 Attacks

Page fusion has been previously used in various attacks: as a prelude to a `FLUSH+RELOAD` attack [44], as a side channel to fingerprint software or build covert channels [25, 34, 39, 43], and as a way to brute-force ASLR [11, 13] or passwords [13]. Finally, the Flip Feng Shui attack [37] uses page fusion for physical memory massaging to compromise cryptographic keys of a victim VM.

In all cases, the traditional mitigation is to disable page fusion, wasting memory. An alternative is to disable active page fusion and only fuse swapped pages within a compressed in-memory cache. This is the approach taken by the current *Windows Memory Combining* implementation (previously the name of the active page fusion system on Windows, now disabled). This design, however, misses substantial fusion opportunities compared to active page fusion. In contrast, VUsion preserves page fusion benefits, mitigates all existing attacks, and even protects against a number of new attack vectors that we presented in this paper.

10.2 Defenses

The only existing defense against information leakage via page fusion is HexPADS [36]. HexPADS is an anomaly detection system that uses performance counters to detect suspicious behavior. Given the anomaly detection nature of HexPADS, it is prone to false positives and false negatives, providing attackers with the opportunity to tune their attacks and easily bypass HexPADS. Furthermore, HexPADS does not protect against physical memory massaging. In comparison, VUsion secures page fusion by design, improving performance as a by-product, and does not have any of the aforementioned weaknesses.

The **S⊕F** design principle relies on the copy-on-access technique, which has also been previously used in different applications such as post-copy live migration [22] and defending against cache attacks [45]. In contrast, VUsion uses copy-on-access as a building block for securing page fusion and combines it with protection against `prefetch`-based and other attacks.

Timing attacks against page fusion and side-channel attacks in general can be (partially) mitigated by reducing the timer accuracy. At the software level, major browsers such as Chrome, Firefox, and Microsoft Edge have reduced the accuracy of their timers to prevent side-channel attacks from JavaScript. Kohlbrenner and Shacham [28] propose introducing noise in the timer and the event loop of JavaScript to hinder timing measurement of system events. At the hardware level, TimeWarp [31] reduces the fidelity of timers and performance counters to make it difficult for attackers to distinguish between different microarchitectural events. Unfortunately, degrading the timer has performance implications and as shown by Gras et al. [19] can easily be bypassed.

11 CONCLUSION

Page fusion reduces memory pressure in modern cloud and consumer platforms, but existing approaches have been plagued by security weaknesses that lead to information disclosure and control over physical memory. This paper shows that these weaknesses are not fundamental and proposes a secure page fusion system with marginal degradation of fusion rates. The new design stops known and new attacks against page fusion, while also addressing inefficiencies. Our solution provides evidence that it is possible to support both secure and efficient page fusion in real-world settings.

ACKNOWLEDGEMENTS

We would like to thank our shepherd Jay Lorch and the anonymous SOSP reviewers for their extensive and helpful feedback. The IT for research group at the Vrije Universiteit Amsterdam provided us with one of the servers used in VUsion's evaluation. This work was supported by the European Commission through project H2020 ICT-32-2014 SHARCS under Grant Agreement No. 644571 and by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI Dowsing.

REFERENCES

[1] 2015. Idle Page Tracking. (2015). Retrieved 25.8.2017 from https://www.kernel.org/doc/Documentation/vm/idle_page_tracking.txt

[2] 2016. (March 2016). Retrieved 25.8.2017 from http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3272

[3] 2016. STREAM: Sustainable Memory Bandwidth in High Performance Computers. (2016). Retrieved 25.8.2017 from https://www.cs.virginia.edu/stream/

[4] 2017. memtier benchmark: A High-Throughput Benchmarking Tool for Redis and Memcached. (2017). Retrieved 25.8.2017 from https://github.com/RedisLabs/memtier_benchmark

[5] 2017. WRK - a HTTP Benchmarking Tool. (2017). Retrieved 25.8.2017 from https://github.com/wg/wrk

[6] Advanced Micro Device. 2013. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations *(SEC'16)*.

[8] Andrea Arcangeli. 2010. Transparent Hugepage Support. *KVM Forum* (2010).

[9] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing Memory Density by Using KSM *(OLS'09)*.

[10] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. 2012. An Empirical Study of Memory Sharing in Virtual Machines *(USENIX ATC'12)*.

[11] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. 2015. CAIN: Silently Breaking ASLR in the Cloud *(WOOT'15)*.

[12] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems *(ASPLOS XIII)*.

[13] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector *(SP'16)*.

[14] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, , and Anna Shubina. 2011. Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation *(;login:)*.

[15] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F. Haratsch. 2017. Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques *(HPCA'17)*.

[16] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. 2011. An Empirical Study on Memory Sharing of Virtual Machines for Server Consolidation *(ISPA'11)*.

[17] Google. 2017. Android Low RAM Configuration. (2017). Retrieved 25.8.2017 from https://goo.gl/Rz4B6I

[18] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: Bare-metal Performance for I/O Virtualization *(ASPLOS XVII)*.

[19] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU *(NDSS'17)*.

[20] Daniel Gruss, Cementine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR *(CCS'16)*.

[21] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. 2017. SmartMD: A High Performance Deduplication Engine with Mixed Pages *(ATC'17)*.

[22] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy Live Migration of Virtual Machines *(OSR'09)*.

[23] Intel. 2017. Intel Clear Containers: Building a Virtualization Continuum. (2017). White paper.

[24] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual*.

[25] Gorka Irazoqui, Mehmet Sinan IncI, Thomas Eisenbarth, and Berk Sunar. 2015. Know Thy Neighbor: Crypto Library Detection in Cloud *(PETS'15)*.

[26] Samira Khan, Donghyuk Lee, and Onur Mutlu. 2016. PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM *(DSN'16)*.

[27] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors *(ISCA'14)*.

[28] David Kohlbrenner and Hovav Shacham. 2016. Trusted Browsers for Uncertain Times *(SEC'16)*.

[29] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens *(OSDI'16)*.

[30] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical *(SP'15)*.

[31] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks *(ISCA'12)*.

[32] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications *(CCS'15)*.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES *(CT-RSA'06)*.

[34] R. Owens and Weichao Wang. 2011. Non-Interactive OS Fingerprinting Through Memory De-Duplication Technique in Virtual Machines *(IPCCC'11)*.

[35] PaX Team. 2003. Address Space Layout Randomization. Phrack, March 2003.

[36] Mathias Payer. 2016. HexPADS: A Platform to Detect "Stealth" Attacks *(ESSoS'16)*.

[37] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack *(SEC'16)*.

[38] Mark Seaborn. 2015. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges *(BHUSA'15)*.

[39] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Memory Deduplication As a Threat to the Guest OS *(EUROSEC'11)*.

[40] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms *(CCS'16)*.

[41] VMWare. 2015. Disallowing inter-Virtual Machine Transparent Page Sharing. (2015). Retrieved 25.8.2017 from https://goo.gl/uH0zNP

[42] Mark Wagner. 2011. KVM Performance Improvements and Optimizations. *KVM Forum* (2011).

[43] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2012. A Covert Channel Construction in a Virtualized Environment *(CCS'12)*.

[44] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack *(SEC'14)*.

[45] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A Software Approach to Defeating Side Channels in Last-Level Caches *(CCS'16)*.