A Low-cost Memory Remapping Scheme for Address Bus Protection

Lan Gao Jun Yang Marek Chrobak Youtao Zhang* San Nguyen Hsien-Hsin S. Lee†

Computer Science and Engineering Department
University of California, Riverside
Riverside, CA 92521
{lgao,junyang,marek,snguyen}@cs.ucr.edu

*Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260
zhangyt@cs.pitt.edu

†School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
leehs@gatech.edu

ABSTRACT

The address sequence on the processor-memory bus can reveal abundant information about the control flow of a program. This can lead to critical information leakage such as encryption keys or proprietary algorithms. Addresses can be observed by attaching a hardware device on the bus that passively monitors the bus transaction. Such side-channel attacks should be given rising attention especially in a distributed computing environment, where remote servers running sensitive programs are not within the physical control of the client.

Two previously proposed hardware techniques tackled this problem through randomizing address patterns on the bus. One proposal permutes a set of contiguous memory blocks under certain conditions, while the other approach randomly swaps two blocks when necessary. In this paper, we present an anatomy of these attempts and show that they impose great pressure on both the memory and the disk. This leaves them less scalable in high-performance systems where the bandwidth of the bus and memory are critical resources. We propose a lightweight solution to alleviating the pressure without compromising the security strength. The results show that our technique can reduce the memory traffic by a factor of 10 compared with the prior scheme, while keeping almost the same page fault rate as a baseline system with no security protection.

Categories and Subject Descriptors:
C.1 [Processor Architectures]: Miscellaneous; K.6 [Management of Computing and Information Systems]: Security and Protection

General Terms: Design, Performance, Security.

Keywords: Secure Processor, Address Bus Leakage Protection.

1. INTRODUCTION

In typical computer systems, host machines are protected from the exploits of malicious applications. For example, applications downloaded from unauthorized sources may contain malicious code such as viruses and worms. We address the other side of the problem where the confidentiality of an application should be protected from being revealed by the host machine itself. This is an often overlooked but realistic problem in prevalent computing models such as distributed computing.

In distributed computing, for instance, application tasks are dispatched to geographically distributed machines that are often contributed by volunteers. Even though distributed security protocols authenticate and authorize resources and users, there is no protection mechanism once a job starts executing on a remote node. It is difficult to recognize if the execution was tampered with, or if any secret it carries was compromised locally. A host machine, having full access to all the local resources, can launch a background program to analyze the execution of an active job and extract confidential information such as the encryption key [21]. Also, the memory contents could be altered either through software breaches or by privileged users so that the computation time of a job is lengthened [12]. This is especially harmful in a commercial environment where resources and services are charged by hour [2].

There have been a number of proposals that address the attacks from a host machine to its guest programs. The attacks can originate from a privileged user [12, 18, 27], a normal user [21], or even physical accesses [12, 18, 27, 31, 33]. A common countermeasure is to encrypt memory contents to prevent secrecy violation, and to authenticate the memory at runtime to prevent memory corruption induced misbehavior during execution. Both can be provided by a secure processor architecture such as XOM [18] or AEGIS [27].

In addition, the dynamic execution *address* sequence can be observed locally and analyzed to expose program's critical information such as the private key in the RSA algorithm [15]. Losing a key allows an attacker to impersonate a trusted user or to delegate a victim's access right to other malicious users. Exposing dynamic address sequence can also reveal the program's control flow information to enable a commercial software developer to reconstruct a core algorithm from a competitor [13, 14, 32, 33]. Hardware tapping devices for extracting addresses from the system bus, or even injecting traffic into the bus are readily available. For example, an FPGA-based programmable device can be attached to an SMP bus to perform on-line cache emulations taking real-time bus traffic from real workloads [20]. There are commercially available mod-chips that can be soldered onto the bus on the motherboard in a Sony Playstation or Microsoft Xbox to misguide the console to play pirated games [4].

Several techniques have been designed to combat the address information leakage: Goldreich *et al.* proposed three program transformation methods [13, 14] to construct a data-oblivious memory access pattern. Unfortunately, these software-only methods suffer from either great performance penalty or memory explosion. Recently, two hardware assisted schemes have been developed to trim down the overhead. The *HIDE* scheme [33] permutes memory subspace at certain intervals by reading them on-chip and writing them back after permutation. The *Shuffle* scheme [32] randomly shuffles the memory content whenever a block is read on-chip so that it will be written to a randomly different location later on. Both schemes

PACT'06, September 16–20, 2006, Seattle, Washington, USA.

try to randomize the address sequence appeared on the bus to hide easily recognized memory access patterns such as loops.

We have observed that the HIDE scheme increases memory access by a factor of 12 to 32 for page sizes between 4KB and 64KB, due to sequential reads/writes and redundant permutations. On the other hand, the Shuffle scheme could induce a large number of page faults in high-performance systems with memory paging. Designs that significantly increase the memory or disk demand do not scale well in future machines incorporating multi-threaded or multi-core processors in which memory and disk bandwidth are both the first-order constraints. It is therefore compelling to take those constraints into considerations while designing a secure architecture.

In this paper, we address the two main problems of the aforementioned security methods: memory access increase and page fault increase. These problems are mainly due to the following reasons: (1) excessive memory reads and writes on every permutation, (2) wasteful permutations, and (3) non-restricted block relocation. We propose a lightweight on-chip address permutation that effectively addresses all the three problems and achieves the lowest memory demands and page faults occurrence. Our main idea is to permute only *on-chip* cached blocks, to avoid the memory sweeps in HIDE, and launch a permutation for only those addresses that have not been remapped since they are read on-chip. Our scheme incurs only a $0.88\times$ increase in memory accesses and a close-to-base page fault rate, without compromising the security strength.

The remainder of this paper is organized as follows. Section 2 describes the motivation of this work. Section 3 gives an overview of the two schemes that we improve, and the in-depth analysis of them. Section 4 introduces our proposed scheme, followed by its architecture design issues in Section 5. Section 6 presents our experimental results. Section 7 discusses the related work and Section 8 concludes this paper.

2. MOTIVATION

The address sequence recorded from the CPU-memory address bus may disclose the control flow information of a program under execution. This is true even in secure processors such as XOM [18] or AEGIS [27] in which the memory contents are all encrypted (the CPU-memory data bus transfers only ciphertext) but the addresses are left in plaintext. Such plaintext address sequence can lead to critical information leakage and is the main problem we tackle in this paper. First of all, the sequence can be split into code and data sequences with explicit reads and writes. This is because code and data are in separate memory regions. Code regions are read-only and typically accessed sequentially at the start-up of the program execution. The obtained code sequence shows the control transfers only at a coarse granularity since most of the instruction reads are serviced by the on-chip caches. This is not difficult to circumvent as the on-chip caching can be disabled through setting proper control register bits [6], or minimized by running a concurrent thread that competes the shared cache with the victim thread [21, 22].

The sequence obtained hereafter can be used to derive the control flow graph (CFG)[1]. A sequence of "`abc abc abc`" clearly shows a loop with "`a`" being possibly the loop starting and "`c`" being the loop ending instruction. Whereas a sequence of "`abcd abd abcd abd`" indicates a conditional branch after "`b`" inside a loop containing "`a, b, c`" and "`d`". Most software nowadays have a high percentage of *reuse code* [19] — those that reuse prebuilt libraries from the public domains or a third party. In other words, the reused portions of a software can be identified once their CFGs are constructed. This could ease the identification of the non-reused part of the code, leading to a potential intellectual property theft.

More seriously, the timing attacks to Diffie-Hellman, RSA and other security algorithms exploit the actual directions of a branch instruction inside a simple loop to reveal the private key bit-by-bit [15]. The loop iterates for a number of times equivalent to the bit width of the private key. Once the address sequence of the loop is exposed, the private key can be recovered.

Finally, addresses to data region can also expose control flow since some data are only accessed by one path of the program. Therefore, protecting the data addresses should be carried with the code addresses. Next, we will briefly review two existing algorithms on address sequence protection, followed by a performance evaluation for each of them.

3. OVERVIEW OF ADDRESS SEQUENCE PROTECTION

The basic idea of address sequence protection is to break its correlation with the CFG of the program. For example, the sequence of "`a a+4 a+8`" does not correspond to sequential instructions in the code. Also a sequence of "`abc abc`" does not indicate a loop structure. Prior schemes (HIDE and Shuffle) incorporated randomization of the memory contents from time to time. Next, we will explain the processor-memory and hardware support model in those schemes.

3.1 Model and premises

We choose to use the secure processor models proposed recently [16, 18, 27] as our foundation. Both HIDE and the Shuffle scheme fall into this framework. In such a secure model, the processor is physically secure such that once the data and code are brought onto the chip, they cannot be tampered with. If any data are sent off-chip to the memory, they are always encrypted to ensure their confidentiality. Therefore, attacks can only be mounted to components external to the processor, such as the buses and the memory. Since the crypto operations are always performed between the processor and the memory, we assume a fast crypto mechanism is available on-chip to accelerate the process [26, 28, 30].

3.2 Chunk level permutation

The basic idea of the HIDE technique [33] is to permute the address space at suitable intervals to break the correlation between repeated memory addresses. Ideally, before an address recurs, a permutation is initiated to map it to a different location. In reality, it is not practical to remember all the addresses between permutations, nor is it efficient to search the history list to detect the recurrence. Hence, HIDE proposes to augment the L2 cache replacement policy with additional locking control — *a block that is newly read from the memory or becomes dirty since its last permutation cannot be replaced after a permutation is performed*. This is because both cases could cause their addresses to recur on the bus in future reads if they were allowed to be replaced earlier. Hence, those entries are temporarily *locked* in the cache. Only blocks that are not locked could be freely replaced. Each permutation permutes a *chunk* (one to several pages) of continuous blocks, mapping each block to a different address within the chunk, so that recurring addresses on the bus may not indicate the same block. Fig. 1 shows how this scheme works.

We assume a two way, two set associative cache, and a two-page memory with eight blocks in each page. Each chunk consists of one page, with even-addressed blocks mapped to set 0, and odd-addressed blocks mapped to set 1. Initially the cache is empty and

[1]The control flow graph is a directed graph that shows the transfer among instruction basic blocks.

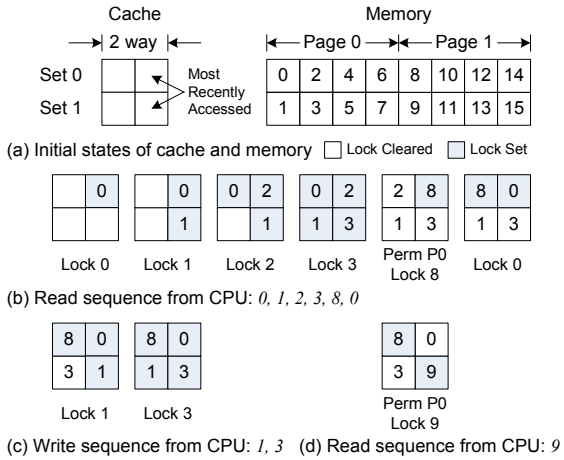

Figure 1: Example of the HIDE scheme

no block is locked. When the CPU reads blocks 0,1,2,3,8,0, the actual sequence on the bus is 0,1,2,3, permutation traffic, 8,$\pi_1(0)$. $\pi_1(0)$ is the permuted address of block 0. Permutation traffic consists of sequential reads and writes of all blocks in page 0, and this permutation is triggered when block 8 replaces the locked block 0. After permuting page 0, all cache blocks of page 0 are cleared for their locks. Any block brought on-chip is locked afterwards. Next when the CPU writes block 1 and 3, both are locked again in the cache. The last read at block 9 causes a writeback of block 1. Since it is locked, a second permutation of page 0 is initiated, during which the locks on block 0 and 3 are cleared.

Through permuting the entire chunk, control transfers within the chunk are invisible on the bus, thus reducing the likelihood of information leakage. However, the strengthened security comes at a high cost of memory accesses as all the blocks in the chunk are swept through — read on-chip and then written off-chip — on every permutation. In Fig. 2, we categorize the HIDE memory accesses into true memory requests ("true") and permutation induced accesses ("perm"). All the accesses are normalized to the first bar for 4K-byte chunk size. It is surprising to see that the percentage of true and useful memory accesses account for only 7.5% and 3% of the total for 4KB and 64KB chunk sizes respectively. In other words, the HIDE scheme increases the memory demands by a factor of 12 (4K) or 32 (64K). Using larger chunk sizes has its own advantages: 1) it further reduces the control flow exposed on the bus; and 2) it reduces the frequency of permutations since the chances of clearing the locks are higher. However, the dramatic increase in memory traffic using large chunks creates a serious bottleneck in the system, which overrides its benefit in security. We break down the extra traffic into two sources:

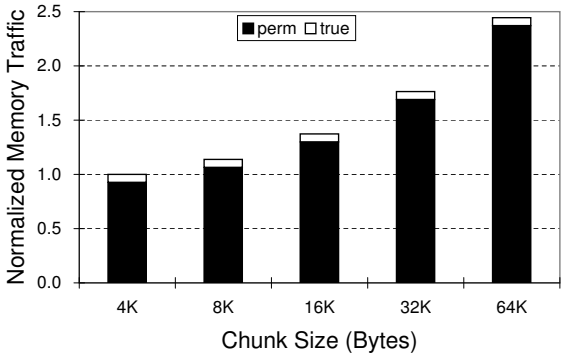

Figure 2: Memory traffic breakdown for different chunk sizes.

Excessive memory accesses on permutation. Fig.2 shows that most memory accesses in HIDE are useless to the program. They are redundant accesses simply to *hide* the traces of the useful memory addresses. Here, "useful blocks" refer to those that have been accessed by the program. We observed that this quantity is fairly low between two permutations, an indication of large redundancy in HIDE.

In Fig. 1(b), only 4 blocks in Page 0 are accessed when the permutation is invoked. During the permutation, they are read on-chip again with the other half that is never touched. Then the whole page is written back, increasing the traffic by fourfold. We studied the excessive accesses in a real system where the page size is 4KB and each page consists of 128 32-byte blocks. Fig. 3 is a histogram of the pages with their numbers of accessed blocks from zero to 128 upon a permutation. That is, when a new permutation of a page is initiated, if it accessed i blocks since the last permutation, we increment the i^{th} bar in the histogram. The graph is averaged over 11 SPEC2K benchmark programs for a 1.1B-instruction run. We can see that only 17% of pages are fully accessed between two permutations. For those pages, efficiency is good since there is no reading of useless memory blocks. However, 65% of pages triggering a permutation are accessed less than a quarter of the blocks.

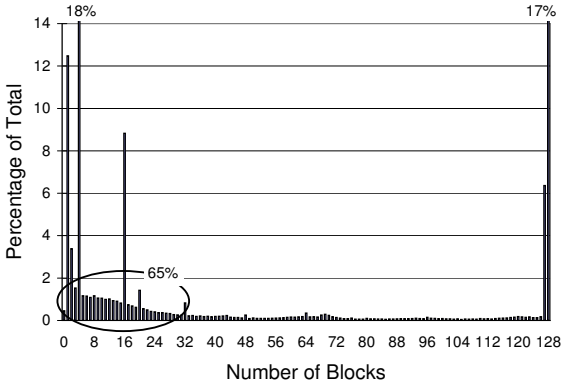

Figure 3: Histogram of pages with 0-128 blocks accessed between permutations.

In fact, if we take a global view of the block usage in a page during the entire simulation, the percentage of pages that are fully used is quite high (Fig. 4). More than half (57%) of the total pages are fully used, much more than that in Fig. 3 because some blocks inside those pages are recalled before the rest are touched. Each recall triggers a scan of the entire page and unaccessed portion of the page can be permuted multiple times due to multiple such recalls. Using the example in Fig. 1, if block 4-7 are accessed in the future, then page 0 would be fully used. However, before this could happen it is already permuted twice.

Redundant permutations. Due to the difficulty of tracking if an address is a repeated access, the permutation is triggered preventively. That is, before a dirty block is written back to the memory, a permutation of the hosting page is triggered, anticipating that the block will be read again in the future. This is why the second permutation in Fig. 1(d) is performed. Such permutations induced by write locks trigger multiple permutations while only one is necessary. The example below illustrates such a scenario.

Let blocks `A`, `B`, `C` belong to page `P`. They are initially read on-chip and then locked. When `B` has to be replaced, a permutation of `P` is triggered since `B` holds a read lock. Afterward, `A`, `B`, `C` are all mapped to different addresses, `B` is written back to a new address while the others are still on-chip. The permutation also clears off their read locks. Suppose a later write locks `A` again. When `A`

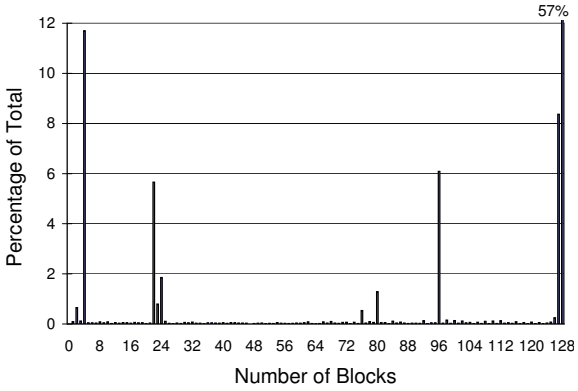

Figure 4: Histogram of pages with 0-128 blocks accessed during the entire simulation.

is replaced at some point, a second permutation of `P` is initiated, which we can see is unnecessary because `A` has been mapped to a new location in the first permutation. Similarly, `B` and `C` are both re-mapped in the second permutation, which are also redundant. Note that between permutations, no accesses to page `P` is necessary, indicating that a page could be permuted repeatedly without even a single access. This also explains why the 0^{th} bar in Fig. 3 is positive (0.46%).

3.3 Memory shuffle

A simpler scheme was proposed in [32] for embedded systems, since they are more vulnerable to the address attacks. The approach is to relocate a block if it is brought on-chip so that it will be written to and read from a different memory address in the future. The new address is chosen randomly from the program's memory space. To implement it efficiently, a small portion of memory blocks is stored in an on-chip *shuffle buffer*. A random block is selected from this buffer to swap with the requested block read on-chip. Effectively, a memory read is always followed by a memory write, where the read is demanded by the processor while the write is a swap randomly picked from the buffer. The block being read still resides in the buffer (as well as the caches) so that the shuffle buffer always has a fixed number of memory blocks for swapping.

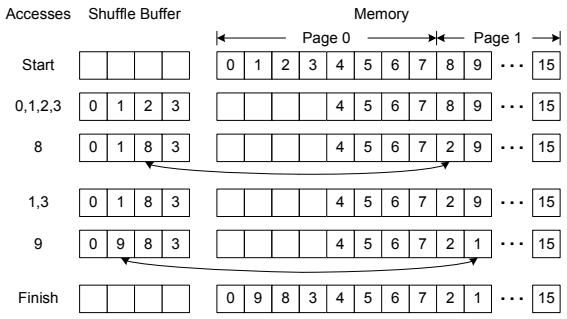

Figure 5: Example of the Shuffle scheme (adapted from [31]).

Fig. 5 shows how this scheme works. We assume the on-chip shuffle buffer only stores 4 blocks. The first 4 accesses fill up the buffer. Starting from the fifth access, a random swapping of blocks between memory and the shuffle buffer is performed. For example, the swapping of block 8 and block 2. If the access hits in the buffer (e.g. block 1 and 3), no swapping is necessary. Finally, all the blocks in the shuffle buffer are written back to the very first four empty slots in memory.

As shown, the memory traffic in the Shuffle scheme – as we will term it in this paper – is only two times of an unprotected system, because a read is always followed by a write. This overhead is significantly lower than the HIDE scheme. However, the blocks are swapped in the entire program memory space, e.g., the swapping of block 8 in page 1 and block 2 in page 0. So this scheme needs to remember the block mapping, i.e., which address is a block mapped to, using the *full-width* block address, whereas in the HIDE scheme, only the offsets in the chunk need to be remembered as the blocks are re-mapped only within a chunk. The storage overhead of the Shuffle scheme is therefore greater than in the HIDE scheme (10% vs. 3.5% as reported previously).

More importantly, in order to make memory accesses look "random" in its memory space, the Shuffle scheme shuffles blocks in the entire program memory. As a result, it destroys the program's locality, and blocks in hot pages may be mapped to cold pages. Eventually, in the long run, all pages are roughly equally warm. This may increase the page faults if not all pages are in the memory. If the Resident Set Size[2] (RSS) is 100%, i.e., all pages of the program reside in memory, such a randomization will not incur extra page faults, and all page faults are cold page faults. However, if RSS is 50%, i.e., at any time of the program execution, only half of the pages reside in memory, any access to the pages not in memory will incur a page fault. In this case, random swapping of blocks will increase the page faults.

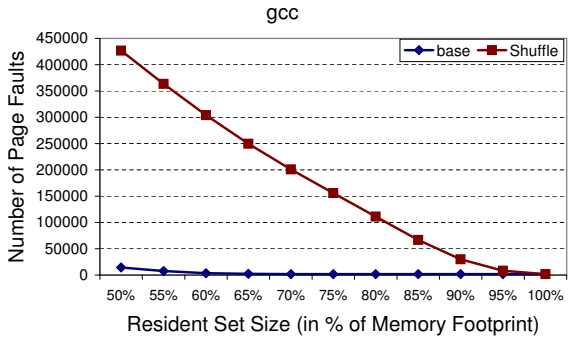

Figure 6: Page fault curve for gcc.

Fig. 6 plots the trend of page faults versus the RSS for `gcc`. As we can see, the number of page faults increases by 257× when RSS drops from 100% to 50% of the total memory pages. However, the curve is almost flat for the base case. This shows that preserving locality is critical to program performance. Similar patterns can be observed in other programs as well. Here we assumed a 4KB page size and perfect LRU memory page replacement policy.

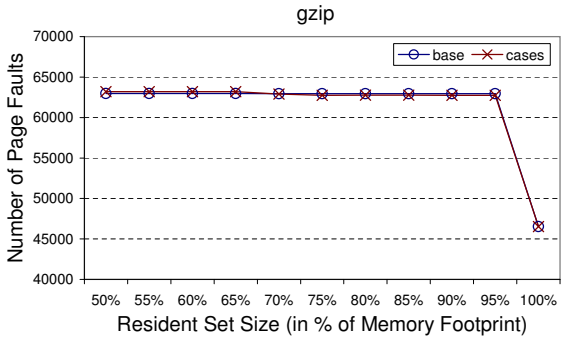

Figure 7: Page fault curve for gzip.

However, if the shuffling happens to keep or improve the program's locality, e.g., page 0 and page 1 in Fig. 5 are always accessed together, the number of page faults could remain the same

[2]The number of virtual pages resident in RAM

or even decrease. As we can see from `gzip` in Fig. 7, the number of page faults in the Shuffle scheme is the same as that in the base case. This is because the shuffle buffer of `gzip` stores mainly those blocks within the current working set because of its special memory access pattern.

It is also worth noting that the Shuffle scheme can introduce some extra memory writes as well. This is because the shuffle buffer effectively holds the recently fetched memory blocks. On a new read from an address `Addr` for example, a block `B` is randomly picked from the shuffle buffer and written back to `Addr`. However, `B` itself might be too recent to be updated by the processor. Later when `B` is updated, it needs to be written back again, which suggests that the previous write was unnecessary. Our measurements show that such redundant writes account for $\sim$5% of the total memory traffic on average.

3.4 Summary

The HIDE scheme permutes blocks only within a chunk, practically one page, and hence has little impact on the memory paging. However, it incurs extremely high overhead of memory accesses, which makes it hard to fit into contemporary processors where memory is still a performance bottleneck and one of most power-hungry components. The Shuffle scheme, on the other hand, introduces mild extra memory demands. Yet its demand on the disk access limits itself to embedded systems only where most applications have small memory footprints. Accessing memory is usually several orders of magnitude faster than accessing the most technologically advanced hard-drives [5]. Hence, in a demand paging system, it is important to keep the page fault rate low. We will next introduce our lightweight design in which both the memory accesses and the page faults are low.

4. PROPOSED INEXPENSIVE ADDRESS PERMUTATION

Our scheme aims at achieving three goals. The first goal is to avoid wasteful memory reads and writes in each permutation. The second goal is to eliminate the wasteful permutations so as to reduce the total number of permutations. The third goal is to preserve locality and keep the page fault rate low. The approach we take is to permute selective blocks instead of a whole page, aided by a good decision of when a permutation should happen.

4.1 The permutation mechanism

The idea We propose to perform permutations only on those *on-chip* blocks. This is because their addresses have occurred once on the bus, and could recur if they are evicted out and read in again in the future. It is therefore necessary to relocate them (i.e., permute them) before they are replaced. However, not every replacement should be preceded by a permutation because if the victim block has been permuted and mapped to a different address, it is safe to release it off-chip. It is only those blocks that are recently read on-chip (RR blocks) but have not participated in any permutations that should be permuted. More clearly, a block `B` of a page `P` is an RR block if `B` is fetched from memory after `P`'s last permutation. Hence, a permutation is started only if an RR block is to be replaced. Afterward, all the RR blocks that are involved in the permutation are turned into normal blocks which can be safely released off-chip because the permutation has masked the earlier traces of accesses and their addresses have been mapped to other random places.

Since during a permutation only on-chip blocks are involved, the need of reading chunks of blocks from memory and writing them back as in HIDE is eliminated. This is the main reason why our scheme can save most memory traffic. Moreover, a great advantage of doing so is that we do not need to update the memory right away. Since the permuted blocks are on-chip, we simply perform the permutation of their *addresses* and remember the mapping. When they need to be replaced some time in the future, they refer to the mapping to obtain their new addresses to which they should be written. Note that we write every replaced block into the memory because it is relocated to a different address even if it is not dirty.

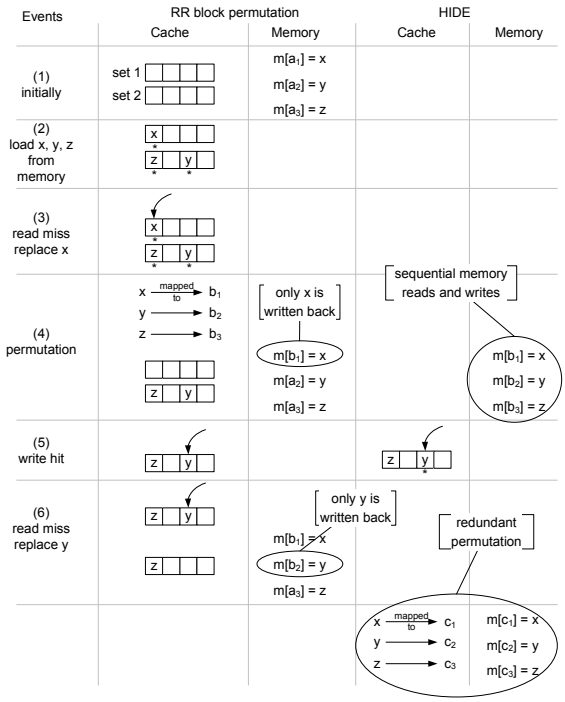

Figure 8: Comparison of on-chip block permutation and the HIDE scheme. x, y, z are from the same memory page. x is mapped to cache set `set 1`. y and z are mapped to `set 2`. The block with a $*$ means an RR block in our scheme, and a locked block in HIDE. '`m[a]` = x' means x is stored at memory address `a`.

An example We now walk through an example in Fig. 8 to show our permutation mechanism. Assume initially, x, y, z are from the same page in memory with address a_1, a_2, and a_3 respectively. When x, y and z are loaded on-chip, they are marked as RR blocks. If later a read miss happens in x's set (event 3) and x is to be replaced, a permutation must be performed because x is an RR block. The permutation generates a random mapping (event 4) for all the on-chip blocks of the same page as x (and some other blocks which will be explained later) and then clears off the marks for those RR blocks being permuted. The mapping is kept on-chip. After that, x is written back to the memory at new address b_1. Note that x may not be dirty but is still copied back. y and z are not written back until they are evicted from cache. A write hit on y (event 5) does not set the RR-bit again. Finally when y is replaced (event 6), it is written back to the new address b_2 assigned during the latest permutation.

We also illustrate the actions taken by the HIDE scheme in the same figure as a comparison to our scheme. This example is along the same line as Fig. 9. The main differences are in the type and number of blocks involved in each permutation and the time a permutation is triggered. In event (4), the HIDE performs a sequential reads and writes to the entire page so that all the blocks in the memory are physically permuted. However, this is wasteful because y

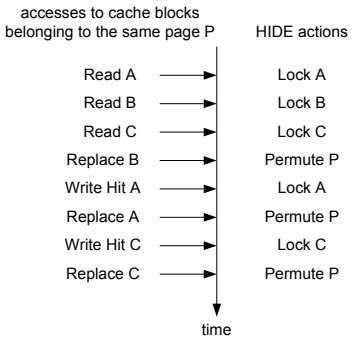

Figure 9: An example showing redundant permutations in HIDE.

and z are still on-chip to serve future requests from the CPU. Moreover, the write hit to y in event (5) locks y again, which triggers a second permutation upon a replacement of y in event (6). The entire block is again sequentially read and written to the memory although a remapping of all three blocks at this time is indeed unnecessary.

Let us mention a few subtleties here to make this example more complete. First, the initial addresses a_is are not the true physical addresses of x, y and z. They have been randomized in previous permutations or an initial permutation of all memory pages as the program started. This is a requirement in HIDE and Shuffle as well. Therefore, one cannot infer, for example "a_1 is mapped to b_1 as the permutation is triggered by their cache conflict", because what caused the conflict are not really a_1 and b_1 on-chip. They have been remapped randomly. Second, permutation on event (4) of HIDE may not happen at the same time as in our scheme because of the LRU replacement was modified in HIDE to delay such permutation. However, it can be delayed but cannot be avoided in the future. Third, the mapping created by the two schemes may not be identical because they use different block sets. We let them equal in the example for ease of illustration and comparison only.

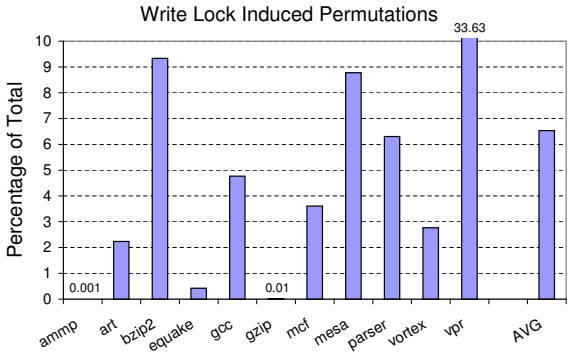

Figure 10: The percentage of saved permutations due to write locks.

In brief, our scheme permutes only on-chip blocks, and a permutation takes place only when an RR block is replaced. We save a significant amount of memory traffic compared to HIDE. The overhead we pay here is only the writebacks of the non-dirty blocks. Further, we eliminate those permutations due to the write locks in the HIDE scheme. Fig. 10 shows the percentage of such permutations our scheme has saved.

4.2 Permuting sufficient number of blocks

During each permutation, there should be enough number of blocks involved to ensure the strength of the randomization. For example, in our 4KB page setting, the HIDE permutes 128 blocks every time, generating 128! possible mappings or a probability of 1/128 to guess one mapped address correctly. If there are only 32 on-chip blocks in a page and if we only permute among those blocks, then the probability is increased to 1/32, much higher than before and thus not preferred.

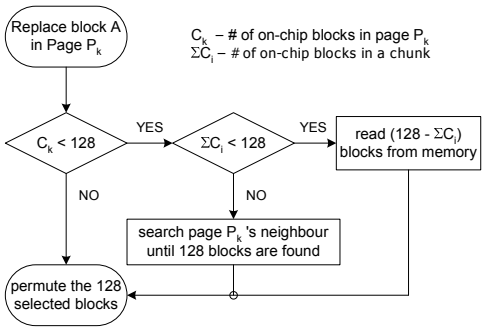

Figure 11: The procedure of searching for sufficient number of blocks to permute.

Therefore, we increase the available blocks by looking at the *chunk* level instead of page level. The HIDE scheme could not truly scale to chunk level due to its prohibitive increase in memory demand. Whereas we do not have this concern as we only look at the on-chip blocks, and thus our scope can be scaled up to include more pages. Our goal here is to permute the same number of blocks per permutation as the HIDE scheme. However, the number of blocks on-chip are dynamic for every page, we cannot be certain how many pages should be included in order to have 128 blocks in every permutation. Therefore, we develop a simple incrementally growing mechanism, as illustrated in Fig. 11 to solve this problem.

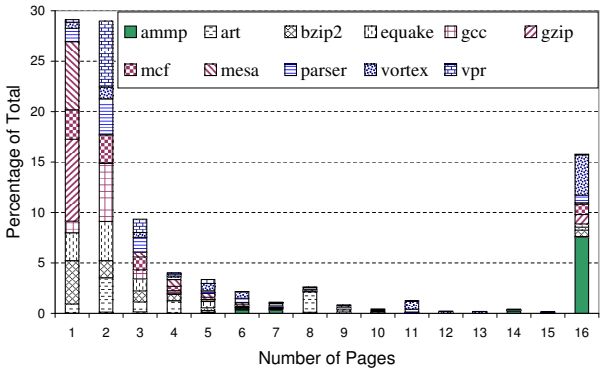

Figure 12: Average number of pages that supply sufficient on-chip blocks per permutation.

The procedure When a permutation is started by a replacement of an RR block `A` in page `P`, we first check if `P` has enough, e.g., 128, blocks on-chip. If so, we simply permute the 128 addresses of those blocks. Otherwise, we expand the search scope one page at a time in its neighborhood until we have enough blocks to permute. When there are sufficient blocks, giving priority to the RR blocks over normal blocks can reduce the total number of permutations because only an RR block triggers a permutation and it becomes a normal block after the permutation. However, our search space does not grow unbounded. It is limited by the chunk size, i.e., the chunk boundary where `P` falls within. If we still cannot find enough blocks in the entire chunk, to ensure the same level of security, we choose to read extra blocks from memory within the chunk as padding blocks to the permutation input. As we can see, the chunk size should be a reasonable number as too small the chunk size

leads to a HIDE-like scheme because we need to read many extra blocks, but too big the chunk size leads to a Shuffle-like scheme because we lose locality among the permuted blocks. Therefore, choosing an appropriate chunk size depends on the trade-off between the memory accesses and the incurred page faults.

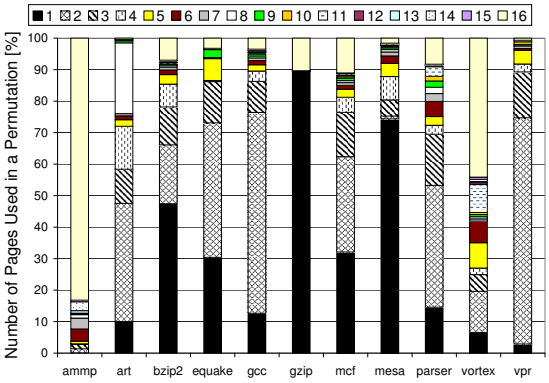

Figure 13: Number of pages involved in each permutation.

Typically, the collection of enough blocks can be satisfied in just a couple of pages. The extreme of searching in the entire chunk does not happen very often. Fig. 12 shows the average number of pages searched in order to find enough blocks on-chip. We set the chunk size to 16 pages, which will be explained later. The figure is averaged over all 11 SPEC2K benchmark programs. Each bar also shows the weight contributed by each program. In most cases (∼58%), up to two pages are enough to provide 128 blocks on-chip. However, searching through the entire chunk does happen more than 15%. This number is mainly contributed by program `ammp` and `vortex`, both having poor locality and low number of blocks per chunk in the cache. Note that when the search stops at 16 pages, padding might be needed. If so, they automatically fall into the 16th bar. An anatomy of the number of pages touched per permutation for each program is given in Fig. 13. As expected, one or two pages are sufficient for most programs, which implies that the searching algorithm can terminate quickly.

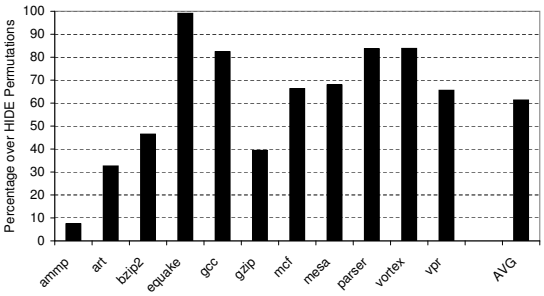

Figure 14: Total number of permutations in percentage of the HIDE permutation number.

The benefit Permuting across multiple pages on-chip helps greatly to reduce the number of permutations. For example, suppose a replacement of an RR block in page P_1 triggers a permutation, followed by a replacement of an RR block in P_2. In HIDE, both replacements cause permutations while in our scheme, P_2 could be permuted together with P_1 in the first permutation, which might clear off the RR block in P_2 and save the second permutation. Such an effect is the major contribution to the total permutation reductions shown in Fig. 14. On average, our scheme saves nearly 40% of the permutations in HIDE including those due to write locks shown earlier in Fig. 10.

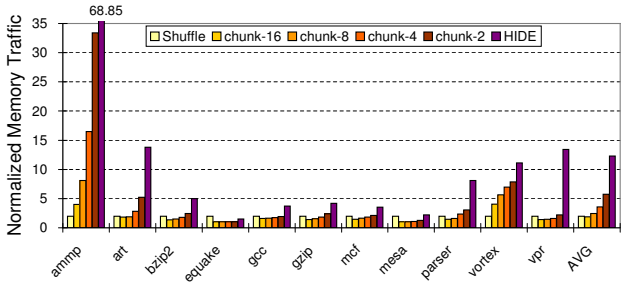

Figure 15: Memory traffic comparison among different schemes. Chunk size is varied from 2 to 16 pages.

Choosing the chunk size As mentioned earlier, a large chunk size may result in more page faults as the permutation spans over a wider address space. A small chunk size may generate more extra memory accesses since not all pages are fully cached on-chip. Hence it is important to find a good break-even point between the page faults and memory accesses. We compare the total number of memory accesses in Fig. 15 with varying chunk sizes in terms of number of pages. The results are normalized to that in a base system where no address protection is present. The Shuffle scheme doubles the memory traffic while HIDE leads to an increase by a factor of 12. As expected, the larger the chunk size, the less the traffic. A chunk of 16 pages only brings 88% more traffic on average, even better than Shuffle, while other sizes increase the traffic by a factor of 2.46, 3.59, 5.73, respectively. Our memory traffic is lower than the Shuffle scheme because the aforementioned reason that Shuffle introduces ∼5% extra writes while our scheme can avoid those. The major portion in the 88% extra traffic comes from the writebacks of non-dirty blocks. To show the page fault increases with larger chunk sizes, Figure 16 plots the curves with gradually decreasing memory RSS. The results are averaged over all the benchmarks. As we can see although larger chunk size generates a little more page faults (less than 2% increase from chunk-2 to chunk-16), they all have far fewer page faults than the Shuffle scheme. So overall speaking, a chunk size of 16 pages has the lowest memory traffic increase with a reasonable page fault increase.

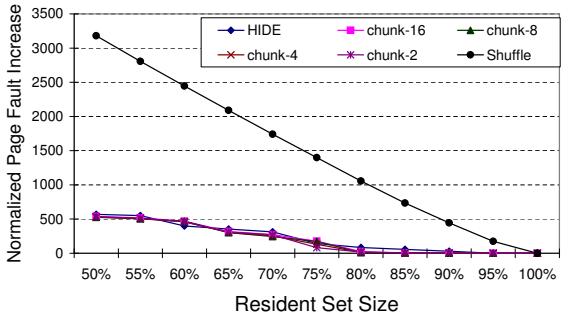

Figure 16: Page faults comparison among different schemes averaged over all benchmarks.

4.3 Security analysis

To find out the address recurrence from our randomized address sequence, an observer must be able to conjecture at least one address mapping correctly between two permutations. Our scheme has many uncertainties that help reduce the probability of a successful guess. For example, there is no clear indication of when a permutation really happened from off-chip observation since most permutations do not involve off-chip accesses and only a subset of off-chip writebacks would cause a permutation. Also, even if a permutation is identified, it is hard to know which set of on-chip

blocks participated in the permutation since they are decided internally. Furthermore, it is not clear what set of blocks are really on-chip soon after some initial execution as writebacks are remapped so that it is hard to know what blocks are returned to the memory.

Nevertheless, let us assume a case where an observer can make a correct guess with the highest probability. Other situations are significantly complex than this case and the probabilities will not be higher. Suppose a page P is entirely read on-chip. Later on a block is written back to an address A in P. Since all the blocks of P were read on-chip and were initially RR blocks, any replacement would invoke a permutation inside P only. Hence, it is some block in P that is mapped and written back to A, and the probability is $\frac{1}{128}$, assuming there are 128 blocks per page. If a second writeback to A′ in P occurs, a block being mapped to A′ has a probability of $(1 - \frac{1}{128}) \times \frac{1}{128}$. Similarly, a block being mapped to the n^{th} writeback has a probability of $(1 - \frac{1}{128})^{n-1} \times \frac{1}{128}$. This is a decreasing function with n. Hence, it is best for an observer to make a guess in the first writeback after a latest permutation. This analysis is along the same line as the Shuffle scheme. Both achieve the same probability of guessing one mapping correctly.

Similar to HIDE, what an observer can see on the bus is the chunk-level transition, instead of fine-grained page level or even block level transition that can reveal control flow easily. The HIDE technique has shown that a chunk size of 64KB, i.e., 16 4KB pages, can cover 95% of the program's control flow (some compiler assisted layout optimization may be necessary to achieve this number). However, the HIDE scheme cannot afford to operate on real 64KB chunk, while our scheme performs the best with this size.

By permuting mostly the on-chip blocks with occasionally off-chip padding blocks, our scheme makes full use of the on-chip cache. The more blocks stored on chip, the fewer addresses are transferred on the bus, and the more secure it is. If an attacker maliciously runs a simultaneously executed thread to inject well crafted cache accesses in order to push some cache blocks off-chip, the program under attack would create more memory accesses as fewer blocks are on-chip, leading to a more HIDE-like effect. Essentially, the address protection scheme plays against the cache-centric attack until the program runs at very low speed. Now it is probably good to raise an alarm indicating a possible attack.

5. IMPLEMENTATION ISSUES AND ARCHITECTURE DESIGN

5.1 Virtual Cache vs. Physical Cache

As with the HIDE address mapping scheme, our permutation should be performed on virtual addresses (VA) because physical address (PA) space is managed by the OS which would map a virtual page to different physical pages dynamically. Hence, with the permutation interface, a VA is always first mapped to another VA which is then translated to a PA by the TLB in MMU and finally sent to the memory. This process is carried beneath L2 (or L3 if there is one) since we only need to protect the addresses that occur on the processor-memory bus. Therefore, the implementation of the permutation is different for a virtual L2 than for a physical L2.

If the L2 is physically indexed, the MMU first translates the VA request sent from L1 (which is typically virtually indexed for performance advantage) into a PA and uses it to access the cache, as shown in solid line path in Fig. 17(a). If it is a cache hit, the data is returned to the CPU and L1 and no further action is needed. If a miss happens, however, the PA should be sent to memory to fetch the data. In a memory address protected system such as HIDE and ours, the original VA is remapped to a VA' and hence the current PA should not be used for memory access. Instead, we should first look up the new mapping for the original VA to find VA' in the Block Address Table (BAT) [32], and then translate the VA' to a PA'. Only after this, can we have the true physical address to access memory. As we can see, the TLB lookup happens *twice* on every such L2 miss. This procedure is illustrated in the dotted path in Fig. 17(a).

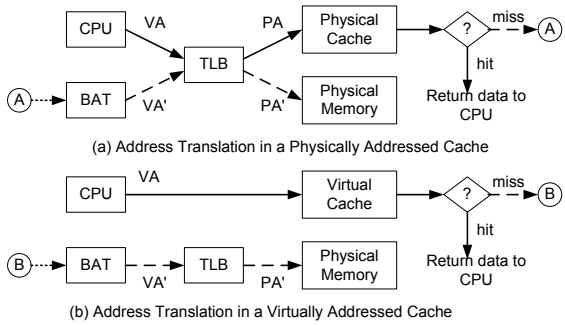

Figure 17: Double and single TLB lookups in a physically and virtually addressed L2.

The problem with the second TLB lookup is that VA' is a mapped address within a chunk of pages in both HIDE and our scheme. Hence, it may belong to a *different* page than VA, which might result in a miss on the TLB lookup because VA and VA' are translated to different physical pages. As we know, a TLB miss triggers a memory access for the proper page table entries, which penalizes the performance. Therefore, in the HIDE scheme, if the chunk size is larger than one, the scheme requires that the physical pages be contiguous if the virtual pages are so that only one TLB access is necessary for the entire chunk. Also, the physical pages belonging to the same chunk should be swapped in and out together. As we can see, this could be a high requirement to the OS.

While in a virtual L2, TLB lookups proceed the L2 misses. Hence, with the address permutation interface, the missed VA is first searched in the BAT for its mapping, and then translated through the TLB to get the true physical address. Only one TLB lookup is necessary, as shown in Fig. 17(b). Putting the TLB below a virtual L2 has the advantage that its coverage on page table entry is better than the TLB in a physical L2, since the former holds page entries for L2 misses [23]. Therefore, the TLB for a virtual L2 has lower miss rates than a physical L2. To see this, we measured the TLB misses in Fig. 18 for a virtual L2 ("VL2"), our address permutation on a virtual L2 ("VL2-perm"), and on a physical L2 ("PL2-perm"), normalized to the TLB misses in a baseline with a physical L2.

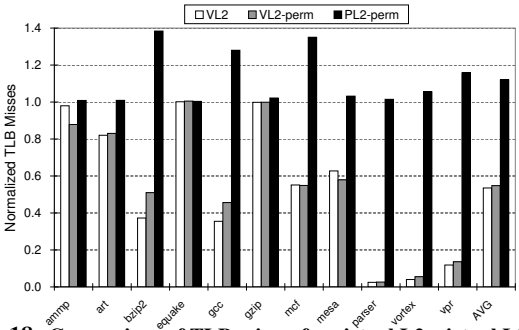

Figure 18: Comparison of TLB misses for virtual L2, virtual L2 with permutation (our scheme), and physical L2 with permutation. Results are averaged over 11 SPEC2k benchmarks.

The results show that using a virtual L2 instead of a physical L2 can reduce the TLB misses by half on average. When enhanced

with address permutation in a virtual L2, TLB misses increase only by 1% on average. However, if permutation is added to a physical L2, the TLB misses are increased by 12%. These data show that a virtual L2 cache is especially useful in our scheme. Using a virtual L2 has some issues such as the synonyms and exception handling. They have been addressed well in the literature with certain hardware assisted schemes [23, 24, 25].

5.2 The permutation architecture

To put everything together, we need the following hardware components to assist the address protection scheme. First of all, we need a permutation unit that can generate a random permutation for 128 blocks. A shuffle algorithm performing 128 swaps would suffice as long as a hardware true random number generator is present. Notice that in most cases, we only need to permute the addresses of the on-chip blocks. Occasionally we also need to read some padding blocks from memory to participate in the permutation. Thus, a buffer for temporarily storing them is necessary. Its size is 127 blocks as there must be at least one block on-chip in the chunk. Also, storing the 128 block offsets for permutation requires 128×7 bits in total.

Next, we need an information table for every chunk that remembers the block address mappings (in BAT) and the block status. All address mappings are stored in the chunk information table. Each chunk has an entry, consisting of two bitvectors and a virtual address BAT. One bitvector remembers if the block is on-chip, and another remembers if it is an RR-block. The BAT stores the virtual-to-virtual address mappings for each block, with each mapping used only once before being relocated again. The record size is small compared with the size of a chunk. If we assume the block size is 32 bytes, and each page is 4KB, then for a 64KB chunk, the storage overhead is only 5%, since there are only two status bits per block, and 11 bits for the mapped block offset within the chunk. Hence, there is approximately the same overhead as in HIDE with the same chunk size. When compared with the Shuffle scheme where there is a 10% storage overhead for the BAT, we have saved almost by half.

Last, we need an address generation unit that can quickly find 128 on-chip addresses to permute. This can be easily performed through wide shift registers and an adder. We can first AND two status bitvector of a page into the shift register. And then shift the register one bit at a time. The adder increments the block offset on each shift, and outputs the offset to the permutation unit if the bit is set, i.e., the block address should be used for permutation. Once a vector for a page is finished, the unit starts with the next page. It might take two iterations to do so as we may not have enough RR blocks in one round. In the second round, we shift only the on-chip status bitvectors. The starting point should be randomly selected to inject some randomness in selecting the on-chip cache blocks. Since the adder only operates on block offsets, it is only 7-bit wide. Hence, a conventional 32-bit adder can perform four adds in parallel, producing $4\times$ the throughput.

The architecture design of the hardware components and the datapath are illustrated in Fig. 19. When a cache miss happens, the BAT is searched for the mapped address, followed by a TLB access to obtain the physical address. When a writeback happens, if the victim is not an RR block, the writeback just needs to go through address mapping and TLB translation as with the cache miss. If the victim is an RR block, a permutation must be initiated before it can go off-chip. The permutation starts by sending the status bitvectors to the address generation unit which then emits 128 addresses for the permutation unit. When a new mapping is ready, it is then updated into the BAT table. The victim can now be sent off chip with a new mapping.

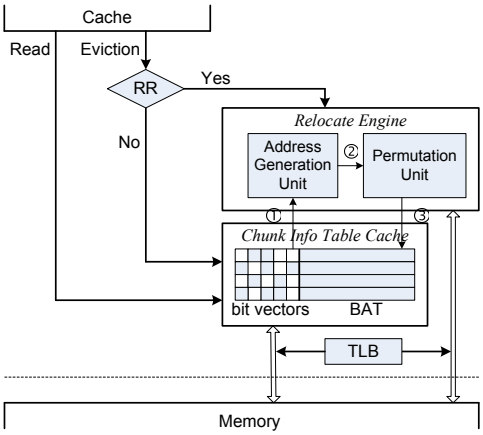

Figure 19: Architecture of the permutation unit.

We assume the entire permutation takes 300 cycles on a GHz processor, 128 for the address generation unit, 128 for the permutation, and the rest for address updates (BAT can be multiported). Since the number of permutations in our scheme is low, and the permutation is done on the write path which is non-critical, such a latency does not bring forth noticeable performance penalty.

6. EVALUATION

For all the data shown earlier in the paper, we used the SimpleScalar Tool set 3.0 [7] to run 11 SPEC2K benchmark programs. All programs were simulated for 1.1 billion instructions. The parameters used are listed in Table 1.

Table 1: Simulation parameters

Clock freq.	1GHz	Unified L2	1MB, 4way, 32B, 12 cycle
RUU/LSQ size	128/64	L1 I- and D-cache	8KB, 32B Direct-map, 1 cycle
Fetch queue	32 entries	Memory bus	200MHz, 8 byte wide
Decode/Issue/ Commit width	8/8/8	Memory latency	80(critical), 5(inter) cycles
TLB miss	30 cycles	Chunk size	4KB∼64KB

We implemented our chunk level permutation scheme, the HIDE scheme, and the Shuffle scheme. We assumed perfect auxiliary on-chip storage for the chunk info table in our scheme, the page info record in the HIDE scheme, and the block address table in the Shuffle scheme. We have shown the reductions of them earlier in Fig. 15 and 16. This is mainly due to the removal of memory page sweeps during each permutation, and the reduction in total number of permutations due to chunk level permutations shown in Fig. 14.

Memory Energy Consumption We also compare the memory energy consumptions for different schemes to show the impact of memory access increase on its total energy. Note that the total energy may not be proportional to the total access numbers because burst reads and writes consume less energy than sparse accesses. The energy is also a function of internal banking and row buffering. We used a detailed trace driven DRAM simulator [10, 11] with the latest power model. Due to the simulation speed, the traces were generated over 100M instructions after fast-forwarding one billion instructions and the chunk size is set to one page. We simulated our benchmarks using the DDR2 specification [3] for a 1Gbit memory with 1 rank, each rank having 5 chips with the 5th being ECC, 32 bit interface with total bandwidth of 2.67GB/s running at 667MHz on a 4GHz processor. Hence, our memory model projects into future processor architectures that will have high bandwidth requirement.

As expected, HIDE and Shuffle consume more energy on aver-

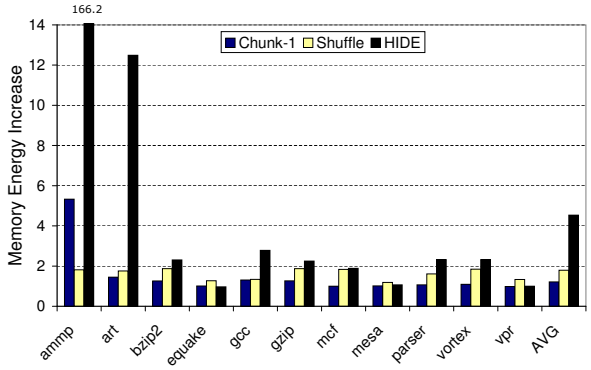

Figure 20: Memory energy consumption increase normalized to the base.

age than ours. As shown in Fig. 20, on average, HIDE and Shuffle increase energy consumption level by a factor of 4.53 and 1.79 respectively. In contrast, our scheme shows a modest increase in energy consumption of 1.21× the baseline. In the case of benchmark `ammp`, our scheme incurs higher memory traffic than Shuffle as shown in Fig. 15, thus resulted in an increase of memory energy consumption. However, in the same benchmark HIDE increases energy by a factor of 166. This is because the trace for `ammp` during the collected interval shows much higher memory demand than the overall results (68.85× in Fig. 15) for 1.1 billion instructions.

7. RELATED WORK

Apart from the address sequence randomization techniques [13, 14, 32, 33], there are some other techniques that aim at protecting the CFG of a program as well. Earlier attempts have taken a software obfuscation approach to transform a code into a form that is harder to reverse engineer [9]. However, it is theoretically uncertain whether a generated transformation can ensure a required level of protection, as studied and proved in [8, 29]. In the world of embedded computing, protecting the program runtime execution trace has been adopted in commercial products since many embedded processors are used in financial applications in which secrecy is highly required (e.g., DS5000 and DS5002FP by Dallas Semiconductor [1]). The protection is done through encrypting the off-chip address and data buses, which unfortunately creates a fixed mapping for addresses that can be easily cracked [17].

8. CONCLUSION

We propose an efficient address permutation scheme for protecting the information leakage from the address bus under physical tampering. Our technique addresses two main issues of the previously proposed HIDE scheme: the excessive memory accesses per permutation and redundant permutations. We also avoid the large number of page faults that incurred in the Shuffle scheme. On average, our scheme reduces the memory traffic in HIDE from 12× to 1.88×, and brings the memory energy consumption from 4.53× down to 1.21×.

9. ACKNOWLEDGEMENT

This work is supported in part by NSF grants CCF-0430021, CCF-0541456, CCF-0447934, CCF-0326396, CNS-0325536, and a DOE Early CAREER Award.

10. REFERENCES

[1] DS5002FP secure microprocessor chip data sheet.
[2] http://www.buyya.com/ecogrid.
[3] http://www.micron.com.
[4] http://www.modchip.com.
[5] 2.5-inch enterprise disc drives. Technology Paper, Feb 2005.
[6] IA-32 Intel Architecture Software Developer's Manual, Vol. 3A, Part 1. ftp://download.intel.com/design/Pentium4/manuals/25366818.pdf, Jan 2006.
[7] T. M. Austin. The SimpleScalar Toolset. http://www.simplescalar.com, Simplescalar LLC.
[8] B. Barak, O. Goldreich, R. Impagliazzo, and etc. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18.J, 2001.
[9] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, 1997.
[10] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. High-Performance DRAMs in Workstation Environments. *IEEE Trans. on Computer*, 50(11):1133–1153, 2001.
[11] V. Cuppu, B. L. Jacob, B. Davis, and T. N. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *the 26th International symposium of Computer Architecture*, pages 222–233, 1999.
[12] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *The 9th International Symposium on High Performance Computer Architecture*, pages 295–306, 2003.
[13] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *The 19th Annual ACM Symposium on Theory of Computing*, pages 182–194, 1987.
[14] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, May 1996.
[15] P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *International Cryptology Conference*, 1996.
[16] M. G. Kuhn. The trustno1 cryptoprocessor concept. Technical Report CS555 Report, Purdue University, 1997.
[17] M. G. Kuhn. Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp. *IEEE Transactions on Computers*, 47(10):1153–1157, Oct 1998.
[18] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.
[19] C. McClure. Software reuse planning by way of domain analysis. Technical Paper, Extended Intelligence, Inc. http://www.reusability.com.
[20] A. Nanda, K. S. Kwok-Ken Mak, R. K. Sahoo, V. Soundararajan, and T. B. Smith. MemorIES: A programmable, Real-time Hardware Emulation Tool for Multiprocessor Server Design. In *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, November 2000.
[21] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *RSA Conference*, February 2006.
[22] C. Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005.
[23] X. Qiu and M. Dubois. Towards Virtually-Addressed Memory Hierarchies. In *the 7th International Symposium on High-Performance Computer Architecture*, pages 51–62, 2001.
[24] X. Qiu and M. Dubois. Tolerating late memory traps in dynamically scheduled processors. *IEEE Transactions on Computers*, 53(6):732–743, June 2004.
[25] X. Qiu and M. Dubois. Moving address translation closer to memory in distributed shard-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):612–623, July 2005.
[26] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 14–24, 2005.
[27] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *the 17th International Conference on Supercomputing*, pages 160–171, 2003.
[28] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *the 36th International Symposium on Microarchitecture*, pages 339–350, 2003.
[29] C. Wang. *A security architecture for survivability mechanism*. PhD thesis, University of Virginia, October 2000.
[30] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *the 36th International Symposium on Microarchitecture*, pages 351–360, 2003.
[31] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta. SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. In *The 11th International Symposium on High-Performance Computer Architecture*, pages 352–362, 2005.
[32] X. Zhuang, T. Zhang, H.-H. S. Lee, and S. Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 292–302, 2004.
[33] X. Zhuang, T. Zhang, and S. Pande. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 72–84, 2004.