Camouflage: Memory Traffic Shaping to Mitigate Timing Attacks

Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff
Electrical Engineering Department
Princeton University
Princeton, NJ, United States
yanqiz@princeton.edu, swagh@princeton.edu, pmittal@princeton.edu, and wentzlaf@princeton.edu

***Abstract*—Information leaks based on *timing side channels* in computing devices have serious consequences for user security and privacy. In particular, malicious applications in multi-user systems such as data centers and cloud-computing environments can exploit *memory timing* as a side channel to infer a victim's program access patterns/phases. Memory timing channels can also be exploited for covert communications by an adversary.**

We propose Camouflage, a hardware solution to mitigate timing channel attacks not only in the memory system, but also along the path to and from the memory system (e.g. NoC, memory scheduler queues). Camouflage introduces the novel idea of shaping memory requests' and responses' inter-arrival time into a *pre-determined distribution* for security purposes, even creating additional fake traffic if needed. This limits untrusted parties (either cloud providers or co-scheduled clients) from inferring information from another security domain by probing the bus to and from memory, or analyzing memory response rate. We design three different memory traffic shaping mechanisms for different security scenarios by having Camouflage work on requests, responses, and bi-directional (both) traffic. Camouflage is complementary to ORAMs and can be optionally used in conjunction with ORAMs to protect information leaks via both memory access timing and memory access patterns.

Camouflage offers a tunable trade-off between system security and system performance. We evaluate Camouflage's security and performance both theoretically and via simulations, and find that Camouflage outperforms state-of-the-art solutions in performance by up to 50%.

***Keywords*-hardware; security; memory system;**

I. Introduction

Running VMs on the same physical machine has become prevalent in Clouds and data centers. Workload consolidation and novel architectures improve server utilization though they can compromise the security of VMs due to leaked information caused by resource sharing [1], [2], [3], [4]. While Secure software [5] and hardware have been proposed, including an authentication circuit [6], Intel's TXT [7], eXecute Only Memory (XOM) [8], Aegis [9], and Ascend [10], they do not prevent information leakage through side channels caused by resource sharing.

Applications from different security domains can share hardware resources (caches, on-chip networks, memory channels, etc.). Contention or interference in shared resources results in information leakage across security domains. For instance, memory queuing delay and scheduling delay highly depend on co-running workloads. Interference between different security domains can create performance interference on some applications, posing a security threat by creating an opportunity for timing channels [11], [12]. For example, when two VMs share a memory system, their memory requests will be co-scheduled by a centralized memory controller. The memory request service time of one application can be greatly impacted by the other application. Figure 1 shows a timing channel attack, where the malicious VM can infer information from the co-running VM by observing its own memory service time (response time).

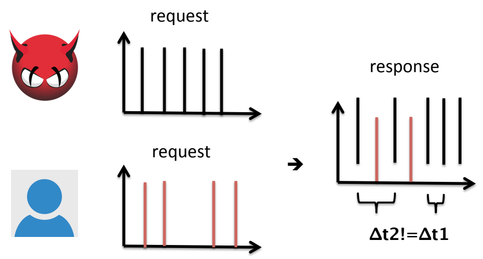


Figure 1. An Example of Timing Leakage. Attacker measures its own response latency to estimate a co-scheduled VM's memory traffic.

In our work, we focus on timing channel attacks in a shared memory system and in the shared channel (NoC, etc.) connecting the processor cores and memory controllers, where the attacker can measure the timing of memory requests and responses, and statistically infer information of victim VMs. We make no assumption on who the malicious entity is. It can be a malicious server, who controls the processor and wants to learn more about the user's data by monitoring the processor's I/O pins or memory buses. Alternatively, it can be a malicious client, who tricks the server into being scheduled on the same physical machine as the victim and infers the victim's memory timing information by measuring its own memory access latency.

Side-channel attacks and their countermeasures have been widely studied in the context of shared caches, on-chip networks, and shared memory channels. Most of them rely on static spatial/temporal partitioning of resources and reducing interference between security domains, thus incurring significant performance degradation. Higher security can be achieved at the cost of performance or higher hardware overhead. An efficient way to remove side-channels is to use a static scheduling or partitioning algorithm. For instance, rather than using a First Ready-First Come First Serve (FR-FCFS) memory scheduling algorithm to improve row buffer hit rate, a leakage-aware scheduler can allocate a static scheduling window for every process/thread that shares the memory system. This potentially impairs throughput and utilization because of its fixed scheduling windows. Alternatively, memory banks/ranks can be partitioned so that each thread accesses a different bank/rank. However, this reduces the effective memory capacity for each thread and could potentially lead to imbalanced memory accesses.

We propose a novel memory traffic shaping and traffic generation mechanism, Camouflage, that is able to camouflage the timing information of memory requests and responses. In order to create a memory traffic distribution exactly matching a pre-determined one, Camouflage limits its traffic rate as well as generates fake traffic. Different from ORAM, we do not solve the memory address/data encryption/obfuscation problem, but only focus on timing channel attacks. However, Camouflage is complementary to address/data encryption/obfuscation or techniques like ORAM. Camouflage can mitigate multiple threat models when used together with ORAM. We don't time partition the memory scheduling window or spatially partition memory capacity, but rather only camouflage memory requests and responses to make it difficult or impossible for a VM to infer any useful information. Specifically, Camouflage uses three different strategies to address different attack scenarios. Request Camouflage (ReqC) shapes only memory request inter-arrival distribution at the processor core side. This limits obtaining timing information from probing or monitoring I/O pins or the path from the core to and from memory. Response Camouflage (RespC) shapes only memory response inter-arrival distribution at the egress of the memory controller. This prevents an untrusted VM from inferring traffic patterns of other VMs by observing its own memory response latencies. Bi-directional Camouflage (BDC) shapes both requests and responses, securing both request and response timing information. The hardware mechanism takes inspiration from the MITTS hardware memory traffic shaper [13], but applies memory traffic shaping for security and not for increasing performance.

Camouflage provides a richer security/performance trade-off space compared with a constant rate shaper (CS) [14]. Compared with Temporal Partitioning (TP) [15], Camouflage mitigates memory timing leakage while not compromising performance and hardware efficiency. Different from Fixed Service (FS) [16], which relies on constant rate shaping and spatial (bank/rank) partitioning, Camouflage does not rely on spatial partitioning for higher performance and is scalable to larger number of threads (more than the total number of banks/ranks). Figure 2 shows the trade-off space provided by Camouflage compared with CS, TP, and FS. Camouflage can be configured to be a constant rate shaper by using only one bin, but also can be used

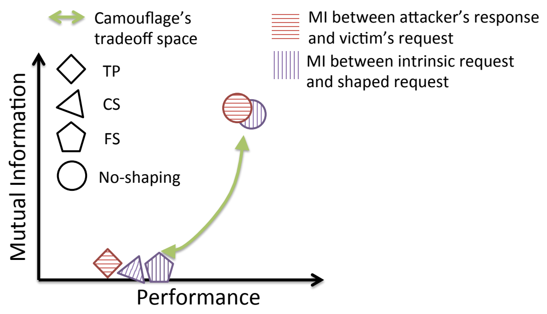


Figure 2. Camouflage Security and Performance Trade-off Space. This plot is based on analysis of the mutual information vs. performance.

to explore alternative security/performance trade-off points. In simulation, Camouflage on average improves program throughput by 1.12x, 1.5x, and 1.32x compared with CS, TP, and FS respectively. We also conduct a real covert channel attack and show that Camouflage is able to mitigate practical covert channel attacks.

Overall, our work demonstrates the feasibility of new design points for mitigating timing channel leaks that provide a combination of strong security and performance, and can serve as a key enabler for the practical deployment of hardware-based timing defenses. Our main contributions include:

1) We design three hardware traffic shaping and generation mechanisms that shape memory requests, responses, and both in terms of inter-arrival times, and generate fake traffic if needed. This enables security-sensitive VMs to camouflage their memory request and response timing.
2) Camouflage protects timing information not only in the memory system, but also the shared channel between processor cores and memory controllers.
3) We leverage information theory and the idea of mutual information to analyze the amount of information that Camouflage leaks. We show that it is less than 0.1% of the transmitted information.
4) We design Camouflage to provide a larger security and performance trade-off space. Camouflage can be configured to leak zero (constant rate shaping), or can be configured to leak slightly more (optimized for performance).

II. Motivation

A. Threat Model

This paper focuses on mitigating or preventing two types of threats: memory-based side-channel and covert-channel attacks, and monitoring of I/O pins or memory buses.

Memory Side-Channel and Covert-Channel: The adversary measures its own overall execution time or memory response latencies in order to estimate its co-scheduled application VM's memory intensity over time. Applications create interference in the memory system due to contention in hardware resources such as NoCs, queues, row buffers, and the memory scheduler. Increasing memory intensity of one application is very likely to slow down other applications' service rate. Therefore, by monitoring the memory response rate change or the overall execution time change, the adversary can infer the memory access pattern of another application.

Pin/Bus Monitoring: The adversary (eg. a data center administrator) has physical access to the processor's I/O pins, system buses, and peripherals. Such information includes the program's start and termination time, the addresses and data sent to and read from the main memory, and when each memory access occurs. In this threat scenario, we focus on the timing aspect of memory access and assume the address and data are protected by ORAM [17] or are encrypted. For example, the adversary can observe communications over the bus between the processors and the memory, in terms of access number and frequency. We assume only off-chip components are insecure while on-chip components are secure. Moreover, we assume the adversary can conduct fine-grain timing measurements, at a per-memory-request level. These fine-grain measurements can lead to direct information leakage of the victim's program characteristics.

B. Timing Protection Overheads

Temporal Partitioning (TP) [15] divides time into fixed-length turns during which only requests from a particular security domain can be issued. It provides security against timing based side-channels. Static temporal partitioning reduces the amount of flexibility in a scheduler, impairing throughput and utilization. For example, application memory traffic is unlikely to be a constant. When there are not sufficient requests from a specific process in its own time division, it is desirable to schedule requests from other processes. Temporal Partitioning applications based on several security domains is feasible, however, it is not scalable if hundreds of applications don't trust each other. For example, if one hundred processes are running on a manycore processor and each ask for separate security domains, TP will have trouble providing high bandwidth, as each of them only receives $\frac{1}{100}$ of the memory bandwidth.

A secure processor, Ascend [18] prevents leakage over the ORAM timing channel by forcing ORAM to be accessed at a single, strictly periodic rate. An enhanced version [14] splits a program into coarse-grain time epochs and chooses a new ORAM rate out of a set of rates at the end of each epoch. This technique bounds the leakage to $E \times \log R$, where E is total number of epochs and R is total number of rates.

Fixed Service (FS) [16] forces every thread to have a constant memory injection rate and a uniform memory access pattern. Similar to CS, it provides little tradeoff opportunity in selecting between security and performance. Combined with memory bank/rank partitioning, it improves performance compared with TP. Spatial partitioning memory reduces the effective memory capacity and bandwidth for each thread and could potentially create imbalanced accesses to a few banks/ranks. Spatial partitioning does not work well when there are massive number of threads that is greater than the total number of ranks or banks.

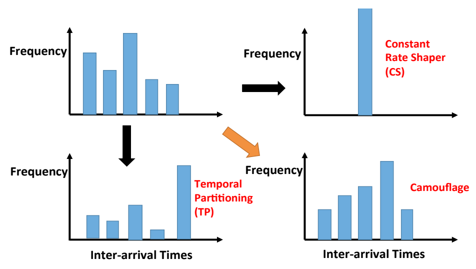


Figure 3. Conceptual Difference between Camouflage and Two Prior Work (CS [14] and TP [15]). CS has only requests/responses in one bin. TP has more requests/responses in high latency bins due to time multiplexed resource.

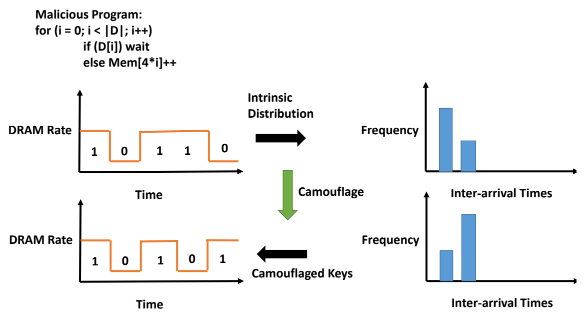

```
Malicious Program:
for (i = 0; i < |D|; i++)
        if (D[i]) wait
        else Mem[4*i]++
```

Figure 4. Camouflage a Vector of Keys. Camouflage slightly changes the request inter-arrival time distribution to distort the inferred keys.

These three techniques negatively impact program performance and provide little tradeoff opportunity for choosing between security and performance. Having more than three points in the security and performance tradeoff space is highly desirable.

C. Memory Traffic Distribution and Timing Channel Protection

Ideally, Camouflage only tunes the traffic pattern slightly so that a malicious user cannot infer the desired information from the camouflaged traffic pattern without significantly changing the intrinsic traffic distribution. There are two different aspects of memory access patterns, bulk bandwidth (total number of memory requests within a timing window) and burstiness. Executing a certain branch of code results in changes in memory bulk bandwidth. A program that deliberately conveys a bit array of sensitive key information can encode that information in memory burstiness. Camouflage uses a memory inter-arrival time distribution to encapsulate both aspects of memory accesses. A distribution describes how an application's memory requests/responses occur at different intervals, and what percentage of requests/responses fall into a specific inter-arrival time.

In our proposed memory distribution, the horizontal axis

Techniques	Capable of Preventing		Performance
	Pin/Bus Monitoring	Memory Side-Channel Covert-Channel	
ReqC	Yes	No	High
RespC	No	Yes	High
BDC	Yes	Yes	High
TP [15]	No	Yes	Impacted by the number of security domains
CS [14]	Yes	No	Low for workloads with non-constant memory request rates
FS [16]	No	Yes	Requires spatial partitioning for better performance

Table I
DIFFERENT MEMORY TIMING PROTECTION TECHNIQUES. FIRST THREE ARE CAMOUFLAGE.

represents the time difference between two subsequent memory requests, while the vertical axis determines how frequent a request falls into a certain inter-arrival category. The inter-arrival time along with the frequency at which memory requests occur with that inter-arrival time determines the bandwidth consumed.

Camouflage shapes memory request/response inter-arrival times into pre-determined statistical distributions. This pre-determined distribution is independent of the intrinsic distribution, thus reducing leaked timing information. Different from conventional static partitioning or static rate limiting, Camouflage enables the choice of flexible distributions of request/response inter-arrival times, which improves performance and memory channel utilization while still hindering timing attacks. As shown in Figure 3, Camouflage does not necessarily shape the intrinsic traffic into a constant rate as a constant rate shaper, or delay a significant number of requests due to time partitioning of the scheduling window (which results in a large number of entries in the high latency bar for Temporal Partitioning). However, Camouflage can be configured as a constant rate shaper if necessary. Intuitively, Camouflage is able to hide the frequency domain information in a more generalized and efficient way. Figure 4 shows an example where a malicious program leaks a vector of secret keys. With Camouflage, we slightly change the request inter-arrival time distribution so that the inferred keys are distorted. Table I summarizes the differences between different techniques.

III. ARCHITECTURE

A. Hardware Design

Camouflage can shape either memory requests, memory responses, or even a combination of both.

As shown in Figure 5, a memory request shaper is placed locally after a processor core's LLC to limit memory request rate for a particular core or thread. The request shaper (ReqC) can transform a process's intrinsic traffic into a fixed pre-determined inter-arrival time distribution. This prevents timing leakage in multiple shared channels, such as NoC (SC1), the memory controller (SC2, SC4), and DRAM (SC3). The request shaper needs to be able to throttle down the request rate when the desired request rate is lower than the intrinsic request rate, as well as generate **fake** requests if the desired request rate is higher than the intrinsic request rate. The post-shaper memory traffic distribution does not vary with different program phases or branches. Request

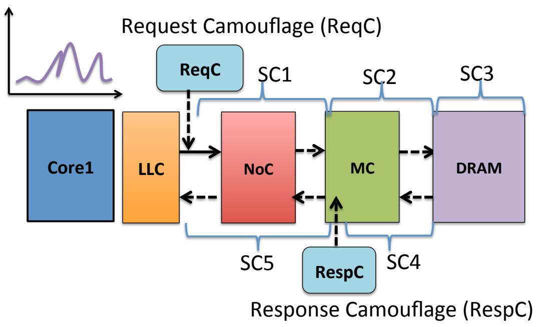


Figure 5. Request, Response, and Bi-directional Camouflage

Camouflage can effectively mitigate I/O pin or memory bus timing channel attack, when combined with address/data encryption/obfuscation techniques.

Similarly, a memory response shaper (RespC) is placed at the egress of the memory controller, before entering a particular processor's LLC. The response shaper eliminates timing leakage generated by the memory system (SC3), providing timing security for shared channels (SC) 4 and 5. For instance, issuing requests from different threads to the same memory rank or bank creates contention on memory bus and row buffers. This can lead to memory side-channel or covert-channel attacks. The RespC throttles down response rate by buffering the response and accelerates response rate by signaling the memory scheduler to give higher priority to the requesting application or generating fake memory responses. This technique reduces the correlation between the victim's request rate and the attacker's response rate and effectively camouflages the interaction occurred in the memory system.

Bi-directional Camouflage (BDC) shapes both memory requests and responses by combining ReqC and RespC. This technique is desirable when we require a shaping mechanism for both memory requests and responses and do not want to change memory controller scheduling policies.

Camouflage is able to shape memory requests and responses into arbitrary statistical distributions, many more options than a fixed rate (Ascend). Requests/responses occur at the attack point at different rates, while the overall inter-arrival distribution within a time window is fixed. This obfusticates the timing information between the shaped distribution compared with the intrinsic one, thereby significantly reducing mutual information of the intrinsic traffic timing and the shaped traffic timing.

1) Bin-based Traffic Shaper: Camouflage uses a bin-based request/response shaper to shape inter-arrival times. In order to control the Camouflage hardware, the hypervisor writes special purpose control registers to configure the shape of the request/response distributions. Each individual core has its own request/response shapers. The use of distributed memory bandwidth traffic shaping can scale up with multicore and manycore systems, and it doesn't necessarily require changes to the centralized memory controller. The traffic shaper tracks cache miss information and measures request/response inter-arrival times. It generates a stall signal to the processor core when the request/response rate exceeds the pre-determined value. The traffic shaper also generates fake traffic if the memory request rate is lower than the pre-determined value. In order to track fine-grained inter-arrival times, we have multiple bins that contain available credits for memory requests. Each bin contains credits that represent one memory transaction at a certain request interval determined by the bin. The scheduling of a memory transaction consumes a single credit. If the memory is shaped into a constant request/response rate, there will be only one of the hardware bins that contains credits. Bin configuration can be arbitrary. Instead of being forced to choose a constant rate, Camouflage enables better performance optimization and leverages applications' constructive traffic.

The maximum number of credits in a bin is bounded by the total memory bandwidth that a memory controller can serve. The request/response shaper enforces that a core's memory request/response distribution does not exceed the prescribed/pre-determined distribution by delaying (stalling) a memory transaction if there are no credits available in a bin that represent lower or equal to the memory transaction's inter-arrival time. The memory transaction will be delayed enough until its inter-arrival time matches a corresponding bin that has credits, or until credits have been replenished.

Each bin is a container holding credits for memory requests with a certain request inter-arrival time. The total number of bins N can be determined by how fine-grain the quantization of inter-arrival interval is desired. We choose ten bins in our design in order to enable the traffic shaper to choose from enough distinct inter-arrival times.

2) Bin Credits Replenishment and Fake Request Generation: Credits are replenished with a fixed period. During credit replenishment, if the hardware detects any unused credits, it saves the credits in another array of unused credits. Whenever there are credits in the unused credit registers, the processor core generates non-cached **fake** memory requests to random memory addresses and enqueues these fake requests to the miss handling registers. However, the fake traffic generated always has lower priority than the intrinsic requests, and will only be generated when there isn't a real memory request at the same cycle.

3) Hardware Overhead: Camouflage's implementation is very similar to MITTS (less than 0.1% in area compared to a two-way OoO processor) with the addition of registers and logic for fake traffic generation [19]. Each Camouflage hardware module (of which in each design there may be multiple) contains a register per bin to track current credits, a register per bin to hold the number of credits to replenish, and a register per bin to track unused credits at each replenishment interval. We assume ten bins where each bin is 10-bits. This added hardware overhead is minimal when compared to the size of most security mechanisms such as hardware implementations of ORAM.

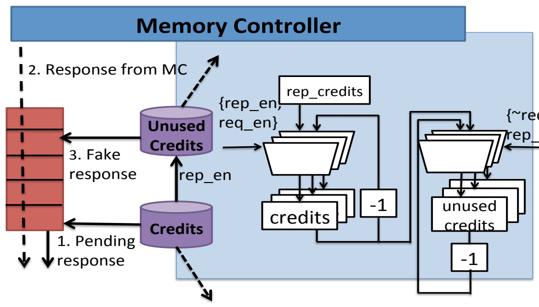


Figure 6. Response Queue. Responses are queued after credits are depleted. Credits are given to queued responses first. 1. There are available credits. 2. There are available credits but no pending responses. 3. There are available unused credits but no pending responses or responses directly from MC.

B. Prevention Mechanisms

1) Memory Side-Channel and Covert-Channel: To combat side-channel and covert-channel attacks caused by response inspection, we propose "Response Camouflage" (RespC). RespC puts shaping hardware at the egress of memory on a per-core or a per-application basis. The hardware accelerates as well as throttles memory responses. For throttling, a response queue buffers responses when not enough credits are available. After bin replenishment, the response queue will be checked to dequeue any response that has been buffered, as shown in Figure 6. However, accelerating responses is challenging if the intrinsic memory intensity is lower than the desired one such that there might not be enough memory responses available. Not enough responses can be caused by the memory being hogged by co-running applications, which slows down the affected application's memory responses. In this case, Camouflage accelerates responses by giving high priority to that particular application in the memory scheduler. The RespC hardware monitors the response inter-arrival time distribution and compares the distribution with a target one. If the response rate is lower than the required one, the RespC sends a warning to the memory scheduler, asking for higher priority for the affected application. At each replenishment, the response shaper sums up unused credits in the hardware bins, and sends the total number of credits along with the warning signal to the memory controller. The memory scheduler will give more priority to the affected application in proportion to the number of unused credits. However, this alone cannot handle changes to a VM's request distribution. In order to maintain a fixed response distribution when a VM drops its request rate, Camouflage generates fake memory responses. Similar to fake request generation, a fake response generator creates fake responses when there are not any pending responses or new responses from the memory controller and there are unused credits accumulated, as shown in Figure 6.

2) Pin/Bus Monitoring: For the I/O pin or memory bus inspection attack, we propose "Request Camouflage"

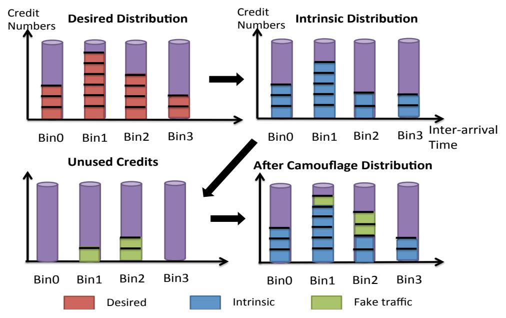


Figure 7. Request Shaping and Fake Requests Generation. Use another set of bins to store unused credits.

(ReqC). ReqC puts the request shaping hardware at the processor core side or after the LLC, so that any distribution of intrinsic memory traffic can be camouflaged into a totally different distribution. We are not addressing address/data obfuscation or encryption problem, but rather only focus on memory timing channel. In order to guarantee the generated memory traffic distribution matches the predetermined distribution, the hardware needs to be able to both throttle and accelerate memory request rate. If the desired request rate is higher than the actual request rate, the hardware generates fake memory requests so that the memory traffic adds up to the desired distribution.

In order to generate fake traffic, we add a register per bin to store unused credits for each replenishment period. Using these credits, Camouflage generates fake requests in the next period to random addresses with the needed distribution. Figure 7 gives an example of generating fake requests. At the end of one replenishment period, Camouflage detects unused credits in Bin1 and Bin2. It immediately saves the unused credits to the unused credit bins. The next replenishment cycle, if the application's intrinsic traffic remains the same as the previous one, the extra fake traffic and the intrinsic traffic will add up to the desired distribution. Even if the distributions of adjacent replenishment periods are different, the added fake traffic compensates for requests missing from the previous replenishment period. The overall traffic distribution will match the desired one.

3) Bi-directional Prevention: Bi-directional technique is desirable when both memory requests and responses are required to be shaped or memory scheduling policies cannot be changed. In order to configure the hardware bins, a software runtime is co-designed to achieve higher performance. BDC can effectively camouflage any suspicious VM's requests and responses when the request distribution changes.

With Bi-directional Camouflage, a malicious VM is unlikely to detect timing channel leakage on the memory bus, as memory traffic from the VMs under protection is fixed. A malicious VM can hardly infer memory traffic of the co-running applications, as its own memory response distribution is fixed. In order to utilize Bi-directional Camouflage, we use a genetic algorithm described in Section IV-C to optimize request and response distributions for each application.

Core	2.4GHz, 4-wide issue, 128-entry instruction window
Number of Cores	4
L1 Caches	32 KB per-core, 4-way set associative, 64B block size, 8 MSHRs
L2 Caches	64B cache-line, 8-way associative, Single-program: 128KB, multi-program: 128KB private
Memory controller	32-entry transaction queue depth
Memory	Timing: DDR3, 1333 MHz Organization: 1 channel, 1 rank-per-channel, 8 banks-per-rank, 8 KB row-buffer

Table II
BASE SIMULATION CONFIGURATION

Camouflage can be configured to constant shape the requests and responses by giving each core the same number of credits in the same bin. By provising more credits than can be used over the replenishment period, Camouflage degenerates into a constant rate shaper that turns into C. Fletcher et al's work [14], which showed no information leakage for unchanging rates. Different from Fletcher et al's work, the return traffic can be shaped by Camouflage as well. Constant response shaping eliminates leakage generated by interference in the DRAM.

IV. EVALUATION

A. Simulation and Workloads

The CPU core and memory system are modeled with a cycle-accurate simulator called SDSim which is adapted from the cycle-accurate core simulator SSim [20], and the DRAM simulator DRAMSim2 [21]. SSim's frontend is integrated with DRAMSim2. This enables us to model out-of-order cores with out-of-order memory systems. SSim is driven by the GEM5 Alpha ISA full system simulator [22]. We model a shared LLC and memory system for the multi-program workloads. Table II shows the details of the simulated system.

We evaluate the SPECInt 2006 benchmark suite together with the Apache web server benchmark. In the evaluation of Response Camouflage and Bi-directional Camouflage, we run two workloads (ADVERSARY, astar, astar, astar) and (ADVERSARY, mcf, mcf, mcf) each time. We name them w(ADVERSARY, astar) and w(ADVERSARY, mcf) for short. ADVERSARY is the untrusted application that inspects the memory bus or its own memory responses to infer use information from the co-scheduled VMs. astar and mcf have wildly different memory intensities and access patterns and are used to simulate the change in memory access. We run 11 workloads (SPECInt 2006 and Apache web server) as the ADVERSARY and compares the performance impacts of Camouflage on both the ADVERSARY and victims.

B. Security Analysis

In order to analyze the effectiveness of different schemes, we leverage mutual information (MI) which comes from classical information theory [23], [24], [25], [26], [27], [28].

Mutual information (MI) is a rigorous metric to characterize statistical privacy in the security/privacy community. In fact, recent work demonstrate strong connections between mutual information and even differential privacy [29].

1) Metric: Mutual Information: The mutual information (MI) of two random variables is a measure of the mutual dependence between the two variables. Specifically, it quantifies the amount of information obtained about one random variable, through the other one. Equation 1 shows the mathematical formulation of MI of variables X and Y.

$$I(X,Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) log(\frac{p(x,y)}{p(x)p(y)}) \tag{1}$$

Camouflage is considered to be secure if the MI between the intrinsic memory inter-arrival distribution and shaped inter-arrival distribution is zero. In this experiment, we compare the long term MI between memory request inter-arrival time distribution before and after Camouflage. More specifically, X and Y contain timing intervals t_i. $p(x = t_i)$ and $p(y = t_i)$ are probability densities for the intrinsic memory request/response and shaped memory request/response at inter-arrival time t_i. $p(x = t_i, y = t_i)$ is the joint probability density at inter-arrival time t_i. In our implementation, we have ten different intervals.

2) Mutual Information Measurement: In order to prove that the MI between traffic distributions before and after Camouflage is minimal, we measure memory request inter-arrival time distributions before and after ReqC and compute the MI between the intrinsic request distribution and the shaped distribution. Without any traffic shaping, the mutual information will be $I(X, X) = H(X)$, which is just its self information. After a traffic shaper, the MI becomes $I(X, Y)$, and ideally should be zero. In our experiment, we evaluate MI for a non-shaping case, a constant rate shaper, and ReqC for w(ADVERSARY, bzip). While the non-shaping case has a MI of 4.4, a constant rate shaper (without fake traffic) reduces the MI to 0.002, ReqC (without fake traffic) reduces the MI to 0.006. With fake traffic, a constant shaper further reduces the MI to 0, and ReqC reduces the MI to 0.002. Other benchmarks generate similar results. This result implies that Camouflage at most leaks 0.1% of the information compared to non-shaping case, and leaks slightly more than the constant rate shaper. In other words, ReqC leaks only 0.1 byte if a non-shaping scheme leaks 100 bytes.

3) Mutual Information with BDC: In this section, we analyze the MI of BDC. We show below that it is never worse than ReqC or RespC. Assume there are two VMs sharing the memory channel. Request inter-arrival time of Alice is converted to A_i by ReqC. Assuming the best case scenario for an adversary Bob, the memory response B_r contains all possible information about the response to Alice (which is A_r) and thus the request of Alice (which is A_i). Now the RespC modifies this response (B_r) to give Bob a further shaped response B. Schematically,

$$A \rightarrow A_i$$
$$B \leftarrow B_r = A_r$$

The arrows represent ReqC and RespC, which make the inputs more unlikely to leak information. We will complete this argument using a data processing inequality for the best case scenario for an adversary Bob ($A_r = B_r$). According to data processing inequality, the inequality loosely states that processing data does not increase MI[1]. This implies:

$$I(A; B) \leq I(A; A_i) \text{ and using it the other way}$$
$$I(A; B) \leq I(B; A_i)$$

So the overall system will be at least as good as the following:

$$I(A; B) \leq min(I(A; A_i), I(B, A_i))$$

which in other words BDC will be at least as good as the best of ReqC and RespC. Since ReqC and RespC use the same throttling and fake traffic generation mechanism, BDC will be at least as good as ReqC, as described in section IV-B2.

4) Leakage Within A Replenishment Window: Camouflage focuses on preventing leakage of long term timing information (analyzed above), on the order of thousands of cycles (longer than the replenishment period of Camouflage), because long term timing information is the easiest to exploit with practical attacks. Nevertheless, we analyze Camouflage in the extreme case where very fine grain timing information can be used within a replenishment period to gain information in a side-channel attack. As a worst case attack, we assume the adversary can receive a bit of information for every request of its own. If its request is delayed, it knows the victim had a request at the same time, else the victim did not have a request. *We also make a conservative assumption that the adversary knows its shaped distribution, the distribution of the victim, and it can specifically control the timing of its requests.* Given these conservative assumptions, leaked information is bounded by the number of credits that the adversary has. In practice, this leakage is mitigated by the lack of the timing accuracy that the adversary can use to time memory requests. This severely degrades the number of (fractional) bits that can be leaked per memory conflict. Also, any time that the victim is actively shaped or fake traffic is created by Camouflage, the intrinsic timing information is obfuscated. If leakage within replenishment window is considered to be a significant threat, the timing of memory requests can be randomized within the interval (that a credit represents) to increase the timing uncertainty and probability of memory conflict in a

[1]More formally, if $X \rightarrow Y \rightarrow Z$ is a Markov Chain i.e., Z conditioned on Y is independent of X, then $I(X; Y) \geqslant I(X; Z)$

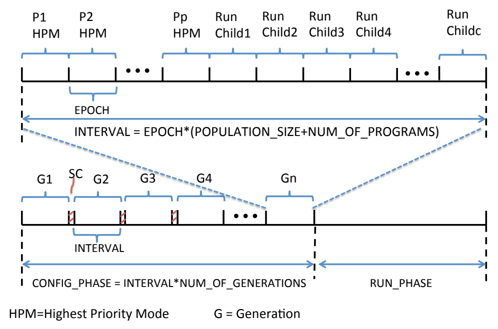

P1 HPM
P2 HPM
Pp HPM
Run Child1
Run Child2
Run Child3
Run Child4
Run Childc
EPOCH
INTERVAL = EPOCH*(POPULATION_SIZE+NUM_OF_PROGRAMS)
G1
SC
G2
G3
G4
Gn
INTERVAL
CONFIG_PHASE = INTERVAL*NUM_OF_GENERATIONS
RUN_PHASE
HPM=Highest Priority Mode
G = Generation

Figure 8. Online Genetic Algorithm. Same GA as in MITTS [13]

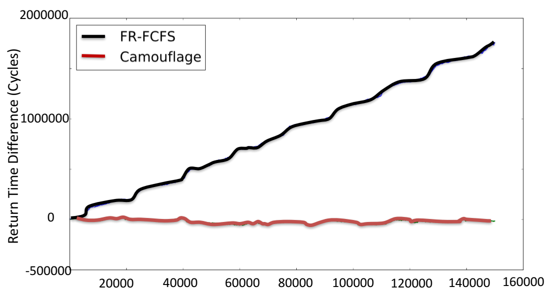


Figure 9. Memory Request Return Time Difference of Two Techniques

randomized manner. Also, short term information leakage can be mitigated by reducing the size of the replenishment window.

C. Optimizing Bin Configuration

Camouflage uses a genetic algorithm (GA) to optimize the hardware bin configurations initially, and guarantees the request/response inter-arrival time distributions do not change due to application interference. Genetic algorithms work well for non-convex search spaces like what we have.

For the BDC, we need an algorithm to optimize performance while camouflaging the memory traffic. With a 10-bin Camouflage that shapes both requests and responses, the search space could be ($MAX\_CREDITS^{20}$), where $MAX\_CREDITS$ is the total number of credits allowed in a bin. For all of the results presented for the BDC, we use an online genetic algorithm to optimize bin configurations. The online genetic algorithm trades off program performance for timing information leakage by reconfiguring the hardware bins. When a constant amount of information leakage is allowed, the online genetic algorithm reconfigures the request/response hardware bins after a fixed amount of time or after a program phase change. As a result, distributions are fixed within a reconfiguration window, but are different across configuration windows. To avoid leakage due to reconfiguration, the online genetic algorithm can be used at the beginning of the program, the proposed configuration will be used for the rest of the program after the configuration phase.

Genetic Algorithms are flexible enough to optimize for any objective functions, such as program performance, multi-program system throughput, fairness, program security, or a combination of all. In our example, we are interested in preventing timing information leakage within a multi-program system without compromising system throughput. Specifically, the Genetic Algorithm optimizes for multi-program average slowdown, by minimizing $\frac{\sum_{i=1}^{n} slowdown_i}{n}$.

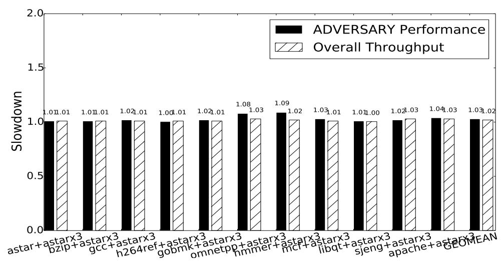
The online genetic algorithm in Figure 8 configures Camouflage at the the beginning of a program phase (Config_Phase), and uses the optimal configuration for the rest of program phase (Run_ Phase). The GA needs to run several generations (typically 20 or 30) with around 20-30 configurations tested in each generation. We name each configuration a child within a generation.

The genetic algorithm is able to search for optimal bin configuration for either a single-program workload or a multi-program workload. For a multi-program workload, the GA optimizes all bins from all programs simultaneously. We use MISE's [30] online profiling design to measure application slowdown ($slowdown\ of\ an\ App = (1-\alpha)(\alpha\frac{Request\ Service\ Rate\ with\ Highest\ Priority}{Shared\ Request\ Service\ Rate})$, $\alpha = \frac{Cycles\ spent\ stalling\ on\ memory\ requests}{Total\ number\ of\ cycles}$). At the beginning of each generation, the GA runs each workload with highest priority in the memory controller in order to measure workload performance without interference in the memory system. This information is combined with performance evaluated in each child configuration to measure workload slowdown when a workload is mixed with other applications. After each run, the runtime saves the measured objective function in memory. After all candidates in one generation are evaluated, the software GA selects the few best configurations and uses them to create the genomes for the subsequent generation (children). We have evaluated different configuration sizes (child size), and have decided to use 20000 cycles for an individual configuration measurement. We run 20 generations in total for each reconfiguration, with about 5000 cycles runtime overhead each generation.

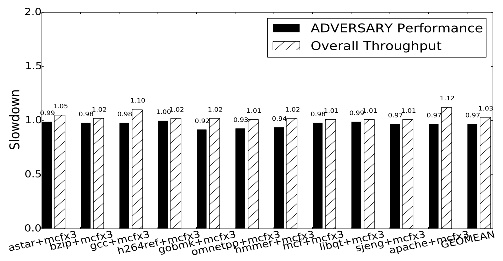
D. Memory Side-Channel and Covert-Channel

In this section, we measure the effectiveness of RespC in preventing malicious VMs from inferring timing information by measuring their own memory response times. We use multi-program workloads made out of SPEC2006 benchmarks and the Apache web server to verify interference can be largely removed.

1) Leakage Evaluation: We empirically evaluate the security features of Camouflage. We measure the accumulated response time difference of w(ADVERSARY, astar) and w(ADVERSARY, mcf) observed by the ADVERSARY application, where ADVERSARY is the adversary application that is stealing timing information from the co-running applications. In order to prevent the ADVERSARY from



(a) astar



(b) mcf

Figure 10. Performance and Throughput Comparison Between Response Camouflage and No Shaping.

inferring memory traffic patterns from the co-running applications, Camouflage guarantees that the ADVERSARY's memory response time for each request won't noticeably change when the co-running application changes its request distribution. As shown in Figure 9, Camouflage maintains a flat line of accumulated response time difference between the above workloads, indicating minimal timing channel leakage via response time inspection. First Ready-First Come First Serve (FR-FCFS) memory scheduling on the contrary, has an increased accumulated response time difference. We ran this experiment with all benchmarks we used in Section IV-E, and see similar profiles for all the benchmarks.

2) Response Camouflage Performance: We evaluate Camouflage on two different cases when an application becomes more memory intensive or less intensive. We use workload w(ADVERSARY, astar) and w(ADVERSARY, mcf) to measure memory request return time for the adversary benchmark. We run each application in our 11 workloads as the ADVERSARY. In the IaaS Cloud, the adversary can be any workload in a lower security domain. As we protect astar and mcf from the adversary application, we name astar and mcf as applications under protection. As mcf is more memory intensive compared with astar, the adversary will notice significant response time increase when co-running with mcf. Ideally, a secure memory scheduler will not change the response time of a benchmark no matter how the co-running benchmarks change. In this case, we put a response shaper at the egress of the memory controller for the adversary application. The bin configuration is set the same as the response distribution as workload w(ADVERSARY, astar). Because the actual response rate is slower than the desired response rate, Camouflage will

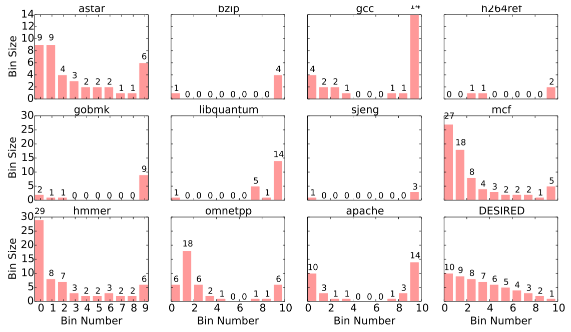


Figure 11. Camouflage Shapes Different Request Inter-arrival Time Distribution into a Desired One

send a signal to the memory controller asking for higher priority for the adversary application. Similarly, if we need to maintain the response distribution of the adversary application for workload w(ADVERSARY, mcf), we set the response shaper bin configuration of w(ADVERSARY, astar) the same as w(ADVERSARY, mcf). In this case, response rate is throttled as the adversary experiences higher response rate compared with workload w(ADVERSARY, mcf).

We measure the ADVERSARY application performance slowdown and overall throughput slowdown due to response Camouflage. Figure 10(a) shows the results of shaping w(ADVERSARY, astar) to the same response inter-arrival distribution as w(ADVERSARY, mcf). And Figure 10(b) shows the result of shaping w(ADVERSARY, mcf) to the same response inter-arrival distribution as w(ADVERSARY, astar). As mcf is more memory intensive than astar, shaping the ADVERSARY's response of w(ADVERSARY, astar) slows down the ADVERSARY, resulting in an illusion that it is still running with mcf. In the other case, shaping mcf requires requesting higher memory request priority in the memory controller to match the behavior of running with astar. Therefore, the ADVERSARY's performance improves because of higher scheduling priority. The throughput, however, is degraded as mcf has lower priority compared with no shaping case. This shows the tradeoff between security and performance and Camouflage guarantees security at a minor cost of performance.

E. Pin/Bus Monitoring

1) Security Evaluation: **Distribution Accuracy** We evaluate Camouflage's effectiveness in shaping any request distribution into another desired one. In this section, we measure 11 application's intrinsic memory request distribution, and set the desired request distribution to a fixed DESIRED distribution. The DESIRED distribution has decreasing size of bins, as shown in the bottom right of Figure 11. As shown in Figure 11, different applications have totally different request distributions. We use another hardware bin to measure the post-Camouflage memory request distribution, and find all the applications have the same distribution as the DESIRED one. This result shows that Camouflage can

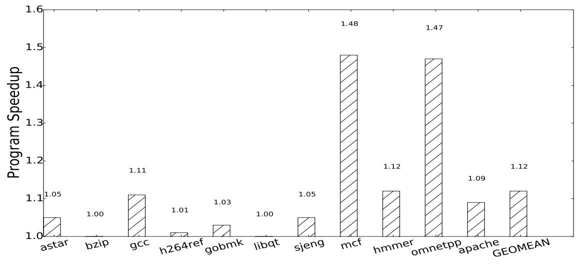


Figure 12. Performance Gain of Request Camouflage Compared with Static Rate Limiter

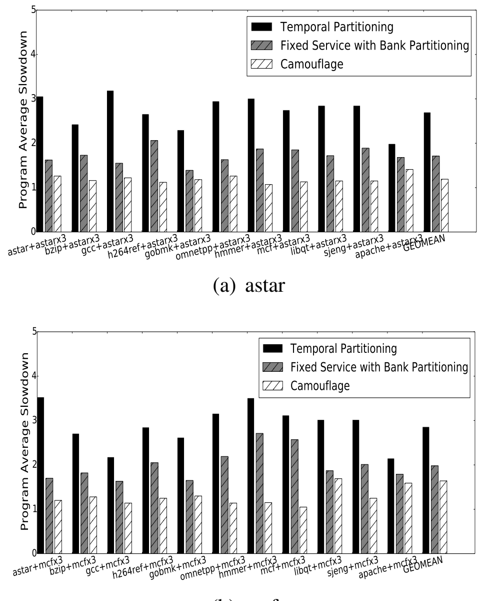


Figure 13. Workload Average Slowdown Compared with TP and FS (with bank partitioning only). Workloads under protection are astar and mcf respectively.

camouflage any request distribution into a totally different one, mitigating memory timing channel attack at I/O pins and memory buses.

2) Performance Evaluation: We evaluate Camouflage's speedup compared with a static memory request shaper. The static shaper limits a program's memory requests into a constant rate but it cannot take into account inter-arrival times. By comparing program speedups, we show Camouflage always outperforms the static rate limiter with the same average bandwidth. In this experiment, we choose 1GB/s bandwidth for each application. The constant rate limiter only allows $\frac{1GB}{REQUEST\_SIZE}$ requests/second of request rate, which Camouflage is able to shape the requests into a distribution which adds up to 1GB/s.

Figure 12 shows the performance gain that Camouflage achieves. Compared to constant rate shaper, Camouflage has 1.12x better geometric mean. This shows Camouflage does not compromise performance while traffic shaping.

F. Bi-Directional Camouflage Performance

In this section, we evaluate BDC by having request shapers for applications under protection and have a response shaper for the adversary application. We first run

Algorithm 1 Covert Channel Attack

```
1: procedure GENERATE COVERT CHANNEL
2:     Keylen ← length of Key
3: top:
4:     if $i > Keylen$ then return 0
5:     end if
6: loop:
7:     if $Key(i) = 1$ then
8:         while $ElapsedTime < PULSE$ do
9:             $BigBuffer[NextCacheLine] \leftarrow 1$. ▷ Generate cache miss for duration of time
10:            $NextCacheLine \leftarrow NextCacheLine + CacheLineSize$.
11:        end while
12:        goto loop.
13:        close;
14:    else
15:        while $ElapsedTime < PULSE$ do
16:            $DoNothing$
17:        end while
18:    end if
19: end procedure
```

the workload w(ADVERSARY, astar) with an online genetic algorithm to configure the hardware bin configurations of the shapers. The genetic algorithm optimizes overall throughput of the workload. In order to evaluate Camouflage's effectiveness in request/response shaping when workload traffic pattern changes, we apply the same configuration to workload w(ADVERSARY, mcf), and evaluate the throughput under traffic shaping. This mimics a workload traffic pattern change from astar to mcf. Camouflage keeps fixed request/response inter-arrival distributions. Then we first run workload w(ADVERSARY, mcf) with the online genetic algorithm and apply the optimal configuration for workload w(ADVERSARY, astar). As we shape both requests and responses, the response distribution is guaranteed to be the same for these two workloads. We run the experiments, and find the response distributions match in two workloads.

We compare Camouflage with TP [15] where each application is allocated a fixed timing channel in the memory scheduler and FS [16] with bank partitioning. As we use only one rank for all the evaluation section, we did not evaluate FS with rank partitioning. As shown in Figure 13(a) and Figure 13(b), Camouflage has minimal impact on workload throughput compared with TP and FS. Camouflage provides better performance than FS because FS still requires a constant memory request rate while Camouflage does not.

G. Covert Channel Prevention

In order to empirically evaluate Camouflage, we implement an algorithm to conduct a covert channel attack. The program will generate memory requests for a fixed amount of time (PULSE) by writing data to different cache lines if the indexed bit in the key is one, otherwise, it does nothing until the same fixed amount of time has passed. Algorithm 1 shows the pseudocode for this algorithm.

We use Camouflage request shaping as a demonstration

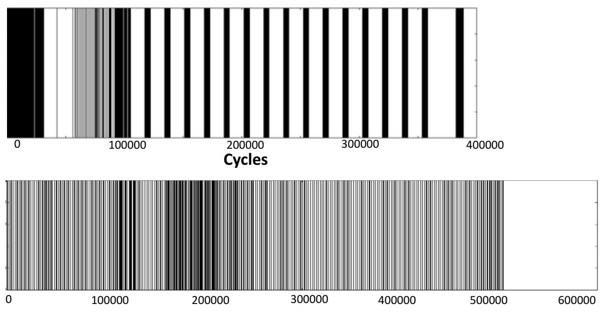


Figure 14. Memory Traffic Before and After Camouflage. Key:32hx2AAAAAAA.

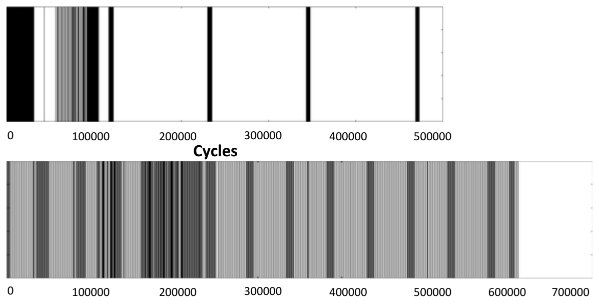


Figure 15. Memory Traffic Before and After Camouflage. Key:32hx01010101.

of covert channel protection. From Figure 14 and Figure 15, we can see Camouflage hides the intrinsic traffic effectively by shaping the memory distribution into another distribution. During the idle period, Camouflage detects unused credits and generates fake memory traffic as a result.

V. Related Work

Covert Channels and Side Channels: Processor architecture features such as simultaneous multithreading, branch prediction, and shared caches inadvertently introduce covert channels and side channels [31], [32]. Camouflage complements existing covert channel and side channel protection techniques, preventing memory timing attacks without compromising performance.

Memory Attacks: Data encryption and oblivious RAM [17] have been used to prevent leaking sensitive information. While data encryption does not prevent information leakage through statistical inference, ORAM conceals memory access pattern by continuously shuffling and re-encrypting data as they are accessed. Camouflage focuses on memory timing channel protection, and can be combined with data encryption and ORAM for other security aspects.

Constant Rate Shaping: A secure processor, Ascend [18] prevents leakage over the ORAM timing channel by forcing ORAM to be accessed at a single, strictly periodic rate. A later enhanced version [14] splits programs into coarse-grain time epochs and chooses a new ORAM rate out of a set of allowed rates at the end of each epoch. These two design points are subsets of Camouflage. Camouflage provides a larger security and performance tradeoff space as it is more flexible in determining memory traffic inter-arrival times.

Interference Reduction: There has been a growing interest in the study of timing channel attacks and mitigation through micro-architectural states, such as cache interference [33], [34], branch predictors [35], and on-chip networks. Non-interference in various hardware resources, such as memory controllers and NoCs, have been studied to prevent timing channels. Ethearal proposed a time-division multiplexed virtual circuit switching network [36] to provide guaranteed services for applications with real-time deadlines. Temporal Partitioning (TP) divides the memory scheduling window into multiple security domains, and only allows applications in the same security domain to be scheduled in the same time window. Camouflage provides a scalable solution compared with TP, as application performance is not influenced by the number of security domains. A bandwidth reservation technique is proposed [37] to avoid information leakage. Camouflage leverages statistical memory inter-arrival time distributions, rather than relying on a fixed scheduling policy.

Fixed Service: FS [16] is a memory controller design that guarantees memory requests to be issued at a constant rate. Combined with spatial partitioning (bank/rank partitioning), it improves performance compared with TP. It requires modifications to the memory controller which Camouflage does not necessarily require. It does not handle timing leakage in shared channels such as NoCs and the path to and from memory nor does it work well with thread counts greater than the number of available memory partitions. Camouflage can furthermore guarantee fixed response distribution in face of request distribution change.

MITTS: MITTS [13] is a distribution-based memory bandwidth shaper that is designed for manycore memory system fairness/throughput and fine-grain pricing in IaaS systems. Camouflage is an extension of MITTS that addresses the memory system's timing-channel leakage problem. Unlike MITTS, Camouflage places traffic shapers at different locations (both request and response channels). Unlike MITTS, Camouflage generates fake traffic for security purposes which generally hurts performance and is antithetical to MITTS' purpose. Finally, Camouflage communicates with the memory controller in certain circumstances to rate limit responses and prevent overflow on the return channels.

VI. Conclusion

Camouflage is a hardware mechanism designed to mitigate memory channel timing attacks. Camouflage shapes memory requests/responses inter-arrival time into a predetermined distribution, even creating additional traffic if needed. This prevents malicious parties from inferring information from another security domain by probing the memory bus, or analyzing memory response rate. We compare Camouflage with Temporal Partitioning, constant rate limiting (Ascend) and Fixed Service (with bank partitioning). We find Camouflage on average improves program performance by 1.5x, 1.12x, and 1.32x respectively. Camouflage provides a larger performance/security tradeoff space than CS, TP, and FS. We analyze the mutual information which can be communicated with Camouflage in place. We also show Camouflage defends against a real covert channel attack.

Acknowledgement

This work was partially supported by the NSF under Grants No. CCF-1217553, CCF-1453112, CCF-1438980, and CNS-1409415, AFOSR under Grant No. FA9550-14-1-0148, and DARPA under Grant No. N66001-14-1-4040.

References

[1] Y. Zhang *et al.*, "Secure information and resource sharing in cloud," ser. CODASPY '15. New York, NY, USA: ACM, 2015, pp. 131–133.

[2] J. Demme *et al.*, "Side-channel vulnerability factor: A metric for measuring information leakage," in *ISCA*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 106–117.

[3] Y. Zhou *et al.*, "CASH: Supporting IaaS customers with a sub-core configurable architecture," ser. ISCA, 2016.

[4] M. Shahrad and D. Wentzlaff, "Availability knob: Flexible user-defined availability in the cloud," in *SoCC*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 42–56.

[5] G. E. Suh *et al.*, "Secure program execution via dynamic information flow tracking," *SIGARCH Comput. Archit. News*, vol. 32, no. 5, pp. 85–96, Oct. 2004.

[6] J. W. Lee *et al.*, "A technique to build a secret key in integrated circuits for identification and authentication applications," in *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, June 2004, pp. 176–179.

[7] *The Intel Safer Computing Initiative: Bulding Blocks for Trusted Computing*, Intel, 2006.

[8] D. L. C. Thekkath *et al.*, "Architectural support for copy and tamper resistant software," in *ASPLOS*, ser. ASPLOS IX, 2000, pp. 168–177.

[9] G. E. Suh *et al.*, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *ICS*, ser. ICS '03, 2003, pp. 160–171.

[10] C. W. Fletcher *et al.*, "A secure processor architecture for encrypted computation on untrusted programs," in *STC*, ser. STC '12, 2012, pp. 3–8.

[11] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," ser. NOCS '12, 2012, pp. 142–151.

[12] H. M. G. Wassel *et al.*, "Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip," in *ISCA*, ser. ISCA '13, 2013, pp. 583–594.

[13] Y. Zhou and D. Wentzlaff, "MITTS: memory inter-arrival time traffic shaping," in *ISCA 2016, Seoul, South Korea, June 18-22, 2016*, 2016, pp. 532–544.

[14] C. Fletcher *et al.*, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs," in *HPCA*, Feb 2014, pp. 213–224.

[15] Y. Wang *et al.*, "Timing channel protection for a shared memory controller," in *HPCA*, 2014, pp. 225–236.

[16] A. Shafiee *et al.*, "Avoiding information leakage in the memory controller with fixed service policies," ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 89–101.

[17] E. Stefanov *et al.*, "Path ORAM: An extremely simple oblivious ram protocol," in *CCS*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 299–310.

[18] C. Fletcher *et al.*, "A secure processor architecture for encrypted computation on untrusted programs."

[19] J. Balkind *et al.*, "Openpiton: An open source manycore research framework," in *ASPLOS*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232.

[20] Y. Zhou and D. Wentzlaff, "The sharing architecture: Subcore configurability for iaas clouds," in *ASPLOS*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 559–574.

[21] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[22] N. Binkert *et al.*, "The GEM5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[23] S. Guiasu, *Information Theory with Applications*. New York: McGraw-Hill, 1977.

[24] L. Sankar *et al.*, "Utility-privacy tradeoffs in databases: An information-theoretic approach," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 6, pp. 838–852, June 2013.

[25] J. Dautrich and C. Ravishankar, "Tunably-oblivious memory: Generalizing oram to enable privacy-efficiency tradeoffs," ser. CODASPY '15. ACM, 2015, pp. 313–324.

[26] H. Yamamoto, "A source coding problem for sources with additional outputs to keep secret from the receiver or wiretappers (corresp.)," *IEEE Transactions on Information Theory*, vol. 29, no. 6, pp. 918–923, Nov 1983.

[27] L. Sweeney, "K-anonymity: A model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 10, no. 5, pp. 557–570, Oct. 2002.

[28] A. Wyner and J. Ziv, "The rate-distortion function for source coding with side information at the decoder," *IEEE Transactions on Information Theory*, vol. 22, no. 1, pp. 1–10, Jan 1976.

[29] P. Cuff and L. Yu, "Differential privacy as a mutual information constraint," *CoRR*, vol. abs/1608.03677, 2016.

[30] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013, pp. 639–650.

[31] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *MICRO*, Dec 2014, pp. 216–228.

[32] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," ser. ACSAC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 473–482.

[33] O. Aciiçmez, "Yet another microarchitectural attack:: Exploiting i-cache," ser. CSAW '07, 2007, pp. 11–18.

[34] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, ser. ISCA '07, 2007, pp. 494–505.

[35] O. Aciiccmez *et al.*, "Predicting secret keys via branch prediction," ser. CT-RSA'07, 2006, pp. 225–242.

[36] K. Goossens, J. Dielissen, and A. Radulescu, "Thereal network on chip: Concepts, architectures, and implementations," *IEEE Des. Test*, vol. 22, no. 5, pp. 414–421, 2005.

[37] A. Gundu *et al.*, "Memory bandwidth reservation in the cloud to avoid information leakage in the memory controller," ser. HASP '14. New York, NY, USA: ACM, 2014, pp. 11:1–11:5.