Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-offs

Christopher W. Fletcher†*, Ling Ren†, Xiangyao Yu†, Marten Van Dijk‡, Omer Khan‡, Srinivas Devadas†

† Massachusetts Institute of Technology – {cwfletch, renling, yxy, devadas}@mit.edu
‡ University of Connecticut – {vandijk, omer.khan}@engr.uconn.edu

Abstract

Oblivious RAM (ORAM) is an established cryptographic technique to hide a program's address pattern to an untrusted storage system. More recently, ORAM schemes have been proposed to replace conventional memory controllers in secure processor settings to protect against information leakage in external memory and the processor I/O bus.

A serious problem in current secure processor ORAM proposals is that they don't obfuscate when ORAM accesses are made, or do so in a very conservative manner. Since secure processors make ORAM accesses on last-level cache misses, ORAM access timing strongly correlates to program access pattern (e.g., locality). This brings ORAM's purpose in secure processors into question.

This paper makes two contributions. First, we show how a secure processor can bound ORAM timing channel leakage to a user-controllable leakage limit. The secure processor is allowed to dynamically optimize ORAM access rate for power/performance, subject to the constraint that the leakage limit is not violated. Second, we show how changing the leakage limit impacts program efficiency.

We present a dynamic scheme that leaks at most 32 bits through the ORAM timing channel and introduces only 20% performance overhead and 12% power overhead relative to a baseline ORAM that has no timing channel protection. By reducing leakage to 16 bits, our scheme degrades in performance by 5% but gains in power efficiency by 3%. We show that a static (zero leakage) scheme imposes a 34% power overhead for equivalent performance (or a 30% performance overhead for equivalent power) relative to our dynamic scheme.

1 Introduction

As cloud computing becomes increasingly popular, privacy of users' sensitive data is a huge concern in computation outsourcing. In an ideal setting, users would like to "throw their encrypted data over the wall" to a cloud service that can perform arbitrary computation on that data, yet learn no secret information from within that data.

One candidate solution for secure cloud computing is to use tamper-resistant/secure processors. In this setting, the user sends his/her encrypted data to trusted hardware, inside which the data is decrypted and computed upon. After the computation finishes, the final results are encrypted and sent back to the user. Many such hardware platforms have been proposed, including Intel's TXT [11] (which is based on the TPM [34, 1]), eXecute Only Memory (XOM) [18], Aegis [33] and Ascend [7, 39].

While it is assumed that adversaries cannot look *inside* tamper-resistant hardware, secure processors can still leak information through side channels. Preventing information leakage over the memory I/O channel, for example, is a hard problem. Even if all data stored in external memory is encrypted to hide data values, the *memory access pattern* (i.e., read/write/address tuples) can still leak information [41].

Completely preventing *access pattern leakage* requires the use of Oblivious RAM (ORAM). ORAMs were first proposed by Goldreich and Ostrovsky [9], and there has been significant follow-up work that has resulted in more efficient, cryptographically-secure ORAM schemes [24, 23, 5, 4, 10, 16, 37, 29, 32]. Conceptually, ORAM works by maintaining all of memory in encrypted and shuffled form. On each access, memory is read and then reshuffled. Thus, any memory access pattern is computationally indistinguishable from any other access pattern of the same length.

Recently, ORAM has been embraced in secure processor designs [7, 26, 19]. These proposals replace a conventional DRAM controller with a functionally-equivalent ORAM controller that makes ORAM requests on last-level cache (LLC) misses. This direction is promising, and allows large secure computations (whose working sets do not fit in on-chip cache) to be performed on the cloud with reasonable overheads.

1.1 Problems

1.1.1. *When* ORAM is accessed leaks privacy. Consider the malicious program that runs in an ORAM-enabled secure processor in Figure 1 (a). In this example, at every time step t that an ORAM access can be made, the malicious program is able to leak the t^{th} secret bit by coercing a Last Level Cache (LLC) miss if the t^{th} secret bit equals 1. Even when the program is not intentionally malicious, however, the nature of on-chip processor caches make ORAM access rate correlate to access pattern locality. See Figure 1 (b): clearly, when main memory is accessed in large

*Christopher Fletcher was supported by a DoD National Defense Science and Engineering Graduate Fellowship. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract N66001-10-2-4089. The opinions in this paper don't necessarily represent DARPA or official US policy.

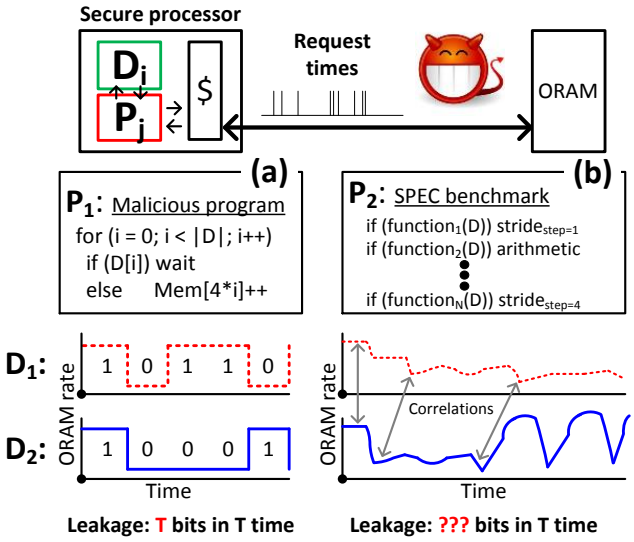


Figure 1: ORAM leakage over the timing channel. D_1 and D_2 are two sets of secret data and equal 10110_2 and 10001_2, respectively. Cache line size is assumed to be 4 words.

programs is program/data-dependent. Abstractly, we can think of a large program as many phases ($\text{function}_1(D)$, $\text{function}_2(D)$, etc.) where the access pattern of each phase is data-dependent to *some* extent. A serious security issue is that we don't know how much leakage is possible with these programs. If we place security as a first-order constraint, we have to assume the worst case (e.g., Figure 1 (a)) is possible.

An important point is that monitoring ORAM rate assumes a similarly-capable adversary relative to prior works, i.e., one who can monitor a secure processor's access pattern can usually monitor its timing. We describe a mechanism to measure a recent ORAM scheme's timing in § 3.2.

1.1.2. Timing protection comes at high overheads. A recent secure processor, called Ascend [7], prevents leakage over the ORAM timing channel by forcing ORAM to be accessed at a single, strictly periodic rate. For Ascend, this rate is chosen offline, i.e., before the program runs. While running, if the program needs ORAM before the next periodic access, the program must wait — hurting performance. If no request is needed when one is forcibly made, an indistinguishable *dummy*[1] access is made instead — wasting energy.

While secure, this approach incurs high overheads. We will show in § 9.3 that forcing a static ORAM rate incurs an over 50% performance/power overhead across a range of SPEC programs. These overheads are *in addition* to the $\sim 2\times$ overhead to use ORAM *without* timing protection [26]. This is not surprising considering that main memory pressure changes dramatically across programs and across inputs to the same program [12]. For example, in Figure 2

[1] A dummy ORAM access is an access made to a fixed program address. By ORAM's security definition, this access looks indistinguishable from a real access.

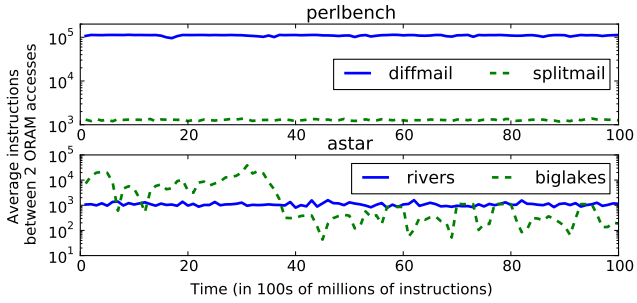


Figure 2: ORAM access rate for perlbench and astar across different inputs, using a 1 MB LLC.

(top) perlbench accesses ORAM 80 times more frequently on one input relative to another. In Figure 2 (bottom), for one input to astar a single rate is sufficient whereas the rate for the second input changes dramatically as the program runs.

2 Our Solution: Leakage Aware Processors

Given that complete timing channel protection is prohibitely expensive (§ 1.1.2) yet no protection has unknown security implications (§ 1.1.1), this paper makes two key contributions. First, we develop an architecture that, instead of blocking information leakage completely, *limits* leakage to a small and *controllable* constant. We denote this constant L (where $L \geq 0$), for *bit leakage* ***L**imit*. Second, we develop a framework whereby increasing/decreasing L, a secure processor can make more/less *user data-dependent* performance/power optimizations. In short, we propose mechanisms that allow a secure processor to trade-off information leakage and program efficiency in a provably secure and disciplined way.

L can be interpreted in several ways based on the literature. One definition that we will use in this paper is akin to deterministic channels [31]: "given an L-bit leakage limit, an adversary with perfect monitoring capabilities can learn no more than L bits of the user's *input data* with probability 1, over the course of the program's execution." Crucially, this definition makes no assumption about *which* program is run on the user's data. It also makes no assumption about *which* L bits in the user's input are leaked.

Importantly, L should be compared to the size of the encrypted program input provided by the user – typically a few Kilobytes – and *not* the size of the secret symmetric key (or session key) used to encrypt the data. As with other secure processor proposals (e.g., Ascend or Aegis [33]), we assume all data that leaves the processor is encrypted with a session key that is *not accessible* to the program (§ 5). This makes the ORAM access rate (for any program and input data) independent of the session key.

To simplify the presentation, we assume a given processor is manufactured with a fixed L. We discuss (along with other bit leakage subtleties) how users can set L per-session in § 10.

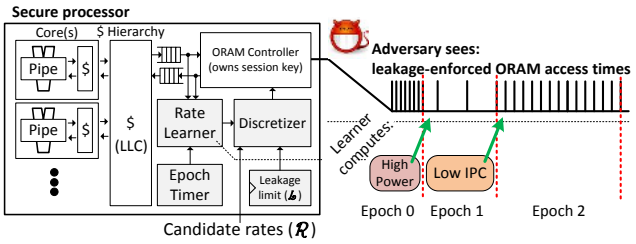


Figure 3: Our proposal: novel hardware mechanisms (shaded grey) to limit and enforce leakage while optimizing for performance/power.

2.1 Calculating & Bounding Bit Leakage

At any time while the program is running, a bound on the number of bits the program has leaked can be calculated by counting the number of possible *observable traces* (referred to as *traces*) the program could have generated up to that point ([31] calls these *equivalence classes*). In our setting, an observable trace is a timing trace that indicates *when* each ORAM access has been made. Using information theory, the worst-case bit leakage is then given by the logarithm base 2 of the number of traces [31, 15]. For the rest of the paper, $\lg$ implies $\log_2$.

Example 2.1. Consider the malicious program (P_1) from Figure 1 (a). We will use the following notation: an ORAM rate of r cycles means the next ORAM access happens r cycles after the last access *completes*. Then, the trace for D_1 can be read as "slow, fast, slow, slow, fast." In general, given T time to run, P_1 can generate 2^T distinct traces of this type (i.e., a distinct trace for each distinct input). Using information theory, the worst-case bit leakage of P_1 after T time is $\lg 2^T = T$ bits. This matches Figure 1 (a), which shows P_1 leaking T bits in T time. On the other hand, accessing ORAM at an offline-selected, periodic rate (§ 1.1.2) yields exactly 1 distinct trace, leaking $\lg 1 = 0$ bits through the ORAM timing channel as expected.

As architects, we can use this leakage measure in several ways. First, we can track the number of traces using hardware mechanisms, and (for example) shut down the chip if leakage exceeds L before the program terminates. Second (our focus in this paper), we can re-engineer the processor such that the leakage approaches L asymptotically over time.

2.2 An Overview of Our Proposal

At the implementation level, our proposal can be viewed as two parts. First, using ideas similar to those presented in [2] and [20], we split program execution into coarse-grain time epochs. Second, we architect the secure processor with learning mechanisms that choose a new ORAM rate out of a set of allowed rates *only at the end of each epoch*. Figure 3 illustrates the idea. We denote the list of epochs, or the *epoch schedule* (§ 6), as $\mathcal{E}$; the list of allowed ORAM rates is denoted $\mathcal{R}$. Each epoch is denoted by its length, in cycles.

2.2.1. Leakage Dependent Factors. Importantly, $|\mathcal{R}|$ and $|\mathcal{E}|$ impact leakage growth. Specifically, if we have $|\mathcal{E}|$ epochs, and allow the secure processor to choose one of the $|\mathcal{R}|$ ORAM rates for each epoch, the number of possible rate combinations is $|\mathcal{R}|^{|\mathcal{E}|}$, resulting in a leakage of $|\mathcal{E}| * \lg |\mathcal{R}|$ bits. For perspective, § 9.3 shows that our dynamic scheme, setting $|\mathcal{R}| = 4$ and $|\mathcal{E}| = 16$, achieves $>$ 30% speedup/power efficiency over static schemes (§ 1.1.2). Plugging in, this configuration can leak $\leq 16 * \lg 4 = 32$ bits over the ORAM timing channel.

2.2.2. Leakage Independent Factors. A key observation is that the values in $\mathcal{R}$, and *which* new rate is chosen at the end of each epoch, *does not* impact leakage growth.

To choose a good ORAM rate in $\mathcal{R}$ for each epoch, we architect a learning module (§ 7) inside the secure processor called a *rate learner* (see Figure 3). Rate learners are circuits that, while the program is running, determine how well (in terms of power/performance) the system would perform if it ran using different ORAM rates. In Figure 3, the rate learner is able to decide that in Epoch 0, the program is accessing ORAM too frequently (i.e., we are wasting energy). Correspondingly, we slow down the rate in Epoch 1.

3 Background: Path ORAM

To provide background we summarize Path ORAM [32], a recent Oblivious-RAM (ORAM) scheme that has been built into secure processors. For more details, see [26, 19]. We will assume Path ORAM for the rest of the paper, but point out that our dynamic scheme can be applied to other ORAM protocols.

Path ORAM consists of an on-chip *ORAM controller* and untrusted external memory (we assume DRAM). The ORAM controller exposes a cache line request/response interface to the processor like an on-chip DRAM controller. Invisible to the rest of the processor, the ORAM controller manages external memory as a binary tree data structure. Each tree node (a bucket) stores up to a fixed number (set at program start time) of blocks and is stored at a fixed location in DRAM. In our setting, each block is a cache line. Each bucket is encrypted with probabilistic encryption[2] and is padded with dummy blocks to the maximum/fixed bucket size.

At any time, each block stored in the ORAM is mapped (at random) to one of the leaves in the ORAM tree. This mapping is maintained in a key-value memory that is internal to the ORAM controller. Path ORAM's invariant is: *If block d is mapped to leaf l, then d is stored on the path from the root of the ORAM tree to leaf l (i.e., in one of a sequence of buckets in external memory).*

3.1 ORAM Accesses

The ORAM controller is invoked on LLC misses and evictions. We describe the operation to service a miss here. On an LLC miss, the ORAM controller reads (+ decrypts)

[2] With probabilistic encryption, the same data encrypted multiple times will yield a completely different looking ciphertext each time.

and writes-back (+ re-encrypts) the path from the root of the ORAM tree to the leaf that the block is mapped to at the beginning of the access. After the path is read, the requested block is forwarded to the LLC and *remapped* to a new random leaf. Block remapping is the critical security step and ensures that future ORAM accesses to the same block occur to randomly/independently chosen paths. Note that with Path ORAM, we can make an indistinguishable dummy ORAM access (§ 1.1.2) by reading/writing a path to a random leaf.

Overheads. [26] demonstrates how the ORAM controller can be built with < 200 KB of on-chip storage. We assume similar parameters (specified in § 9.1.2). To give readers a sense upfront, we note that each ORAM access returns a single cache line, transfers 24.2 KB over the chip pins and has a 1488 cycle latency. This bit movement is due to ORAM reading/writing tree paths on each access.

3.2 Measuring Path ORAM Timing

If the adversary and secure processor share main memory (e.g., a DRAM DIMM), a straightforward way to measure ORAM access frequency is for the adversary to measure its own average DRAM access latency (e.g., use performance counters to measure resource contention [20, 35]).

Even in the absence of data-dependent contention and counters, however, the adversary can accurately determine Path ORAM access frequency by repeatedly performing reads to a single DRAM address. Since all ORAM accesses *write* an ORAM tree path to main memory using *probabilistic encryption* (§ 3), each ORAM access causes bits to flip in main memory. Further, every Path ORAM tree path contains the root bucket and all buckets are stored at fixed locations. Thus, by performing two reads to the root bucket at times t and t' (yielding data d and d'), the adversary learns if ≥ 1 ORAM access has been made by recording whether $d = d'$.

This attack assumes that the secure processor's main memory *can* be remotely read (i.e, through software) by an adversary, which we believe is a realistic assumption. Much focus is given to protecting physical DRAM pages in [11] (i.e., in the presence of DMA-capable devices, GPUs, etc. that share DRAM DIMMS). This indicates that completely isolating DRAM from malicious software is a challenging problem in itself. For example, bugs in these protection mechanisms have resulted in malicious code performing DMAs on privileged memory [38]. Of course, an insider that is in physical proximity can measure access times more precisely using probes.

4 Threat Model

Our goal is to ensure data privacy while a program is running on that data in a server-controlled (i.e., remote) secure processor. The secure processor (hardware) is assumed to be trusted. The server that controls the processor is assumed to be curious and malicious. That is, the server wants to learn as much as possible about the data and will interfere with computation if it can learn more about the user's data by doing so.

4.1 Secure Processor Assumptions

The secure processor runs a potentially malicious/buggy program, provided by the server or the user, on the user's data. The secure processor is allowed to share external resources (e.g., the front-side bus, DRAM DIMMs) with other processors/peripherals. As with prior ORAM work [7, 26, 19], we assume that the secure processor runs a program for one user at a time. Thus adversaries that monitor shared on-chip resources (e.g., pipeline [6], cache [35]) are out of our scope. We give insight as to how to extend our scheme to cache timing attacks in § 10.

ORAM ensures that all data sent on/off chip is automatically encrypted with a symmetric session key. We assume that this key cannot be accessed directly by the program. For timing protection, we additionally require that all encryption routines are fixed latency.

4.2 Monitoring The Secure Processor

The server can monitor the processor's I/O pins, or any external state modified through use of the I/O pins (i.e., using techniques from § 3.2). I/O pins contain information about (a) when the program is loaded onto the processor and eventually terminates, (b) the addresses sent to the main memory and data read from/written to main memory, and (c) *when* each memory access is made. For this paper, we will focus on (a) and (c): we wish to quantify ORAM timing channel leakage and how a program's termination time impacts that leakage. We remark that ORAM, without timing protection, was designed to handle (b).

4.3 Malicious Server Behavior

We allow the server to interact with the secure processor in ways not intended to learn more about the user's data. In particular, we let the server send wrong programs to the secure processor and perform replay attacks (i.e., run programs on the user's data multiple times). Our L-bit leakage scheme (without protection) is susceptible to replay attacks: if the server can learn L bits per program execution, N replays will allow the server to learn $L * N$ bits. We introduce schemes to prevent this in § 8. Finally, we do not add mechanisms to detect when/if an adversary tampers with the contents of the DRAM (e.g., flips bits) that stores the ORAM tree (§ 3). This issue is addressed for Path ORAM in [25].

4.4 Attacks Not Prevented

We only limit leakage over the digital I/O pins and any resulting modified memory. We do not protect against physical/hardware attacks (e.g., fault, invasive, EM, RF). An important difference between these and the ORAM timing channel is that the ORAM channel can be monitored through software, whereas physical and invasive attacks require special equipment. For this same reason, physical/hardware attacks are not covered by Intel TXT [11].

5 User-Server Protocols

We now describe an example protocol for how a user would interact with a server. We refer to the program run on the user's data as P, which can be public or private. Before we begin, we must introduce the notion of a maximum program runtime, denoted T_{max}. T_{max} is needed to calculate leakage only, and should be set such that all programs can run in $< T_{max}$ cycles (e.g., we use $T_{max} = 2^{62}$ cycles at 1 GHz, or ≈ 150 years). The protocol is then given by:

1. The user and secure processor negotiate a symmetric session key K. This can be accomplished using a conventional public-key infrastructure.
2. The user sends $\mathsf{encrypt}_K(D)$ to the server, which is forwarded to the processor. $\mathsf{encrypt}_K(D)$ means "D encrypted under K using symmetric, probabilistic encryption". Finally, the server sends P and leakage parameters (e.g., $\mathcal{R}$; see § 2.2) to the processor.
3. **Program execution (§ 6-7).** The processor decrypts $\mathsf{encrypt}_K(D)$, initializes ORAM with P and D (as in [7, 26]) and runs for up to T_{max} cycles. During this time, the processor can dynamically change the ORAM rate based on $\mathcal{E}$ and $\mathcal{R}$ (§ 2.2).
4. When the program terminates (i.e., before T_{max}), the processor encrypts the final program return value(s) $\mathsf{encrypt}_K(P(D))$ and sends this result back to the user.

6 Epoch Schedules and Leakage Goals

A zero-leakage secure processor architecture must, to fully obfuscate the true termination time of the program, run every program to T_{max} cycles. On the contrary, the protocol in § 5 has the key property that results are sent back to the user *as soon as the program terminates* instead of waiting for T_{max}. We believe this *early termination* property, that a program's observable runtime reflects its actual runtime, is a requirement in any proposal that claims to be efficient and of practical usage.

The negative side effect of early termination is that it can leak bits about the private user input just like the ORAM timing channel. If we consider termination time alone, program execution can yield T_{max} timing traces (i.e., one for each termination time). Further applying the theoretic argument from § 2.1, at most $\lg T_{max}$ bits about the inputs can leak through the termination time per execution.

In practice, due to the logarithmic dependence on T_{max}, termination time leakage is small. As we discussed in § 5, $\lg T_{max} = 62$ should work for all programs, which is very small if the user's input is at least few Kilobytes. Further, we can reduce this leakage through discretizing runtime. (E.g., if we "round up" the termination time to the next 2^{30} cycles, the leakage is reduced to $\lg 2^{62-30} = 32$ bits.)

6.1 $O(\lg T_{max})$ Epochs $\rightarrow O(\lg T_{max})$ Leakage

Since programs leak $\leq \lg T_{max}$ bits through early termination (§ 6), we will restrict our schemes to leak at most that order ($O(\lg T_{max})$) of bits through the ORAM access timing channel. To obtain this leakage, we split program runtime into at most $\lg T_{max}$ epochs.

Recall the setup from § 2.2: First, we denote the list of epoch lengths (the *epoch schedule*) as $\mathcal{E}$ and the set of allowed ORAM access rates at each epoch as $\mathcal{R}$. Second, while running a program on the user's secret data *during a given epoch*, the secure processor is restricted to use a single ORAM access rate. Given T_{max}, $\lg T_{max}$ epochs and $\mathcal{R}$, there are $|\mathcal{R}|^{\lg T_{max}}$ distinct epoch schedules. An upper bound[3] on the number of timing traces (including ORAM timing and early termination) is then given by the number of epoch schedules times the number of termination times — i.e., $|\mathcal{R}|^{\lg T_{max}} * T_{max}$. Thus, bit leakage is $\lg T_{max} * \lg |\mathcal{R}| + \lg T_{max}$ and our $O(\lg T_{max})$ leakage bound holds. We note that in practice, $|\mathcal{R}|$ is a small constant (e.g., $|\mathcal{R}| = 4$ is sufficient; see § 9.5).

6.2 Epoch Doubling

We now discuss how to set the cycle length for each of the (at most) $\lg T_{max}$ epochs. For the rest of the paper, we assume a simple family of epoch schedules where each epoch is $\geq 2\times$ the length of the previous epoch. We refer to the special case where each epoch is twice the length of the previous epoch as *epoch doubling* (whose inspiration came from the slow-doubling scheme in [2]).

Example 6.1. Suppose we run the epoch doubling scheme and set the first epoch's length to 2^{30} ($\approx$ 1 billion) cycles. If $|\mathcal{R}| = 4$ and we run for up to $T_{max} = 2^{62}$ cycles, we expend $\lg 2^{62-30} = 32$ epochs. Thus, the number of possible traces is 4^{32}, resulting in a bit leakage of $\leq \lg 4^{32} = 64$ bits (counting ORAM timing only) and $\leq 64 + \lg T_{max} = 126$ bits with early termination. For perspective, the number of possible traces with no ORAM timing protection is given by $\sum_{t=1}^{T_{max}} \sum_{i=1}^{\lfloor t/\mathsf{OLAT} \rfloor} \binom{t-i(\mathsf{OLAT}-1)}{i}$, where OLAT is the cycle latency per ORAM access.[4] For secure processors, OLAT will be in the thousands of cycles (§ 3.1), making the resulting leakage astronomical.

Clearly, epoch doubling and similar schemes with larger epochs satisfy the $O(\lg T_{max})$ leakage requirement. They also ensure that programs with very different runtimes will still see epoch transitions throughout their execution. This is important for efficiency — if any epoch schedule "runs out of epochs" long before the program terminates, a later phase change in the program may result in a suboptimal and uncorrectable ORAM rate.

For the rest of the paper, we assume the first epoch's length is 2^{30} cycles (as in Example 6.1). The initial epoch should be large enough so that the rate learner has enough

[3] An interesting subtlety is that the exact number of traces depends on the cycle length of each epoch (i.e., the values in $\mathcal{E}$). During the i^{th} epoch, each termination time contributes $|\mathcal{R}|^{i-1}$ traces to the number of possible traces, whereas our bound assumes each termination time contributes $|\mathcal{R}|^{\lg T_{max}}$ traces. We will use this bound to simplify leakage calculations. We note that the choice of rates in $\mathcal{R}$ does not impact leakage.

[4] Conceptually this is, for every termination time t, the number of t-bit bit strings such that any 1 bit must be followed by at least $\mathsf{OLAT} - 1$ repeated 0 bits.

time to determine the next rate (§ 7). A larger initial epoch also means less epochs total, reducing leakage. The initial epoch should be small enough to not dominate total runtime; for the workloads we evaluate in § 9, 2^{30} represents a small fraction of execution time. During the initial epoch, the ORAM rate can be set to any (e.g., a random) value.

7 Rate Learners

We now explain how rate learners select new ORAM rates in $\mathcal{R}$ at the end of each epoch, and how these learners are built into hardware.

7.1 Performance Counters and Rate Prediction

Our rate learner is made up of three components: performance counters, a mechanism that uses the performance counters to predict the next ORAM rate and a discretization circuit that maps the prediction to a value in $\mathcal{R}$.

7.1.1. Performance counters. The performance counters (called AccessCount, ORAMCycles and Waste) are added at the ORAM controller and track LLC-ORAM queue state over time. At each epoch transition, all counters are reset. AccessCount tracks the number of *real* (i.e., not dummy; see § 1.1.2) ORAM requests made during the current epoch. ORAMCycles is the number of cycles each *real* ORAM request is outstanding, summed over all real ORAM accesses. Waste represents the number of cycles that ORAM has real work to do, but is either (a) waiting because of the current rate or (b) performing a dummy access because of the current rate. Conceptually, Waste counts the number of cycles lost due to the current epoch's ORAM rate.

We show what data the performance counters track in Figure 4. Req ❶ illustrates an overset rate, meaning that we are waiting too long to make the next access. Recall our notation from § 2.1: an ORAM rate of r cycles means the next ORAM access happens r cycles after the last access completes. If the rate is overset, Waste can increase per-access by $\leq r$. In our evaluation, ORAM latency is 1488 and rates in $\mathcal{R}$ range from 256 to 32768 cycles (§ 9.2). Thus, oversetting the rate can lead to a much higher performance overhead than ORAM itself.

Req ❷ illustrates an underset rate, meaning that ORAM is being accessed too quickly. When the rate is underset, the processor generates LLC misses when a dummy ORAM request is outstanding (forcing the processor to wait until that dummy access completes to serve the miss). This case is a problem for memory bound workloads where performance is most sensitive to the rate (§ 9.2).

Req ❸ illustrates how multiple outstanding LLC misses are accounted for. In that case, a system without timing protection should perform ORAM accesses back to back until all requests are serviced. To model this behavior, we add the rate's cycle value to Waste.

7.1.2. Rate prediction. At each epoch transition, a dedicated hardware block computes the following averaging

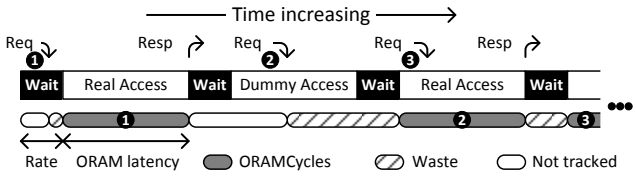


Figure 4: Example timing trace that illustrates what the performance counters (§ 7.1.1) track. Req/Resp stand for request/response.

statistic based on the performance counters:

$$\mathsf{NewIntRaw} = \frac{\mathsf{EpochCycles} - \mathsf{Waste} - \mathsf{ORAMCycles}}{\mathsf{AccessCount}} \tag{1}$$

where EpochCycles denotes the number of cycles in the last epoch. Conceptually, NewIntRaw represents the offered load rate on the ORAM. Note that since ORAMCycles is the *sum* of access latencies, this algorithm does not assume that ORAM has a fixed access latency.

7.1.3. Rate discretization. Once the rate predictor calculates NewIntRaw, that value is mapped to whichever element in $\mathcal{R}$ is closest: i.e., $\mathsf{NewInt} = \arg\min_{r \in \mathcal{R}}(|\mathsf{NewIntRaw} - r|)$. As we show in § 9.5, $|\mathcal{R}|$ can be small (4 to 16); therefore this operation can be implemented as a sequential loop in hardware.

7.2 Hardware Cost and Optimizations

As described, the learner costs an adder and a divider. Since this operation occurs only once per epoch (where epochs are typically billions of cycles each, see § 6.2), it is reasonable to use a processor's divide unit to implement the division operation. To make the rate matcher self-contained, however, we round AccessCount up to the next power of two (including the case when AccessCount is already a power of 2) and implement the division operation using 1-bit shift registers (see Algorithm 1). This simplification may underset the rate by as much as a factor of two (due to rounding), which we discuss further in § 7.3.

Algorithm 1 Rate predictor hardware implementation.

```
NewIntRaw = EpochCycles − Waste − ORAMCycles
while AccessCount > 0 do
  NewIntRaw = NewIntRaw ≫ 1 {Right shift by 1}
  AccessCount = AccessCount ≫ 1
end while
```

In the worst case, this operation may take as many cycles as the bitwidth of AccessCount — which we can tolerate by starting the epoch update operation at least that many cycles before the epoch transition.

7.3 Limitations of Prediction Algorithm

Our rate learner's key benefit is its simplicity and its self-containment (i.e., it only listens to the LLC-ORAM controller queue and computes its result internally). That said, the predictor (Equation 1) has two limitations. First, it is

oblivious to access rate variance (e.g., it may overset the rate for programs with bursty behavior). The shifter implementation in § 7.2 helps to compensate for this effect. Second, it is oblivious to how program performance is impacted by ORAM rate.

We experimented with a more sophisticated predictor that simultaneously predicts an upper bound on performance overhead for each candidate rate in $\mathcal{R}$ and sets the rate to the point where performance overhead increases "sharply." What constitutes "sharply" is controlled by a parameter, which gives a way to trade-off performance/power (e.g., if the performance loss of a slower rate is small, we should choose the slower rate to save power).

As we have mentioned, however, an important result in § 9.5 is that $|\mathcal{R}|$ can be small (recall: $|\mathcal{R}| = 4$ is sufficient). This makes choosing rates a course-grain enough operation that the simpler predictor (§ 7.1) chooses similar rates as the more sophisticated predictor. We therefore omit the more sophisticated algorithm for space.

8 Preventing Replay Attacks

Clearly, the set of timing traces (denoted $\mathcal{T}$) is a function of the program P, the user's data D, and the leakage parameters (e.g., $\mathcal{E}$ and $\mathcal{R}$). If the server is able run multiple programs, data or epoch parameters, it may be able to create $\mathcal{T}_1$, $\mathcal{T}_2$, etc (i.e., one set of traces per experiment) such that $\log \prod_i |\mathcal{T}_i| > L$ — breaking security (§ 2.1).

One way to prevent these attacks is to ensure that once the user submits his/her data, it can only be 'run once.' This can be done if the secure processor "forgets" the session key K after the user terminates the session. In that case, Step 1 in the user-server protocol (§ 5) expands to the following:

1. The user generates a random symetric key, call this K', encrypts K' with the processor's public key, and sends the resulting ciphtertext of K' to the processor.
2. The processor decrypts K' using its secret key, generates a random symetric key K (where $|K| = |K'|$) and sends $\mathsf{encrypt}_{K'}(K)$ back to the user. The processor stores K in a dedicated on-chip register.

The user can now continue the protocol described in § 5 using K. When the user terminates the session, the processor resets the register containing K.

The key point here is that once the user terminates the session, K is forgotten and $\mathsf{encrypt}_K(D)$ becomes computationally un-decryptable by any party except for the user. Thus, $\mathsf{encrypt}_K(D)$ cannot be replayed using a new program/epoch schedule/etc. The downside is a restriction to the usage model – the user's computation can only proceed on a single processor per session.

8.1 Broken Replay Attack Prevention Schemes

Preventing replay attacks must be done carefully, and we now discuss a subtly broken scheme. A common mechanism to prevent replay attacks is to make the execution environment and its inputs fixed and deterministic. That is, the user can use an HMAC to bind (the hash of a fixed program P, input data D, $\mathcal{E}$, $\mathcal{R}$) together. If the server runs that tuple multiple times (with the corresponding program P) in a system with a fixed starting state (e.g., using [11]), the program will terminate in the same amount of time and the rate learners (§ 7) will choose the same rates each time. Thus, the observable timing trace should not change from run to run, which (in theory) defeats the replay attack.

This type of scheme is insecure because of non-deterministic timing on the main memory bus (e.g., FSB) and DRAM DIMM. Different factors—from bus contention with other honest parties to an adversary performing a denial of service attack—will cause main memory latency to vary. Depending on main memory timing, the secure processor will behave differently, causing IPC/power to vary, which causes the rate learner to [potentially] choose different rates. Thus, the tuple described above (even with a deterministic architecture) does not yield deterministic timing traces and the replay attack succeeds. This problem is exacerbated as the secure processor microarchitecture becomes more advanced. For example, depending on variations in main memory latency, an out-of-order pipeline may be able to launch none or many non-blocking requests.

9 Evaluation

We now evaluate our proposal's efficiency overheads and information leakage.

9.1 Methodology

9.1.1. Simulator and benchmarks. We model secure processors with a cycle-level simulator based on the public domain SESC [27] simulator that uses the MIPS ISA. We evaluate a range (from memory-bound to compute-bound) of SPEC-int benchmarks running reference inputs. Each benchmark is fast-forwarded 1-20 billion instructions to get out of initialization code and then run for an additional 200-250 billion instructions. Our goal is to show that even as epochs occur at sparser intervals (§ 6.2), our efficiency improvements still hold (§ 9.4).

9.1.2. Timing model. All experiments assume the microarchitecture and parameters given in Table 1. We also experimented with 512 KB - 4 MB LLC capacities (as this impacts ORAM pressure). Each size made our dynamic scheme impact a different set of benchmarks (e.g., `omnetpp` utilized more ORAM rates with a 4 MB LLC but `h264ref` utilized more with a 1 MB LLC). We show the 1 MB result only as it was representative. We note that despite the simple core model in Table 1, our simulator models a non-blocking write buffer which can generate multiple, concurrent outstanding LLC misses (like Req ❸ in § 7.1.1).

In the table, 'DRAM cycle' corresponds to the SDR frequency needed to rate match DRAM (i.e., $2 * 667$ MHz $=$ 1.334 GHz). We model main memory latency for insecure systems (`base_dram` in § 9.1.6) with a flat 40 cycles. For ORAM configurations, we assume a 4 GB capacity Path ORAM (§ 3) with a 1 GB working set. Additional ORAM parameters (using notation from [26]) are 3 levels of recursion, $Z = 3$ for all ORAMs, and 32 Byte blocks for recursive ORAMs. As in [26], we simulate our ORAM on

Table 1: Timing model; processor clock = 1 GHz.

Core	
Core model	in-order, single-issue
Pipeline stages per Arith/Mult/Div instr	1/4/12
Pipeline stages per FP Arith/Mult/Div instr	2/4/10
Fetch Buffer	256 B, 1-way
[Non-blocking] write buffer	8 entries
On-Chip Memory	
L1 I/D Cache	32 KB, 4-way
L1 I/D Cache hit+miss latencies	1+0/2+1
L1 eviction buffers	8 entries
L1 I/D, L2 cache output bus width	256/64/256 bits
Unified/Inclusive L2 (LLC) Cache	1 MB, 16-way
L2 hit+miss latencies	10+4
Cache/ORAM block size	64 Bytes
Memory system	
DRAM frequency/channels	667 MHz (DDR)/2
Off-chip pin bandwidth	16 Bytes/DRAM cycle

Table 2: Processor energy model; technology = 45 nm.

Component	Energy (nJ)	Source
Dynamic energy		
ALU/FPU (per instruction)	.0148	[8] (FM add)
Reg File Int/FP (per instruction)	.0032/.0048	[8]
Fetch buffer (256 bits)	.0003	[30] (CACTI)
L1 I Cache hit/refill (1 cache line)	.162	" "
L1 D Cache hit (64 bits)	.041	" "
L1 D Cache refill (1 cache line)	.320	" "
L2 Cache hit/refill (1 cache line)	.810	[17]
DRAM Controller (1 cache line)	.303	§ 9.1.3
Parasitic leakage		
L1 I Cache (per cycle)	.018	[30]
L1 D Cache (per cycle)	.019	[30]
L2 Cache (per hit/refill)	.767	[17]
On-chip ORAM Controller		
AES (per 16 Byte chunk, 170 Gbps)	.416	[21]
Stash (per 16 Byte rd/wr)	.134	[30]
Total (1 cache line)	984	§ 9.1.4

DDR3 SDRAM using DRAMSim2 [28]. Our ORAM parameters, coupled with our CPU/DRAM clock and DRAM channel count (Table 1), gives us an ORAM latency of 1488 cycles per cache line. Further, each ORAM access transfers 24.2 KB (12.1 KB for each of the path read/write) over the chip pins.

9.1.3. Power model. To estimate the relationship between the energy of on-chip components (e.g., pipeline, cache) and ORAM, we account for energy-per-component in Table 2. We account for energy from the pipeline to the on-chip DRAM/ORAM controller and do not model external DRAM power consumption. To calculate Power (in Watts): we count all accesses made to each component, multiply each count with its energy coefficient, sum all products and divide by cycle count.

We account for dynamic power only except for parasitic leakage in the L1/L2 caches (which we believe will dominate other sources of parasitic leakage). To measure DRAM controller energy, we use the peak power reported in [3] to calculate energy-per-cycle (.076 nJ). We then multiply this energy-per-cycle by the number of DRAM cycles that it takes to transfer a cache line's worth of 16 Byte chunks (our pin bandwidth; see Table 1) over the chip pins.

9.1.4. Path ORAM power consumption. During each Path ORAM access (§ 3.1), we count the energy consumption of the ORAM controller and on-chip DRAM controller (that would serve as a backend for the ORAM controller). For every 16 Bytes (the AES-128 chunk size) read along the accessed path, the ORAM controller performs AES decryption and writes the plaintext data to an SRAM internal to the ORAM controller ([26] calls this memory the *stash*). For every 16 Bytes written, the operation is reversed: the stash is read and the chunk is re-encrypted.

See Table 2 for AES/stash energy coefficients. AES energy is taken from [21], scaled down to our clock frequency and up to a 1 AES block/DRAM cycle throughput. Stash read/write energy is approximated as the energy to read/write a 128 KB SRAM modeled with CACTI [30]. We assume the ORAM's DRAM controller constantly consumes the peak power from [3] during the entire ORAM access (1488 processor cycles, or 1984 DRAM cycles).

Thus, the energy-per-ORAM-access is given as the chunk count $\times$ (AES energy + Stash energy) + Cycle latency $\times$ DRAM controller cycle energy. Each ORAM access moves 24.2 KB of data (§ 9.1.2) which is $2 * 758$ 16-Byte chunks. Given the power coefficients from Table 2, energy-per-access is then $2 * 758 * (.416 + .134) + 1984 * .076 \approx 984$ nJ.

9.1.5. Baseline information leakage. To calculate leakage, we fix $T_{max} = 2^{62}$. Thus, the *baseline leakage* through the early termination channel (without ORAM) is 62 bits and we will compare our scheme's additional leakage to this number. Of course, the SPEC programs run for a significantly smaller time and leak fewer bits as a result.

9.1.6. Baseline architectures. We compare our proposal to five baselines:

1. **base_dram**: All performance overheads are relative to a baseline insecure (i.e., no security) DRAM-based system. We note that a typical SPEC benchmark running base_dram with our timing/power model (§ 9.1.2-9.1.3) has an IPC between 0.15-0.36 and a power consumption between 0.055-0.086 Watts.
2. **base_oram**: A Path ORAM-based system without timing channel protection (e.g., [26]). This can be viewed as a power/performance oracle relative to our proposal and is insecure over the timing channel.
3. **static_300**: A Path ORAM-based system that uses a single static rate for *all* benchmarks. This follows [7] and can be viewed as a secure (zero leakage over the ORAM timing channel) but strawman design. We swept a range of rates and found that the 300 cycle rate minimized average performance overhead relative to base_dram. This point demonstrates the performance limit for static schemes and the power overhead needed to attain that limit.
4. **static_500** and **static_1300**: To give more insight, we also compare against static rate schemes that use 500

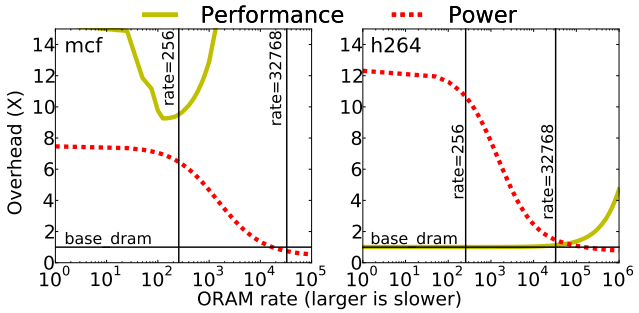


Figure 5: The relationship between power and performance overhead for a memory bound (mcf) and compute bound (h264ref) benchmark.

and 1300 cycle ORAM rates. static_500/static_1300 has roughly the same performance/power (respectively) as the dynamic configuration we evaluate in § 9.3. Thus, static_500 conveys the *power overhead* needed for a static scheme to match our scheme in terms of performance (and vice versa for static_1300).

All static_ schemes assume no protection on the early termination channel and therefore leak ≤ 62 bits (§ 9.1.5).

9.2 Choosing the Spread of Values in $\mathcal{R}$

To select extreme values in $\mathcal{R}$, we examine a memory (mcf) and compute bound (h264ref) workload (Figure 5). In the figure, we sweep static rates and report power/performance overhead relative to base_dram for each point. For our timing/energy model, rates below 200 cycles lead to unstable performance for mcf as the rate becomes underset on average (§ 7.1.1). On the other hand, rates much larger than 30000 cycles cause h264ref's power to drop below that of base_dram, telling us that the processor is waiting for ORAM (idling) a majority of the time. Thus, for our experiments we will use a 256/32768 cycle lower/upper bound for rates.

Once the high and low rates are chosen, we select ORAM rate candidates in between that are spaced out evenly on a lg scale. For example if $|\mathcal{R}| = 4$, $\mathcal{R} = \{256, 1290, 6501, 32768\}$. Intuitively, the lg scale gives memory-bound (ORAM-sensitive) workloads more rates to choose from; whereas 32768 is a suitable rate for all compute bound workloads. During the first epoch, we set the rate to 10000 for all benchmarks (§ 6.2).

9.3 Comparison to Baselines

Our main result in Figure 6 shows the performance/power benefits that dynamically adjusting ORAM access rate can achieve. Performance results are normalized to base_dram. Power results are actual values in Watts. The white-dashed bars at the bottom represent power consumption from non-main-memory components, which is similar across different systems because instructions-per-experiment is fixed; the colored bars indicate the power consumption of the DRAM and ORAM controllers.

base_oram has the lowest overhead—3.35× performance and 5.27× power relative to base_dram—but can leak an unbounded amount of information through the timing channel (§ 1.1). Our dynamic scheme is given by dynamic_R4_E4, where R4 means $|\mathcal{R}| = 4$ and E4 means epoch i is $4\times$ as long as epoch $i-1$. This configuration represents a high performance, low leakage point in § 9.5. Given our choice of T_{max}, dynamic_R4_E4 expends 16 epochs, giving us a leakage of $16 * \lg|\mathcal{R}| = 32$ bits. This dynamic configuration has a performance/power overhead of 4.03×/5.89×. Compared with base_oram, this is 20% performance overhead and 12% power overhead.

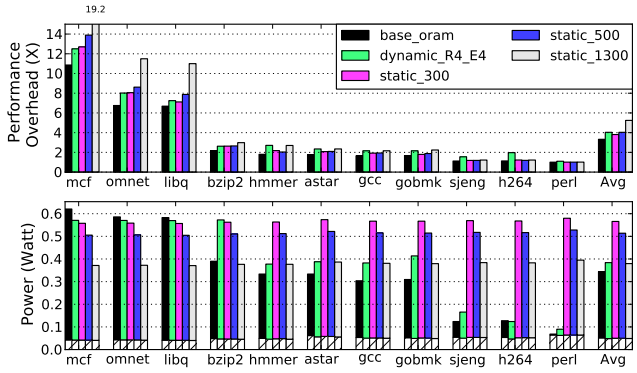


Figure 6: Performance overhead and power breakdown (§ 9.3) of baseline ORAM, global average access rates and our dynamic scheme.

static_300 incurs 3.80×/8.68× performance/power overhead. This is 6% better performance and 47% higher power consumption relative to dynamic_R4_E4. Also compared to the dynamic scheme, static_500 incurs a 34% power overhead (breaking even in performance) while static_1300 incurs a 30% performance overhead (breaking even in power). Thus, through increasing leakage by ≤ 32 bits (giving a total leakage of $62 + 32 = 94$ bits; § 9.1.5) our scheme can achieve 30% / 34% performance/power improvement depending on optimization criteria.

9.4 Stability

Figure 7 shows that our dynamic scheme has stable performance as epoch length increases. The figure compares the IPC of dynamic_R4_E2 (our dynamic scheme with epoch doubling), base_oram and static_1300 over time in 1-billion instruction windows. We discuss three representative benchmarks: libquantum, gobmk and h264ref. To complete 200 billion instructions, these benchmarks ran for $1 \sim 5$ trillion cycles and completed 9-11 epochs. libquantum is memory bound and our scheme consistently incurs only 8% performance overhead relative to base_oram. gobmk has erratic-looking behavior but consistently selects the same rate after epoch 6 (marked **e6**). After epoch 6, our dynamic scheme selects the 1290 cycle rate (see § 9.2 for rate candidates) which is why its performance is similar to that of static_1300. We found that astar and gcc behaved similarly.

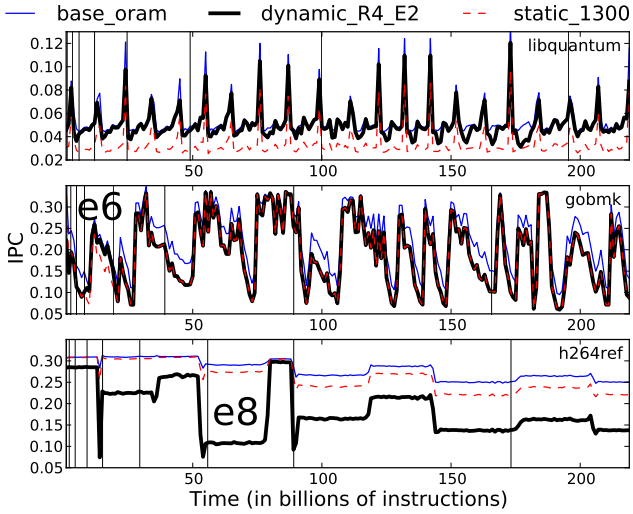


Figure 7: IPC in 1-billion instruction windows over time on selected benchmarks (§ 9.4). Vertical black lines mark epoch transitions for dynamic_R4_E2.

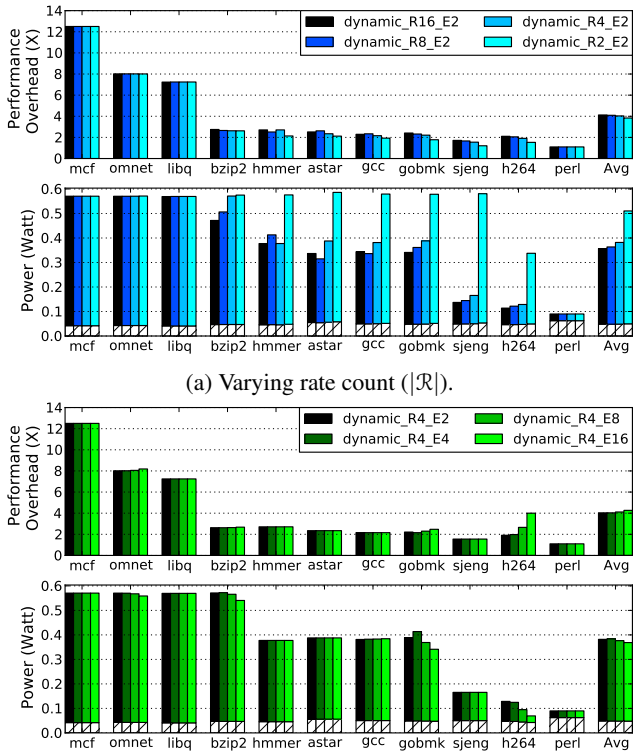
h264ref is initially compute bound (choosing the 32768 cycle rate) but becomes memory bound in epoch 8 (marked **e8**). At the next epoch transition, the rate learner switches to the 6501 cycle rate. To give insight into trade-offs, we ran h264ref fixing the rate to different values in $\mathcal{R}$ after epoch 8. If the rate learner had not switched to the 6501 cycle rate, h264ref would have incurred a 2.88× performance hit but only decreased power by 2.38×. On the other hand, selecting the next fastest rate (1290 cycles) would have improved performance by 52% (relative to the 6501 rate), but increased power consumption by 2.36×. Thus, we believe the 6501 cycle rate to be a reasonable trade-off.

9.5 Reducing the Leakage Bound

We can control leakage by changing the number of candidate access rates $|\mathcal{R}|$ and the epoch frequency $|\mathcal{E}|$.

Figure 8a shows the impact of changing $|\mathcal{R}|$, given the lg spread of rates (§ 9.2) and the epoch doubling scheme. Varying $|\mathcal{R}| = 16$ to $|\mathcal{R}| = 4$, performance improves by 2% and power consumption increases by 7% — but leakage drops by 2× bits. Small $|\mathcal{R}|$ improve performance more for benchmarks that are neither compute nor memory bound (e.g., gobmk, gcc). This makes LLC misses less regular, which the rate learner's averaging mechanism does not account for (§ 7.3). Note that when $|\mathcal{R}| = 2$, power increases significantly for these same benchmarks. In that case, $\mathcal{R} = \{256, 32768\}$ — neither of which match well to non-extreme workloads.

Using $|\mathcal{R}| = 4$, we next study the effect of less frequent epochs in Figure 8b. Over 200 billion instructions, dynamic_R4_E2 expends 8-12 epochs and dynamic_R4_E16 expends 2-3 epochs. Most benchmarks do not lose performance with less epochs. h264ref is the largest exception, which chooses a slow rate before the memory bound region (see § 9.4) and is stuck with that rate for a longer time. It is



(a) Varying rate count ($|\mathcal{R}|$).

(b) Varying epoch count ($|\mathcal{E}|$).

Figure 8: Leakage reduction study (§ 9.5).

possible that running the workload longer will fix this problem; e.g., we saw the same behavior with gobmk but the 200 billion instruction runtime was enough to smooth out the badly performing epoch. Despite this, dynamic_R4_E16 (8 epochs in $T_{max} = 2^{62}$ cycles) reduces ORAM timing leakage to 16 bits (from 32 bits) and only increases average performance overhead by 5% (while simultaneously decreasing power by 3%) relative to dynamic_R4_E4.

10 Discussion

Letting the user choose L (the bit leakage limit, see § 2). So far, we have assumed that L is fixed at manufacturing time for simplicity. To specify L per session, the user can send L (bound to the user's data using a conventional HMAC) to the processor during the user-server protocol (§ 5). When the server forwards leakage parameters (e.g., $\mathcal{R}$, $\mathcal{E}$) to the processor, the processor can decide whether to run the program by computing possible leakage as in § 6.1.

Supporting additional leakage channels. A key implication of § 6.1 is that *bit leakage across different channels is additive*, making our proposal scalable to additional leakage channels. Suppose we wish to protect N channels (early program termination and ORAM timing being two). If channel i can generate traces $\mathcal{T}_i$ in isolation, then the whole processor can generate $\prod_{i=1}^{N} |\mathcal{T}_i|$ trace combinations, resulting in a leakage of $\sum_{i=1}^{N} \lg |\mathcal{T}_i|$ bits. This property allows our work to be extended to others that calculate bit leakage—e.g., [40] which studies interactive programs

and [14] which studies cache timing attacks.

Can our scheme work without ORAM? Our scheme can be used without ORAM if dummy memory operations are indistinguishable from real memory operations.[5] If this property holds, the adversary's view of the memory timing channel is a single periodic rate per epoch and the number of measurable timing traces is as we calculated in § 2.2. With ORAM, this property holds by definition (§ 3). With commodity DRAM, it may be possible for this property to hold by adding additional mechanisms. For example, we must prevent an adversary from using DRAM state to tell dummy from real operations—e.g., by disabling DRAM row buffers or placing them in a public state after each access. We must also ensure that the adversary cannot scan DRAM to determine access frequency (§ 3.2)—e.g., by physically partitioning DRAM between devices.

Bit leakage interpretation subtleties. As stated in § 2, the adversary can learn any L bits of the user's data. This is an issue if (a) some of the user's bits are more "important" than others (e.g., if the user's data *itself* contains a cryptographic key) and (b) the adversary can run any program of its choosing on the user's data. For example, in Figure 1 (a) the program writer chooses what bits to leak. The user can mitigate this issue somewhat by binding a certified program hash to its data with an HMAC, which restricts the processor to run that program only. This strategy assumes (a) that the program *can* be re-written to not reveal important bits, (b) that ORAM is integrity verified [25] and (c) that the adversary cannot arbitrarily influence the well-written program by introducing traffic to main memory (as in § 8.1).

A second subtlety is that bit leakage can be probabilistic [31]. That is, the adversary may learn $> L$ bits of the user's data with some probability < 1. Suppose a program can generate 2 timing traces. Our leakage premise from § 2.1 says we would leak $\leq \lg 2 = 1$ bit. The adversary may learn L' bits (where $L' > L$) per trace with the following encoding scheme: if L' bits of the user's data matches a complete, concrete bit assignment (e.g., if $L' = 3$ one assignment is 001_2) choose trace 1; otherwise choose trace 2. If the user's data is uniformly distributed bits, the adversary learns all L' bits with probability $\frac{2^L - 1}{2^{L'}}$.

11 Related Work

11.1 Foundational Work

This paper builds on recent work on Path ORAM and information-theoretic approaches to bounding leakage.

Path ORAM's theoretic treatment is given in [32]. Path ORAM has been studied in secure processor settings using software simulation [26] and FPGA implementation [19]. Path ORAM integrity protection mechanisms are covered in [25]. *None* of above works protect the ORAM timing channel. To our knowledge, the only work to protect against the ORAM timing channel is [7], which imposes a strict, periodic rate that we evaluate against (§ 9).

The most relevant information theoretic work is Predictive Mitigation [2] and leakage bounding techniques for on-chip caches [14]. We use ideas similar to Predictive Mitigation to break programs into epochs (§ 6.1), although our setting is somewhat different since [2] does not permit dummy accesses to fool an adversary. [14] applies the same information-theoretic framework to bound leakage in on-chip caches. The key difference to our work is that [14] focuses on *quantifying* the leakage of different schemes. Our focus is to develop hardware mechanisms to bound leakage and trade-off that leakage to get efficiency.

More generally, timing channel attacks and related protections have been a hot topic since it was discovered that RSA and other crypto-algorithms could be broken through them [13]. We cannot list all relevant articles, but two related papers are Time Warp [20] and Wang et al. [35]. Time Warp also uses epochs to fuzz architectural mechanisms (e.g., the RDTSC instruction) and, using statistical arguments, decrease timing leakage. Wang et al. propose novel cache mechanisms to defeat shared-cache timing attacks.

11.2 Secure processors

The eXecute Only Memory (XOM) architecture [18] mitigates both software and certain physical attacks by requiring applications to run in secure compartments controlled by the program. XOM must be augmented to protect against replay attacks on memory. Aegis [33], a single-chip secure processor, provides integrity verification and encryption on-chip so as to allow external memory to be untrusted. Aegis therefore is protected against replay attacks.

A commercialized security device is the TPM [34] — a small chip soldered onto a motherboard and capable of performing a limited set of secure operations. One representative project that builds on the TPM is Flicker [22], which describes how to leverage both AMD/Intel TPM technology to launch a user program while trusting only a very small amount of code (as opposed to a whole VMM).

The primary difference between our setting and these works is the threat model: none of them require main memory address or timing protection. Address leakage is a widely acknowledged problem (outside of ORAM, [41] shows how program control flow can be determined through memory access pattern). Although main memory timing leakage has not been addressed, a lesson from prior work (§ 11.1) is that when there is a timing channel, attackers will try to exploit it.

11.3 Systems that enforce non-interference

There is a large body of work that is built around systems that *provably enforce non-interference* between programs ([36] is a represenative paper). Non-interference is the guarantee that two programs can coexist, yet any actions taken by one program will be invisible (over the timing channel in particular) to the other program. In our setting, non-interference is akin to a single, strict rate that per-

[5] For perspective, in Figure 6 an average of 34% of ORAM accesses made by our dynamic scheme are dummy accesses.

mits no leakage [31]. We believe that our proposal, which permits *some* interference, may be applicable and useful to these works.

12 Conclusion

We propose mechanisms that provably guarantee a small upper-bound on timing channel leakage and achieves reasonable performance overheads relative to a baseline ORAM (with no timing channel protection). Our schemes are significantly more efficient than prior art which was restricted to choosing a static rate of accessing memory.

References

[1] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *S&P*, 1997.

[2] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *CCS*, 2010.

[3] M. N. Bojnordi and E. Ipek. Pardis: a programmable memory controller for the ddrx interfacing standards. In *ISCA*, 2012.

[4] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious RAM practical. MIT-CSAIL-TR-2011-018, 2011.

[5] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[6] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *ISCA*, 2012.

[7] C. Fletcher, M. van Dijk, and S. Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *STC*, 2012; an extended version is located at http://csg.csail.mit.edu/pubs/memos/Memo-508/memo508.pdf (Master's thesis).

[8] S. Galal and M. Horowitz. Energy-efficient floating-point unit design. *IEEE Transactions on Computers*, 60:913–922, 2011.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. In *J. ACM*, 1996.

[10] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[11] D. Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[12] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

[13] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, 1996.

[14] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *CAV*, 2012.

[15] B. Köpf and G. Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *CSF*, 2010.

[16] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[17] S. Li, K. Chen, J.-H. Ahn, J. Brockman, and N. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *ICCAD*, 2011.

[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 *ASPLOS*, 2000.

[19] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. CCS, 2013.

[20] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *SIGARCH Comput. Archit. News*, 40(3):118–129, June 2012.

[21] S. Mathew, F. Sheikh, A. Agarwal, M. Kounavis, S. Hsu, H. Kaul, M. Anders, and R. Krishnamurthy. 53gbps native gf(24)2 composite-field aes-encrypt/decrypt accelerator for content-protection in 45nm high-performance microprocessors. In *VLSIC*, 2010.

[22] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*

[23] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.

[24] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, 1997.

[25] L. Ren, C. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *HPEC*, 2013.

[26] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *ISCA*, 2013.

[27] J. Renau. Sesc: Superescalar simulator. Technical report, UIUC ECE department, 2002.

[28] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, jan.-june 2011.

[29] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt*, 2011.

[30] P. Shivakumar and N. J. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, Feb. 2001.

[31] G. Smith. On the foundations of quantitative information flow. In *FOSSACS*, 2009.

[32] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *CCS*, 2013.

[33] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ICS*, 2003.

[34] Trusted Computing Group. TCG Specification Architecture Overview Revision 1.2, 2004.

[35] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, 2007.

[36] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood. Surfnoc: a low latency and provably non-interfering approach to secure networks-on-chip. In *ISCA*, 2013.

[37] P. Williams and R. Sion. Round-optimal access privacy on outsourced storage. In *CCS*, 2012.

[38] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Another way to circumvent intel trusted execution technology: Tricking senter into misconfiguring vt-d via sinit bug exploitation.

[39] X. Yu, C. Fletcher, L. Ren, M. van Dijk, and S. Devadas. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *CCSW*, 2013.

[40] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *CCS*, 2011.

[41] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS*, 2004.