Ascend: An Architecture for Performing Secure Computation on Encrypted Data

by

Christopher W. Fletcher

B.S. in Electrical Engineering and Computer Science, University of California, Berkeley, 2010

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2013

Author . . . .
Department of Electrical Engineering and Computer Science
May 4, 2013

Certified by . . . .
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Ascend: An Architecture for Performing Secure Computation on Encrypted Data

by

Christopher W. Fletcher

Submitted to the Department of Electrical Engineering and Computer Science on May 4, 2013, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis considers encrypted computation where the user specifies encrypted inputs to an *untrusted* batch program controlled by an untrusted server. In batch computation, all data that the program might need is known at program start time. Encrypted computation on untrusted batch programs can be realized through fully homomorphic encryption (FHE) techniques, but FHE's current overheads limit its applicability. Secure processors (e.g., Aegis), coprocessors (e.g., TPM) or hardware extensions (e.g., TXT) typically require trust in the entire processor, the host operating system and the *program* that computes on the inputs. In this thesis, we design a secure processor architecture, called *Ascend*, that guarantees privacy of data given *untrusted* batch programs.

The key idea in Ascend to guarantee privacy is *parameterizable, obfuscated program execution*. From the perspective of the Ascend chip's input/output and power pins, an untrusted server cannot learn anything about private user data regardless of the program run. Ascend uses Oblivious RAM (ORAM) techniques to hide memory access patterns and differential-power analysis (DPA) resistance techniques to hide data-dependent power draw. For each of the input/output and power channels, an Ascend chip exposes a set of public knobs that *fully specify the observable behavior of the chip given any batch program and any input to that batch program.* These knobs (e.g., specifying strict intervals for when external memory should be accessed) are controlled by the server and can be tuned, based on the server's apriori knowledge of the program, to trade-off performance and power without impacting security.

Experimental results when running Ascend on SPEC benchmarks show an average $3.6\times/6.6\times$ and $5.2\times/4.7\times$ performance/power overhead—when hiding memory access pattern and power draw—using two schemes that capture the server's apriori knowledge in different ways. Furthermore—when hiding memory access pattern only—performance/power overheads drop to only $2.6\times/2.2\times$. These surprising results mean that it is viable to only trust hardware and not software in some security-conscious applications.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

For the rest of the thesis, I, Chris Fletcher, will refer to the author as *we*. This is because Ascend has been a team effort from start to present, and has only been possible through my efforts, combined with those of Srini Devadas, Marten van Dijk, Ling Ren and Xiangyao Yu. I would first like to thank Srini Devadas and Marten van Dijk. Srini, your incredible drive and boundless support has made this last year the best of my academic life so far. Both directly and indirectly, the only reason I am here right now is because you put up with all of those fizzled projects in my first year. For giving me the freedom to try things that would never work, I thank you. Marten, I feel truly lucky to be working with you. This project has been an eye opener for me, especially since I came in with no security background. Thank you for all you have taught me; on the flip side, I hope I have been able to teach some systems design! To Ling and Xiangyao, you guys have been excellent this past term. I can tell how hard you have been working, and it showed with the ISCA submission this last November.

I would also like to thank my peers and members of CSAIL. First, thanks to Rachael Harding and Omer Khan for many very helpful discussions not only computer systems design but also security. Coming out of a security meeting to talk systems design, it is always great to pitch an idea to you and think *fantastic, they understand exactly where I am coming from!* Second to other members of CSAIL: Charles O'Donnell, Keun sup Shim, Ilia Lebedev, Michel Kinsy, Mieszko Lis, Nirav Dave, Raluca Ada Popa, and Alessandro Chiesa. To the entire Hornet group, for putting up with security talks and offering feedback from the perspectives of no less than three areas. I would also like to thank Sanjit Seshia, Omer Khan, and many others for giving me early feedback on the thesis itself.

I feel very privileged to be and have been mentored by some amazing people outside of CSAIL. In particular, I would like to thank Joe Pasquale, John Wawrzynek, Garry Nolan and Greg Gibeling. Without any one of you, I don't know where I would be right now but it certainly wouldn't be here.

Finally, I would like to thank my girlfriend Cassie, my brother and my parents. Cassie, you have been my rock these past two years; I cannot begin to tell you how important your support has been. Nor could I ask for a better girlfriend. To my brother, Sam: you are one of my best friends, and probably the person who knows me the best. Once I turn in this thesis, we're going to Vegas. To Mom and Dad: simply put, I couldn't have had better parents. I consider myself blessed. Without your support and enthusiasm, not only would I not have gotten this far, it would not have been nearly as meaningful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Privacy of data is a huge problem in cloud computing, and more generally in outsourcing computation. From financial information to medical records, sensitive user data is being sent to and computed upon by the cloud. Computation on sensitive data requires that the data be exposed to cloud servers, which may be attacked by malicious applications, hypervisors, operating systems, or by insiders.

This thesis addresses how a computationally-limited user can outsource *batch* computations on private data to an untrusted (cloud) server without revealing that private data to the server. (For the rest of the thesis, we will use "server" and "adversary" interchangeably). The user is motivated to outsource computation because that user is using an embedded or mobile device—the cost of spending energy or memory on computation is disproportionally higher than the cost of transporting that data to a more powerful server that may itself contain data that the program needs. In a batch computation, all data that the program will need must be present in program memory at program initialization time.[1] When outsourcing a batch computation, the user sends the server an encryption of its private data that will be the input to a batch program. The server then runs the batch program on the user's data, along with any public data that the server thinks the program will need, to produce an encrypted result. This encrypted result is sent back to the user who decrypts it to get the actual result.

One example of such a batch computation determines health diagnoses given a user's private medical records [45]. In this case, a mobile device is constantly monitoring its user's symptoms, and wants to know the likelihood that the user has some condition given these symptoms. To outsource computation, the device sends a cloud server an encryption of {symptoms, condition of interest}, which will be inputs to a program that calculates likelihood of the condition (call this `MedComp()`). `MedComp()` may be proprietary and contain expert knowledge, learned models or access elements from a database of known size. After running `MedComp()` on the user's private data, the server sends an encryption of the result (e.g., "there is a 55% likelihood that you have the condition") back to the user. To maintain privacy, the server must never learn anything about the user's private inputs—the symptoms or diseases of interest—at any time before, during or after the computation completes. More examples are given in [45].

One way to completely protect private data when outsourcing is for the user to encrypt his/her data and trust that no server entity will see the data in its plaintext form. This ideal

[1] That is, if the program requires access to a file, that file must be loaded into memory at program start time.

can be achieved with Fully Homomorphic Encryption (FHE) techniques [3, 39], which allow an untrusted server to perform computation directly on an encrypted ciphertext without having access to the decryption key. FHE follows the batch computation model because it evaluates programs as circuits: all input data that the program needs must be present at the start. Data is never decrypted by the server under FHE, and *privacy is independent of the program running on the data*. Unfortunately, FHE currently incurs huge overheads, about a billion times slowdown for straight-line code, and this overhead increases with general-purpose batch programs. Further, FHE approaches by themselves cannot guarantee integrity.

Another candidate solution for achieving data privacy when outsourcing is for the server to use tamper-resistant hardware [60, 12, 20]. Here, the user's computation takes place inside a secure hardware compartment on the server side that protects the user's private data while it is being computed upon. The smaller the trusted computing base (TCB) the better from a security perspective: if a trusted component is compromised, the security of the entire system fails.[2] At the same time, removing components from the TCB typically impacts performance and/or energy efficiency because untrusted components in the system need to be retrofitted to prevent information leakage, or the interface to these components has to change. Despite the hit in efficiency, the computationally-limited user is still motivated to outsource computation since compute, energy and memory resources are significantly cheaper for the server than the user.

A serious limitation with current secure hardware solutions is that they (unlike FHE) assume that the *program* running on the secure hardware is trusted. Batch applications like `MedComp()` are seldom trustworthy. To trust the program, the user has to "sign-off" on that program, saying that it is free of malicious code and bugs that may leak sensitive data. Verifying bug-free and malicious behavior is a hard problem—intractable for sufficiently complex programs—and involves more work for the user in any case. Frequent software patches, which are typical in modern software development, only confound the issue. Furthermore, the user may not have access to the program in the first place as the program's algorithms may be proprietary. *How can a user trust a secure processor system when the program running on the secure processor cannot be trusted?*

In this thesis, we take a step towards solving the problem of placing trust in programs by designing a secure processor, called Ascend[3], that can run *untrusted* programs. This approach is directly applicable to batch programs like `MedComp()`. Here, the user would send the server an encryption of its medical information, along with a *request* to use a program that calculates the likelihood of a certain condition, and would expect the server to send back a meaningful result. The server would then run the user's private data on an Ascend processor with a `MedComp()` implementation based on the user's request. As with FHE, privacy leakage is independent of whether the `MedComp()` program is buggy, malicious or being patched on a regular basis.

1.1 The Problem with Running Untrusted Programs on Secure Hardware

Today's secure processors (e.g., Intel+TXT, XOM or Aegis) leak private information when untrusted, badly written, or malicious programs are run. Consider the following scenario.

[2]In this sense, FHE has no TCB.

[3]Architecture for Secure Computation on ENcrypted Data.

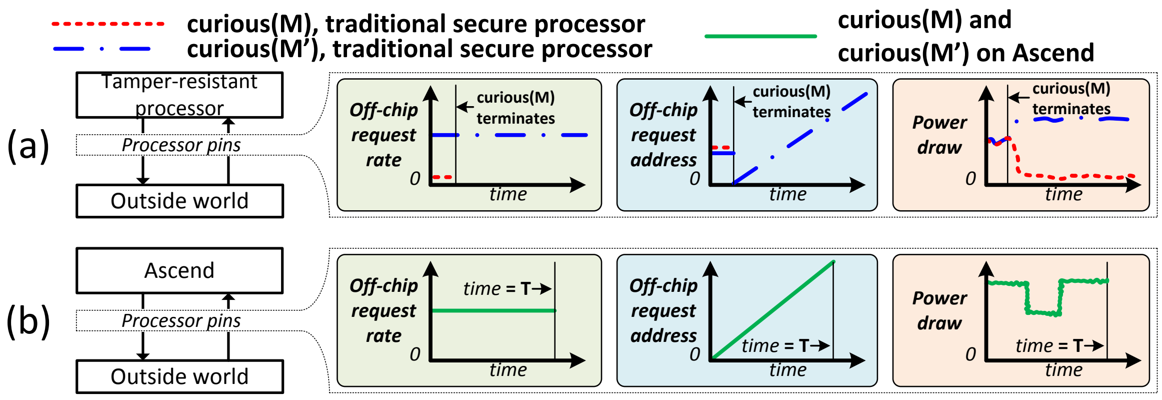


Figure 1-1: Signals from the perspective of (a) a tamper-resistant processor's pins and (b) an Ascend processor's pins when the server runs the `curious()` program (Section 1.1) on two user inputs `M` and `M'`. `M` satisfies the condition `M[I] & 0x1 == 0` and `M'` satisfies the condition `M'[I] & 0x1 == 1`.

A user wants to use the `MedComp()` program (described above), and sends the server an encryption of its symptoms and condition of interest (ciphertext `M`) plus a request to use `MedComp()`. The user expects the server to run the `MedComp()` program. If the server is curious and wants to learn about the user's data, however, it may decide to run the program `curious()` instead, shown in Program 1.

Program 1 The `curious()` program. `M` is the private user memory.

```
int I = 0; // chosen by the server
void curious(M) {
  if (M[I] & 0x1) {
    for (int j = 0;;j++) M[j]++;
  } else { return; }
}
```

As shown in Figure 1-1(a), `curious()` forces a tamper-resistant processor to interact differently with the outside world (which is visible to the server) depending on which branch of the `if` statement is taken. Specifically, a tamper-resistant processor will either appear to run forever and send requests to memory repeatedly or terminate almost instantly. Which event happens depends on a private bit in the user's data, which can be selected by the server by changing `I` and the bitmask `0x1`. Through this attack, the server can learn the value of the bit of interest.

This attack is difficult to prevent. Encrypting data that leaves the secure processor or obfuscating the memory access pattern won't help because the attack succeeds if the server sees either (a) that *some* external request is made or (b) that `curious()` is running forever.[4] Furthermore, adding on-chip cache to the secure processor (to add noise to the memory access pattern as mentioned in [24]) and/or growing the TCB to include (say) the main memory will not help because the server can just rewrite the `curious()` program to ensure that visible resources (e.g., disk) will be accessed.

[4]Both of these signs indicate that the processor entered the `for` loop.

The server can repeat this experiment by running `curious()` multiple times, with different bit masks and values for `I`, in order to leak multiple bits in the user's data. `curious()` may be a standalone program, or embedded in `MedComp()` by an inside party. Of course, even if the server does run a non-malicious program, program characteristics or bugs can leak private information in the same way as `curious()`.

1.2 Ascend: Program Obfuscation in Hardware

Ascend defeats attacks like those posed by the `curious()` program by performing *program obfuscation* in hardware. Formally, Ascend guarantees that: *given an untrusted batch program P, a public length of time T and two arbitrary inputs to P namely M and M': running $P(M)$ for T time is indistinguishable from running $P(M')$ for T time from the perspective of Ascend's external pins.* Ascend has both input/output (I/O) and power pins like a normal processor. We assume that the server can watch these pins, and we will discuss techniques to obfuscate both I/O requests on the I/O pins and the power signature on the power pins. This thesis will not cover invasive attacks on Ascend, nor electromagnetic/radio-frequency-based side-channels (discussed further in Section 2.3).

Unlike a normal processor, Ascend runs for a parameterizable amount of time T that is chosen by the user before the program runs, and is public to the server. (In some settings, the server may also choose T). Since T is set a priori, it may or may not be sufficient to complete the program given the user's inputs. In either case, Ascend's pins must carry signals that leak no information about private user data for the entire T time, regardless of the program run within Ascend. If the program terminates before T time has elapsed, Ascend will perform indistinguishable dummy work so that the server doesn't learn that the program terminated.[5] After T time is complete, Ascend will output either the final program state or intermediate program state, encrypted with the user's key. The server therefore cannot determine whether the program completes or how much forward progress was made, provided program completion time depended on private user data.[6]

Putting these ideas together, Figure 1-1(b) shows how the server cannot learn about the user's input through the `curious()` program from Section 1.1. Regardless of whether `curious()` enters the `for` loop or not, Ascend runs for T time and behaves in an indistinguishable way from the perspective of the chip pins.

1.3 Big Idea: Public Parameters that Control Observable Behavior

A central idea that we will weave throughout the rest of the thesis is that *since the program P is public, the server can collect knowledge about the program (offline). We will add mechanisms to Ascend so the server can use this knowledge to improve Ascend's performance/energy efficiency when running P on private (hidden) inputs.* This idea plays a role during three stages:

[5] Making dummy work indistinguishable from real work is important and will be discussed throughout the rest of the thesis.

[6] Note that in the case of a public program an external observer may be able to tell what instruction is being executed some or all of the time, e.g., at certain points in the program such as the start of the program or if the entire program is made up of straight-line code. Since this is the same for any input M or M', however, it leaks no private information.

At processor manufacturing time, Ascend is architected with a set of public knobs/-parameters that, when specified, fully characterize Ascend's *observable behavior*[7] given any program P and input M. (Given any Ascend implementation, we will refer to that Ascend chip's public parameters as the set $\mathbb{P}$). For example, one such parameter might specify the interval at which Ascend's instruction pipeline should access its on-chip memory. (Assume that accessing the on-chip memory causes an observable power signature). Once the corresponding parameter is specified, Ascend will access the memory at strictly the specified rate, regardless of what is actually needed by the running program. If the memory is not needed when it is accessed, an indistinguishable dummy access will be performed instead (as introduced in the previous section).

Offline, for each program P, the server is allowed to extract information about P—e.g., data-independent control flow, coarse-grain program phase behavior, etc—through static analysis, profiling on its (the server's) own inputs or other means. We make no assumptions about the level of analysis, or amount of analysis that the server performs. The server may even skip this step and perform no analysis for some programs.

Online, the server will initialize Ascend with the program P, the user's private inputs (call this M as before), and specific values for each public parameter in $\mathbb{P}$ that governs Ascend's observable behavior. *Crucially, the server may set each public parameter based on its offline analysis of program P.* Returning to the on-chip memory example, suppose that by profiling program P offline, the server determines that on most inputs to P, the memory will be accessed once every 100 clock cycles. If the server's profiling is representative of the real behavior for $P(M)$, Ascend becomes more efficient while running $P(M)$: memory is accessed less often than a naïve scheme (which might be to make an access as frequently as possible) and always performs useful work. If the server's profiling is incorrect, Ascend will still access the cache at the same rate, to hide the true behavior of $P(M)$.

The public parameter governing the on-chip memory was just an example—Ascend may support any number of any type of public parameters at any granularity in principle to get more efficiency in different parts of the chip. Furthermore, the server may decide to set these parameters once (at program start time), or dynamically (e.g., if through offline profiling the server sees that program P has temporal phase behavior and can estimate when each phase occurs). *The key point is that the public parameter system allows the server to optimize for common case program behavior.*

1.4 Thesis Contributions

This thesis has three primary contributions:

1. First, we introduce and apply the general principle of a processor chip's observable behavior being fully specified by a set of public knobs, each of which are exposed to and controllable by adversaries.

2. We then introduce principles for designing a general-purpose power and I/O-obfuscated hardware architecture, given *untrusted* programs, and detail a complete design as an example. The idea of public knobs is applied throughout.

3. Finally, we optimize the primitives (Oblivious RAM for the I/O channel and Wave Dynamic Differential Logic), that provide the foundation for protecting each channel,

[7]For example, when/where/how external memory is accessed, when on-chip memories are accessed, etc.

for our setting.

As noted in the Acknowledgements section, Ascend has been a collaborative effort and the authors have already published pieces of the work. [51] presents techniques to perform encrypted computation (using FHE as a primitive) while exploiting apriori knowledge of public/untrusted programs. An initial publication on Ascend was published in [50]. Lastly, material in Chapters 3 and 6 appears in [59].

1.5 Organization

The rest of the thesis is organized as follows. We describe our implementation philosophy, security model and a user-server interaction protocol in Chapter 2. Chapter 3 describes optimized primitives to obfuscate signals on Ascend's I/O pins. Chapter 4 discusses optimized primitives to obfuscate signals on Ascend's power pins. Chapter 5 puts the ideas from Chapters 3 and 4 together and presents a complete microarchitecture for a particular implementation of the Ascend processor. Chapter 6 describes a scheme for guaranteeing integrity of execution. Chapter 7 gives an argument for security and then evaluates performance and power overheads. In Chapter 8 we discuss related/future work and we conclude with Chapter 9.

1.6 Conventions and Assumed Background Knowledge

We will use capital letters to define variables that have meaning throughout the entire thesis. For example, T will always refer to Ascend's public time threshold, P will always refer to the public/server-specified program, etc. For reader convenience, we have consolidated all capital symbols in Table 1.1, along with where they are defined in the writing. Lowercase *italicized* symbols are treated as "temporary variables," which will be defined, used for a short period of time, and re-defined as needed in later sections. A function called func is referred to as $\mathsf{func}()$; we refer to the probability that a random variable $R = x$ as $\mathsf{Prob}(R = x)$ and the expected value for R as $\mathsf{Expected}(R)$.

The thesis assumes basic knowledge in security and computer architecture. That is, the reader is expected to be familiar with symmetric encryption, key sharing, and hashing. Readers aren't expected to be familiar with side-channel countermeasures: e.g., ORAM and power analysis-resistant logic. Readers are expected to be familiar with *basic* computer architecture and digital design: namely that processors contain instruction pipelines, on-chip caching (and that caches are organized in terms of sets, each of which may have multiple ways), possibly networks and memory controllers. On the other hand, readers aren't required to know the difference between inclusive and exclusive caches, or the power/performance trade-offs in instruction pipeline design (for example). Basic clock terminology (edge behavior, clock gating), flip-flop, logic gate and SRAM memory behavior are also assumed.

Table 1.1: Variables with consistent meaning that are used throughout the thesis.

Symbol name	Description	Defined in Section...
$\mathbb{P}$	The set of public parameters that control Ascend's observable behavior	1.3
P	Public/server-specified program running on Ascend	1.2
M	Program memory	1.2
T	Public time threshold	1.2
K	Secret (symmetric) key, shared with the user	2
R	Number of bytes returned by two-interactive protocol	2.2
I	Public interval ($I \in \mathbb{P}$)	2.4.2.1
Q	ORAM transcript (sequence of addresses)	3.2
L	Bits per Path ORAM leaf index	3.2.1
N	Number of buckets in a Path ORAM tree	3.2.1
Z	Blocks per Path ORAM bucket	3.2.1
B	Bits per cache line/ORAM block	3.2.1
U	Bits per program memory address	3.2.1
C	Path ORAM local cache capacity	3.2.1.4
F	Cache data bus width	5.4.2
S	Public circuit policy ($S \in \mathbb{P}$)	5.6
E	Energy consumption per processor operation	7.2.1.3

Chapter 2

Framework

To start, we present a general framework for performing computation under encryption for batch programs (Sections 2.1-2.2). Section 2.3 then discusses our attacker model and security assumptions. Finally, Section 2.4 gives an implementation philosophy which we will use throughout the next two chapters.

We assume black-box symmetric-key $\mathsf{encrypt}_K(\dots)$ and $\mathsf{decrypt}_K(\dots)$ functions, which take a plaintext/ciphertext and key K (where K has been shared with the user) as inputs and return the corresponding ciphertext/plaintext using probabilistic/randomized[1] encryption or decryption under K. We will refer to this as randomized encryption/decryption for the rest of the thesis. Time is measured in processor clock cycles.

2.1 Modeling the Ascend Chip

Ascend is a single-chip coprocessor running on the server. While running any program, Ascend stores a symmetric (secret) session key K in an on-chip register. Aside from this register, Ascend chip is composed of symmetric encrypt/decrypt circuits (which use K to perform randomized encryption/decryption), an interface to external memory, and a means to run programs (i.e., an instruction pipeline and on-chip cache memory). The key K is accessible only to the encrypt/decrypt units—not to the program running on Ascend.

Ascend interacts with the server to initialize, terminate and run a program. While Ascend is running, it may communicate over its I/O pins to a fixed-size external memory that is controlled by and visible to the server. All data sent to external memory from Ascend is encrypted under K; further, Ascend decrypts any encrypted data it receives under K. Addresses sent to external memory, and whether a given operation is a read or a write (the *opcode*), are always plaintext. That is, off-chip accesses are to public memory locations whose contents are encrypted. Schemes must be in place to prevent the plaintext address and opcode from leaking private user data—a problem we address in Chapter 3. Ascend has power pins that draw current based on what circuits within the chip are currently being used and what data values are being sent to/from those circuits (like a normal processor).

[1] E.g., AES in CTR mode.

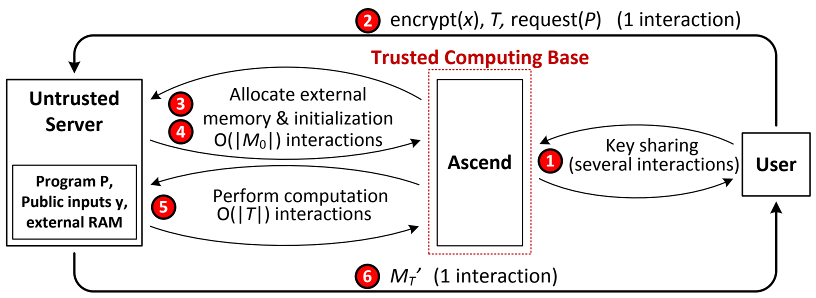

2 encrypt(x), T, request(P) (1 interaction)
Trusted Computing Base
Untrusted Server
Program P, Public inputs y, external RAM
3 4 Allocate external memory & initialization O(|M0|) interactions
5 Perform computation O(|T|) interactions
Ascend
1 Key sharing (several interactions)
User
6 MT' (1 interaction)

Figure 2-1: The two-interactive protocol between a user, server and Ascend. Numbers follow Section 2.2.

2.2 Two-interactive Protocols

We use a two-interactive protocol to initiate and terminate sessions between a user, an untrusted server and Ascend. The user wants to perform computation on private inputs x using a batch program P. P is stored on the server-side and may have a large amount of public data y associated with it (e.g., the contents of a database). The result of the user's computation is denoted $P(x, y)$, meaning that program P needs only the inputs x and y to run which matches our definition of batch computation from Chapter 1. Formally, a two-interactive protocol for computing $P(x, y)$ works as follows (shown graphically in Figure 2-1):

1. The user shares a secret (symmetric) key K securely with Ascend. We assume that Ascend is equipped with a private key and a certified public key. As described in Section 2.1, K is used by Ascend's encrypt/decrypt units.

2. The user encrypts its inputs x using K to form ciphertext $\mathsf{encrypt}_K(x)$ and then chooses a number of cycles T, which is the *public* time budget that the user is willing to pay the server to compute on P. In certain circumstances, the server may choose T. The user then transmits $(\mathsf{encrypt}_K(x), T, \mathsf{request}(P))$ to the server, where $\mathsf{request}(P)$ is a request to use the program P.

3. After receiving the pair $(\mathsf{encrypt}_K(x), T, \mathsf{request}(P))$, the server sends $\mathsf{encrypt}_K(x)$, P, y and $\mathbb{P}$[2] to Ascend's I/O pins as a stream of data. As Ascend receives each block of data, it decrypts the block (if the data is part of $\mathsf{encrypt}_K(x)$), (re-)encrypts the block (unless that block contains all or part of $\mathbb{P}$) with a different random pad and writes it to external memory. Each parameter in $\mathbb{P}$ is stored in on-chip registers where it will be read by Ascend to create a variety of observable behaviors. At the end of this step: all of x, P and y will be stored in Ascend's external memory and encrypted under K (we refer to this as M_0, the program state after 0 time).

4. The server chooses a *public* size R, which is the number of bytes reserved for the final result/output of P's computation on x and y (R can also be supplied in Step 2 by the user). The server sends T and R to Ascend in the clear.

5. Ascend spends a number of clock cycles, corresponding to T, making forward progress

[2]Specific values for each of this Ascend chip's public parameters (see Section 1.3).

in P after which Ascend extracts *exactly* R bytes as indicated[3] by P. While running, Ascend may make requests to its external memory to fetch more instructions or data. During this time the server may also decide to load new values for any parameter in $\mathbb{P}$ at any time. Schemes must be in place to prevent leakage of private user data from I/O requests or chip power consumption regardless of the program run or when that program terminates (Chapters 3 and 4).

6. After Ascend runs for T time, the external memory contains M_T. The ciphertext $M'_T = \mathsf{encrypt}_K(r)$ (r are the R bytes extracted from M_T in the previous step) is sent back to the user.

7. The user decrypts M'_T and checks whether T was sufficient to complete $P(x, y)$. Without loss of generality, we assume that the algorithm outputs an "I am done" message as part of its final encrypted result.

A correct execution of the two-interactive protocol outputs to the user the evaluation $P(x, y)$ (if T was sufficient) or some *intermediate result*.

2.3 Security Model

The server in the two-interactive protocol is malicious in that it (a) wants to learn about the user's private data and (b) wants to cheat the user out of its computation.

2.3.1 Privacy

We assume the server can monitor the Ascend chip's I/O and power pins. We make no assumptions as to *how* the server monitors the pins. That is, an insider might attach an oscilloscope to the pins directly, or create a sniffer program running on another server chip which monitors what bits change in Ascend's external memory. The Ascend chip itself is assumed to be tamper-resistant: the server cannot remove packaging/metal layers and watch bits once they have been decrypted. The server is assumed to know Ascend's architecture and microarchitecture, e.g., its cache block size, cache capacity, the cycle latency for each instruction, etc.

In this thesis, we assume the server cannot learn from side-channels other than the pins (e.g., EM or RF-based channels). These additional leakage channels separate our work from FHE—since FHE has no TCB, it is impervious to all physical side-channels. We consider these additional attack surfaces to be a part of our future work.

To learn about the user's private inputs, the server can perform experiments with Ascend and $\mathsf{encrypt}_K(x)$, and watch Ascend's pins for data-dependent behavior. To perform an experiment, the server initializes Ascend with arbitrary $\mathsf{encrypt}_K(x)$,[4] P, y, T, R and $\mathbb{P}$ (i.e., any of the parameters from Section 2.2) and monitors how Ascend interacts with the outside world through the chip pins.[5] While Ascend is running, the server can modify the

[3]This can be coded in a separate subroutine in P or can be supplied as a separate extraction algorithm in conjunction with P.

[4]Since P is untrusted, the server can force Ascend to produce ciphertexts of x, y, or arbitrary transformations on these quantities. Since `encrypt()` is probabilistic, however, any ciphertexts manipulated directly by the server (i.e., flipping arbitrary bits) will cause the underlying plaintext to be decrypted to garbage with overwhelming probability.

[5]Running the `curious()` program from Section 1.1 is an experiment where the server ran a malicious program.

contents of Ascend's external memory at any time, perform a denial-of-service attack by not returning requested data, send any data back to Ascend at arbitrary times, or modify any value in $\mathbb{P}$. The server does not need to return the results of its experiments back to the user. These experiments can be run offline, e.g., after the computation requested by the user takes place. Chapters 3-5 will show how the server learns negligible amounts of information about $\mathsf{encrypt}_K(x)$ by monitoring Ascend's pins, even if the server performs arbitrary experiments of this kind.

2.3.2 Integrity

To cheat the user, the server can send the user the result of running a different program (a program P' different from that corresponding to $\mathsf{request}(P)$), run P or P' for some time threshold not equal to T, or tamper with external memory by changing bits—all of which cause the user to get back some incorrect result. The server is motivated to do the minimal amount of work needed to satisfy the user, or to trick the user into interacting with the server more times in the future. Chapter 6 will show an efficient implementation for *certified execution*, which allows the user to detect this cheating. Given that the user can detect cheating, the server's motivation changes: it wants as much business from the user as possible and therefore wants to return as complete and correct a result as possible.

2.4 Implementation Principles

There are different implementation strategies that can attain the security level described in Section 2.3. Furthermore, the schemes used to secure the I/O channel may be different than the schemes used to secure the power channel. In general, for either channel to be completely obfuscated for T time, it must be obfuscated in two domains: *value* and *time*. That is, *when* and *how* signals on that channel change must not give away any private information to the untrusted server for T time.[6] We will now explain the ideas for obfuscating value and time for each channel. Chapters 3-5 will detail how these ideas are implemented for each of the I/O and power channels.

2.4.1 Value Domain

To obfuscate a channel in the *value* domain, any particular operation on that channel must look like a random (data-independent) operation. To accomplish this for the I/O channel, Ascend manages its external memory as an Oblivious-RAM (ORAM) (Section 3.2). At a high-level, ORAM is an encrypted and shuffled memory that has the property that any particular off-chip memory access looks like a random access. To protect the power channel, we build Ascend out of differential power analysis resistant (DPA resistant) circuits (Section 4.2). Each circuit in Ascend, whose behavior can be influenced by user data, must have the following property: when that circuit is accessed with some set of inputs, the power signature emitted from the circuit must be indistinguishable from when that circuit is activated with a random set of inputs. Preventing leakage over the I/O is orthogonal to preventing leakage over the power channel. That is, ORAM does not help secure the power channel and DPA resistant circuits do not help the I/O channel. We note that both ORAM and DPA resistant circuits have computational security assumptions (i.e., based on

[6]Recall that after T time, Ascend stops running and therefore will not leak any information after this point.

encryption/PRNG/circuit routing primitives/etc). We discuss these assumptions further in Chapters 3-4.

2.4.2 Time Domain

To obfuscate signals in the *time* domain, both the I/O and power channels apply the following key idea: *that security is independent of the rates at which Ascend accesses internal circuits (e.g., caches, register files, etc) or off-chip memory as long as these rates are data-independent (e.g., specified by the untrusted server).* We architect the Ascend chip such that the rate at which each circuit is activated is controlled by some public parameter in $\mathbb{P}$ (Section 1.3).

Despite the server knowing what program is running inside Ascend, programs in general may have data-dependent behavior that depend on (private) program input. Thus, a data-independent rate scheme may cause circuits to be accessed when they are not needed by the program. In this case, the circuit performs *dummy work* which does not help to make forward progress in the program. We call non-dummy work *real work* for the rest of the thesis. Dummy work must be indistinguishable from real work, which is true if and only if the value domain is obfuscated (see previous section). On the other hand, the program may also need to access a particular circuit before the data-independent rate allows for that circuit to be accessed. In this case, the program will stop making forward progress until the circuit is accessed. Schemes must be in place to prevent the server from determining when or if this ever happens.

2.4.2.1 Public Intervals

One type of public parameter that we will use throughout the thesis is the *public (server-specified) interval*. For each circuit x (or group of circuits), we add a public parameter $I_x \in \mathbb{P}$ which indicates the number of Ascend clock cycles between when the circuit named x completes its last access and starts its next access.[7] As introduced in Section 1.3, the server may decide on a good value for each public interval based on apriori information at the server's disposal: namely P, y, T, etc.

We now give a simple performance model for public intervals to help intuition. Suppose (a) that the server sets the off-chip memory interval to be I_{mem} processor clock cycles, (b) that each memory access takes t_{access} cycles to complete and (c) that Ascend runs for T total cycles. T is now broken into *epochs*, where the nth epoch begins on clock cycle $T_{epoch,n} = n \cdot (I_{mem} + t_{\text{access}})$ for $n \geq 0$. Then a new real or dummy memory access is made on clock cycle $T_{epoch,n} + I_{mem}$ for $n \geq 0$.

Suppose the program needs to make a memory access at some clock cycle t where $t < T$ and $t_{\text{epoch}} = I_{mem} + t_{\text{access}}$ for short. If $i = (t \mod t_{\text{epoch}})$, we have the following cases:

1. If $i < I_{mem}$: the memory access in this epoch will do real work and the program will get a response back in $t_{\text{epoch}} - i$ cycles.

2. If $i \geq I_{mem}$: the memory access in this epoch will do dummy work and the program will get a response back in $2 \cdot t_{\text{epoch}} - i$ cycles.

To meet the security requirements from the previous section, the server must never determine t if t depends on the user's private data.

[7]Note that this definition does not forbid concurrency, but implies that any circuit that supports concurrent operations must have indistinguishable behavior regardless of the interleaving of its operations.

The public interval scheme guarantees that the server learns nothing based on *when* each circuit is accessed, in an unconditional sense. That is, an interval is a flat rate: when each circuit is accessed does not depend on the security of some primitive (e.g., a PRNG) or any computational assumption.

Implementation note. A simple hardware-based way to implement intervals is with dedicated hardware finite state machines (referred to as interval FSMs) that behave like counters. The interval FSM increments an internal counter each processor cycle and triggers either a real or dummy request when it reaches the threshold set by the server. Once the behavior set to the interval completes, the FSM resets and the process repeats. In principle, the server may set/change the interval before or during the computation; for the rest of this thesis, we assume that the interval is specified before the computation starts and is not changed.

Chapter 3

Protecting the I/O Channel

Our goal in this chapter is to develop and optimize primitives that make the *digital* signals on Ascend's I/O pins not leak any information that depends on the user's private data.

Program 2 Curious programs that leak through the I/O channel. `M` is the private user memory.

```
int I = 0; int J = 1; int K = 2;
void curious_addr(M) {
  if (M[I] & 0x1) int a = M[J];
  else            int a = M[K];
}
```

```
int I = 0; int J = 1;
void curious_rate(M) {
  if (M[I] & 0x1) int a = 0;
  else            int a = M[J];
}
```

Ascend makes I/O requests to external memory when it needs more data/instructions and the server may learn about private user data by monitoring the requests themselves (value domain) and/or when requests are made (time domain). Each request is composed of a plaintext address, opcode (read or write) and data block (if the operation is a write). Mitigating leakage over the opcode and data blocks is straightforward: the data block values can be hidden using randomized encryption (Section 2.1) and the opcode can be hidden by forcing each I/O request to perform both a read and a write. Preventing leakage through the plaintext address is a difficult problem. For example, the server can engineer the `curious_addr()` program shown in Program 2 (left). Here, the server learns whether the branch is taken by watching for address `J` or `K` on the pins.[1]

Of course, even if the server can learn nothing by monitoring the specific logic values on the I/O pins, it can still learn by watching when the I/O channel is used. For example, the `curious_rate()` program (Program 2 (right)) either accesses memory once (branch taken) or twice (branch not taken). This example leaks the same amount of information as `curious_addr()`.

[1]Most processors group consecutive data words into blocks, which could make `J` and `K` map to the same cache block and prevent the second request from being made over the pins. The server can always get around this issue by choosing `J` and `K` to correspond to different cache blocks. Recall that the server knows Ascend's architecture (Section 2.3).

3.1 Chapter Overview

The first part of this chapter, Section 3.2, introduces Oblivious-RAM (ORAM): a primitive from the literature that we use to make any I/O request look like a random request, thereby obfuscating particular logic values on the I/O pins (e.g., address, opcode and data). For readers already familiar with ORAM and Path ORAM (a particular ORAM scheme that we use in constructing Ascend), Sections 3.2.1.2 through 3.2.2 discuss optimizations to Path ORAM to make it implementable in an Ascend setting. Next, Section 3.3, improves the ORAM primitive by creating a zero-failure ORAM based on *background eviction* and discusses attacks against background eviction schemes. Lastly, Section 3.4 discusses how the public interval introduced in Section 2.4.2.1 is applied to the improved ORAM primitive to prevent leakage over the time domain.

3.2 Value Obfuscation: Oblivious RAM (ORAM)

Oblivious RAM (ORAM), first described in [4, 5, 7], makes any sequence of memory requests look like data-independent requests. Formally: suppose we are given batch program P with input M and any other batch program P' with input M' and compare the first z memory requests made by each. We denote the sequence of requests as $Q_z(P(M))$ and $Q_z(P'(M'))$.[2] Each request is composed of an address, operation (read or write) and data (if the operation is a write). Oblivious RAM (ORAM) guarantees that $Q_z(P(M))$ and $Q_z(P'(M'))$ are computationally indistinguishable. Crucially, this is saying that the access pattern is independent of the program and data being run, which defeats the attack posed by `curious_addr()` from the beginning of the chapter.

Notice that a simple ORAM scheme that satisfies the above property is to read/write the entire contents of the program memory to perform every load/store. To hide whether a particular block was needed in the memory scan (and if it was, whether the operation was a read or a write), every block must be encrypted using randomized encryption. With this scheme, the access pattern is independent of the program or its data but clearly will have unacceptable overheads (on order the size of the memory). Modern ORAM schemes use pseudo-random number generation to achieve polylogarithmic overheads: any address sequence of a fixed length must be computationally indistinguishable from a random address sequence.

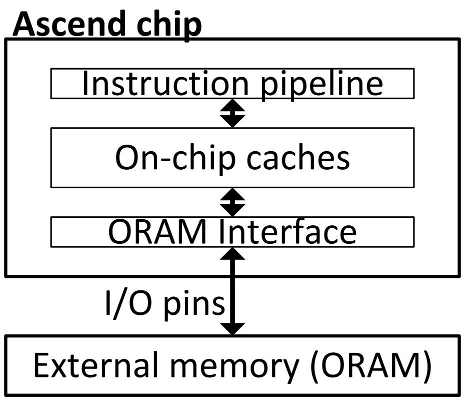


Figure 3-1: ORAM organization in Ascend.

Any ORAM scheme is made up the ORAM itself (implemented on top of some untrusted memory) and trusted ORAM client logic. In our setting, the ORAM is implemented over the external memory from Section 2.1, whose input/output bus and contents are visible to the server at all times. The ORAM client logic (which we refer to as the *ORAM interface*) is built inside Ascend and is responsible for translating program memory requests into random-looking requests that will be sent to the untrusted memory/ORAM itself. The ORAM interface is analogous to a mem-

[2] If z_P and $z_{P'}$ are the total number of requests made by P and P' respectively, then $\min(z_P, z_{P'}) \geq z \geq 0$.

ory controller—i.e., all digital communication by Ascend's instruction pipeline and on-chip caches (if any) go through the ORAM interface (Figure 3-1). Thus, I/O channel security in the value domain reduces to the security of the ORAM. We note that the ORAM interface's internal state is trusted and must be hidden from the server.

ORAM was first introduced to maintain software confidentiality. Recently, two new proposals were published in [46, 57]. Stefanov and Shi also constructed a much more simple and practical ORAM construction called Path-ORAM [56] which we use and improve here.

3.2.1 Path ORAM

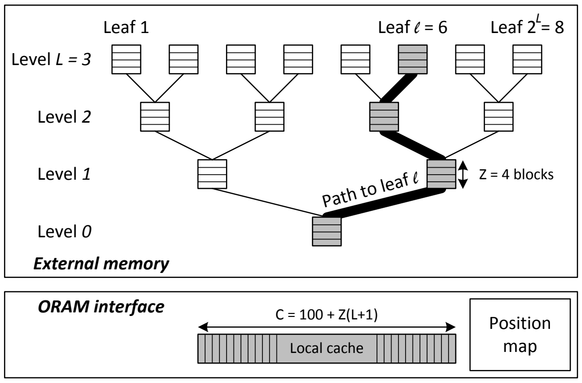


Figure 3-2: A Path ORAM for $L = 3$ levels. At any time, a block mapped to leaf $l = 6$ can be located in any of the shaded structures. The local cache capacity (C) is discussed in Section 3.2.1.4.

In Path ORAM, the external memory is structured as a balanced binary tree, where each node is a *bucket* that can hold up to Z blocks. The root is referred to as level 0, and the leaves as level L. This gives a tree with $L + 1$ levels, holding up to $Z(2^{L+1} - 1)$ data blocks (each of which is analogous to a processor cache block in our setting). The remaining space is filled with *dummy blocks* that can be replaced with real blocks as needed. As with data blocks in a naïve memory scan scheme (Section 3.2), each block in the ORAM tree is encrypted with randomized encryption.

The ORAM interface for Path ORAM is composed of three parts: a *local cache*[3], a *position map*, and associated control logic. The position map is a lookup table that associates the program address of each data block with a leaf in the ORAM tree. The local cache is a memory that stores up to a small number of data blocks from the ORAM tree at a time. Since both structures store secret information that the server must never learn (Section 3.2), a strawman approach may implement both in SRAM memory within Ascend.

Now we describe how Path ORAM works. Readers can refer to [56] for a more detailed description. At any time, each data block stored in the ORAM is mapped (at random) to one of the 2^L leaves in the ORAM tree via the position map (i.e., $\forall$ leaves l and blocks b, $\mathsf{Prob}(b \text{ is mapped to } l) = 1/2^L$). Path ORAM's invariant (Figure 3-2) is: *If l is the leaf currently assigned to some block b in the ORAM, then b is stored (a) 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), or (b) in the local cache within the ORAM interface.*

Initially the ORAM is empty and the position map associates each possible program address with a random leaf. Suppose a program wants to access some block b with program address u and that this block is currently mapped to leaf l. The program using the ORAM makes requests through the ORAM interface via $\mathsf{accessORAM}(u, op = read/write, b')$:

1. Look up the position map with u, yielding the leaf label l.

[3]This is ORAM terminology, not to be confused with on-chip processor caches.

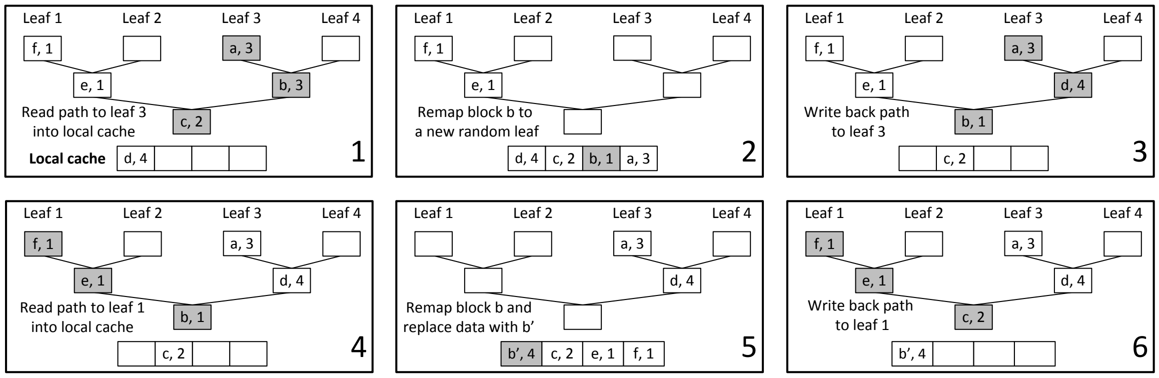

Leaf 1
Leaf 2
Leaf 3
Leaf 4
f, 1
a, 3
e, 1
b, 3
Read path to leaf 3 into local cache
c, 2
Local cache
d, 4
1
Remap block b to a new random leaf
d, 4
c, 2
b, 1
a, 3
2
d, 4
Write back path to leaf 3
b, 1
c, 2
3
Read path to leaf 1 into local cache
4
Remap block b and replace data with b'
b', 4
e, 1
f, 1
5
Write back path to leaf 1
6

Figure 3-3: An example Path ORAM operation (where $Z = 1$ and the local cache has a 4 block capacity) for reading some block b and then (in a second access) replacing the value of that block with a new value b'. Each block has the format: 'block identifier','leaf label' (e.g., $b, 3$). If the program address for b is u, the top three boxes (from left to right) correspond to $\mathsf{accessORAM}(u, read, -)$ and the bottom boxes are $\mathsf{accessORAM}(u, write, b')$ from Section 3.2.1. The server sees the ORAM interface read/write the path to leaf 3 and then read/write the path to leaf 1, and therefore cannot figure out what operations really occurred.

2. Read the buckets along the path from the root of the ORAM tree to leaf label l (in that order). Decrypt all real blocks on the path and add those blocks to the local cache. Path ORAM's invariant guarantees that if block b exists, it must be in the local cache at this point.

3. If $op = read$, return b if it exists; otherwise return nil. If $op = write$, replace b with b' if it exists; otherwise add a new block b' to the local cache.

4. Replace the label l associated with b with a new randomly-selected l'.

5. Evict and encrypt as many blocks, from the updated local cache, into the path to leaf l in the ORAM tree. If there is space in any of the buckets along the path that cannot be filled with data blocks, fill that space with encryptions of dummy blocks. As with reading the path, blocks must be written back in a data-independent order (e.g., starting with the root and ending with the leaf).

Later in the thesis, we will refer to steps 2-5 as $\mathsf{accessPath}(u, l, l', op, b')$.

Step 4 is the key to Path ORAM's security: whenever a block is accessed, that block is randomly *remapped* to a new leaf in the ORAM tree (see Figure 3-3 for an example). $\mathsf{accessORAM}()$ leaks no information on the address accessed, because a randomly selected path is read and written on every access regardless of the program memory address sequence. Furthermore, since data/dummy blocks are put through randomized encryption, the server will not be able to tell which block (if any) along the path is actually needed.

Step 5 is similar to the ORAM 'shuffle' operation from the literature [7]. As paths are read into the local cache, the local cache will begin to fill. In order to keep the necessary local cache capacity as small as possible (which reduces on-chip storage requirements), step 5 tries to writeback as many blocks to the tree as possible. To perform writeback, we scan the local cache (from right to left in Figure 3-3); for each scanned block in the local cache, we write that block as close to the leaf bucket in the path as possible. In the top right box

(# 3) in Figure 3-3: $a, 3$ is scanned first and mapped back to leaf 3; $b, 1$ only shares the root bucket in common with the path to leaf 3 so it is written to the root; $c, 2$ can no longer be written back to the tree at all (since it only shared the root bucket in common with the path to leaf 3, and the root bucket is now full); finally, $d, 4$ is mapped back to the open bucket between the leaf and the root.

Despite the writeback operation, the local cache still has a chance to *overflow*, causing the ORAM to fail. If the ORAM fails, it may return functionally incorrect results at any point in the future, or be forced to recover somehow (which may leak privacy). Failure can occur due to the block remapping process. Suppose that on some access to block b, b is remapped from path p to path p'. Let $\mathsf{CP}(p, p')$ (for common path) yield the set of buckets that are shared between paths p and p'. Then, if there is no space for block b in any bucket in $\mathsf{CP}(p, p')$, the local cache occupancy will increase by one. We notice that if p and p' are chosen at random, $\mathsf{Expected}(|\mathsf{CP}(p, p')|) = 2 - \frac{1}{2^L}$. Thus, the likelihood of b getting stuck in the local cache is not negligible: on average p and p' share less than two buckets.[4] Note that if $p = p'$ (which happens with negligible probability, $\mathsf{Prob}(p = p') = \frac{1}{2^L}$), the local cache is guaranteed not to grow in occupancy during the access to block b. We will introduce schemes to eliminate failure probability and simultaneously get performance in Section 3.3.

The ORAM tree and local cache stores (leaf, program address, data) 3-tuples for each data block. Suppose the ORAM capacity in data blocks is given by $N = Z(2^{L+1} - 1)$. Then each leaf is labeled by L bits and each block's associated program address is stored in $U = \lceil \log_2 N \rceil$ bits. If B is the data block size in bits, each bucket contains $Z(L+U+B)$ bits of plaintext. As mentioned, the protocol requires randomized encryption over each block (including dummy blocks) stored in external memory, which imposes additional overheads. We now discuss two schemes for performing randomized encryption:

3.2.1.1 Strawman Encryption Scheme

A strawman scheme to fully encrypt a bucket (used in a preliminary version of Ascend [50]) is based on AES-128: On a per-bucket basis, apply the following operation to each block in the bucket:

1. Generate a random 128-bit key K' and encrypt K' using the processor's secret key K (i.e., $\mathsf{AES}_K(K')$).

2. Break up the B plaintext bits into 128-bit chunks (for AES) and apply a one-time-pad (OTP) to each chunk that is generated through K' (i.e., to encrypt $chunk_i$, we form the ciphertext $\mathsf{AES}_{K'}(i) \oplus chunk_i$).

The encrypted block is the concatenation of $\mathsf{AES}_K(K')$ and the OTP chunks, and the encrypted bucket is the concatenation of all of the Z encrypted blocks. Thus, this scheme gives a bucket size of $BucketSize = Z(128 + L + U + B)$ bits where $Z(L + U + B)$ is the number of plaintext bits per bucket from the previous section. Note that since we are using OTPs, each 3-tuple of $(L+U+B)$ bits does not have to be padded to a multiple of 128 bits.

[4] Note that it is not straightforward to bound the probability on block b getting stuck in the local cache—and furthermore the percent chance that the ORAM will fail for a given local cache size—since Step 5 in $\mathsf{accessORAM}()$ can move blocks from buckets in $\mathsf{CP}(p, p')$ to locations higher in the ORAM tree before block b is considered for writeback.

3.2.1.2 Optimization: Counter-based Encryption Scheme

The downside to the strawman scheme is the extra 128 bits of overhead per block that is used to store $\mathsf{AES}_K(K')$. We can reduce this overhead by a factor of $2 \cdot Z$ by introducing a 64-bit counter per bucket (referred to as $BucketCounter$). To encrypt a bucket:

1. $BucketCounter \leftarrow BucketCounter + 1$.[5]
2. Break up the plaintext bits that make up the bucket into 128-bit chunks. To encrypt $chunk_i$, apply the following OTP: $\mathsf{AES}_K(BucketID||BucketCounter||i) \oplus chunk_i$, where $BucketID$ is a unique identifier for each bucket in the ORAM tree.

The encrypted bucket is the concatenation of each chunk along with the $BucketCounter$ value in the clear.

This scheme works due to the insight that buckets are always read/written atomically. $BucketCounter$ is set to 64 bits so that the counter value will not roll over (as in AES-CTR). Note that $BucketCounter$ does not need to be initialized; it can start with any value. Also note that seeding the OTP with $BucketID$ is important: it ensures that two *distinct* buckets in the ORAM tree will not have the same OTP. With this scheme, $BucketSize = Z(L + U + B) + 64$ bits which we assume for the rest of the paper.

3.2.1.3 Path ORAM Performance and Hardware Assumptions

In this thesis, we assume that the ORAM tree is stored in some commodity external memory (e.g., DRAM) that lives next to the Ascend processor on a shared board. The processor receives/sends data over P 1-bit data pins (our evaluation assumes $P = 128$ to be competitive with modern processors). The ORAM tree itself is laid out flat in memory, level by level.[6] That is, the root bucket is laid out contiguously at the low-order external memory address. The two buckets in level 1 of the ORAM tree are stored immediately after the root bucket, etc. Depending on memory technology, buckets may be padded for addressing reasons which we will discuss below.

ORAM access latency correlates to (a) the number of buckets read per access ($= 2 \cdot (L + 1)$) and (b) the number of bits in each bucket, as given in Section 3.2.1.2. The Path ORAM reads buckets along a path: while each bucket is stored contiguously in external memory, two buckets in adjacent levels are generally not adjacent. In fact, between levels l and $l + 1$ for large l, the two buckets on a path are likely stored in different memory banks or different memories altogether (since the number of buckets per level grows exponentially with the depth of the tree). In our performance evaluation, we assume that:

1. An entire bucket can be read through a burst/fast page command to the DRAM such that P-bits of the bucket arrive each cycle until the bucket is fully sent/received.
2. If bucket$_i$ is currently being sent/received and bucket$_{i+1}$ is the next bucket, the first P bits for bucket$_{i+1}$ arrive the cycle after the last P bits in bucket$_i$.

[5]Note that since the counter always increments by one, it is important for Ascend to use a different user session key K during each run of each program, to avoid a replay attack. Alternatively, the counter could be changed to a random number, re-generated when each bucket is re-encrypted. In that scheme, however, there is a small probability that the same random number is generated twice, which means the same OTP will be used twice.

[6]We note that for some ORAMs it is not possible to completely fill the last ORAM tree level with buckets to get the desired ORAM capacity. In that case, some branches may be 1 level longer than other branches. This doesn't impact the flat layout because the change only impacts the leaves.

3. Each bucket is aligned to the nearest P in DRAM. That is, each bucket takes up $P \cdot \lceil \frac{BucketSize}{P} \rceil$ bits, stored contiguously. This is done to avoid needing alignment/shifting hardware in the ORAM interface.

Based on these requirements, an implicit assumption is that Ascend's ORAM interface is fully pipelined and able to process (i.e., decrypt, store, re-encrypt) P-bits per cycle. With these assumptions, Path ORAM access latency is given by

$$\begin{aligned} CyclesPerAccess &= Lat_{AES} + Lat_{DRAM} + 2 \cdot (L+1) \cdot \left\lceil \frac{BucketSize}{P} \right\rceil \\ &= Lat_{AES} + Lat_{DRAM} + 2 \cdot (L+1) \cdot \left\lceil \frac{Z(L+U+B)+64}{P} \right\rceil \end{aligned} \tag{3.1}$$

where $Lat_{AES} = 100$ is the latency to decrypt and encrypt a single bucket and $Lat_{DRAM} = 100$ is the number of cycles needed to perform a single DRAM read plus single DRAM write. Note that AES and DRAM costs are only incurred once due to pipelining.

The most important assumption for our performance evaluation is #2 above: that the first P-bit chunk for bucket$_{i+1}$ arrives the cycle after the last chunk in bucket$_i$. We argue that this is reasonable because the sequence of buckets needed for a particular ORAM access is known as soon as the position map lookup is complete. That is, the ORAM interface can prefetch each bucket after the position map read occurs, or send the leaf to an external (insecure) controller local to the DRAM chips that generates a series of bucket address requests at a rate which matches the Ascend chip's pin throughput.

3.2.1.4 Path ORAM Space Requirements

The ORAM tree (external memory) can store up to $N \cdot B = Z(2^{L+1}-1) \cdot B$ data bits while the ORAM tree data structure uses $(2^{L+1}-1) \cdot BucketSize = (2^{L+1}-1) \cdot (Z(L+U+B)+64)$ bits. Suppose the block capacity in the local cache is given by C. Then the local cache uses $C \cdot (L+U+B)$ bits and the position map requires $N \cdot L$ bits—both of which must be built out of dedicated on-chip memory. We will restrict C to be a small constant number (e.g., 100) plus the length of one path—giving us $C = 100 + Z(L+1)$ blocks which amounts to tens to hundreds of Kilobytes in systems we evaluate. The position map, on the other hand, is very large because N grows linearly with the capacity of the ORAM. Thus, we must use a Recursive ORAM construction as described below.

3.2.2 Recursive Path ORAM

The $N \cdot L$-bit position map (Section 3.2.1.4) is usually too large to fit in a secure processor's on-chip storage. For example, a 4 GB Path ORAM with a block size of 128 bytes and $Z = 4$ has a position map of 93 MB. The Recursive Path ORAM addresses this problem by storing the large position map in an additional ORAM (this idea was first mentioned in [57]).

We will refer to the first ORAM in the recursive construction as the data ORAM or $ORam_1$. $ORam_1$'s position map will now be stored in a second ORAM: $ORam_2$ (which we will refer to as a *position map ORAM*). The secure processor's new on-chip storage requirement is the local cache for each ORAM and the position map for $ORam_2$. If $ORam_2$'s position map is still too large for on-chip storage, we can repeat the process with an $ORam_3$ or with however many ORAMs are needed. To perform an access to the data ORAM in a recursive construction of X ORAMs labeled $ORam_1, \ldots, ORam_X$, we first look up the on-chip position map for $ORam_X$, then perform an access to $ORam_X, ORam_{X-1}, \ldots, ORam_1$. That is, each ORAM lookup yields the leaf index for the next ORAM lookup.

To be concrete, we give an example with a 2-level Recursive Path ORAM. Let N_2, L_2, B_2, C_2, and Z_2 be the parameters for $ORam_2$ (variable names are analogous to those defined up to this point). Since the position map of $ORam_1$ has size $N \cdot L$ and each block in $ORam_2$ is able to store $k_2 = \lfloor B_2/L \rfloor$ labels, $ORam_2$'s capacity in data blocks must be at least $N_2 = \lceil N/k_2 \rceil + 1 \approx N \cdot L/B_2$ (where the $+1$ represents the extra dummy all-zero address). The number of levels in $ORam_2$ is equal to $L_2 = \lceil \log_2 N_2 \rceil - 1$.

Initially both ORAMs are empty and $ORam_2$'s position map is full of random leaf labels (as in Section 3.2.1). The Recursive Path ORAM maintains that, when some program address u has an associated data block b written in $ORam_1$:

1. $\forall$ leaves l in $ORam_1$, $\mathsf{Prob}(b \text{ is mapped to } l) = 1/2^L$.

2. b is either in some bucket along the path to leaf l in $ORam_1$'s tree, or in $ORam_1$'s local cache (the Path ORAM invariant).

Furthermore, if the above holds for u, then there exists a block in $ORam_2$ (call it b_2) such that the block's address is $u_2 = \lfloor u/k_2 \rfloor$ and the value at leaf offset $i = u - u_2 k_2$ in b_2 is l. The invariant for $ORam_2$ is that:

1. $\forall$ leaves l_2 in $ORam_2$, $\mathsf{Prob}(b_2 \text{ is mapped to } l_2) = 1/2^{L_2}$.

2. b_2 is either in some bucket along the path to leaf l_2 in the $ORam_2$ tree, or is in $ORam_2$'s local cache (the Path ORAM invariant).

Given the above invariants, the following algorithm describes a complete 2-level Recursive ORAM access $\mathsf{accessRORAM}(u, op, b')$:

1. Generate random leaf labels l' and l'_2. Determine i and u_2 as described above.

2. Lookup $ORam_2$'s position map with u_2, yielding l_2.

3. Perform $\mathsf{accessPath}(u_2, l_2, l'_2, write, b'_2)$ on $ORam_2$, yielding a block b_2 (as described in the invariant). Record l, the leaf at offset i in b_2, then replace l in b_2 with l', yielding b'_2.

4. Perform $\mathsf{accessPath}(u, l, l', op, b')$ on $ORam_1$. This will complete the operation.

We define $\mathsf{accessPath}()$ in Section 3.2.1.

To give the reader an idea, a preliminary paper on Ascend [50] reports 5880 clock cycles per $\mathsf{accessRORAM}()$ operation (using hardware assumptions from Section 3.2.1.3). In our evaluation, we assume (using optimizations from throughout this chapter) a 3090 cycle access latency (Section 7.2.2).

3.2.3 Optimization: Early Completion

Although the Path ORAM $\mathsf{accessORAM}()$ operation requires a path to be read *then* written, the program running within Ascend can start making forward progress as soon as the block of interest is read into the local cache (i.e., during Step 3 in $\mathsf{accessORAM}()$). In this case, the path writeback operation happens "in the background." In fact, since the block requested by the program may be in the local cache when the ORAM access begins, or may be read into the local cache before the last bucket is read, the program may be able to continue even sooner.

We refer to these optimizations as “early completion.” Early completion does not impact security over the I/O channel because from the server’s perspective, accessORAM() has not changed: forwarding data to Ascend’s instruction pipeline happens internally. For the same reason, early completion techniques only improve Path ORAM access *latency*, not throughput. For example, suppose two ORAM accesses are needed for two blocks b_1 and b_2 and that both blocks reside in the ORAM local cache when the first access is made. Through early completion, the first access will finish almost instantly. Yet to make Ascend’s behavior independent of this outcome, the access to b_2 can only start after all steps for accessORAM() complete for b_1.[7]

Early completion can also be applied to the Recursive ORAM construction (Section 3.2.2). In accessRORAM(), the block of interest is somewhere on the path in the data ORAM. Recall that each position map ORAM must be accessed before the data ORAM can be accessed. To get a benefit from early completion, the read path operation for each ORAM is performed (Step 3 in accessORAM()) before a single writeback operation (Step 5), for any ORAM, is performed. Once the data ORAM’s path is read, each ORAM performs its writeback operation. Thus, early completion can reduce the Recursive ORAM’s access latency by half.

3.3 Path ORAM Failure Probability

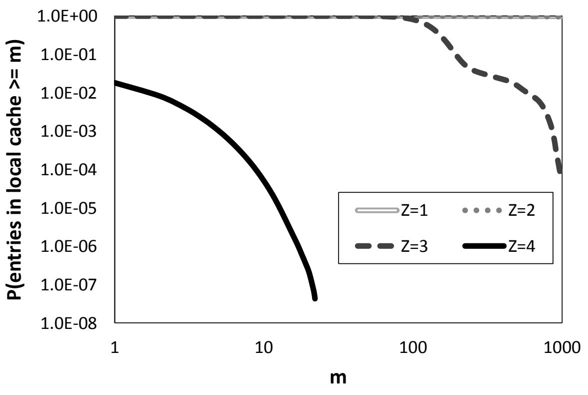


Figure 3-4: The probability that the number of entries in the local cache exceeds a certain number (m), for different Z.

In this section, we will develop *background eviction* schemes that eliminate the Path ORAM’s chance to fail and discuss different attacks that can be made on background eviction schemes in general.

ORAM schemes in general and Path ORAM in particular have the problem that they fail with a non-zero probability. Path ORAM has a chance to fail because the ORAM interface’s local cache has finite capacity and each access remaps the block of interest to a new path (Section 3.2.1). Figure 3-4 shows the local cache occupancy of different Z values for a 4 GB Path ORAM with a 2 GB working set, assuming an infinitely large local cache. To plot each curve, we modeled a Path ORAM making accesses to random program addresses and recorded the number of blocks in the local cache after each access. For $Z \leq 2$, the local cache always has more than 1000 entries. For $Z = 3$ the probability that local cache has more than 1000 entries is $\sim 10^{-4}$. These results indicate that to support $Z \leq 3$, the

[7]The astute reader may ask whether performing accessORAM() is necessary in the case when b_1 is in the local cache. That is, the local cache can be scanned prior to any buckets being requested which will prevent the server from finding out that an access was to be made in the first place. When considering the entire Ascend system, this trick can only be used if ORAM is nevertheless accessed at data-independent times (Section 3.4).

ORAM interface needs a very large local cache. This problem can be alleviated by making $Z \geq 4$, in which case failure is possible but extremely unlikely. The trade-off for larger Z is performance: as seen in Equation 3.1, the number of bits moved per access increases (almost) linearly with Z.

Since smaller values of Z lead to performance improvements but increase the chance of local cache overflow, a natural question is whether one can keep a small Z and have the ORAM interface perform a *local cache eviction* to clear out blocks when a local cache overflow is in danger of occurring. Referred to as *background eviction*, such schemes reduce the probability of Path ORAM failure to 0% and simultaneously allow for $Z \leq 3$ ORAMs with a small local cache.

3.3.1 Properties of and Attacks Against Background Eviction Schemes

We will now discuss background eviction schemes in general and how a naïvely constructed scheme can cause security problems. Suppose that for any batch program P given any given input M, the program address request sequence for $P(M)$ is $Q(P(M))$, and the corresponding sequence of external memory requests[8] for $P(M)$ is $\mathsf{PORAM}(Q(P(M)))$. A background eviction scheme is a scheduler $\mathcal{S}$, that uses some scheduling algorithm—*but only has access to the number of blocks in the local cache at any given time*—to create a new sequence of external memory requests $\mathsf{EVICT}(Q(P(M)))$, which are interleaved with $\mathsf{PORAM}(Q(P(M)))$ on the I/O channel to avoid local cache overflows (call this combined sequence $\mathsf{BEPORAM}(Q(P(M)))$).

By extending the ORAM security definition, security breaks if $\exists P', M', z$ such that $\mathsf{BEPORAM}_z(Q(P(M)))$ and $\mathsf{BEPORAM}_z(Q(P'(M')))$—i.e., the first z elements of each sequence—are computationally distinguishable. A key insight with Ascend and Path ORAM is that local cache occupancy (and by extension, the behavior of $\mathcal{S}$) depends on P, M and the Ascend hardware architecture given P and M. Thus—even if $\forall P', M', z$: $\mathsf{EVICT}_z(Q(P(M)))$ is computationally indistinguishable from $\mathsf{EVICT}_z(Q(P'(M')))$ and $\mathsf{PORAM}_z(Q(P(M)))$ is computationally indistinguishable from $\mathsf{PORAM}_z(Q(P'(M')))$—the server can write a curious program that tries to influence the local cache occupancy if certain private conditions in the user's data are met, and then watch for how background eviction operations are interleaved with normal ORAM operations. Notice that we can avoid this attack by constructing $\mathcal{S}$ to not have access to local cache occupancy. For the same reason that security breaks, however, this change will prevent $\mathcal{S}$ from eliminating failure probability completely (unless $\mathcal{S}$ performs a naïve strategy such as scanning the entire ORAM). We will now discuss some examples of how local cache occupancy depends on program, data and chip design.

3.3.1.1 Address Sequence and Local Cache Occupancy

The program virtual address sequence issued to the ORAM interface impacts local cache occupancy due to the Path ORAM block remapping process. For example, consider that at ORAM access t, the local cache contains c blocks, where c can now be arbitrary. If, from that point on, the ORAM interface continually accesses the same block b repeatedly, the local cache occupancy after any number of accesses is guaranteed to be $\leq c + 1$. This is because the only block in the path that can get stuck in the local cache is b. On the other hand, if m distinct blocks $b_1, b_2, \ldots, b_m$ are accessed, the local cache may contain as

[8]Each external memory request is a read/write to a single location in external memory that is fully visible to the server on the I/O pins.

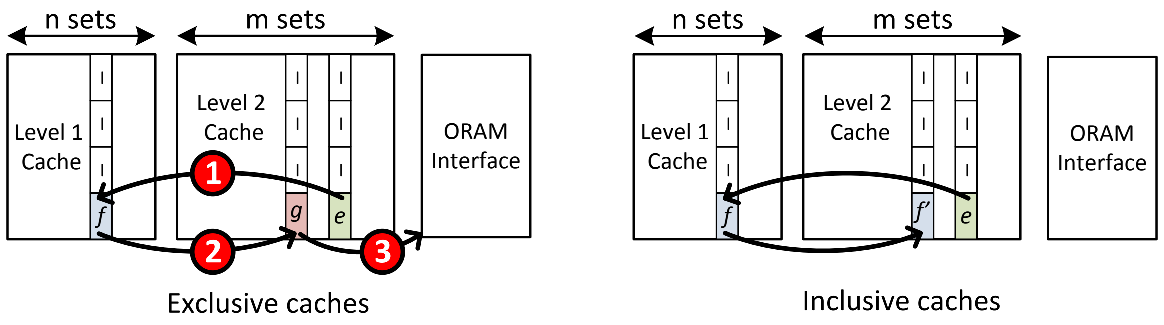


Figure 3-5: Inclusive vs. exclusive behavior for a 4-way L1/L2 cache hierarchy (typically $n \ll m$). '$-$' indicates a valid data block in the cache. The exclusive hierarchy can hit in the L2 cache yet still evict a block g to the local cache.

many as $c + m$ blocks. Thus, the server can create a curious program that will increase the likelihood of a background eviction by conditionally performing a scan through memory.

3.3.1.2 Chip Architecture and Local Cache Occupancy

Ascend's hardware architecture can impact local cache occupancy because of last-level cache eviction behavior. Suppose that Ascend is made up of an instruction pipeline, on-chip cache hierarchy and the ORAM interface as shown in Figure 3-1, and that the instruction pipeline/cache hierarchy behave as they would in a normal processor. When a data block is evicted from on-chip cache, the evicted block is moved to the ORAM interface's local cache where it will later be pushed to the ORAM tree. Hence, on-chip cache eviction causes a security problem: if the program causes an on-chip cache eviction, a block is added to the ORAM interface local cache asynchronously.

How many blocks can be evicted to the local cache per unit time differs depending on the on-chip caches' coherence policy. Traditional cache hierarchies can be inclusive, exclusive or non-inclusive (we discuss inclusive and exclusive hierarchies here). Inclusive caches maintain the *inclusivity property*: if some cache block b is present in some level (e.g., the level 1 cache, or L1), a (possibly stale) copy of b must be present in all the lower levels (e.g., the L2). In an exclusive hierarchy, it is guaranteed that b will not be present in the lower levels. In terms of background eviction security, inclusive vs. exclusive differs in the following way:

Inclusive caches only evict blocks to main memory (the ORAM interface) in the case of a last-level on-chip cache miss.

Exclusive caches may evict blocks to main memory due to a miss at any level of the on-chip cache hierarchy. Note that if the miss is not at the last-level cache, some lower cache level may still hit.

The difference security-wise between these cases is that an exclusive cache enables a curious program to add a block to the ORAM local cache once per on-chip cache access, whereas the inclusive hierarchy limits this eviction rate to one eviction per ORAM access. Since ORAM latency is orders of magnitude higher than on-chip cache access latency, the exclusive cache design allows a curious program to trigger background evictions with very high certainty at precise moments in time.

The attack on the exclusive design (illustrated on a 2-level cache hierarchy, see Figure 3-5) is performed in two phases: First, the curious program fills the L1 and L2 caches with blocks (which it can do because the server knows Ascend's cache organization). Suppose, at this point, that some block e is present in the (exclusive) L2 cache (Figure 3-5 (left)). The curious program will now make an L1 request for e (❶). Since the L1 set where e is to be mapped is currently full, another block (call this f) will be evicted from the L1 cache to make room for e (❷). Even though f and e map to the same cache set in the L1, they generally will not map to the same set in the L2.[9] Furthermore, since the L2 set that f is mapped to is full of blocks, f will evict another block (call this g) from the L2 to the ORAM interface (❸). Despite this, the exclusive hierarchy still hits in the L2 and the curious program will be able to make forward progress soon after (at which point it can repeat the process to force-evict another block).

On the other hand, if block f is evicted from the L1 in an inclusive hierarchy, there must be a (possibly stale) copy of f (call this f') in the L2 cache to satisfy the inclusivity property. When f is evicted, it will replace f' and this action will not add a block to the ORAM interface local cache. In fact, the only way that an inclusive hierarchy can add an evicted block to the local cache is if block e (the requested block) was in ORAM and could not be found in the on-chip cache hierarchy. In that case, the curious program must wait until e is fetched from the ORAM which prevents it from repeating its attack until the ORAM access completes. With the inclusive design, the on-chip cache to local cache pressure is limited which makes such attacks more difficult to mount.

3.3.2 Probabilistic Background Eviction

To prevent the attacks from the previous section, we propose a background eviction scheme where background eviction operations are indistinguishable from regular ORAM accesses. Conceptually, a background eviction performs a dummy ORAM access. Suppose the Path ORAM local cache can hold at most C blocks. To prevent local cache overflow, we stop serving real memory requests and instead issue dummy requests whenever the number of blocks in the local cache exceeds $C - Z(L+1)$.[10] A dummy access reads and decrypts a random path and writes back (after re-encryption) as many blocks from the local cache to the path as possible. No block is remapped on a dummy access, meaning (a) all the real blocks on that path can at least be written back to their original places and (b) the dummy access will at least not add elements to the local cache after the writeback. If the read path was not full of real blocks, there is a possibility that some blocks in the local cache will find places on this path. We keep issuing dummy accesses until the number of blocks in the local cache drops below the $C - Z(L+1)$ threshold.

Probabilistic background eviction can be easily extended to a Recursive Path ORAM. If the local cache of any of the ORAMs in the hierarchy exceeds the threshold, we issue a dummy request to each of the Path ORAMs in the same order as a normal access, i.e., the smallest Path ORAM first and the data ORAM last.

Table 3.1 shows the ratio of dummy to real accesses for the Z settings in Figure 3-4 with background eviciton, assuming $C - Z(L+1) = 100$. Background eviction effectively reduces the local cache size and prevents the overflow problem for $Z = 1, 2$ and 3 by introducing $7.81\times, 0.69\times$ and $0.015\times$ dummy accesses per real access, respectively. The downside of background eviction is that dummy accesses may hurt performance. The upside is that

[9] This is due to the L2 typically being larger and/or having a different associativity than the L1 cache.

[10] Recall: this leaves just enough space for a new path.

Table 3.1: The ratio between dummy accesses to real accesses for a 4 GB Path ORAM with 2 GB working set, varying Z.

Z=1	**Z=2**	**Z=3**	**Z=4**
7.81	0.69	0.015	0.0

small Z settings, which yield very good throughput but were previously prohibited due to high failure probability, are now possible. For instance, $Z = 3$ reduces the number of bits that are moved per ORAM access by $\sim 25\%$, relative to $Z = 4$ (Equation 3.1), and requires less than one dummy access per 50 real accesses.

3.3.3 Security of Probabilistic Background Eviction

Recall that the original Path ORAM scheme (with an infinite local cache and no background eviction) is secure because, independent of the memory requests, the server will observe a sequence of random paths being accessed (for each Path ORAM in a Recursive Path ORAM). Denote the sequence as

$$\mathsf{PORAM}(Q(P(M))) = \{p_1, p_2, \ldots, p_k, \ldots\},$$

for any program P and input M, where p_k is the path that is accessed on the kth memory access. Each p_k for $k > 0$ follows a uniformly random distribution and is independent of p_j $\forall j > 0$ $s.t.$ $(k \neq j)$, also in the sequence. Background eviction interleaves, in an arbitrary manner, another sequence of random paths q_m for $m > 0$, producing a new sequence

$$\mathsf{BEPORAM}(Q(P(M))) = \{p_1, p_2, \ldots, p_{k_1}, q_1, \ldots, p_{k_2}, q_2, \ldots\}.$$

where q_m $\forall m > 0$ is chosen from a uniformly random distribution. In particular, q_m is independent of p_k $\forall k > 0$ and q_n $\forall n > 0$ $s.t.$ $(n \neq m)$. Thus, $\forall P, M, z$: $\mathsf{BEPORAM}_z(Q(P(M)))$ follows the same uniformly random distribution as $\mathsf{PORAM}_z(Q(P(M)))$, and thus is indistinguishable from $\mathsf{PORAM}_z(Q(P(M)))$. Subscript z means the first z elements in the sequence as before. Crucially, how background evictions are interleaved with normal ORAM accesses does not impact security.

Livelock. Our background eviction scheme has a very low probability of *livelock*. Livelock is a pathological situation and different than ORAM failure: Suppose that the ORAM tree is in an arbitrary state and that the local cache contains $c = C - Z(L+1)$ blocks (i.e., just enough space to fit a new path from the ORAM tree). In this case, the ORAM interface must perform background evictions until the local cache contains $< c$ blocks. We say that the background eviction scheme has *livelocked* if no number of background evictions can decrease the local cache occupancy to $< c$ blocks. If this happens, the ORAM will effectively perform background evictions forever and no longer be able to do useful work. We note that this doesn't impact security because background evictions are indistinguishable from real ORAM accesses.

Clearly, the probabilistic background eviction scheme can livelock if every block in the local cache is mapped to leaf l, and the path to leaf l is already full of blocks that also map to leaf l (recall that this was beginning to happen to leaf 1 in Figure 3-3). Since this livelock is extremely unlikely, however, our background eviction scheme works well in most

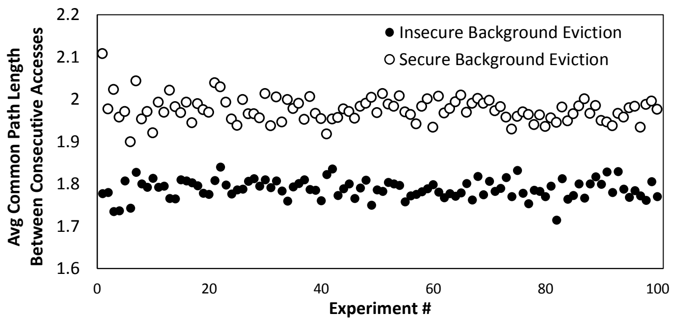


Figure 3-6: Average Common Path Length (CPL) between consecutively-accessed paths with the insecure eviction scheme and our secure background eviction scheme. This attack compromises the insecure eviction scheme.

cases.

3.3.4 Insecurity of Block Remapping Eviction Schemes

We point out that attempts to eliminate livelock are very dangerous security-wise. Consider the following scheme (called the block remapping scheme): *when the number of blocks in the local cache reaches a threshold, we choose a block already present in the local cache at random and perform a normal ORAM access for that block.* This scheme will not livelock because the block in question will get *remapped* on each access and eventually 'escape' congested paths. In fact, this scheme is more efficient (issues less dummy accesses) than our proposed background eviction. The reason may be that the blocks that remain in the local cache are more likely to have been mapped to some congested paths, and this scheme helps these blocks escape the congested paths. Unfortunately, this is also the reason why security breaks, as we show below.

We first define $\mathsf{CPL}(p, p')$ as the Common Path Length of path p and p', which is the number of buckets shared by the two paths.[11] Using Figure 3-2 as an example, $\mathsf{CPL}(1,2) = 3$ and $\mathsf{CPL}(3,8) = 1$. Given a ORAM tree of $L+1$ levels, if p and p' are drawn from a uniform distribution, we have

$$\mathsf{Prob}\left(\mathsf{CPL}(p,p') = l\right) = \frac{1}{2^l},\ 1 \le l \le L, \tag{3.2}$$

$$\mathsf{Prob}\left(\mathsf{CPL}(p,p') = L+1\right) = \frac{1}{2^L}, \tag{3.3}$$

$$\mathsf{Expected}\left(\mathsf{CPL}(p,p')\right) = 2 - \frac{1}{2^L}. \tag{3.4}$$

For the sequence $\mathsf{PORAM}(Q(P(M)))$ of paths accessed without eviction, if we average $\mathsf{CPL}(p_k, p_{k+1})$ for $k = 1, 2, 3, \ldots$, we will get a value very close to $2 - \frac{1}{2^L}$, since each p_k follows an independently uniform random distribution.

For the sequence $\mathsf{BEPORAM}(Q(P(M)))$ derived from the block remapping eviction scheme, each q_m (an access for eviction) is the leaf label of a block u_m that is in the local cache at that point. *Note that a block mapped to path q_m is less likely to be evicted to the ORAM tree if the paths accessed before it share a shorter common path with q_m.* So if this eviction scheme is used, averaging the common path length of consecutive paths accessed

[11] That is, $\mathsf{CPL}(p,p') = |\mathsf{CP}(p,p')|$ as defined in Section 3.2.1.

in $\mathsf{PORAM}(Q(P(M)))$ will yield a result that is significantly smaller than the expected value $2 - \frac{1}{2^L}$. We mount this attack 100 times on a Path ORAM with $L = 5$, $Z = 1$ and $C - Z(L+1) = 2$. Figure 3-6 shows that the two schemes produce different path sequences. For our probabilistic background eviction scheme, the average CPL value is 1.979, which is very close to the expected value $2 - \frac{1}{2^L} \approx 1.969$. For the insecure scheme, the average CPL value is centered around 1.79. So our attack is able to detect evictions when using the block remapping scheme.

There are other block remapping eviction schemes that are less easy to break. For example, an eviction scheme can randomly remap one of the blocks in the local cache and then access a (different) random path. Then there would be no dependency between consecutive accesses and the above attack is defeated. However, such a scheme still tends to remap blocks from congested paths to less congested paths, and can be broken by more sophisticated attacks.

3.4 Time Obfuscation: Public Intervals

Obfuscating the I/O channel in the time domain is straightforward with public intervals because the I/O channel's behavior is fully determined by the ORAM interface's behavior. The simplest secure scheme is for the ORAM interface to be set to a single interval—referred to henceforth as I_{ORAM}, where $I_{ORAM} \in \mathbb{P}$—as described in Section 2.4.2.1. In this case, the ORAM interface will make exactly one real/dummy ORAM access per interval.[12] When an ORAM access completes, the ORAM interface drives the I/O pins to a fixed logic value and does not respond to additional requests made by Ascend's on-chip caches until the next ORAM access. The server may set I_{ORAM} by estimating the number of cycles that the given program will need between two accesses to external memory, given Ascend's on-chip resources and apriori knowledge of that program.

Note that background eviction operations in our secure scheme (Section 3.3.3) are indistinguishable from dummy accesses. This property makes our background eviction scheme especially well suited for Ascend. If Ascend makes a dummy access because the program does not need ORAM at that time, that dummy access also performs the job of background eviction. This decreases the overhead of background eviction for programs that are not memory bound. Note that if a background eviction is needed, it must wait for the next interval like any other access.

Denial of service. Since making ORAM requests entails sending/receiving data to/from the outside world (i.e., no longer under Ascend's control), the I/O channel must also protect against denial of service attacks. That is, Ascend behavior should be independent of the amount of time it takes the external memory to return data (on a read) or complete a write. Of course, variable latency to main memory may also be the result of benign congestion on a network. We point out that our definition of public intervals—specifically that the next ORAM access is made I_{ORAM} cycles after the previous ORAM access completes—protects against privacy leakage through denial of service attacks with-

[12] If Recursive ORAM (Section 3.2.2) is being used, each ORAM access will be broken up into multiple ORAM accesses. This does not change the security of the single-interval scheme: when the ORAM interface triggers an ORAM access, it will always trigger a fixed sequence of ORAM accesses until the Data ORAM is accessed. A more sophisticated interval scheme can create a separate interval for *each* ORAM in the recursive construction, and keep a small table (similar to a page table) available for accessing recently used blocks in the position map ORAMs. The insight is that each block in the position map is used to lookup a range of program addresses and that such blocks can have page-level locality.

out change. Suppose that at time t Ascend begins a new ORAM access. If ORAM has an access latency of t_{access} cycles and the server forces the request to take d more/less cycles, Ascend will always perform the next ORAM access at cycle $t + t_{access} + d + I_{oram}$. Since all of these quantities are public to the server, the server learns nothing that it wouldn't have been able to determine apriori.

Chapter 4

Protecting the Power Channel

Our goal in this chapter is to develop and optimize primitives that make the *analog* signals on Ascend's power pins not leak any information that depends on the user's private data, thereby making Ascend resistant to power analysis-based attacks. Using power analysis to break secure cryptographic implementations gained notice in the late 1990s when DES implementations were shown to leak secret keys through their power traces [11]. Since then, there have been numerous successful attacks based on power analysis and proposed counter measures [15, 22, 21, 25, 29, 36].

Program 3 Curious programs that leak through the power channel. `M` is the private user memory.

```
int J = 1; int K = 2;
void curious_dpa(M) {
  int ci = 1 + M[J];
  int di = 1 + M[K];
}
```

```
int I = 0; // chosen by the server
void curious_spa(M) {
  int ai = M[0];  float af = (float) ai;
  if (M[I] & 0x1) float cf = 1.0f + af;
  else            int ci =   1 + ai;

}
```

When monitoring the power pins, the server sees an aggregate signal that represents the instantaneous power consumption of all the circuits in the chip.[1] As with the I/O channel, how each circuit is activated (i.e., the bit values of its inputs, its internal state, etc) and when it is activated (i.e., when the clock that drives the circuit is running) both influence pin behavior. We distinguish between a circuit's inputs and the clock signal for technological reasons (Section 4.2).

In the literature, detecting *how* a circuit is activated is done through differential power analysis (DPA) and detecting *when* that circuit is activated is done through simple power analysis (SPA) [11]. To perform DPA, the server collects many power traces[2] and tries to

[1]In actuality, modern processors expose many power pins. Typically, each pin connects to an array of wires that span the chip and supply V_{dd} to every circuit. Each power pin may carry a different signal based on the where each circuit is located relative to the pin. We will not make assumptions about which pins the server is monitoring.

[2]For example, the server may apply different public inputs to an encryption circuit to try and deduce some hidden state (the key). The server can construct a curious program to do exactly this in an Ascend setting.

detect correlation using statistical techniques. For example, the `curious_dpa()` program (Program 3 (left)) has the same control flow regardless of `M`. Even so, the server can learn whether `M[J]` and `M[K]` are different by observing small variations in the power draw from the adder circuit over different executions. This type of leakage happens because a CMOS circuit's input values and transition behavior both impact power consumption.[3] With SPA, the server examines a power trace to determine which circuits are activated to perform a computation and when those circuits are activated. For example, the server can create the `curious_spa()` program (Program 3 (right)), which conditionally activates either the floating point unit or a fixed point arithmetic unit based on user inputs. Unlike `curious_dpa()`, the SPA-based attack requires very few samples (perhaps a single sample) because the difference in power consumption between the floating point and fixed point units can be large. Many variations of the attack are possible; the server just has to activate different circuits with distinct power profiles.

We note that most symmetric-key cryptographic circuits like DES or AES are usually attacked through DPA as opposed to SPA. This is not because SPA is ineffective in general, but rather because these circuits are relatively hardwired and perform a similar set of operations regardless of the key (e.g., AES performs a public number of rounds, and the same mixing operations per round), making SPA less effective [11]. Both SPA and DPA are effective in our context of untrusted programs, which may activate different circuits based on private user inputs (e.g., `curious_spa()`).

4.1 Chapter Overview

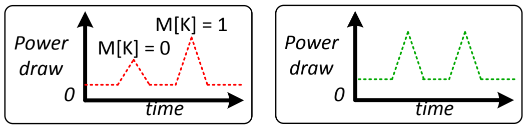
This chapter is structured in a similar way as Chapter 3. To begin, Section 4.2 introduces DPA resistant circuits: a set of primitives that make a circuit give off a data-independent power draw when that circuit is activated, thereby obfuscating the power channel in the value domain. This section is analogous to ORAM for the I/O channel. Our goal will be to create a complete cell library which can be used to build the rest of the Ascend chip. Throughout the discussion, we will make optimizations to the DPA resistant logics that are suited for general-purpose processors. Section 4.3 discusses obfuscation in the time domain. Since the power channel is impacted by many circuits simultaneously—as opposed to the I/O channel which is fully specified by the ORAM—obfuscating the time domain for the power channel is significantly more involved. In addition to public intervals we introduce a new type of public parameter, that is specific to the power channel, called the public circuit policy.

4.2 Value Obfuscation: DPA Resistant Circuits

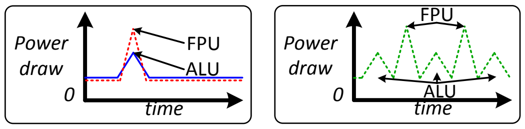
To prevent DPA, we will build Ascend entirely out of DPA resistant circuits, preventing attacks posed by programs like `curious_dpa()`, as shown in Figure 4-1a. At a high level, if a particular circuit is built out of DPA resistant logic, the server will not be able to determine the internal state of that circuit if that state could not have been determined apriori.

DPA resistant logics are typically based on dual-rail (differential) logic [15, 22] or bit masking techniques [21]. More recent proposals have combined the two [25, 29]. The big

[3]For example, circuit glitches can cause fewer or more circuit transitions based on circuit inputs. In the case of `curious_dpa()`, `1 + M[J]` will provoke additional transitions if the low bit in `M[J]` is equal to 1, due to adder carry logic.



(a) (Left) A power trace from running `curious_dpa()` (Section 4). The server can determine the value of `M[K]` by watching the power consumption of the adder. (Right) The same program when Ascend is built out of DPA resistant logic.



(b) (Left) A power trace from running `curious_spa()` (Section 4). The server can see if the floating point unit (FPU) or arithmetic unit (ALU) was activated. (Right) The same program, where ALU/FPU accesses are made in a strict data-independent interleaving.

Figure 4-1: DPA and SPA resistance.

idea with dual-rail logic is to make circuit elements have data-independent power draws. Regardless of whether a circuit transitions or has different inputs/output values, the same amount of power should be dissipated. Bit masking techniques instead apply a random mask bit that changes randomly each cycle to decorrelate the input signal with the output signal. For both techniques, the server learns that a particular circuit was activated but not what inputs/outputs were sent to/from that circuit or whether the circuit underwent a logic transition. Both techniques are at the digital logic level and do not necessitate program-level transformations.

Ascend can be built with any DPA resistant logic, in principle. To get a circuit library that is sufficient to build the entire chip, we first need DPA resistant logic gates and flip-flops. Gates and flip-flops are used to build processor pipelines, control logic and small memories throughout the chip. Second, we need DPA resistant SRAM cells, which will be used to implement on-chip cache and register files.

4.2.1 Logic Gates and Flip-Flops

We will use a dual-rail technique—wave dynamic differential logic (WDDL) [22]—to describe Ascend's logic gates and flip-flops because WDDL can be built on top of normal ASIC standard cells and even be implemented on an FPGA. WDDL specifies how to build DPA resistant logic gates and flip-flops: Each insecure logic gate can be replaced with a functionally equivelent secure version that has a *data-independent power draw*. WDDL flip-flops, on the other hand, hide the value they are currently storing and when that value changes.

Figure 4-2a depicts a sequential parity calculator that takes a stream of bits as input and calculates the running parity. Unshaded gates/flip-flops represent the insecure parity checker; shaded circuits represent additional circuits added to support WDDL. Both shaded and unshaded circuits are ordinary CMOS circuits. In this example, the circuit surrounded with a red dashed box is considered the secure circuit; everything outside may be insecure (i.e., not built using WDDL). Suppose the server/adversary controls the `in` signal. The adversary's goal is to deduce the initial value on wire `p2`, with probability better than guessing, by monitoring the secure circuit's power pins. We now explain the salient design principles in WDDL and how WDDL prevents the server from learning `p2`:

1 **(see Figure 4-2a) Compound gates.** Each insecure logic gate is paired with its complement as given by De Morgan's law, forming *compound gates*. For example, an

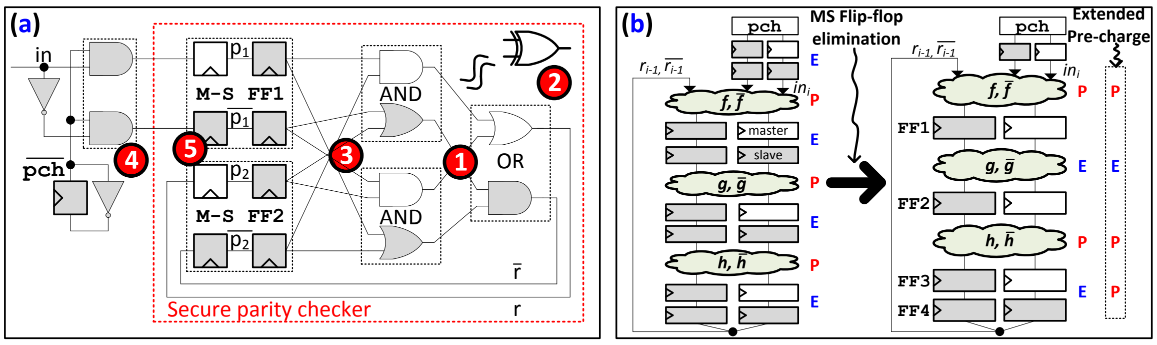


Figure 4-2: (a) A sequential parity checker built using WDDL techniques. Grey shaded circuits are added to implement WDDL: an insecure version requires only the unshaded circuits; further the insecure XOR can be implemented directly as an XOR gate. Each individual circuit is an ordinary/insecure CMOS circuit; all flip-flops are assumed to be positive edge-triggered. (b) A 3-stage reduction operation in WDDL optimized for performance/area/power using the techniques in Sections 4.2.1.1-4.2.1.3.

insecure AND gate becomes an AND gate and an OR gate, where the OR gate's inputs are inverted. This is done to hide whether the output of each insecure gate is logic 0 or 1: each compound gate will always output the pair 1,0 or 0,1. We refer to compound gates as having real,complement outputs and the notation 1,0 means the real output = 1. Because power signals are aggregated, WDDL assumes that the server will not be able to tell which outcome happens (i.e., $1 + 0 = 0 + 1 = 1$ in both cases). This assumption holds when the power consumption of interconnect wires dominates, which as [22] points out becomes more true as the silicon process shrinks.

2 **Positive gates.** Boolean networks must be made up of *positive* compound gates only—that is AND gates and OR gates. For example, in the figure we have replaced the XOR gate with the functionally equivalent two AND gates and an OR gate. This is done to avoid glitches, which reveal circuit inputs: *WDDL maintains the invariant that each gate transitions at most one time in a given clock cycle.*[4]

3 **Implementing inverters.** To implement inverters (NOT gates), WDDL crosses inverted inputs (which are always available through compound gates). Inverter gates need not be implemented directly.

4 **Circuit Pre-charge.** WDDL circuits switch between a *pre-charge* phase and *evaluation* phase every other cycle. During the pre-charge phase, all outputs to all flip-flops are pulled to 0, which in turn causes all compound logic gates to output 0,0. During the evaluation phase, each pair of flip-flops outputs either 1,0 (logic 1) or 0,1 (logic 0) which the compound gates evaluate to produce the next state. Adding a pre-charge phase is necessary to hide whether the logic value output by each compound gate *changes* between two consecutive cycles. That is, the aggregated signal in each evaluation phase is $1 + 0 = 0 + 1 = 1$ and $0 + 0 = 0$ in each pre-charge phase, regardless of the input bit stream. Thus, from the server's perspective, each compound gate transitions every cycle regardless of the input bit stream sent to the secure circuit ([22] refers to this as every compound logic gate having

[4]XOR does not satisfy this property. Suppose that the inputs to an XOR gate toggle one time each per cycle, but that the toggles happen at different moments in the cycle; e.g., 0,0 → 0,1 → 1,1. In this example, the XOR's output transitions from 0 → 1 → 0. This is a glitch that can reveal the XOR's inputs to the server.

100% switching factor). Pre-charge requires that the server send new inputs to the secure circuit at half the original rate (pre-charge does not do useful work).

❺ **“Master-slave” flip-flops.** Each flip-flop in WDDL is converted into a compound “WDDL master-slave[5] flip-flop” that can be built using four insecure CMOS flip-flops. The construction works as follows: First, each insecure flip-flop must be copied (vertically in the figure) to store the De Morgan’s complement value. Second, each flip-flop pair must be copied (horizontally: forming a master and slave). WDDL does this to maintain the following invariant: *during each cycle, each logic stage (separated by flip-flops) must have all of its inputs in the pre-charge phase or all of its inputs in the evaluation phase.* In our example, if p_1, $\overline{p_1}$ is in the pre-charge phase: p_2,$\overline{p_2}$ is in the pre-charge phase and r,$\overline{r}$ is in the evaluation phase (and vice versa). On the other hand, if either the master or slave flip-flops are removed from `M-S FF2`, $p_2 = r$,$\overline{p_2} = \overline{r}$. In that case, it is impossible to interleave pre-charge and evaluation.

We now discuss the overheads in complementary pre-charge logics like WDDL:

Area and power. Since each WDDL gate requires the original gate as well as a complementary gate, WDDL incurs at least $2\times$ area overhead. Furthermore, each flip-flop is replaced by four flip-flops to implement the master-slave design. [22] estimates that the power overhead correlates to the area overhead: in evaluating AES and DES test circuits, [22] reports a $3-4\times$ area/power overhead for an ASIC and $6\times$ for an FPGA. Note that generating the pre-charge signal incurs negligible overhead and is similar to generating a no-op to the secure circuit every other cycle.

Performance. Combinational delay for a logic network in WDDL increases when non-positive gates such as XOR are needed (i.e., WDDL implements XOR using AND and OR as shown in Figure 4-2a). Pairing each gate with its complement may also complicate routing which will lengthen the critical path wire delay. Furthermore, throughput decreases by half and cycle latency doubles, because of circuit pre-charge and the master-slave flip-flop design. The decrease in throughput is fundamental to pre-charge logics for security reasons: a WDDL circuit may never perform two evaluation operations in two consecutive cycles to hide real logic transitions. We will use an optimization to reduce the latency penalty in Section 4.2.1.1.

Security-wise, DPA resistant logics are evaluated using normalized energy/standard deviation (NED/NSD) metrics, which indicate how power consumption varies over the set of possible inputs to the circuit. [22] reports that WDDL decreases NED/NSD by $50\times$, which is roughly $2\times$ less effective than a similar technique that requires full custom cells [15].

WDDL has several possible vulnerabilities. [28] reports that place and route tools can cause leakage because of uneven differential wire lengths. That is, if the real and complement outputs for a given WDDL compound gate are routed differently, the capacitive load for the gate’s 1,0 and 0,1 states will not be the same. [32] reports that WDDL may also be able to leak through the so-called early propagation effect.

4.2.1.1 Optimization: Flip-flop Elimination

To reduce the cycle latency and area overhead in WDDL constructions, we point out that the master-slave flip-flop construction is unnecessary in cases where *by removing the slave flip-flops the invariant stated in* ❺*, above, is not violated.* Stated again: it must be the

[5] “Master-slave” is WDDL terminology: the same (not inverted) clock signal is sent to each flip-flop.

case that for every logic stage in every clock cycle, all inputs to that stage must be in *either* pre-charge or evaluation mode.

Consider a circuit that computes the logic function $r_i = h(g(f(r_{i-1}, in_i)))$—a reduction over an input stream for $i = 0, \ldots,$ where f, g, h are arbitrary combinational functions. We will focus on optimizing this form of reduction circuit over the next several sections because many general-purpose processor components (e.g., pipelines and on-chip caches) share its structure. We show the corresponding WDDL circuit construction in Figure 4-2b (left). Pipeline stages in pre-charge are marked **P** and evaluation stages are marked **E**. Stages change phase every cycle. To make the example compelling, we say that f, g and h have substantial combinational delay—one would like to pipeline these circuits to achieve a better design clock frequency.

Pipelining has limitations with WDDL, however, because an additional pipeline stage introduces *two* cycles of latency due to the master-slave flip-flop design. Furthermore, only one of the two cycles can perform useful work. On the one hand, we cannot eliminate each slave flip-flop in Figure 4-2b (left) because this causes the $r, \overline{r}$ input to f to be in pre-charge when the in_i input is in evaluation (and vice versa). An insight is that we can eliminate *most* of the slave flip-flops, by padding the circuit such that its feedback path is an *even number of pipeline stages in length*.[6] We point out that baseline WDDL accomplishes this trivially: since each master-slave flip-flop is two flip-flops in series, every path has an even length in all cases. After eliminating slave flip-flops as shown in Figure 4-2b (right), our circuit has a 4 cycle latency instead of 6 and requires 10 flip-flops instead of 16.[7]

We note that eliminating flip-flops in this way complicates an automatic tool's job when transforming an insecure circuit into its WDDL equivalent. Indeed, the master-slave design implies that the circuit designer need not change RTL (register-transfer level) design before converting that RTL into WDDL.

4.2.1.2 Optimization: Extending Pre-Charge

To reduce the power consumption for the baseline WDDL construction we point out that interleaving evaluation with pre-charge, every other clock cycle, is an overly strong condition to maintain security. For a circuit to be secure, it is sufficient to meet the following conditions:

1. The invariant from 5 (see previous section) must be maintained.

2. No pipeline stage can be in the evaluation (**E** phase) for two consecutive cycles. If this condition is violated, the server can learn whether (secret) logic values changed between the two cycles.

3. Each pipeline stage must be in the evaluation and pre-charge phase at data-independent times. This is because the server can measure whether a stage is in **E** or **P** at a given time.

Note that properties 2 and 3 are trivially guaranteed by baseline WDDL: pre-charge and evaluation toggle every other cycle and this toggle depends only on the clock signal.

[6] This is *similar* to saying "the pipeline stages that make up the sequential circuit must be two-colorable." We will show how WDDL differs from this definition in Section 4.2.1.2.

[7] Note that applying this optimization to the parity checker (Figure 4-2a) does not improve that design's cycle latency, since it has a feedback path length of 2.

We can reduce the dynamic power consumption for WDDL circuits by *extending* pre-charge until the circuit is able to accept a new input, if these times are data-independent. The idea is shown in Figure 4-2b (right): it is *publicly known* that after a valid input in_i is presented to f (i.e., f is in the **E** phase), $h(g(f(r_{i-1}, in_i)))$ will be computed *exactly* four cycles later. Thus, the pre-charge circuit at the input (labeled `pch`) can hold the input in pre-charge for the intermediate three cycles and create a new **E** bubble every *fourth* cycle. Extending pre-charge is built on the insight that dynamic power consumption in CMOS logic correlates to the number of times a wire or logic gate toggles. If a given pipeline stage is in **P** for two consecutive cycles, it will not consume dynamic power during the second cycle. For the example in Figure 4-2b (right), only two stages transition between pre-charge and evaluation per cycle, regardless of the circuit's pipeline depth. In baseline WDDL, every pipeline stage transitions every cycle.

Security-wise, the extended pre-charge scheme still has data-independent switching factor. If between two consecutive clock cycles, any pipeline stage transitions between **E**$\rightarrow$**P** or **P**$\rightarrow$**E**, each compound gate has 100% switching factor and is equivalent to baseline WDDL (Section 4.2.1). If a stage transitions between **P**$\rightarrow$**P**, we have 0% switching factor but we have restricted this to happen at data-independent times. Lastly, we design the circuit such that no stage will ever have a **E**$\rightarrow$**E** transition—completing the case analysis.

Performing an extended pre-charge trades off throughput for energy. By definition, the more cycles the secure circuit is in pre-charge, the less attainable throughput the circuit has. The $h(g(f(r_{i-1}, in_i)))$ circuit is a good candidate for our optimization when one input stream is being applied to the circuit at a given time. In this case, throughput does not matter. On the other hand, if multiple input streams were interleaved, we would want to revert back to baseline WDDL for throughput reasons. We will use extended pre-charge in later sections for making processor on-chip cache accesses more efficient—a design decision based on two insights: First, latency (not throughput) is the primary design criterion for single-core caches (due to pipelines tolerating a small number of outstanding loads at a time). Second, caches (and the wires that make up the inter-cache bus) are a large source of energy expenditure in modern processors. As with baseline WDDL, extended pre-charging cannot make a WDDL circuit's throughput better than 50%: pre-charge is required after every evaluation phase for security reasons.

4.2.1.3 Optimization: Additional Flip-flop Elimination

If both optimizations from Sections 4.2.1.1-4.2.1.2 are applied simultaneously, we can remove additional flip-flops in the WDDL construction. Consider the optimized WDDL circuit in Figure 4-2b (right). Without the extended pre-charge optimization, we must add the pair of flip-flops labeled `FF4` as a pad to ensure that each pipeline stage is completely in pre-charge or evaluation at a given time (see invariant from ❺ in Section 4.2.1). When the extended pre-charge scheme is applied, however, `FF4` can be removed. This is because creating fewer evaluation stages per unit time makes it easier to satisfy invariant ❺: for instance, if the circuit were in pre-charge 100% of the time, the invariant from ❺ would trivially hold for all circuits without padding paths or master-slave flip-flops, etc. The resulting circuit has a cycle latency of 3 instead of 4 (which was the result from Section 4.2.1.1) and is made up of 8 flip-flops instead of 10.

4.2.1.4 Creating Dummy Work

By definition, WDDL creates a near-constant power draw regardless of the real logic values present in evaluation mode. Thus, a dummy signal applied during evaluation mode will be indistinguishable from a real signal. Traditional enable logic, passed down alongside each multi-bit signal, determines whether that signal is real or dummy, and prevents dummy state from erasing real state (e.g., by adding a WDDL multiplexer (mux) circuit at the input of flip-flops that need to conditionally update their state).

4.2.2 SRAM Arrays

WDDL and other similar logics only provide secure logic gates and flip-flops; we must assume different techniques to implement SRAM cells. Recent work has developed a DPA resistant SRAM cell using differential logic and pre-charge ideas [31, 36]. For our evaluation and for the rest of the Chapter, we assume the design from [31] and refer to this design as `cell-1`.

4.2.2.1 Secure SRAM Design and Sources of Leakage

[31] points out that SRAM read operations and address decoders do not contribute large data-dependent variations in power, even for insecure SRAM designs, when considering a single SRAM access in isolation. During a read operation, complementary column bitlines are pre-charged to the same logic value and, regardless of the logic value stored in each column of the open row, exactly one bitline per column discharges. Furthermore, regardless of which row is opened, the row address decoder takes $\log n$ bits (for an n row SRAM) as input and asserts exactly 1 out of n word lines (the column decoder behaves the same way). Putting these together, total charge transfer per access is independent of read address and the data read out. The observation in [31] is that *which* word line (is activated) and which bitline (is discharged per column) is very difficult to detect, as SRAM arrays have regular structure (e.g., all word lines have equal length).

We now summarize sources of leakage in an insecure SRAM array (taken from [31]). First, data-dependent leakage occurs during SRAM write operations because the logic value written to each SRAM cell may be the same or different than the old value stored in the cell. Which case occurs is data-dependent and creates a variation in power consumption. Second, accross multiple SRAM operations, charge gets trapped in the row/column address decoders, causing power consumption in the decoders to vary *across accesses* in data-dependent ways. The server may exploit these weaknesses by creating a curious program that repeatedly writes the same word to the same address in memory (for example).

It is possible to mitigate the above leakages using WDDL pre-charge-like ideas. First, before new data is written to each SRAM cell, the cross-coupled inverters that store the bit are brought to a public logic level (making charge transfer independent of write data, as was done for read operations). Second, the internal logic in the address decoder is brought to a public logic value before the next operation starts. Additional details can be found in [31]. The proposed design adds two PMOS transistors per SRAM cell and routes an additional wire down each SRAM column, which controls one of the PMOS transistors.

4.2.2.2 Secure SRAM Overheads

A 256 Byte (32×64) `cell-1`-based SRAM array[8]) was reported to decrease NED (normalized energy deviation) by $10\times$ in terms of what address and data is being accessed, but does not hide whether a request is a read or a write [31] or that some request is being made. Ascend needs to hide whether an access is a read/write. As with the I/O channel, the server can engineer a curious program that conditionally performs a read or write based on private user data. At a high level, one solution to this problem is to perform a read plus write operation for every access to the SRAM. A dummy read can be a read to any address; a dummy write can be a write with arbitrary data to a reserved (unused) location. An important detail implied by this setup is that dummy data takes up a fixed amount of space in the SRAM, no matter how many SRAM accesses do dummy work. We will describe SRAM-related management schemes in more detail when we discuss Ascend's microarchitecture (Sections 5.4.2 and 5.4.6).

Follow-up work to the `cell-1` design was performed in [54]. That work focuses on protecting data values on SRAM write operations (and not address/operation). Since Ascend needs to hide the address[9] as well, we will assume the original `cell-1` design for the rest of the thesis. We point out, however, that the work in [54] uses similar ideas as [31] and reports that the standard deviation in power consumption drops by four orders of magnitude using their techniques—a promising potential improvement over the `cell-1` design.

[36] reports that accessing a `cell-1` cell dissapates $2\times$ the energy of a conventional SRAM cell, per access, and that the 256-Byte SRAM array takes 70% more area than a conventional SRAM of the same dimensions. The authors do not say how much their design increases combinational delay. On the one hand, the `cell-1` design maintains the structure of a conventional SRAM, suggesting that insecure SRAM timing models will provide a good approximation for access latency. On the other hand, the `cell-1` design will add some amount of combinational delay because the SRAM cells activated along a row will have to be pre-charged to a known value before the write operation can complete. We will assume that throughput further drops by a factor of two, because SRAM arrays in Ascend will be controlled by WDDL logic. Following [31], all SRAM operations are synchronous to the clock: i.e., read data will come out a cycle after addresses are presented and a write will complete on the rising edge after address and data is presented.

To our knowledge, the literature has not addressed how `cell-1`-like SRAM cells can be connected with WDDL (or any other DPA resistant logic), which is what Ascend requires. We will assume that mechanisms for this task exist.[10] We point out that control logic within the `cell-1` SRAM array is not WDDL.

4.2.3 Optimization: Controlling Clock Enable

Not previously discussed, another control input to both DPA resistant gates/flip-flops and SRAM cells is clock enable. We will manipulate each circuit's clock enable signal to further

[8]The authors do not discuss whether arrays with different dimensions/aspect-ratios impact security. We point out that the 2 Kb array they present is similar in size to conventional 4 Kb SRAM arrays used in insecure designs.

[9]Since a curious program can leak through the address, a strawman solution that uses [54] may have to scan all possible addresses to hide the addresss of interest.

[10]Using the WDDL assumption that most leakage comes through wire interconnects, one solution may be to route complementary WDDL signals (corresponding to SRAM inputs) up to but not into the SRAM array. Those signals can then be terminated in a flip-flop that has a feedback loop to its input.

decrease the energy consumption for our circuit constructions. When clock enable is set, we say the corresponding circuit is on; otherwise the circuit is off.

For gates and flip-flops, clock enable determines whether pre-charge/evaluation transitions occur. When a WDDL[11] flip-flop is on (clock enable high), it transitions between phases in the usual fashion and all associated logic networks can accept new inputs at each evaluation step. When a WDDL flip-flop is off, it and downstream logic stalls and ceases to consume dynamic power.

Switching circuits to the off state and extending pre-charge are both done to save energy (Section 4.2.1.2), but differ in fundamental respects. First, extending pre-charge has an impact on RTL design. For instance, `FF4` (Figure 4-2b (right)) can only be removed if pre-charge is extended (Section 4.2.1.3). Second, extending pre-charge causes a circuit to lose its internal state if the pre-charge occurs for more cycles than the length of the circuit's feedback path. In Figure 4-2b (right), if 4 pre-charge phases happen in 4 consecutive cycles, the reduction circuit loses its internal state. A circuit may be toggled to the off state for an arbitrarily long period of time without losing state (i.e., the state is stationary in each flip-flop).

For SRAM cells, clock enable controls whether the `cell-1` primitive is currently being accessed. When clock enable is high, the SRAM performs either a read or write during that clock cycle and is said to be in the on state. Otherwise, no access is made and the SRAM is off.

4.2.4 Observable Circuit States

Summarizing the chapter so far, the server has the following view of each type of circuit in Ascend:

Logic gates, flip-flops (WDDL). The server can tell when each flip-flop/logic network is in the on/off state. Furthermore, when a circuit is in the on state, the server can tell whether it is in the pre-charge or evaluation phase.

SRAM cells (`cell-1`). The server can tell if an SRAM cell is being activated in the current cycle (the on) state or not (the off state). If the SRAM cell is in the on state, the server can tell if the access is a read or write.

We refer to these as the *observable states* for each circuit. For the rest of the thesis, we will assume the server can only determine which observable state each circuit is in at any given time, for purposes of abstraction. In practice, the server can see small, additional power fluctuations if complementary wire routing is uneven (Section 4.2.1), etc. As is the case with the I/O channel, Ascend's power resistance is no greater than that of its underlying primitives.

4.3 Time Obfuscation: Public Intervals, Interval Policies and Rate Matching

We now describe techniques to prevent leakage through the power channel in the time domain, given the circuit primitives from the previous sections. The goal is to prevent

[11] Following Section 4.2.1, we assume WDDL for concreteness. Manipulating clock enable, however, also works for other DPA resistant logics.

leakage through any program that uses similar principles as `curious_spa()`. Concretely, *schemes must be in place that force each circuit to transition between observable states at times that are strictly independent of the user's data* (see previous section). Using ideas developed so far, a strawman design can keep all circuits in the on state always, interleave SRAM reads/writes in a 1-1 fashion and interleave pre-charge and evaluation every other cycle for all circuits (as shown in Figure 4-1b).

4.3.1 Public Intervals for Groups of Circuits

To improve upon the strawman scheme, we will add additional public parameters (i.e., elements to $\mathbb{P}$) that control when different circuits toggle to the on state. For instance, the server may run the honest program `power_honest()`, as shown in Program 4. Despite this program being honest, it can still leak privacy if eavesdroppers are present (e.g., since a multiplication operation requires more time/energy than the adder). To minimize the energy cost to complete `power_honest()`, the server can analyze `power_honest()` offline and set public parameters that instruct Ascend to perform three on-chip memory loads (the first of which should also be an ORAM request, in anticipation of a cold miss), a bitwise AND, an addition, multiplication and on-chip memory store—in that order.

Program 4 An honest program that can leak through the power channel. `M` is the private user memory.

```
void honest_power(M) {
  int a = M[0]; int b = M[1];
  if (M[2] & 0x1) a = a + b;
  else            a = a * b;
  M[0] = a;
}
```

In the above example, the server could control each circuit *precisely* (i.e., by streaming a static schedule into Ascend that specifies when each circuit should be activated next). In this thesis, we will implement mechanisms that give the server some of this control by re-using (and augmenting) the public interval idea from Section 2.4.2.1. The primary difference between the I/O and power channels in terms of intervals is that while the I/O channel's behavior is *completely* determined by the behavior of the ORAM interface (Section 3.4), the power channel's signal is determined by the instantaneous behavior of every circuit.

A first step is therefore to partition the circuits within Ascend into groups, where each circuit in the group shares an interval. Circuits within a group toggle to the on state in a manner that is synchronous with the interval (and are otherwise in the off state).[12] Thus, as the interval increases each circuit assigned to that interval is in the off state for a higher percentage of time and dynamic power decreases.

At two extremes, every circuit might be assigned to a single interval or every circuit to a different interval. As the number of intervals increases, the server gets finer-grain control over when circuits are activated, which can lead to efficiency improvements. At the same time, if too many intervals are set based on apriori knowledge of program P on public input

[12]For example, if an on-chip cache is set to an interval, the SRAM arrays are toggled to the on state first, and the logic that checks whether the access was a hit or a miss toggles to the on state, second (as every cache access will require operations to happen in that order).

M, those intervals may "overfit" program P applied to hidden input M'. We will apply the following design principle when deciding how to assign intervals: *if two circuits c_1 and c_2 are always accessed together, those circuits can be assigned to the same interval.*

A second step is to decide how a particular circuit group's interval be restricted relative to other circuit groups' intervals. The next two sections illustrate why this is important. Suppose Ascend contains an instruction pipeline (pipeline for short) and two levels of on-chip cache (connected as shown in Figure 3-1). Further suppose that the pipeline, level 1 (L1) and level 2 (L2) caches are set to *distinct* intervals. Like a normal processor, (a) if the L1 cache is accessed and has a real miss, a real request is sent to the L2 and (b) that the pipeline accommodates only one outstanding memory request. There are now two phenomena to consider (as described in Sections 4.3.2 and 4.3.3):

4.3.2 Producer/Consumer Rate Matching

The L2 is rate limited by the L1 and the L1 is rate limited by the pipeline. For example, if x distinct L2 accesses are made for each L1 access (due to how the server set each interval), the L2 is guaranteed to do dummy work for at least $x - 1$ of those accesses. More generally, suppose that n producer circuits $c_{1,\ldots,n}$ (each of which are controlled by a distinct interval $I_{c1,\ldots,n}$) connect to a consumer circuit c (with interval I_c). Then one solution to the rate matching problem is to restrict the value for I_c to values suggested by a function $\mathsf{rate}_{c_{1,\ldots,n}}(I_{c1,\ldots,n})$, which calculates the maximum rate at which the producer circuits can produce complete sets of inputs to c. For simplicity, we say that a complete set of inputs to c are available after each of $c_{1,\ldots,n}$ are switched on for one cycle. The rate() function depends on each circuit's interval (dynamic) and access time (static), is derived by the chip manufacturer and can be computed offline (given concrete values for $I_{c1,\ldots,n}$) by the server.

Another solution that we will use in future sections is to make c's interval *synchronous* to $I_{c1,\ldots,n}$. That is, instead of c being switched on after some number of cycles (Section 2.4.2.1), c will be switched on after $c_{1,\ldots,n}$ have been switched on I_c times. That is, we will change the semantic meaning of a consumer circuit's interval to obey producer/consumer relationships for all possible values of the interval (e.g., only access the L2 cache after the L1 cache has been accessed I_{L2} times). Note that for this scheme to work, there must be a "base case" circuit—that is, if each circuit $c_{1,\ldots,n}$ receives input from no other circuit, then $I_{c1,\ldots,n}$ must be synchronous to the clock (or some other circuit that behaves independently of any other circuit). If this is violated (e.g., if c is also a producer for circuits $c_{1,\ldots,n}$, and $I_{c1,\ldots,n}$ are synchronous to I_c), the system can deadlock.

4.3.3 Producer/Consumer Data Dependancies

If the level i cache is performing a real access, the pipeline and level $1, \ldots, i-1$ caches will necessarily be doing dummy work if those circuits are in the on state. With our example architecture, this phenomena is an artifact of the pipeline having one outstanding memory request at a time, but is more generally caused by data dependencies: if circuit c has producer circuits $c_{1,\ldots,n}$, c can only perform real work when all of its producers create real work. To take advantage of this behavior and increase chip efficiency, we introduce a new type of public parameter (i.e., add more elements to $\mathbb{P}$) called the *public interval policy.* An interval policy allows the server to specify whether circuit c should always be kept in the on or off state while its producers $c_{1,\ldots,n}$ are in the on state. As discussed before (Section 4.2.3), we care about this distinction because a circuit's off state consumes less energy than its on state.

Interval policies should be set based on the server's *confidence* in its ability to predict the program's behavior given an arbitrary input. For example, if the program running in Ascend is straight line code, the server will be able to predict exactly when the L1 caches need to be accessed. In this case, the server will be able to determine exactly when the L1 cache is doing real or dummy work, for a given interval setting. If the interval is set such that the cache performs real work most (or all) of the time, the correct policy is for the pipeline to be in the off state when the cache is being accessed, to save energy. If the program's behavior depends heavily on its input, however, the right policy is less clear and the server will have to rely on guesswork. On the one hand, if the server sets a policy that forces the pipeline to the on state while the L1 cache is being accessed, either the cache or pipeline will be performing dummy work at all times, which wastes energy. On the other hand, if the pipeline is switched to the off state when the L1 cache is performing dummy work, the system will incur a performance penalty whereas an insecure processor would have been able to make forward progress.

Chapter 5

Ascend Microarchitecture

Our goal in this chapter is to combine the primitives from Chapters 3-4 together and produce a complete, optimized Ascend microarchitecture. The design goal is to explore ways to trade-off performance and energy efficiency while preserving the same level of security. We first describe how the main structures (the instruction pipeline, memory hierarchy and ORAM interface) interact, and then describe each in detail.

5.1 Chip Organization

The design is composed mainly of an instruction pipeline, a two-level cache hierarchy and the ORAM interface. The cache hierarchy is made up of separate level 1 (L1) (I)nstruction and (D)ata caches that both connect to a unified L2 cache. We will use the terms "L2 cache" and "last-level cache" interchangeably. Ascend isn't constrained to this organization, but we will focus on it for exposition purposes and for its similarity to modern processors.

5.1.1 Request and Eviction Buffers

The instruction pipeline interfaces with the L1 caches through request buffers. The L1 caches interface with the L2 and the L2 interfaces with the ORAM interface through separate buffers. When we say "L2 buffer" we are referring to the buffer between the L1 and L2 caches. Note that the L2/ORAM buffers are used to store requests as well as blocks evicted from higher memory levels.

5.1.2 Flow of Requests, Data and Evictions

The pipeline evaluates instructions; some of which may make an (address, operation=read/write, data) request to the L1 DCache request buffer and all of which will make an (address, operation=read) request to the L1 ICache request buffer. The corresponding cache will service the request (read from its request buffer) and return a response at some later time. Like a normal processor: Each request may hit (the data was present) or miss (otherwise); some misses will cause blocks to be evicted from the L1 cache to make space. When a miss or miss and/or eviction occurs, the L2 is accessed as in a normal processor, and if a miss and/or eviction occurs in the L2, the ORAM interface is accessed. Write requests are never made from the pipeline to the L2/ORAM interface directly—data is first read from those sources into the L1 and L2/ORAM writes happen only as a result of block eviction.

5.1.3 Public Intervals

Conceptually, we will assign each cache and the ORAM interface to separate intervals. This is to minimize the number of intervals (which reduces the "search space" for finding good interval values), while setting the most performance-sensitive circuits to distinct intervals. For example, caches and ORAM heavily impact performance/energy consumption and there are only several caches and one ORAM interface per chip. Breaking the instruction pipeline itself into multiple intervals is left to future work.

Following Section 4.3.2, we set each memory's interval to be *synchronous* to when another memory or the instruction pipeline is accessed. The insight for this design choice is that caches have a very clear producer/consumer relationship: *if the level i cache is accessed more frequently than the level $i-1$ cache, the level i cache is guaranteed to be doing dummy work some of the time.* We will use the following notation for the intervals: I_{L1D}, $I_{D\to L2}$, $I_{I\to L2}$, and I_{ORAM}. I_{L1D} is in terms of clock cycles and means "perform an access to the L1 DCache I_{L1D} cycles after the last L1 DCache access completes." We will assume for the rest of the thesis that the L1 ICache is accessed constantly. This is a design decision that will be addressed again in Section 5.2, but the intuition is that every instruction needs an instruction fetch.[1] $I_{D\to L2}/I_{I\to L2}$ mean "perform an access to the L2 cache every $I_{D\to L2}/I_{I\to L2}$ times the L1 DCache/ICache is accessed, respectively. We create two intervals because instruction caches typically have very small miss rates compared to data caches. Finally, I_{ORAM} means "perform an access to ORAM (through the ORAM interface) every I_{ORAM} times the L2 cache is accessed."

We can show that this setup makes each cache get accessed at times that only depend on public intervals through induction: the L1 cache is accessed at data-independent times (i.e., based on the clock) and the L2 is accessed solely based on when the L1 is accessed, etc. The scheme cannot deadlock (see Section 4.3.2) because I_{L1D} is synchronous to the instruction pipeline's clock and the L1 ICache is accessed as frequently as possible.

5.2 Instruction Pipeline

Instructions are executed via a pipeline composed of traditional stages (i.e., fetch/decode/execute/memory/retire) where each stage is separated by a bank of flip-flops. We now describe the pipeline's salient features (see Figure 5-1 (left)):

1. Instructions are executed in-order and the pipeline will only fetch the next real instruction after the previous instruction retires.

2. Each family of instructions is executed via feed-forward logic and is pipelined as needed to decouple the clock frequency from the logic complexity of each instruction. Since the next instruction is not fetched until the previous retires, we don't implement traditional operand bypass logic for simplicity.

3. Instruction/data memory requests are sent to a pair of buffers that decouple the pipeline from the L1 caches. The caches signal to the pipeline when the request is complete.

[1] An optimization to this scheme is to access a single-cache block buffer every cycle and have the L1 ICache refill this buffer at a separate interval. We will not perform this optimization, for Ascend or an insecure processor, in our evaluation for simplicity.

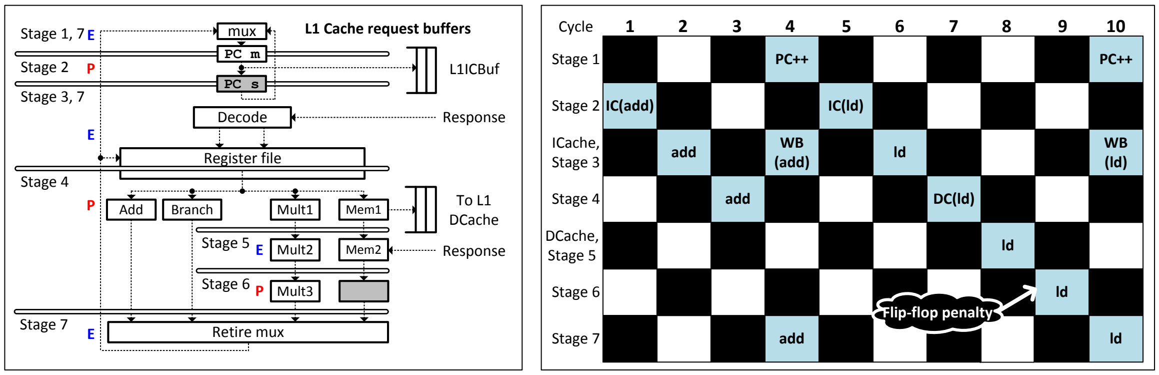


Figure 5-1: (Left) A power-analysis resistant instruction pipeline (only some execution units are shown for simplicity). **P**/**E** indicate which stages are in pre-charge/evaluation (Section 4.2.1.1). Each stage is divided by flip-flops; pipeline latency added as a result of WDDL is shaded grey. (Right) Timing diagram for the instruction pipeline when `add` and `ld` (load from memory) instructions are evaluated, in that order. **Black** squares indicate when a stage is in pre-charge; white squares mean the stage is in the evaluation phase but is doing dummy work; blue/labeled squares mean the stage is in the evaluation phase and is doing real work. For simplicity, the L1 I and DCaches are treated as combinational logic. IC(*inst*) and DC(*inst*) mean that instruction *inst* is writing to the L1 I/DCache buffer in that cycle; WB means register file writeback.

4. Results are merged after the execute stages by multiplexer (mux) circuits, which coordinate writebacks to the register file and PC register.

This design was chosen for its simplicity, its high clock frequency potential and its ability to be transformed into efficient DPA resistant circuits, discussed next. Creating more aggressive pipelines (out-of-order, multiple issue, etc) is left to future work.

To be secure, the pipeline must give off a data independent power draw regardless of what instructions are actually executing. In this discussion we will assume the DPA resistant primitives from previous sections for concreteness. The dashed lines in Figure 5-1 show the set of "paths" taken through the pipeline in each cycle and illustrate the high-level strategy for any DPA resistant logic family: *during any cycle when the pipeline is in the* `on` *state, it must give off the impression of executing any possible instruction.*[2]

If we use WDDL and `cell-1` SRAM specifically, the design changes in the following ways: First, all pipeline flip-flops are implemented as WDDL flip-flop pairs; gates as WDDL compound gates; and the register file/L1 request buffers as `cell-1` SRAM arrays. Second, pipeline bubbles must be added (shown as grey boxes in Figure 5-1 (left)) to feedback paths with odd length, following our optimization from Section 4.2.1.1. Note that WDDL master-slave flip-flops are still needed if their corresponding insecure flip-flop needs to maintain the same state across cycles (such as the PC register—labeled `PC_m` and `PC_s`). Such registers conceptually have feedback paths of length 1. Other flip-flops, that only serve to separate pipeline stages and do not require back-pressure, are only implemented using two flip-flops (to store the real and complement logic values) to reduce instruction cycle latency and area.

We interleave pre-charge and evaluation phases in a 1-1 fashion like baseline WDDL (i.e., do not apply the extended pre-charge optimization from Section 4.2.1.2) for perfor-

[2]Note that real processors exhibit the same behavior to some degree. For example, a processor ALU is typically a large combinational circuit whose output goes through a mux.

mance reasons. In modern processors, some execution units (e.g., multipliers) incur larger combinational delays than others (e.g., adders)—as shown in Figure 5-1 (left). Furthermore, some instructions can take a data-dependent amount of time to complete [37] (e.g., division with early exit on the x86 or floating point division/square root). *Ascend can apply these optimizations better as the percentage of evaluation phases per unit time increases.* Figure 5-1 (right) shows an example where an (`add`) instruction is followed by a load from a data memory (`ld`) instruction. Because every pipeline stage is in the **E** phase (defined in Section 4.2.1.1) every other cycle, the `add` can complete in less cycles than the `ld` and performance improves. Optimizations like "data-dependent early exit" can also be implemented *as long as data is forwarded an odd number of stages down the pipeline*, which prevents a stage from being in pre-charge and evaluation simultaneously.

Alternatively, we could apply the extended pre-charge trick to reduce power consumption. Since (a) this scheme still requires that the **E** phase happen at data-independent times and (b) when the next instruction needs to be fetched is generally data-dependent, this scheme just equalizes instruction cycle latencies and decreases performance for "fast" instructions. One insight when sacrificing energy for performance is that (in practice) for our simple pipeline, energy consumption is very small (relative to ORAM and the cache memories) regardless of the frequency at which evaluations occur.

5.3 Choosing an ISA

Ascend is not bound to a specific instruction set architecture (ISA) but does benefit from simpler (i.e., RISC) ISAs. Since the simple pipeline design (see previous section) activates every circuit path on every cycle, simpler ISAs will have smaller implementation overheads. On the other hand, a CISC ISA that requires a unique/complex execution unit would incur greater overhead if implemented directly (i.e., no microcode). We will assume a MIPS-like ISA for the rest of the thesis, whose relevant features are:

1. Each instruction may read the instruction pipeline register file two times, and write one time. Thus, these three actions occur every cycle to hide real behavior.

2. Each instruction may make one request for one address in memory, using one of several addressing modes.[3] Depending on the instruction, a different number of bytes may be returned to the pipeline. To hide which case occurs, each request performs the work of returning the maximum number of bytes (8 for double words in MIPS). We point out that an ISA whose instructions each access disjoint regions of memory in a data-dependent fashion complicate the power obfuscation process significantly, as will be made apparent in the following sections.

3. Floating point is implemented in hardware. We will assume that the floating point unit (FPU) is activated along with the integer units. Note that Ascend can emulate floating point in software like a normal processor as an alternative.

Exploring alternative ISAs or designing ISAs that map particularly well to an Ascend (power-obfuscated) architecture is left to future work.

[3]MIPS has four addressing modes.

5.4 Cache Hierarchy

In principle, an Ascend chip can support any number of large or small on-chip caches. In this thesis, we assume that Ascend has the two-level cache hierarchy as described throughout Section 5.1. Caches have complex behavior and interact with one another in data-dependent ways. *After we have set each cache buffer to an interval*, we must additionally hide the following:

1. Cache inputs per access: the data, address and operation (read/write for the L1 DCache) signals must be hidden for all accesses to all caches.

2. Cache outcome per access: whether a cache access results in a hit, miss, or miss+eviction (determined by the cache inputs and cache state).

3. Cache coherence behavior: whether a miss or eviction causes other blocks in other caches to be invalidated for data coherence reasons, written to lower cache levels, etc.

This section will detail each of these and apply the following key idea throughout. Suppose a particular cache operation (e.g., accessing the L1 or L2 cache) has multiple possible outcomes (e.g., hit, miss, evict, etc). Then to obfuscate power draw, that operation must perform the union of the work needed to complete all outcomes, to hide the real outcome. *To improve performance and power efficiency despite this overhead, we will make microarchitecture and protocol-level design changes that, for each cache operation, minimizes this worst-case amount of work.*

5.4.1 Energy Efficiency via Intervals+Extended Pre-Charge

For concreteness, we will assume the DPA resistant primitives introduced in Section 4.2. Cache accesses are multi-stage operations whose major data flows are sequential (e.g., we send an L2 request through the network, *then* access the L2 data array, *then* return data to the processor). Thus, we can apply the extended pre-charge optimization (Section 4.2.1.2) as follows: we will only insert WDDL evaluation phases (**E** bubbles) into the cache logic when either real or dummy work could possibly be performed in that stage, at that time. Conceptually, we generate one evaluation phase when the cache access starts and this evaluation phase "bubbles" through the otherwise pre-charged stages. The scheme is still secure because *when* and *how* the evaluation bubble propagates depends strictly on intervals and hardware design. As noted, we won't apply this trick to the instruction pipeline (Section 5.2) because when one instruction retires and another is fetched is data-dependent and, for most programs, not predictable.

5.4.2 Hiding Cache Inputs/Outcomes

Firstly, each cache is built out of DPA resistant primitives. Cache tag and data arrays are implemented using `cell-1` SRAM arrays. Comparator logic to implement associativity, eviction logic and replacement policy logic are built using WDDL. We assume the least recently used (LRU) cache block replacement policy for the rest of the thesis.[4] With these primitives in place, the wires that carry the (address, operation, data) cache inputs are hidden because each wire is DPA resistant as described in Section 4.2.1.

[4]Note that LRU has data-dependent behavior but can be designed not to leak privacy; i.e., if implemented as DPA resistant combinational logic.

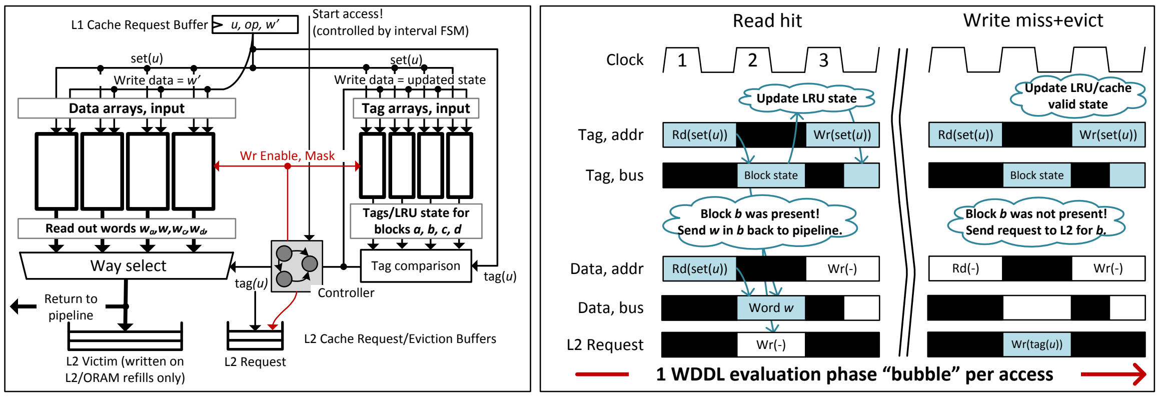

L1 Cache Request Buffer
u, op, w'
Start access!
(controlled by interval FSM)
set(u)
Write data = w'
set(u)
Write data = updated state
Data arrays, input
Tag arrays, input
Wr Enable, Mask
Read out words wa,wb,wc,wd
Tags/LRU state for blocks a, b, c, d
Way select
tag(u)
Tag comparison
tag(u)
Controller
Return to pipeline
L2 Cache Request/Eviction Buffers
L2 Victim (written on L2/ORAM refills only)
L2 Request
Read hit
Write miss+evict
Clock
1
2
3
Update LRU state
Update LRU/cache valid state
Tag, addr
Rd(set(u))
Wr(set(u))
Tag, bus
Block state
Block b was present! Send w in b back to pipeline.
Block b was not present! Send request to L2 for b.
Data, addr
Rd(set(u))
Wr(-)
Rd(-)
Data, bus
Word w
L2 Request
Wr(-)
Wr(tag(u))
1 WDDL evaluation phase "bubble" per access

Figure 5-2: Example architecture (left) and timing information (right) for a 4-way set associative L1 DCache. SRAM arrays have **bold** borders. Note that logically, the cache has inputs and outputs but physically there is a single input/output bus (as indicated in the timing diagram). Timing diagram colors (blue, black, white) match the conventions in Figure 5-1; i.e., solid black means pre-charge. $\mathsf{Rd}(x)/\mathsf{Wr}(x)$ indicate read/write commands to address x. The timing diagram shows two extreme outcomes for the secure design: regardless of the cache access outcome, the sequence of SRAM read/write and logic evaluation/pre-charge transitions is data-independent. Read data is returned to the pipeline at the rising edge between cycle 2 and 3.

We will now discuss how to hide the outcome of a particular cache access, using a 4-way set associative L1 DCache (whose architecture is shown in Figure 5-2) as an example. Suppose the instruction pipeline makes a data memory request for program address u which corresponds to data word w, which is contained in cache block b. For the rest of the thesis, cache block size is the same as the size of a data block in ORAM (i.e., $|b| = B$ from Section 3.2.1). $\mathsf{tag}(u)/\mathsf{set}(u)/\mathsf{offset}(u)$ correspond to the tag/set/offset bits in address u. Generally, $|w| < B$. Furthermore, the cache data bus width F is typically $< B$; we will assume F divides B. An insecure L1 DCache access is now given by $\mathsf{accessL1DCache}(u, op, w')$:

1. Read each tag array at address $\mathsf{set}(u)$ and compare each output tag with $\mathsf{tag}(u)$ to determine which cache way block b is mapped to.

2. If $op = read$:

   (a) If $\mathsf{tag}(u)$ was found in the tag arrays (read hit): perform an data array read at address $\{\mathsf{set}(u), \mathsf{offset}(u)\}$ to fetch F bits from the corresponding cache way. w, a subset of the retrieved bits, will be returned to the pipeline. This read operation may happen in parallel to the tag array read (Step 1, above).

   (b) Otherwise (read miss):

      i. Add an L2 request for $\mathsf{tag}(u)$ to the L2 buffer.

      ii. If $\mathsf{set}(u)$ does not have empty space: choose a block a, that maps to $\mathsf{set}(u)$, to evict through tag LRU logic and perform $\frac{B}{F}$ reads to the cache data arrays to evict a. Add $(\mathsf{tag}(a), a)$ to the L2 eviction buffer (read miss+evict).

3. If $op = write$:

   (a) If $\mathsf{tag}(u)$ was found in the tag arrays (write hit): perform a data array write at address $\{\mathsf{set}(u), \mathsf{offset}(u)\}$ and replace w in b with w'. This write operation must happen after the tag array read (Step 1) completes because the tag array read determines which cache way is given write enable.

(b) Otherwise (write miss): perform Steps 2(b)i-2(b)ii above as if a read miss occurred.

4. Perform a tag table write at address $\mathsf{set}(u)$ to update tag state based on the final cache state (reflecting block evictions, new LRU state, etc).

A naïve/secure implementation can perform the work to complete each cache outcome (of which there are six, corresponding to {read hit, write hit}∪{read miss, write miss}×{evict, no evict}) in sequence. That is, perform the operations for a read hit, *then* a read miss, *then* a read miss+evict, etc. This strategy is highly inefficient and performs on order six times the work per access relative to an insecure system. We will use two ideas to get efficiency, without changing any security assumptions:

1. We will "cover" the common cache outcomes (cache hits) with a *static/minimal* number of SRAM operations, that are performed *in a fixed order*. Note that SRAM reads are distinct from SRAM writes (Section 4.2.2).

2. We will only perform uncommon outcomes (cache evictions) when the L2 cache or ORAM completes a request. This will be referred to as *lazy* cache eviction.

We now illustrate idea 1 (above), also shown in Figure 5-2. Each time the L1 DCache is accessed, according to I_{L1D}, $\mathsf{accessL1DCacheOpt}(u, op, w')$ is performed:

1. **First cycle:** Read the cache data arrays and tag arrays, at address $\{\mathsf{set}(u), \mathsf{offset}(u)\}$ and $\mathsf{set}(u)$ respectively. If the eventual outcome is a read hit, some way at address $\{\mathsf{set}(u), \mathsf{offset}(u)\}$ will contain w, which will be returned to the pipeline. Otherwise, the data array read will be a dummy operation to some (arbitrary) address. In all cases, the tag array read will be used to determine hit/miss/evict information for $\mathsf{tag}(u)$.

2. **Second cycle:** Send word w or dummy data back to the pipeline. If the cache access was a real access, the pipeline may always continue making forward progress at the end of this cycle. If a miss occurred (as determined by the tag array read), write $\mathsf{tag}(u)$ to the L2 buffer.

3. **Third cycle ("in the background"):** Update (write) the tag arrays at address $\mathsf{set}(u)$ with the final state of the cache, invalidating evicted lines and updating LRU information. If the cache access was a real write hit: write the data array, corresponding to the cache way selected by the tag comparison, at address $\{\mathsf{set}(u), \mathsf{offset}(u)\}$ with data w. Write back, to all other ways, the same data that was read during the first cycle of the operation.[5] Note that an insecure processor would only need to write back to the way that contained the data block of interest. We will discuss why Ascend may have to write back to all ways in Section 5.4.3.

Thus, an arbitrary L1 DCache access is made up of one read+write to the tag array and one read+write to the data array. The read occurs first to minimize the time it takes for the pipeline to resume its operation.

[5]Alternatively, write arbitrary data back to a reserved address in the cache that will always be used to implement dummy writes.

Waiting until cycle 3 to perform the write is an artifact of WDDL: each set of input/output wires to the SRAM must be in pre-charge every other cycle. To decrease read latency to 2 cycles, we set the address/operation inputs to the SRAM to evaluate on odd cycles, while the data bus is set to evaluate on even cycles (which implies that pre-charge does not impact cache read hit latency). This may complicate SRAM write timing—i.e., on an SRAM write the address and data for the write are typically asserted in the same cycle. A possible design solution is to (in cycle 3) hold the cache data bus in pre-charge while the clock is high[6] (see Figure 5-2), and switch to the evaluation phase while the clock is low, which places write data at the SRAM input port by the rising edge of cycle 4.

Cache evictions. Cache evictions are only serviced *lazily* when a lower-level memory (the L2 cache or ORAM) performs a refill to the L1 cache. The motivation to postpone this operation is that a cache eviction will require $\frac{B}{F}$ distinct reads (where $\frac{B}{F}$ varies between 1 and 16 in modern processors)—we would like to minimize the number of times that we have to perform this expensive operation. Lazy evictions are based on two insights. First, even if a particular L1 access needs to evict a block, we do not have to "make space" in the L1 cache until the replacement block arrives at the cache. Second, the L2 cache and ORAM are accessed much less frequently than the L1, which means that the lazy scheme will do the work of real/dummy L1 cache evictions less frequently. Putting these ideas together, each L1 DCache refill (which occurs at the end of every L2 or ORAM access) requires $\frac{B}{F}$ cache read operations (which move a real/dummy evicted block into the L2 buffer), followed by $\frac{B}{F}$ cache write operations (to load the replacement block into the L1 cache).

Additional optimizations. Details in the above design are not meant to be limiting; we will now summarize some other possible optimizations. First, the read+write operation may be split into two public intervals if the percentage of dynamic instructions in a given program has a different number of loads vs. stores, on average. Second, the $\frac{B}{F}$ cache read operations that occur per eviction may be set to yet another interval, if inclusive caching is assumed. The insight here is that while almost every real L2 access *will* cause an eviction in the L1 cache (as most programs exceed the L1 cache capacity), many of these evictions may be to clean (unmodified/read-only) data blocks which can just be overwritten in the L1 with the new data block. We point out that the L1 ICache contains only read-only blocks, and can therefore always apply this trick when inclusive caching is used. Third, the entire cache evict+refill operation ($\frac{B}{F}$ reads followed by $\frac{B}{F}$ writes) can be interleaved (i.e., perform a write after each read). The insight is that if the evict+refill operation is doing dummy work, yet looks the same as a series of accessL1DCacheOpt() operations, then the pipeline can perform normal cache accesses as part of the evict+refill operation. Fourth, during an L1 Cache refill, the "critical word" in the incoming data block can be forwarded to the pipeline early like in a normal processor. To obfuscate power draw, however, the system must send real/dummy "critical words" to the pipeline each cycle—regardless of where the critical word is located within the block.

5.4.3 Power Obfuscation and Cache Organization

Due to their size, caches are partitioned into smaller SRAM arrays (called subarrays), where each subarray typically stores a subset of bits for a subset (or all) cache sets in a single cache way. For example, the 256 Byte `cell-1` SRAM array (Section 4.2.2.2) can be thought of as a single subarray. There are different methods to organize subarrays into the complete

[6]This idea is similar in concept to wave-pipelining and was proposed by [22] as an alternative way to launch the pre-charge wave.

cache (e.g., Cacti uses an H-tree structure [13]). A simple organization arranges subarrays in columns and rows where all subarrays for a given row are activated in a given cache access to deliver a complete output. This type of scheme saves power by using pre-decoders to select only the correct row. An important point for power obfuscation is that the number of subarrays per row is the same for all rows.

We discuss two higher-level cache organizations that impact which subarrays are activated per access. First, the tag and data arrays can be accessed in parallel (as done in Figure 5-2). A row of subarrays for each cache way are activated per cache access and the data from the correct way is chosen through a large mux whose select bits are generated through tag comparator logic. We assumed this design for the L1 caches as they are performance sensitive and therefore are usually implemented using such a parallel design. A serial design, however, is widely used in L2 caches (or L3 caches in larger processors) because L2 caches are large, highly associative and more decoupled (performance-wise) from the pipeline because of the L1. In the serial design, the tag arrays are accessed first and used to determine which cache way contains the block/data of interest, allowing for only the subarrays in that way to be activated. This design saves energy (less subarrays are activated per access) but has higher access latency (the tag and data are looked up serially).

It is unclear the extent to which power-obfuscated caches can take advantage of optimizations like the serial cache design, as the power consumption variation when activating a data-dependent subset (but fixed number) of subarrays has not been studied in the literature to our knowledge. On the one hand, each cache way contains a topology of 'identical' SRAM subarrays, which suggests that the total energy dissipated per way, when considering each way in isolation, is very similar across ways. On the other hand, SRAM subarrays are large structures and the physical distance between subarrays may cause time-varying fluctuations in power consumption, depending on which subarrays are activated. For example, in large caches (e.g., the L2/last-level caches), the subarrays span the chip. One solution to this problem might be to balance the input/output wires and the power network (as in an H-tree) such that activating each "row" of subarrays causes variations in the power signature at a time independent of which row of subarrays is activated.

5.4.4 Where DPA Resistance is Not Needed

The astute reader will have noticed that in Section 5.4.2, not all wires carry data-dependent information. For instance, in accessL1DCacheOpt() the tag array is always read *and then* written when the interval initiates a cache access. When the cache is accessed and how the cache read/write signal changes is therefore data-independent, the read/write signal itself and its control logic need not be built out of a DPA resistant logic like WDDL. The same is true for the logic that controls interval FSMs and for the logic that controls interval policy stall signals (which will be discussed in Section 5.6.1). In practice, however, we believe that these types of circuits will be a minority in (at least) our current Ascend design and will not mention where they occur from now on.

5.4.5 Hiding Cache Coherence Behavior

In addition to hiding the outcome of a given cache access, certain cache accesses may trigger other cache accesses to maintain coherence of data throughout the chip. Recall the distinction between inclusive and exclusive caches from Section 3.3.1. Architecturally, an argument for an exclusive hierarchy is its increased aggregate capacity, as cache lines are

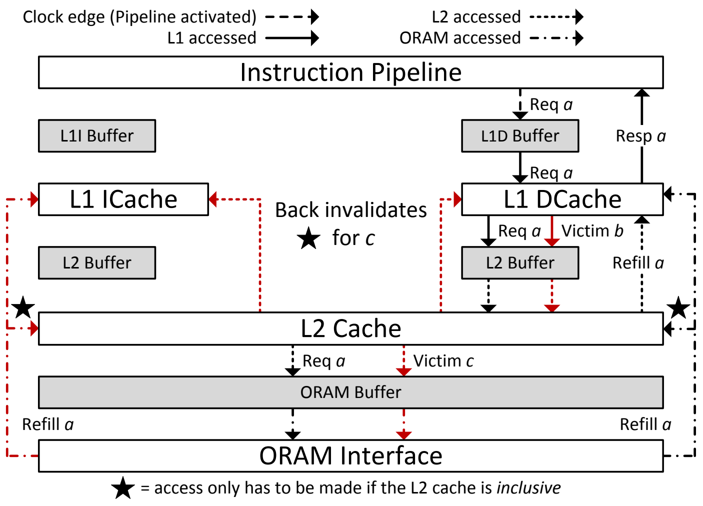


Figure 5-3: For each cache: the set of cache accesses that must be made when a particular cache is accessed. (The L1 ICache-related work is not shown for simplicity). Req is request and Resp is response; a, b, c are three distinct cache blocks. "Victim" means evicted block. Black arrows need to be taken by a normal processor; Ascend must take black *and* red arrows to obfuscate cache evictions and back-invalidations.

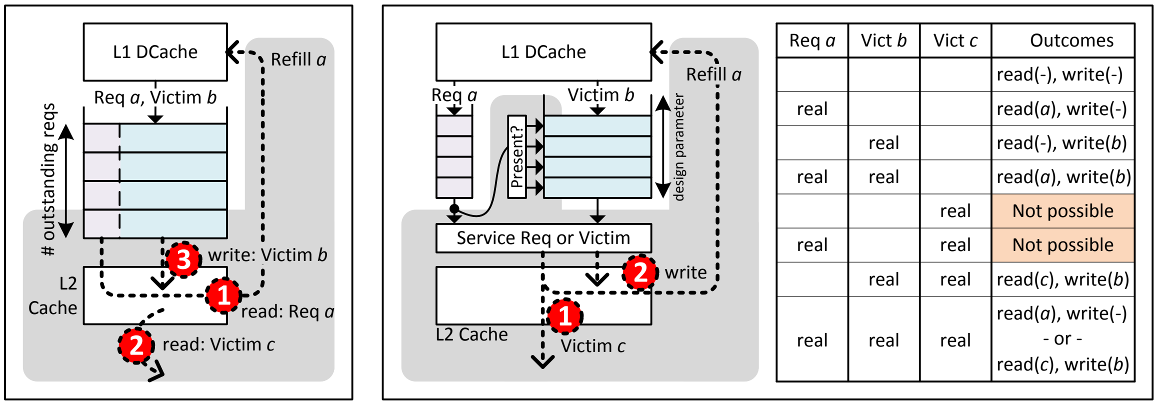
not duplicated to preserve the inclusivity property. An argument for inclusive hierarchies is simplified eviction logic: if a clean block is evicted from some level, it does not have to be written back to a lower level because of the inclusivity property.

Ofbuscating power draw disproportionately increases the number of coherency operations that must be performed for inclusive cache designs. Figure 5-3 illustrates the problem: when block c is evicted from the L2 in an inclusive hierarchy, eviction requests are sent to any L1 cache that also holds a copy of c (to maintain inclusivity). Back invalidations cause data-dependent behavior that can be detected through power analysis, so each L2 access must perform all possible back invalidations (some will be dummy work) or schedule those operations to intervals. Also, as mentioned above one of the benefits to inclusive designs is that clean blocks evicted from the L1 do not have to be written back to the L2. To hide whether an evicted block is clean or dirty, however, Ascend must conservatively perform a real or dummy write back always, or at a public interval—nullifying a traditional benefit in inclusive hierarchies [55].

5.4.6 Hiding Cache Outcome for the L2 Cache

Despite the attractive features of power-obfuscated exclusive L2 caches in terms of maintaining coherency (see previous section), it seems significantly more difficult to obfuscate cache outcome for the exclusive L2 cache. The fundamental issue is that when an L1 cache evicts a block to the L2, that eviction may cause another eviction from the L2 to the ORAM interface (this was also a source of problems in Section 3.3.1). Such chain-evictions will not occur with an inclusive L2 cache.

We now discuss some possible designs to hide L2 cache access outcome, for both inclusive and exclusive caches. When the L2 cache interval I_{L2} signals that an L2 access should be made, there may be both an L2 request (Req a) and an evicted block (Victim b) in the L2 buffer.



Req a	Vict b	Vict c	Outcomes
			read(-), write(-)
real			read(a), write(-)
	real		read(-), write(b)
real	real		read(a), write(b)
		real	Not possible
real		real	Not possible
	real	real	read(c), write(b)
real	real	real	read(a), write(-) - or - read(c), write(b)

Figure 5-4: Comparison between a strawman L2 buffer implementation (left) and an out-of-order implementation (right). Each dashed/numbered arrow represents an L2 cache read or write operation. Shaded grey area covers circuits that are activated on each L2 access. The table on the right illustrates cases for an implementation that uses the out-of-order and lazy L1 optimizations from Section 5.4.6. read(a) means "read L2 data array for Req a; read($-$)/write($-$) are dummy L2 data array operations. Putting this together: the second row of the table means "when there is a real request and no outstanding eviction, perform a read to the set containing Req a and then perform a dummy write."

5.4.6.1 Exclusive L2 Caches

An exclusive L2 cache must perform the following operations to handle both request and eviction:

1. (requests) L2 data array read, to return the block associated with Req a to the L1.

2. (evictions) L2 data array read, to move/evict a Victim block c to the ORAM buffer if writing the block evicted from the L1 (Victim b) itself causes an eviction in the L2.

3. (evictions) L2 data array write, to move Victim b into the L2.

Thus, a strawman L2 access strategy will perform two data array reads and a write per L2 access (Figure 5-4 (left)). Concretely: when the L2 cache's interval indicates that it is time to make an access, the L2 reads from both its request and eviction buffer and makes the above accesses in the order listed. This scheme has a low complexity implementation: First, the L2 buffer only has to be as deep as the number of outstanding memory requests in the system (we assume 1 in this thesis). Second, the buffer will never overflow and therefore does not need back-pressure logic to logically stall the L1 caches or pipeline. (By *logically stall*, we mean that the pipeline/higher level caches will do dummy work until the stall ends, but that those circuits' observable states will be the same as if the stall did not occur).

To reduce the number of L2 SRAM operations per access, we can architect the L2 buffer to service requests and evictions out-of-order (Figure 5-4 (right)).[7] We can't "cover" the two L2 SRAM read operations in a single read (as was done for the L1 DCache, see Section 5.4.2) because both read operations may be real reads. Instead, we allow the L2 to dynamically choose which buffer (request or eviction) it will read on a given access. Once

[7]Note that "out-of-order" does not mean that our memory consistency model has changed: from the program's perspective, memory read/writes occur in order.

chosen, the L2 will first perform a data array read (to either read Req a or Victim c) and then a data array write (to writeback Victim b). The new scheme therefore performs an L2 read+write instead of a read+read+write (strawman).

Reading the L2 buffer out-of-order increases design complexity. First, Req a may correspond to a recently evicted block which is still in the L2 eviction buffer. Thus, the eviction buffer must be scanned prior to every L2 access (in practice, the eviction buffer will only contain several entries). Second, the L2 buffer may fill and need to logically stall the L1 caches and pipeline. For example, suppose each L1 access adds a real request and real evicted block to the L2 buffer, and the L2 cache always services the request (i.e., *not* the evicted block) if there is a real request present. In this case, the pipeline will eventually be able to fill the L2 eviction buffer regardless of the buffer's depth. Two observations indicate that in practice, this happens only rarely: First, as we will show in Chapter 7, a large percentage of L2 accesses are dummy accesses.[8] We define a *dummy L2 access* as an L2 access were Req a is dummy. If there is an evicted block in the L2 buffer when a dummy access occurs, the dummy access can service that evicted block and make space in the eviction buffer. The idea is similar to ORAM background evictions that occur when Ascend makes dummy ORAM accesses (Section 3.4). Second, the case when all three operations—read+read+write—are needed after a single L1 access seems to be rare in our benchmarks. This case seems to happen more in memory bound benchmarks that make ORAM requests continuously, rather than to benchmarks that rely on the L2 capacity.

5.4.6.2 Inclusive L2 Caches

For inclusive L2 caches, we can achieve the read+write design trivially. To service Req a, an L2 read is performed as was done in the exclusive L2 case. To service Victim b, the (possibly stale) copy of block b already in the L2 cache can be overwritten with Victim b. As discussed in Section 5.4.5, this write must still occur to hide whether Victim b was dirty. As with the strawman exclusive design, this design is simple to implement (i.e., the L2 buffer may have a depth of 1, yet guarantee that the pipeline will never have to logically stall). Furthermore, this design allows the server to optimize for the case when most lines evicted from the L1 are clean (i.e., by splitting L2 reads and writes into two intervals as mentioned in Section 5.4.2).

As will be quantified in Chapter 7, performing an L2 data array access (read or write) consumes a significantly larger amount of energy than L1 cache back invalidations. Thus, it seems the exclusive design must apply out-of-order buffer processing to be competitive, in terms of energy efficiency, with the inclusive design.

5.5 ORAM Interface

We leave the bulk of the ORAM interface discussion to Chapter 3, and will only discuss system integration and power obfuscation-related information here.

5.5.1 Security-Related Issues

We note the following security-related points: First, each component in the ORAM interface must be built using DPA resistant logic. At a high level, the ORAM interface is

[8]This seems to be due to noise in the L1 cache miss rate that cannot be captured in fixed intervals.

composed of a position map, the local cache, symmetric encrypt/decrypt units and control logic. To a curious program, learning which element in the position map or local cache is accessed is equivalent to having no ORAM at all. Of course, learning the outcome of an encryption/decryption is also a critical security hole. Second, the accessORAM() algorithm itself (Section 3.2.1) must perform a data-independent amount of work per access. On this topic we point out the following:

1. Reading and writing an ORAM path performs a data-independent amount of work per access (i.e., the number of ciphertext bits in each path is constant).

2. Choosing which blocks should be written back to the Path ORAM tree (Step 5 in accessORAM()) must perform a data-independent amount of work. [56] implicitly assumes a data-dependent sort to decide which blocks to write back—in an Ascend context, this may leak information through the power channel because the number of blocks in the local cache is access-pattern dependent (Section 3.3.1). Fortunately, a simpler linear/data-independent scan that reads the entire local cache once can achieve the same eviction rate and is simpler to implement hardware-wise. Suppose the local cache can hold up to C blocks. Then the eviction algorithm reads the local cache C times; for each read, control logic determines the highest level in the ORAM tree that the corresponding block can be evicted to. This is a Boolean function whose outcome depends on the leaf index assigned to the corresponding block, the current path being read/written and the load on each bucket along the path. The load on each bucket is stored in flip-flops and is updated after each access. This scan uses the insight that if the highest level that blocks b_1 and b_2 can be mapped to is level l, the eviction rate is independent of which block is evicted to level l.

5.5.2 Systems Integration-Related Issues

In addition to storing each cache block's tag in on-chip cache tag tables (as in a normal processor), Ascend must also store the current Path ORAM leaf index associated with each cache block in tag tables as well. This adds an additional L bits per entry for each block in the tag table, where $L < 32$ typically (see Section 3.2.1). Leaf indices are needed to support L2 (last-level) cache evictions: when a cache block is evicted from the L2, it is added back to the ORAM interface's local cache. In order to get evicted to the ORAM tree during the Path ORAM writeback operation (Step 5 in accessORAM()), each block in the local cache must have an associated leaf also stored in the local cache at that time.

Despite being Ascend's last-level memory, Path ORAM can be inclusive or exclusive in the same sense as the on-chip caches. Suppose a block b, whose program address is u, is currently mapped to Path ORAM leaf l. In an exclusive Path ORAM:

1. If Ascend has a last-level cache miss for u: the ORAM interface performs accessORAM$(u, read, -)$ operation to read b, and replaces b with a dummy block in the data ORAM tree.

2. If (b, u, l) is evicted from Ascend's last-level cache, it is added to the ORAM interface local cache and pushed to the ORAM tree during the Path ORAM writeback operation (i.e., in the background).

In an inclusive Path ORAM:

1. If Ascend has a last-level cache miss for u: see corresponding operation for the exclusive case.

2. If (b, u, l) is evicted from Ascend's last-level cache:

   (a) If b is clean: do nothing. Conceptually, the block is removed from the system because an up-to-date copy is already owned by the ORAM interface. To obfuscate the power draw of clean evictions, a dummy SRAM write can be performed to the ORAM interface local cache.

   (b) If b is dirty: perform $\mathsf{accessORAM}(u, \mathit{write}, b)$ to replace the stale copy in the ORAM tree. Note that with small probability, the bucket containing the stale copy of b will be read into the local cache, as part of another ORAM operation, while the dirty copy of b is present in the local cache. For security reasons, the dirty block and stale block cannot be merged if this happens: an explicit $\mathsf{accessORAM}(u, \mathit{write}, b)$ operation must always be made.[9]

The position map ORAMs (if the Recursive Path ORAM is used) still contain the mapping for (b, u) in both inclusive/exclusive setups.

We will assume an exclusive Path ORAM for the following reasons. First, the inclusive case is more complex because it needs to (conditionally) perform explicit Path ORAM operations to write back dirty lines. As noted, this operation may also leak privacy if not performed properly. Second, it is not clear what benefit (from a local cache occupancy standpoint) inclusive ORAMs will have over exclusive ORAMs. Suppose a program is scanning memory (i.e., the on-chip caches will fill) and that each block read is read-only (i.e., each block evicted from the last-level cache will be clean). This is intuitively the best case scenario for the inclusive setup. With an inclusive cache, each last-level cache miss will (potentially) add a block to the local cache (i.e., the block that was read and remapped). The last-level eviction (if any) will not add a block to the local cache because that block was clean. With an exclusive Path ORAM, each last-level cache miss will *not* add a block to the local cache, since that block will have been forwarded to the on-chip caches. On the other hand, each last-level miss for the exclusive case will cause a last-level eviction, which will add a block to the local cache. Thus, in this pathological scenario (which was designed to benefit the inclusive case) the inclusive and exclusive add the same number of blocks to the local cache per last-level miss.

Note that the above argument may not hold when the last-level (L2) on-chip cache is exclusive, since an exclusive L2 may evict blocks to the ORAM interface, yet still hit in the L2 (Section 5.4.6.1). In that case, writing clean blocks to the ORAM interface can have higher impact.

5.5.3 Performance-Related Issues

We will only use the version of early completion (Section 3.2.3) that forwards data to Ascend's pipeline after the entire path has been read into the ORAM interface. This choice is made for energy efficiency reasons: to obfuscate the power draw for the most aggressive early completion scheme (i.e., forward the requested data block as soon as it is read into the local cache), the ORAM interface would have to forward real or dummy data every

[9] If we allow stale and dirty lines to be merged based on other ORAM operations, the leaf accessed during $\mathsf{accessORAM}(u, \mathit{write}, b)$ will be correlated to the sequence of previous accesses. This was also why security broke with the block remapping background eviction scheme (Section 3.3.4).

cycle during the read path step in accessORAM(). We believe that this change would have a disproportionate energy-performance trade-off.

5.6 Interval Policies: Dynamically Trading Off Power & Performance

Once intervals are designed and cache accesses/coherence/etc are obfuscated, each memory can additionally be given an *interval policy* that says whether higher levels (e.g., the pipeline or L1 caches) inside Ascend will continue to be accessed while a lower level (e.g., the L2 or ORAM) is doing real/dummy work. Interval policies were introduced in Section 4.3.3 and apply to all of the instruction pipeline, caches and ORAM because these components interact in a clear producer/consumer fashion. As mentioned in Section 4.3.3, the big idea behind these policies is to improve energy efficiency for programs that have more regular behavior.

In the next several sections we will explain how to set policies so that the server can make performance and energy trade-offs based on its apriori knowledge of the program running on Ascend. We will not discuss secure SRAM or WDDL-related concerns in this section and point out that the interval policy idea generally applies to any DPA resistant logic that can be switched between on/off states (Section 4.2.3).

We allow for each memory level i to have its own policy S_i ($S_i \in \mathbb{P}\ \forall i$) and say that each $S_i = \mathsf{SI}$ (for *speculative*) or $= \mathsf{CI}$ (for *conservative*). (Notation note: "memory level $i = 1$" refers to the L1 DCache). If $S_i = \mathsf{SI}$: memory level i being accessed, as instructed by its interval, does not impact the state of the rest of the Ascend chip. This is implicitly the scheme we have assumed for each memory up to this point. If $S_i = \mathsf{CI}$: the instruction pipeline, L1 ICache[10] and cache levels $1, \ldots, i-1$ are switched to the off state, regardless of their intervals, while memory level i is performing an access. Since no circuit can do work in the off state, the CI policy causes higher levels of the system to logically stall by extension.

To be secure, each S_i causes strictly data-independent behavior: i.e., if $S_i = \mathsf{SI}$, the higher levels never depend on whether memory level i is accessed, and *always* depend on memory level i for the CI policy. As with public intervals, we want the server to be able to set each S_i dynamically so that chip behavior can be tailored to different programs. Figure 5-5 shows one way to implement interval policies (for the architecture in Section 5.1) that matches the above definitions and constraints. The takeaway is that the logic cost to implement dynamic policies as opposed to fixed policies is cheap: several AND/OR gates per memory.

For the interested reader, Section 5.6.1 goes over the signal behavior in Figure 5-5 in detail. Sections 5.6.2-5.6.4 discuss some common policy combinations, and how the server can choose which one to use for a given program.

5.6.1 Interval Policy Gate-level Signal Behavior

This section discusses the timing in Figure 5-5 in detail. Following the figure, assume SI is encoded as logic 1 and CI is encoded as 0. As usual, each memory has an input buffer (shown as ❶ for the L1 DCache) which stores pending requests and data block evictions. We do

[10] There is a 0% chance that any L1 ICache access will do useful work if the pipeline is switched to the off state.

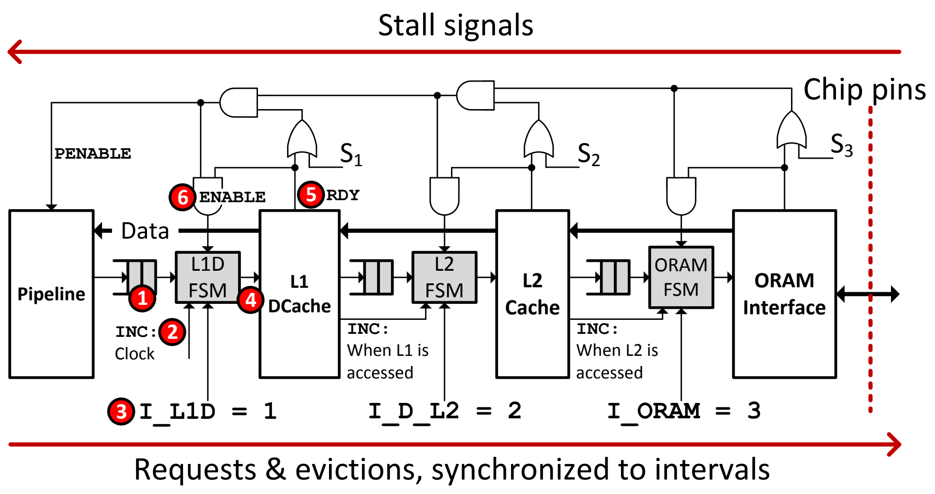


Figure 5-5: Interval policy gate-level implementation. $S_{1,\dots,3}$ implement interval policies as discussed in Section 5.6. In this design, SI is encoded as logic 1 and CI is encoded as 0. When the speculative policy is in effect for the L2 (for example), $S_2 = 1$. When the PENABLE signal is 1, the pipeline is switched to the on state.

not show the L1 ICache because its on/off state is tied to the pipeline (see previous section). At a high level, the buffer passes requests through to its memory only when allowed by its interval FSM (e.g., the L1D/L2/ORAM FSMs in the figure).

Consider how the pipeline interacts with the L1 DCache as an example. The INC signal (2) is pulsed to increment the L1D FSM's internal counter.[11] Initially, the L1 DCache is not performing an access and is in the off state. When the L1D FSM's counter reaches the value corresponding to the I_L1D input (i.e., the L1 DCache interval, 3):

1. The L1 DCache clears its RDY signal (5) to logic 0 and switches to the on state.

2. An access is made to the L1 DCache, which performs the steps discussed in Section 5.4.2. If there is any real data in the L1 buffer at this time, that data is passed to the L1 DCache (4); otherwise dummy work is sent. By the end of this step, any requests or block evictions to the L2 cache are now stored in the L2 buffer.

3. The INC signal into the L2 FSM is pulsed once. If the L2 FSM's counter reaches the interval threshold I_D_L2, repeat these steps for the L2 cache.

4. The L1 DCache sets its RDY signal to logic 1 and switches back to the off state.

The ENABLE signal (the output of the AND gate labeled 6) implements interval policies: we say that the INC signal only increments its FSM's internal counter when ENABLE is set to logic 1. As long as ENABLE is 0, no accesses will be made to the corresponding memory and that memory will not consume dynamic power. Examining the gate behavior in Figure 5-5, the ENABLE signal prevents the L1D FSM's internal counter from incrementing in one of three circumstances:

1. The L1 is currently servicing a request (i.e., its RDY signal is logic 0). We assume no pipelined memory accesses for simplicity.

[11] E.g., INC is pulsed every cycle for the L1D FSM.

2. The L2 cache is currently servicing a request and $S_2 = 0 = \mathsf{CI}$, which means the L2 was given a conservative policy.

3. The ORAM interface is currently servicing a request and $S_3 = 0 = \mathsf{CI}$.

Similar rules apply to the L2 cache and ORAM. With the ORAM, however, we will toggle the `ENABLE` to logic 1 when the ORAM read path operation completes, following the early completion optimization (Section 3.2.3). We will discuss intuition for the server choosing policies in the next few sections.

5.6.2 Example: Fully Speculative Policies

If the server wants (a) to maximize performance for a given interval setting or (b) has low confidence in the correct setting for each memory interval, it should set all memory levels to the *speculative interval* (SI) policy.

Performance maximization. By definition, a fully speculative policy means that the pipeline is always in the on state and all memories are accessed at their respective intervals, regardless of when any other memory is accessed. Thus, if a consumer memory is performing a dummy access, a fully speculative policy boosts performance because there is no program data dependency and the producer (a higher level memory or the instruction pipeline) can continue making forward progress. If the consumer is performing a real access, there is a program data dependency and the producer cannot make forward progress. A normal processor would stall when there is an outstanding dependency; to maintain security, the producer(s) must evaluate dummy work until the dependency is satisfied, wasting energy.

See Figure 5-6a for a detailed example. Almost immediately ($t = 2$ cycles), the pipeline reaches i1 which needs data from ORAM; thus all work performed until $t = 6027$ is dummy work—wasting a significant amount of energy. At the same time, once i1 retires the pipeline is able to execute 10 more instructions and reach the next memory-bound instruction i11—*before* the next ORAM access is made. Thus, the second ORAM access is a real access. The big point here is that the pipeline was only able to reach i11 in time because the pipeline was never switched to the off state while a memory was being accessed.

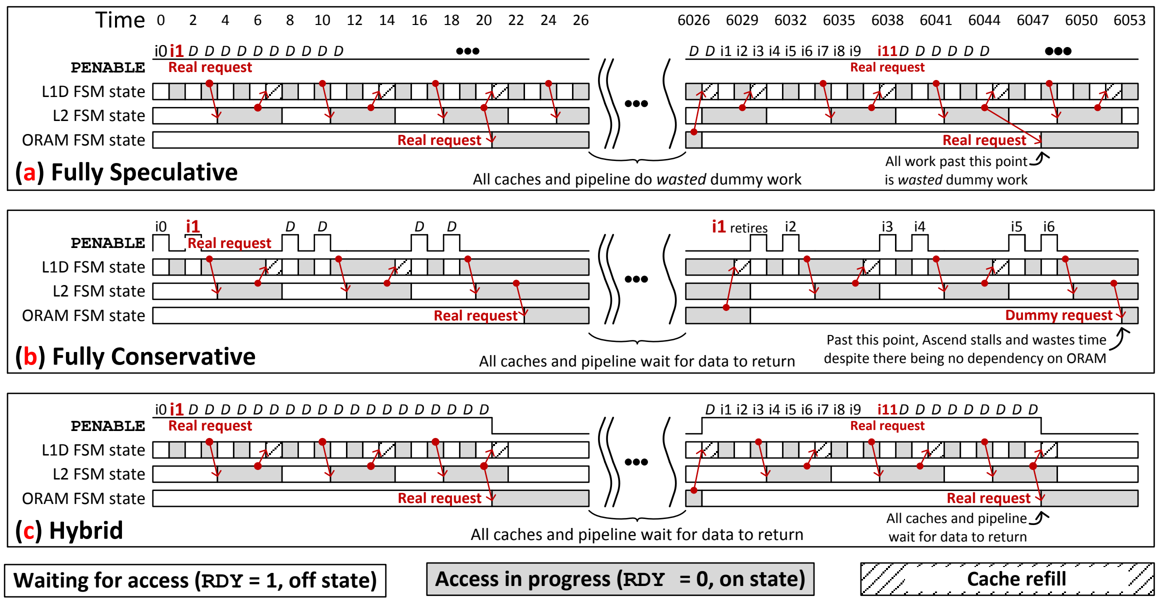
Low confidence. Another way to look at interval policies is through how confident the server is in setting interval values for a specific program. In Figure 5-6, `I_L1D/I_D_L2/I_ORAM` are set to 1/2/3, respectively, which the server must specify. If the server is not confident in this interval setting,[12] it expects Ascend's memories to be doing dummy work some of the time (in effect creating false data dependencies). To minimize performance loss due to false dependencies, the server should opt for the pipeline to never stall on a false dependency—which is the essence of the speculative policy.

5.6.3 Example: Fully Conservative Policies

If the server has high confidence that it set each interval correctly, it should set all memory levels to the *conservative interval* (CI) policy.

With this scheme, all higher levels in the system stall when a consumer is performing real/dummy work. If the consumer memory is performing real work, the conservative policy saves energy because higher levels and the pipeline are waiting on an actual program dependency. If the consumer was doing dummy work, time is wasted because the higher

[12] The program may not have been sufficiently profiled offline or may contain large amounts of data-dependent control flow that cannot be captured well with static, strict intervals.



Time
(a) Fully Speculative
(b) Fully Conservative
(c) Hybrid
PENABLE
L1D FSM state
L2 FSM state
ORAM FSM state
Real request
Dummy request
All caches and pipeline do *wasted* dummy work
All work past this point is *wasted* dummy work
All caches and pipeline wait for data to return
Past this point, Ascend stalls and wastes time despite there being no dependency on ORAM
All caches and pipeline wait for data to return
Waiting for access (RDY = 1, off state)
Access in progress (RDY = 0, on state)
Cache refill

Figure 5-6: How interval policy impacts system timing and energy consumption. Assume the signal names (`RDY`, `PENABLE`) from Figure 5-5. Instructions are labeled `i`#; *D* means dummy work was performed by the instruction pipeline. Arrows represent cache accesses. For each part of the diagram, only `i1` and `i11` access memory and both of these instructions need to access ORAM. `I_L1D`/`I_D_L2`/`I_ORAM` are 1/2/3, respectively. That is, the L1 DCache is accessed every cycle; the L2 cache is accessed every *second* L1 DCache access; the ORAM is accessed every *third* L2 cache access. Accessing the L1/L2/ORAM takes 1/3/6000 cycles, respectively.

levels could be making forward progress but are not allowed to preserve security. Note that the conservative policy only reduces dynamic power consumption: even when a memory or the instruction pipeline is in the off state, it still has leakage power.

As we will show in Chapter 7, the fully conservative policy is very sensitive to each cache's interval setting. See Figure 5-6b: as with the fully speculative policy, the conservative policy is able to reach `i1` before the first ORAM access is made. During the first 6026 cycles, the conservative policy is therefore the right decision: very similar amounts of forward progress are made relative to the speculative scheme but the energy consumption for the conservative policy is far less (i.e., when the ORAM is accessed, the rest of Ascend is clock gated). On the other hand, once the first ORAM access returns, the conservative scheme cannot reach `i11` by the time the second ORAM access is made. This is a result of the pipeline stalling when any memory access is outstanding. Hence, the second ORAM access is dummy and the system unnecessarily stalls for ~ 6000 cycles.

Livelock prevention. We point out that accessing caches and ORAM at synchronous intervals (see Section 5.1) is important in preventing livelock when the system uses a fully conservative policy.[13] Consider a design similar to Figure 5-5 where *every* interval is synchronous to the clock. E.g., the L1 cache is accessed I_{L1} cycles after the last L1 access completes, the L2 is accessed I_{L2} cycles after the last L2 access completes, etc. Given certain interval settings for each memory, the stall signal sent to the instruction pipeline may

[13]Note: this livelock is distinct from the livelock associated with ORAM background eviction; see Section 3.3.3.

always be high and no program forward progress will ever be made. For instance, if each interval is set to a very low value, the union of stall signals (see Figure 5-5) may always be high. By accessing consumer memory level i only after the producer memory level $i-1$ is accessed some integer number of times, the producer necessarily makes forward progress after the consumer completes each access.

5.6.4 Example: Hybrid Policies

Recall that the server can set a different policy for each memory: a hybrid policy refers to any policy where for any two memory levels i and j, $S_i = \mathsf{SI}$ and $S_j = \mathsf{CI}$. (Figure 5-5 supports these hybrids without change).

The hybrid we will focus on in this thesis applies the speculative policy to each cache and the conservative policy to the ORAM. This policy is based on the observation that the ORAM latency is orders of magnitude greater than the L1 or L2 latencies. In other words: the low-hanging fruit to saving energy while preserving performance is to maximize the chance of sending useful work to the ORAM. See Figure 5-6c: the hybrid policy is equivalent to the fully speculative scheme in terms of performance and almost as efficient energy-wise as the fully conservative scheme while the first ORAM access is outstanding. Also like the fully speculative scheme, the hybrid is able to reach i11 by the second ORAM access, once again achieving the best of both worlds.

Chapter 6

Certified Execution

Certified execution allows the user to detect whether a malicious server (a) passed off results obtained from a different program, (b) ran a program for some number other than T cycles, or (c) tampered with Ascend's external memory (i.e., ORAM) during execution. Ascend can obtain a certificate of execution as follows. During initialization Ascend computes and locally stores a hash of its inputs: $h = \mathsf{hash}(P||x||y)$ where $||$ is the concatenation operation. When the processor finishes after T cycles, the result of the computation is signed together with h and T. The signature together with P and y is sent back to the user.[1] In order to guarantee that, by verifying the signature, the user (or third party) is convinced that the result was produced after T cycles of executing $P(x, y)$, the processor needs to integrity-verify the external memory [17]. ORAM requests are made as described in Chapter 3, and as in the Aegis processor [20, 27] encrypted values read from the ORAM tree are integrity-verified.

6.1 Motivation

The first motivation for certified execution is to tolerate malicious servers that try to cheat by running for fewer than T steps, or running a program different from P, since the certificate will not verify. Even if the client is not capable of verifying the program hash against P, however, certified execution is still helpful in a multi-interactive protocol setting.

Multi-interactive protocols generalize the two-interactive protocol from Section 2.2 by allowing the user to send an intermediate result back to the server so that more work can be done. The benefit is that the protocol will yield $P(x, y)$ instead of an intermediate result; the downside is additional privacy leakage. Without certified execution, this leakage can easily break security: for each server-user interaction, the server can return the intermediate result after 1 additional cycle is run (for example). If the user interacts with the server x times, the server will suspect that the program took exactly x cycles to run. With certified execution, *the user controls* leakage: the user may ask the server to run for T_1 cycles, then T_2, etc.[2] The user will know for certain that each interaction will be run for the specified

[1] If a trusted third party certifies $\mathsf{hash}(P)$ to correspond to a trustworthy algorithm with P's specification, then instead of P the server may send back $\mathsf{hash}(P)$ together with y, the third party certificate and $h = \mathsf{hash}(\mathsf{hash}(P)||x||y)$ (generated by Ascend). In the case P is supplied by the server, this protects the server from revealing the detailed code of P. If P is supplied by the user, then P is known to the user, so, it does not need to be sent back.

[2] One strategy might be that $T_{i+1} = 2 \cdot T_i$.

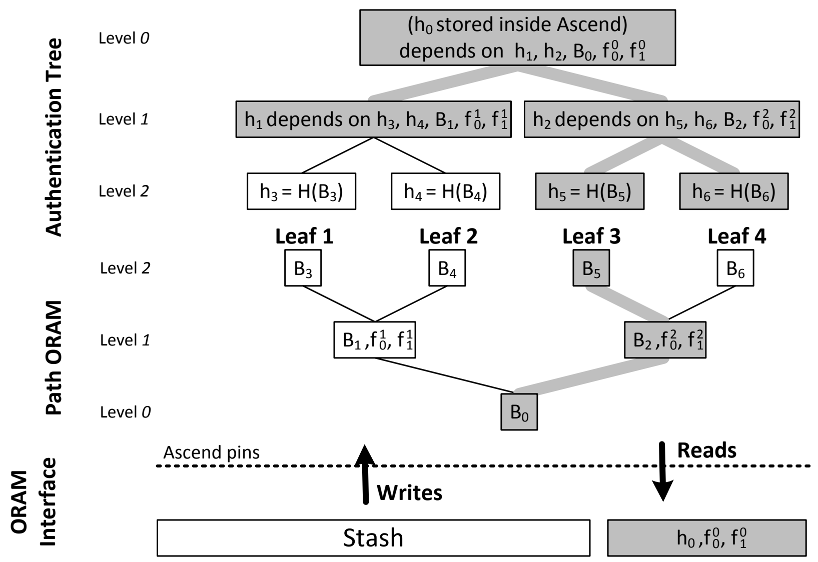


Figure 6-1: Integrity verification on top of Path ORAM. hash() is abbreviated H(). All data touched in external memory per Path ORAM access is shaded.

number of cycles, which allows the user to obfuscate the true running time by its choice of each T_i.

6.2 Integrity Verification

To implement certified execution, we build an integrity verification layer on top of Path ORAM (Section 3.2.1) to verify that all the retrieved blocks from the ORAM tree are *authentic*, i.e., they were produced by Ascend, and *fresh*, i.e., when a block is loaded from ORAM, it corresponds to the latest version that Ascend wrote to ORAM. The integrity verification layer allows Ascend to certify whether its computation is based on correct inputs from ORAM. As discussed previously, Ascend only interacts with the outside world through ORAM.

A strawman approach to implementing the integrity layer is to store a Merkle tree in external memory. Each leaf of the Merkle tree stores a 160-bit hash (e.g., if using SHA-1) of a data block in the ORAM. We note that this scheme would work with any kind of ORAM, and similar ideas are used in [52]. To verify a block, a processor needs to load its corresponding path and siblings in the Merkle tree and check the consistency of all the hash equations. This scheme has large overheads for Path ORAM (triples access latency), because all the $Z(L+1)$ data blocks on a path have to be verified on each ORAM access. So $Z(L+1)$ paths through the Merkle tree must be checked per ORAM access, which contain $Z(L+1)^2$ hashes in total. (Z and L are given in Section 3.2.1).

To avoid having to initialize the authentication tree at program start time,[3] we add two bits to each bucket—labeled f_0^i and f_1^i for bucket i and stored in external memory along with bucket i—that are conceptually valid bits for bucket i's children. We say bucket i is *reachable* from the root bucket if all valid bits on the path, from the root bucket to bucket i, equal 1. We define $\mathsf{reachable}(B_i) = 1$ if B_i was reachable at the start of a particular ORAM

[3]We assume that at start-up time, the authentication and ORAM trees consist of random bits corresponding to the uninitialized DRAM state.

access and $= 0$ otherwise. We maintain the invariant that all reachable buckets from the root bucket have been written to through ORAM operations at some point in the past.

Each intermediate node in the authentication tree now stores the hash of the concatenation of (a) child bucket valid flags, (b) the *corresponding bucket* in the Path ORAM tree, and (c) the sibling hashes for that intermediate node. Authentication works as follows: Suppose the root bucket is labeled B_0 and the root hash/child valid flags (stored inside the ORAM interface) are $h_0/f_0^0/f_0^1$ respectively. We initialize $h_0 = H(0)$ and $f_0^0 = f_1^0 = 0$ at program start time. Following the figure: to perform an ORAM access to block B_5 mapped to leaf $l = 3$, the ORAM interface performs the following operations:

1. ORAM path read: read B_0, B_2 and B_5 and child valid flags f_0^2, f_1^2.

2. Read sibling hashes for the path (h_1 and h_6).

3. Compute $h'_5 = H(B_5)$, $h'_2 = H(f_0^2||f_1^2||(f_0^2 \vee f_1^2) \wedge B_2||f_0^2 \wedge h'_5||f_1^2 \wedge h_6)$, $h'_0 = H(f_0^0||f_1^0||(f_0^0 \vee f_1^0) \wedge B_0||f_0^0 \wedge h_1||f_1^0 \wedge h'_2)$, where '$\vee$' and '$\wedge$' are logical OR/AND operators.[4]

4. If $h_0 = h'_0$, the path is authentic and fresh!

5. Update child valid flags: $f_0^{0\prime} = f_0^0$, $f_1^{0\prime} = f_0^{2\prime} = 1$ and $f_1^{2\prime} = f_1^2 \wedge \mathsf{reachable}(B_2)$. Update the root bucket child valid flags (inside the ORAM interface) to $f_0^{0\prime}$,$f_1^{0\prime}$.

6. ORAM path writeback: evict as many blocks as possible from the stash to the path 3 (forming B'_0, B'_2 and B'_5). Write $f_0^{2\prime}$,$f_1^{2\prime}$ as the new child valid flags for B'_2.

7. Re-compute h_5, h_2 and h_0; write back h_5 and h_2.

All data touched in external memory is shaded in Figure 6-1.

Note that only the sibling hashes need to be read in from the authentication tree. The hashes on the path of interest are computed by the processor, by hashing the buckets read via the Path ORAM operation concatenated to the sibling hashes. We point out that since hashes are computed from the leaves to the root, only the reachable portion of the path in the authentication tree needs to be read per access. That is, if the path to B_5 is being accessed (see above) and $f_0^0 = f_1^0 = 0$ at the time of the access, $h'_0 = H(0||0||0||0||0) = H(0)$, which is independent of any values in the authentication tree. Conceptually, the child valid flags indicate a frontier in the ORAM/authentication trees that has been touched at an earlier time.

In summary, on each ORAM access at most $L \ll (L + 1)^2 Z$ (sibling) hashes need to be read into the processor and L hashes (along the path) need to be written back to the external authentication tree. This operation causes low performance overhead beyond accessing ORAM.

[4]Note that $(f_0^i \vee f_1^i) \wedge B_i = B_i$ if $\mathsf{reachable}(B_i) = 1$ and is only needed to get the correct value for h'_0 before the first access is made. This OR-AND operation is applied to other non-leaf buckets for the sake of consistency, but is not required.

Chapter 7

Evaluation

To evaluate Ascend, we will first provide an argument for security; then, we will explore power and performance trade-offs.

7.1 Security

In order to argue Ascend's security, we need to show that at most a negligible amount of private information about user-specified inputs may leak. We consider an adversary $\mathcal{A}$ who experiments with Ascend by running its own programs on the private inputs with time budgets of its own choice and by controlling the communication from external RAM whose state is known to $\mathcal{A}$ at any time. $\mathcal{A}$ observes the power pins and I/O pins. The power trace from the power pins can be analyzed, e.g., by SPA and DPA (which, together, we refer to as PA for power analysis). The I/O pins only show the digital output of the ORAM interface, this includes the time when bits are outputted.[1] $\mathcal{A}$ uses the combination of all observed information to try to correctly predict some private information with non-negligible probability.

A security proof (sketch) for Ascend relies on three properties:

1. Below, we define power analysis (PA) resistance and use this property to show that if Ascend is PA resistant, then any adversary $\mathcal{A}$ can be simulated by a second $\mathcal{A}'$ who has no access to the power pins at all, but does have access to each data-independent parameter (e.g., server-specified intervals and policies) at every moment in time.

2. By assuming the semantic security of AES based on the user's session key and assuming the indistinguishability of the (keyed) PRNG output from truly random output, the security property of ORAM holds. This property states that the bit sequence that is exchanged over the I/O channel cannot be distinguished from bit sequences that are exchanged with the ORAM interface for random load/store request patterns that come from L2 (last-level) cache. This means that $\mathcal{A}'$ can be simulated by a third $\mathcal{A}''$ that only receives timing information about when output bits are transmitted over the I/O channel. ($\mathcal{A}''$ does not receive the bit values or a power trace).

3. Since the timing of transmissions is solely based on server/adversary specified intervals and interval policies, this timing information can be simulated by a fourth adversary

[1]Power fluctuations on top of the digital output are assumed to be smoothed away by a phase locked loop. An alternate assumption would be that the adversary is not monitoring power on the I/O pins.

$\mathcal{A}'''$ who receives no output from Ascend at all. That is, no non-negligible information about private encrypted inputs leaks to any of the adversaries, in particular, $\mathcal{A}$.

7.1.1 Circuit Requirements for PA Resistance

For our reduction from $\mathcal{A}$ to $\mathcal{A}'$ to work, we need for the circuits that Ascend is built upon to have the following definition of PA resistance (we only give an informal sketch here). Let BN be a circuit for which we want to define PA resistance. We assume adversaries who want to learn/predict information about BN's state variables s as a function of time. For example, if BN represents AES for some symmetric key K, then K is part of s. If BN represents Ascend, then the decrypted private inputs are at one time or another part of s. PA resistance should quantify to what extent the adversary is limited in learning more information about s than what would be possible without access to the power trace.

In our definition we consider probabilistic polynomial time (ppt) adversaries $\mathcal{A}$ who (1) may interact with BN as a black-box by only observing/controlling the power pins and I/O pins and (2) may also have access to some other side channel $\mathcal{O}$ that leaks information about s. We say BN is ϵ-PA resistant if, for all $\mathcal{O}$ and for all $\mathcal{A}$, there exists a ppt simulator $\mathcal{S}$ whose output distribution, given $\mathcal{O}$ but not BN's power trace, is "ϵ-indistinguishable" from the output distribution of $\mathcal{A}$. This definition says that whatever $\mathcal{A}$ predicts can be closely simulated (as quantified by ϵ) by a simulator who does not have access to BN's power trace.

Our definition in terms of simulators and side channels is useful for proving the security of a composition of primitives. Suppose that part of the output of BN_1 serves as part of the input to BN_2 and that $\mathcal{A}$ is given side channel $\mathcal{O}$. Then, $\mathcal{A}$ with access to a new side channel, given by the composition of BN_2 and $\mathcal{O}$, will try to attack BN_1. Since BN_1 is ϵ_1-PA resistant, there exists a simulator $\mathcal{S}'$ such that the output distributions of $\mathcal{A}$ and $\mathcal{S}'$ are ϵ_1-indistinguishable. Now adversary $\mathcal{A}'$ defined as $\mathcal{S}'$ with access to side channel $\mathcal{O}$ attacks BN_2. Since BN_2 is ϵ_2-PA resistant, there exists a simulator $\mathcal{S}$ such that the output distributions of $\mathcal{A}'$ and $\mathcal{S}$ are ϵ_2-indistinguishable. By using a triangle inequality, we conclude that the output distributions of $\mathcal{A}$ and $\mathcal{S}$ are $(\epsilon_1 + \epsilon_2)$-indistinguishable, hence, the composition of BN_1 and BN_2 is $(\epsilon_1 + \epsilon_2)$-PA resistant (notice that aggregation of the power pins to BN_1 and BN_2 in the composed circuit only improves PA resistance).

We note that our definition resembles the line of thought in recent work [48, 58] which corresponds to a definition of PA resistance where "ϵ-indistinguishable" is replaced by a similarity metric on what is observed by $\mathcal{A}$ and what is modeled by $\mathcal{S}$.

7.1.2 Relation to Circuit Primitives

Given the circuit primitives from Chapter 4, ϵ-indistinguishability roughly corresponds to normalized energy/standard deviation (NED/NSD) or the difference in power draw given the set of possible inputs to BN. Informally, our composability definition requires that each circuit's observable state (Section 4.2.4) change only as a function of data-independent events. In other words, the security of the entire system reduces to the security of the WDDL and `cell-1` SRAM primitives.

Given our primitives, BN is composed of an SRAM array, a Boolean network composed of gates/flip-flops or a combination of these. Consider BN_1 and BN_2 being ϵ_1 and ϵ_2-indistinguishable (where ϵ-indistinguishability corresponds to how NED/NSD is defined for each type of circuit—see Sections 4.2.1-4.2.2).

If BN_2 is composed of logic gates and flip-flops, its behavior is defined by when it is in the pre-charge and evaluation phase, which logic values are present at each input when it is in pre-charge and evaluation, and when clock enable is set (i.e., when BN_2 is in the on state—see Section 4.2.3). From previous discussion:

1. When clock enable is set is fully determined by public parameters in $\mathbb{P}$ (Section 4.2.3).
2. Given that clock enable is set: when BN_2 is in evaluation/pre-charge depends only on data-independent counters (Section 4.2.1.2).
3. The values on each wire during pre-charge are always the same (Section 4.2.1).

Thus, BN_1 can only influence BN_2 by changing which logic values are present in each evaluation phase.

If BN_2 is an SRAM array, its behavior is determined by its address a, data d, operation op (read/write) and clock enable inputs. Given the architecture proposed in Chapter 5, both clock enable and op are fully determined by public parameters in $\mathbb{P}$ and static behavior. For example, when the public interval starts an L1 DCache access (Section 5.4.2), a read (cycle 1, after the access starts) is followed by a write (cycle 3, after the access starts). Thus, BN_1 can only influence BN_2 by changing a and d into the SRAM array.

Thus, regardless of BN_2, BN_2 can only be influenced by factors already captured within its definition of NED/NSD. We conclude that the composition between BN_1 and BN_2 is $(\epsilon_1 + \epsilon_2)$-PA resistant.

7.2 Performance and Power

In this section we will evaluate Ascend's power and performance overheads relative to an insecure processor.

7.2.1 Methodology

7.2.1.1 Benchmarks

We evaluate processors like Ascend over a subset of the SPEC06-int benchmarks—SPEC puts stress on the memory hierarchy [34] and therefore tests Ascend's overheads (intervals, memory access obfuscation, the ORAM interface, etc) to the extent possible. We classify SPEC into three categories: memory bound, balanced and compute bound (Table 7.1). These categories are based on observed cache miss rates and will help us explain results.

Table 7.1: Benchmark categories.

Memory bound	**Balanced**	**Compute bound**
bzip2, libq, mcf	hmmer, astar, perlb, gcc, gobmk	h264, omnet, sjeng

7.2.1.2 Comparison Points

We compare the following processors:

baseline: An *insecure* processor that runs programs "in the clear." The insecure system has a normal memory controller to main memory, does not obfuscate circuit behavior/build any circuit out of DPA resistant logic or encrypt/decrypt any data. baseline has no public parameters: when some circuit (e.g., the cache) is given work, the work is carried out as soon as the circuit becomes available.

ascend_io: A version of Ascend that protects the I/O channel (Chapter 3) only. This design uses all ORAM-related optimizations mentioned in the thesis: use of background eviction, early completion, AES pad size reduction. ORAM is accessed at a public interval I_{ORAM} (i.e., $\mathbb{P} = \{I_{ORAM}\}$), but the chip only stalls when a real ORAM request is made.

ascend: The complete Ascend proposal. In addition to ascend_io, this design assumes WDDL/`cell-1` SRAM for concreteness (Section 4.2), applies DPA resistant logic and architecture-level optimizations as discussed throughout Chapters 4-5. This configuration is parameterized by the intervals from Section 5.1.3 and the interval policies from Section 5.6. We note that for SPEC benchmarks, the L1 ICache miss rate is $\sim 0\%$—thus we will fix I_{L1I} and $I_{I\to L2}$ to very large values and abbreviate $I_{D\to L2}$ to I_{L2}. Thus, $\mathbb{P} = \{ I_{L1D}, I_{L2}, I_{ORAM}, S_1, S_2, S_3 \}$, and we will vary these parameters in our experiments.

We show ascend_io to show an intermediate, yet meaningful, level of security. For instance, an adversary may not have access to the power pins at all (i.e., is only capable of performing software-based attacks).

7.2.1.3 Simulator and Metrics

We model processors with a cycle-level simulator based on the public domain SESC [14] simulator that uses the MIPS ISA (recall Section 5.3). Instruction/memory address traces are first generated through SESC's rabbit (fast forward) mode and then fed into a timing model (Table 7.2) and power model (Table 7.3). Each experiment uses SPEC reference inputs, fast-forwards 1-20 billion instructions to get out of initialization code and then monitors performance/power until 3 billion instructions worth of application forward progress is made (dummy instructions do not count). Performance-wise, this is equivalent to an instructions per cycle (IPC) metric. Power is measured in Watts. To get a holistic comparison, we will use a combined *power-performance product* which is given by:

$$\mathsf{PPP}(ascend\_conf) = \frac{\mathsf{IPC}(\mathsf{baseline})}{\mathsf{IPC}(ascend\_conf)} * \frac{\mathsf{PWR}(ascend\_conf)}{\mathsf{PWR}(\mathsf{baseline})}$$

where PWR() gives average power consumption in Watts. In other words, the PPP metric is Ascend's performance overhead times power overhead.

The top section in Table 7.3 (labeled **Events**) shows nanojoules per individual processor operation (assuming a 1 GHz clock, 45 nm technology, and no security countermeasures). Power consumption for baseline is modeled by counting the number of processor operations per simulation, multiplying each count by its energy expenditure in nanojoules, and converting the sum of these products into power expenditure in Watts.

AES energy (Ascend only) is taken from [40], scaled down to our frequency and up to a 1 AES block/cycle throughput; the FPU (E_{fpu}) is modeled after a single-precision fused multiply-add from [44]. We were not able to find comparable energy numbers for

Table 7.2: Processor timing model and baseline architecture. We assume the cache hierarchy organization from Section 5.1.

Core model	
in order, single issue, instructions advance 1 pipeline stage/cycle	
Pipeline stages per Arith/Mult/Div instr	1/4/12
Pipeline stages per FP Arith/Mult/Div instr	2/4/10
On-Chip Memory	
L1 I/D Cache	32 KB, 4-way
L1 I/D Cache hit+miss latencies	1+0/2+1
L2 Unified/Exclusive L2 Cache	1 MB, 16-way
L2 hit+miss latencies	10+4
Cache/ORAM block size ($B/8$)	128 Bytes
L1/L2 Cache bus bandwidth ($F/8$)	64 Bytes/cycle
On/Off-chip network/pin bandwidth (P)	16 Bytes/cycle
Main memory (DRAM/ORAM) capacity	4 GB

the ALU (E_{alu}) so we approximate it to be the same as the FPU.[2] E_{pins} models each pin transition as the amount of energy stored in a capacitor (given by $\frac{1}{2}CV^2$), assuming 1 pF capacitance 1 V. Other energies (caches, register files (rf), cache buffers, etc) are taken from CACTI [13]. To orient the reader, the IPC and power consumption for baseline given our timing and power model is shown in Table 7.4.

The **Compound Ops** section in Table 7.3 shows each *compound processor operation* assumed for ascend. Additional overheads from DPA logic are discussed below, in Section 7.2.1.5. #(**ORAM transfers**) is the number of 128-bit data transfers that occur per ORAM access. The number of compound operations can be converted into Watts in the same way as with the insecure processor.

7.2.1.4 Cache Energies

All L1 caches in our evaluation are designed to be parallel tag/data array lookup (i.e., the design in Figure 5-2). This is a high performance design, representative of real L1 cache designs in the industry. The L2 caches have serial tag/data array lookups, allowing *only a single way* to be activated per access. This again follows industry: larger highly-associative caches that are not on a processor's critical path should employ a more power-conscious design. Recall the discussion from Section 5.4.3: to hide which cache way contains the block of interest in the L2 cache, we increase the energy consumption per L2 access for ascend by 16×—the number of ways in the cache, given by #(**L2 ways**) in Table 7.3. We will not try to approximate the overhead of accessing multiple subarrays per way, but remark that the L1/L2 caches generated by CACTI in Table 7.2 contain 2/4 subarrays per way, respectively.

7.2.1.5 Additional DPA Resistant Logic Overheads

To model the overhead from DPA resistant logic (modeled after WDDL and `cell-1` SRAM cells, see Section 4.2), we:

[2]Note that since the L1 ICache is accessed constantly (Section 5.1.3), the ALU/FPU energies are insignificant.

Table 7.3: Power model for processor operations. $srcL1$ is used to indicate which L1 cache will be refilled on an L2/ORAM access, and can be one of $L1I$, $L1D$, $(L1I, \textsf{ascend})$ or $(L1D, \textsf{ascend})$.

$$
\begin{aligned}
&\textbf{Events}: \\
E_{rf,int} &= .0032 \; // \; \textbf{two read, one write} \\
E_{rf,fp} &= .0048 \; // \; \text{“ ”} \\
E_{fpu} &= .0148 \; // \; \textbf{one instruction} \\
E_{alu} &= E_{fpu} \; // \; \text{“ ”} \\
E_{L1D} &= 0.609 \; // \; \textbf{read/write} \\
E_{L1I} &= E_{L1D} \; // \; \text{“ ”} \\
E_{L1,refill}(srcL1) &= \left\lceil \frac{B}{F} \right\rceil * E_{srcL1} \\
E_{L2} &= \left\lceil \frac{B}{F} \right\rceil * .766 \; // \; \text{“ ”} \\
E_{pin} &= .064 \; // \; \textbf{per 128 bit (P = 16 Bytes) transfer} \\
E_{local\$} &= .0588 \; // \; \textbf{per 128 bit read/write} \\
E_{AES} &= .33 \; // \; \textbf{per 128 bit block} \\
&\textbf{Compound Ops}: \\
E_{I.pipe,\textsf{ascend}} &= (E_{rf,int} + E_{rf,fp} + E_{fpu} + E_{alu}) \\
E_{L1D,\textsf{ascend}} &= 2 * E_{L1D} \; // \; \textbf{read + write} \\
E_{L1I,\textsf{ascend}} &= E_{L1I} \; // \; \textbf{read} \\
E_{L2,\textsf{ascend}}(srcL1) &= 2 * (\#(\textbf{L2 ways}) * E_{L2}) + 2 * E_{L1,refill}(srcL1) \\
&\quad // \; \textbf{read + write to L2, lazy L1 evictions, way obfuscation} \\
E_{ORAM} &= \#(\textbf{ORAM transfers}) * (E_{pins} + E_{AES} + E_{local\$}) + \\
&\quad 2 * (E_{L1,refill}(L1I) + E_{L1,refill}(L1D))
\end{aligned}
$$

Table 7.4: Representative baseline IPC/Power (in Watts) over SPEC reference inputs.

	bzip2	libq	mcf	gobmk	sjeng	hmmer	astar	perlb	gcc	h264	omnet
IPC	0.292	0.298	0.203	0.375	0.391	0.358	0.360	0.328	0.329	0.310	0.348
Power	0.283	0.232	0.218	0.303	0.308	0.301	0.301	0.296	0.299	0.309	0.304

1. Adjust cache access/instruction cycle latencies to take into account circuit pre-charge. With our cycle latency reduction optimizations from Sections 5.2 and 5.4, this typically adds 1 cycle to the baseline latencies in Table 7.2. We do not model additional cycle latency per SRAM access, if there is indeed any in practice (Section 4.2.2.2).

2. Increase all circuit power consumptions in Table 7.3 by 2× (to account for complementary logic and secure SRAM cells) with two exceptions:

   (a) E_{pins} does not increase because the pins themselves are not DPA resistant.

(b) E_{AES} increases by $4\times$: $2\times$ for the complementary logic overhead and $2\times$ because we need two times as many AES blocks to maintain 1 AES block/cycle throughput (due to WDDL pre-charge reducing throughput by $2\times$).

We do not normalize for area between the baseline and ascend, and assume the same clock frequency in all experiments. In some cases, area overhead from primitives such as WDDL is straightforward: For example, if baseline supports an inter-cache bus of F bits (which requires F wires), ascend will require $F' = 2 \cdot F$ to achieve the same bandwidth, as each wire must be complemented. Furthermore, `cell-1` SRAM arrays and related work are reported to incur a 40%-70% area overhead [31, 36]. The area overhead for large WDDL circuits (e.g., the pipeline) is unclear due to unpredictable place-and-route tools.

We remark that should ascend be area-normalized to baseline, the size/associativity/etc parameters for different chip memories/etc (Table 7.2) should be re-evaluated. For example, suppose the area overhead of each ascend component is $2\times$. The naïve solution is to reduce the L2/L1 capacity by 50%. A better solution would be to keep the L1 capacity the same and reduce the L2 by more than 50%.

7.2.2 Path ORAM in Secure Processors

In this section, we will determine which ORAM configuration to use for the rest of the chapter. In all experiments, we assume an ORAM with a 4 GB capacity that stores up to 1 GB of program data (which is sufficient for the SPEC workloads [35]) and only consider configurations whose ORAM interface can be implemented using < 200 KB of dedicated on-chip storage. With these constraints, we will need to use the Recursive Path ORAM construction (Section 3.2.2). Recall that for the Recursive Path ORAM:

1. On-chip storage consists of each ORAM's local cache and the smallest ORAM's position map.

2. Each ORAM needs to be accessed on each memory request, which gives an access latency of $t = \sum_{i=1}^{X} CyclesPerAccess_i$ cycles, where X is the number of Path ORAMs in the recursive construction and $CyclesPerAccess_i$ is the cycle latency to access ORAM i, given by Equation 3.1.

7.2.2.1 Choosing ORAM Block Size

We will first optimize each ORAM's data block size B. From Equation 3.1, we know that decreasing the block size B for ORAM i reduces the cycle latency for ORAM i. From Section 3.2.1.4, we see that to get an ORAM with constant capacity as B decreases, N (the number of data blocks in the ORAM tree) must increase. Thus, the position map for ORAM i increases as B decreases. This gives a trade-off: to decrease access latency for ORAM i, more ORAMs may be needed to reduce the final position map to a reasonable size manageable in on-chip storage.

In all experiments, we will fix the block size B for the data ORAM. Recall that B for the data ORAM is also Ascend's on-chip cache block size (see Table 7.2), which is set to exploit spatial locality in programs. There is, however, no reason to use a large block size for the smaller (position map) ORAMs because all we need from each position map ORAM per access is a leaf label, which is typically smaller than 4 Bytes. Thus, a smaller block size may be preferred for all position map ORAMs.

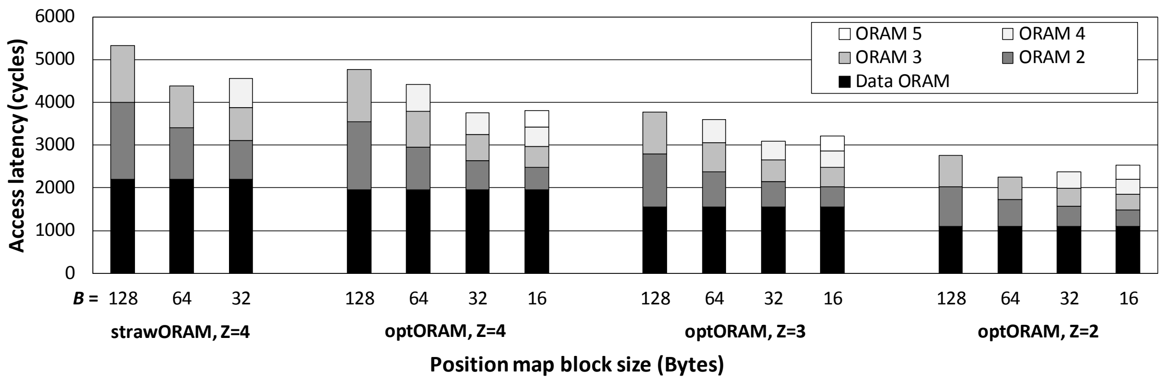


Figure 7-1: Latency breakdown in cycles for each ORAM in the Recursive Path ORAM construction. The number of ORAMs for each configuration is set to minimize overall access latency, subject to the constraint that on-chip storage in the ORAM interface is less than 200 KB.

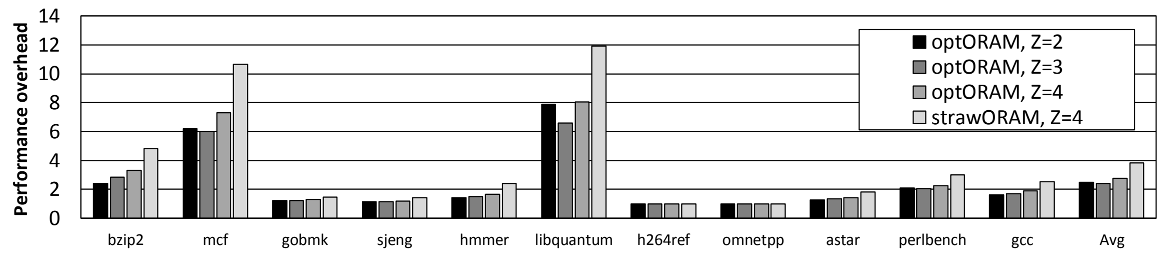


Figure 7-2: Performance impact, varying Z and Path ORAM optimizations.

Figure 7-1 shows the optimal block size for the position map ORAMs. In all experiments, each position map ORAM is given the indicated block size. strawORAM uses the strawman encryption scheme (Section 3.2.1.1) and optORAM uses the counter-based encryption scheme (Section 3.2.1.2). For each block size, we add Path ORAMs to the Recursive construction until the total on-chip storage is less than 200 KB, *assuming each local cache is limited to 100 entries, plus the length of one path.* The local cache constraint is important: in the figure, it is not clear whether the $Z = 3$ or $Z = 2$ configurations will be usable, since those configurations can lead to large, average local cache occupancies (Section 3.3) if our background eviction technique (Section 3.3.2) is not used. *If* $Z = 3$ or $Z = 2$ are usable, they can achieve a 41% and 93% reduction in raw access latency, relative to strawORAM, $Z = 4$.

Notice that for strawORAM, $Z = 4$, decreasing the position map block size hits diminishing returns and can't achieve its best performance point if the position map block size is < 64 Bytes. This is because the overhead to encrypt each block in strawORAM is large: a 128-bit pad per block. optORAM, on the other hand, can achieve better performance with a 32-Byte position map block size, when $Z = 4, 3$—leading to an overall performance increase. optORAM eventually hits diminishing returns: if the position map block size is < 32 Bytes, or if $Z = 2$, the counter dominates the size of each block.

7.2.2.2 Choosing Z, the Number of Blocks Per Bucket

We now evaluate the performance-optimal configurations from Figure 7-1 on the SPEC benchmarks, and show the effect of our optimization techniques.

Table 7.5: ORAM configurations used for performance comparison in Section 7.2.2. strawORAM assumes no optimizations from the thesis and is equivelent to an ORAM configuration used in [50]. optORAM assumes all optimizations. The $x/2$ cycle latency modifier refers to the early completion optimization—ORAM latency for those configurations is $x/2$ cycles, but a new ORAM request can only be made once every x cycles.

Design Pt.	Data/Pos. Map ORAM Block Size B	# ORAMs	Latency (cycles)
strawORAM, $Z = 4$	128/128	3	5362
optORAM, $Z = 4$	128/32	4	3752/2
optORAM, $Z = 3$	128/32	4	3090/2
optORAM, $Z = 2$	128/64	4	2260/2

Figure 7-2 compares end-to-end performance using the designs in Table 7.5. All performance overheads are relative to baseline. All optORAM configurations use background eviction, early completion and the counter-based encryption scheme. The strawORAM uses none of these optimizations—and is therefore limited to $Z = 4$—for comparison purposes. Overall, optORAM, $Z = 3$ with a 32 Byte position map block size performs best, and reduces average execution time by 58% compared to strawORAM, $Z = 4$. Furthermore, optORAM, $Z = 3$ incurs only $2.4\times$ performance overhead relative to baseline. Thus, we will use optORAM, $Z = 3$ as our ORAM configuration for the rest of the chapter.

Noteworthy is that optORAM, $Z = 3$ improves upon optORAM, $Z = 2$ by 3%, despite the smaller access latency for optORAM, $Z = 2$. This is because optORAM, $Z = 2$ requires a greater number of background evictions per access, as was shown in Table 3.1. Not surprisingly, the improvement is most significant for the memory bound benchmarks (libquantum, mcf and bzip2).

7.2.3 Public Parameter Design Space Exploration

In this section, we will show the performance/power impact from public intervals which, together, completely specify Ascend's observable behavior. Recall from Section 1.3: the server can profile each program that it plans to run offline to learn that program's common case behavior. The server can then set each public parameter for each program, when that program is run on hidden user inputs, based on this common case behavior.

To imitate this setup, we will use two different inputs for every benchmark: *a training input* and *a test input*.[3] When possible, both the training and test inputs are SPEC reference inputs. In certain cases (e.g., mcf, sjeng, libquantum) there is only one reference input and the training input is set to a SPEC training input. The only exception is libquantum, where we chose a large pair of composite numbers to factor (mimicking the reference input).[4]

The training input is public and owned by the server. For each benchmark P and corresponding training input, we will sweep values in $\mathbb{P}$ such that power-performance product is minimized—giving us a concrete setting for each parameter, which we refer to as $\mathbb{P}_{P,\mathsf{train}}$. We then compare baseline, running the test input, with an Ascend configuration running the test input, where public parameters in Ascend are set to $\mathbb{P}_{P,\mathsf{train}}$.

[3]This evaluation will only use a single training input per benchmark. Thus, our results may be improved through using additional training inputs and taking the best average parameter setting.

[4]We found that libquantum's behavior was similar for many sufficiently large composite numbers.

Note that for any setting of $\mathbb{P}_{P,\mathsf{train}}$, the server will be able to predict Ascend's power consumption exactly given any user input. (If this was not the case, Ascend would not be secure). Thus, if $\mathbb{P}_{P,\mathsf{train}}$ results in a power consumption that exceeds some power budget, the server can change $\mathbb{P}_{P,\mathsf{train}}$ in a way that minimizes performance degradation for the training input.

7.2.3.1 Performance/Power for ascend_io

This section gives the overhead for ascend_io, which completely hides program behavior on the I/O channel only. Using the results from Section 7.2.2, Figure 7-3 shows the performance and power overheads for each benchmark when the ORAM is set to an interval. In the figure: train shows, for each benchmark and training input, the power/performance overhead for the value of I_{ORAM} (swept between 1 and 2000 clock cycles) that minimizes power-performance product. test shows power/performance on the test input, using the same 'optimal' value for I_{ORAM} as used on the training input for the same benchmark.

Generally, the I_{ORAM} value chosen by the training input yielded similar performance/power overheads for the test input. Several exceptions are astar, hmmer, bzip2 and perlbench—which actually perform better with test than with train. This is due to test having (coincidentally) less dependence on external memory than train. For instance, the L1D/L2 miss rates for bzip2 with train/test are 1.3%/38% and 0.5%/25%, respectively. The other outliers are gcc and mcf, which perform worse with the test input for the same reason that bzip2 performs better. We point out that this is not unexpected for gcc—control flow and data accesses for that benchmark depend heavily on the input. For mcf, L1D/L2 Cache miss rates for the training input are 27%/11% and the corresponding test input miss rates are 14%/27%. This suggests that the training point should perform worse, since its L1 DCache miss rate is substantially higher than that of the test input. Since the test point has a higher L2 Cache miss rate, however, the average number of cycles between each ORAM request is still lower than with the training point.[5] Thus, the dependence on ORAM is greater for the test input and performance suffers.

Overall, preventing leakage on the I/O channel causes an average performance/power overhead of 2.6×/2.2×, respectively, relative to baseline. It is noteworthy that setting ORAM to an interval causes only 9.5% additional performance overhead, relative to running ORAM without intervals (Section 7.2.2.2).

7.2.3.2 The Impact of Intervals and a Fully Speculative Policy

We will now evaluate the fully speculative interval policy (Section 5.6.2) for the complete Ascend proposal (ascend). Figure 7-4 (top/middle/bottom) shows how IPC and power, given the training input, varies as $I_{L1D}/I_{L2}/I_{ORAM}$ vary, respectively. For example, Figure 7-4 (top) fixes I_{L1D} to a specified value, fixes $S_1 = S_2 = S_3 = 1$, and sweeps I_{L2}/I_{ORAM}. Each point (with corresponding power consumption given as the red dot) corresponds to the parameter setting that maximizes performance. We vary I_{L2} and I_{ORAM} between 1 and 50 for all experiments. To constrain the design space, we sweep I_{L1D} from 1 to 5 but additionally show the result for when $I_{L1D} = 50$ for consistency.

[5] We note that the % of dynamic instructions that access memory is similar for training/test inputs. So, to estimate the average number of cycles between when each input needs to access ORAM, one can use $\frac{\text{L1D\$}_{\text{hit}}}{L1D\$_{miss\%}} * \frac{\text{L2\$}_{\text{hit}}}{L2\$_{miss\%}}$, where the numerators refer to cycle latency per access and denominators refer to miss rates. Through this check, the test input needs to access ORAM 27% more than the training input.

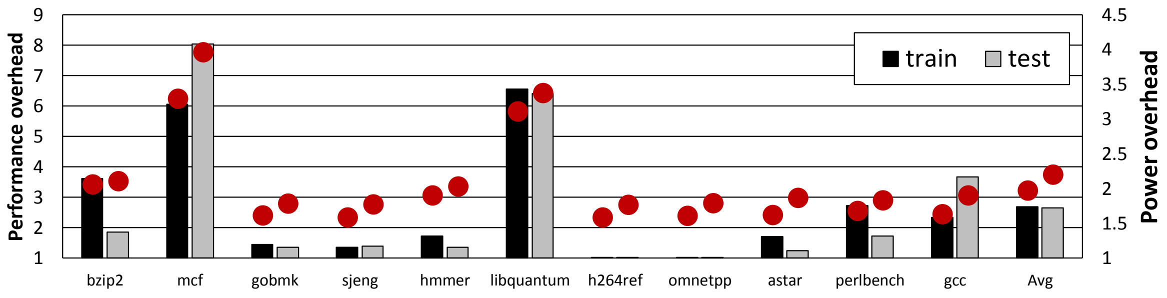


Figure 7-3: The best training point (in terms of power-performance product) and resulting test point given the same values for I_{ORAM}. Power is shown as the red dot.

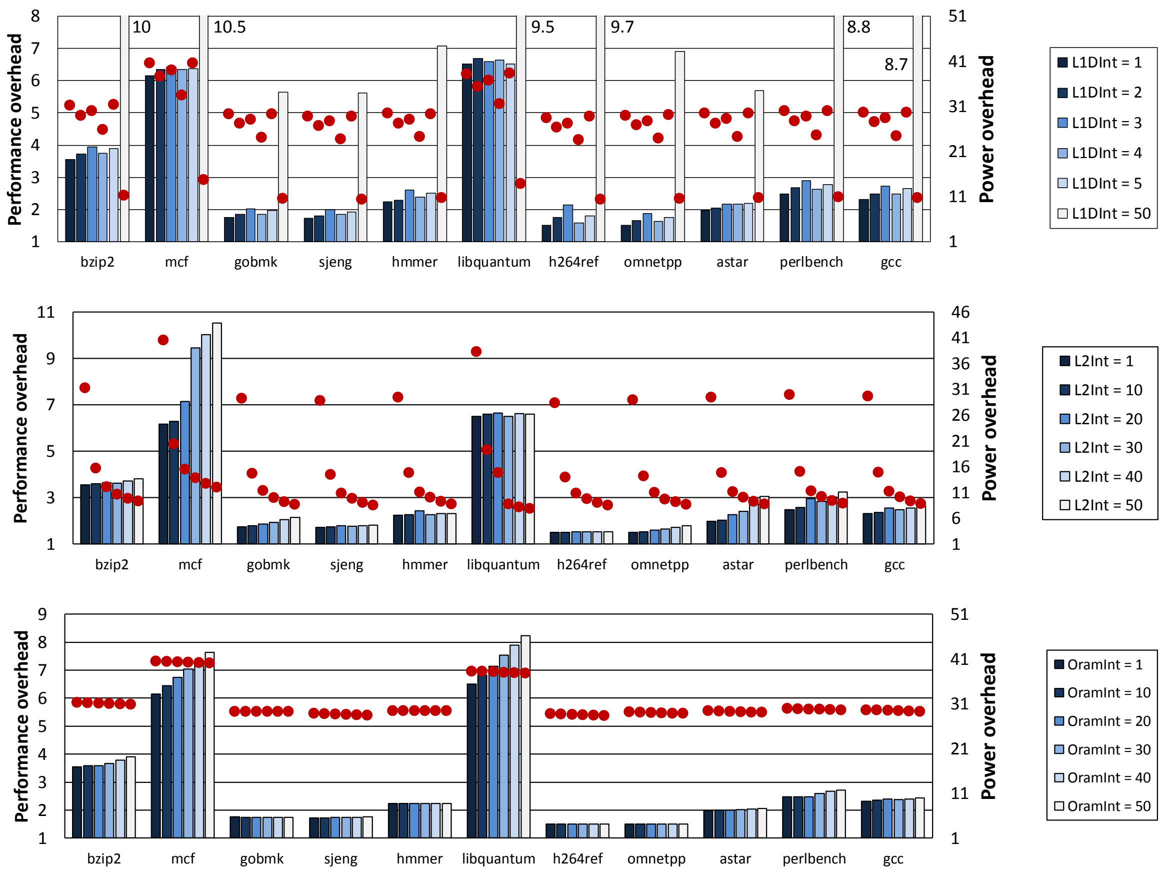


Figure 7-4: Performance and power impact from varying public intervals with a fully speculative policy. Each graph fixes the interval shown in the legend (right) and varies the other intervals to find the point that maximizes performance. Each bar represents performance slowdown and each dot represents power overhead.

Varying I_{L1D} (Figure 7-4 (top)) has the most impact on the compute bound benchmarks. For instance, varying I_{L1D} from 1 to 3 decreases performance for h264 by 42%, but only decreases power consumption by 4.2%. This is because h264 has a very low L1 DCache miss rate (< .1%) and a % dynamic instruction composition that is 52% memory instructions—making its performance very sensitive to the L1 DCache hit latency. Other compute bound benchmarks are similar. (We focus on varying I_{L1D} between 1 and 5 for performance

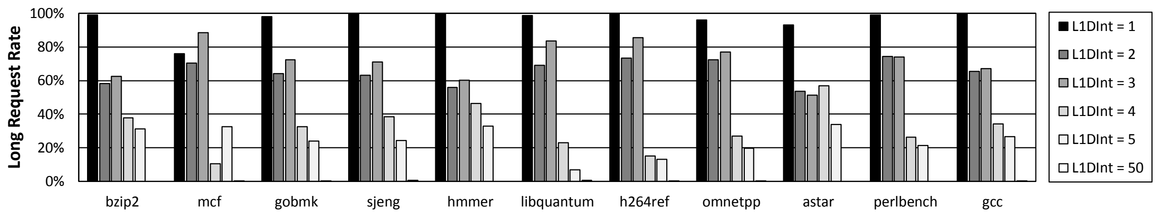


Figure 7-5: The percentage of long requests sent to the L1 DCache, for the points shown in Figure 7-4 (top).

reasons, but show $I_{L1D} = 50$ for consistency).

Notice that peformance does not degrade monotonically as I_{L1D} increases (e.g., $I_{L1D} = 4$ yields better performance, on average, than $I_{L1D} = 3$). This is because our instruction pipeline does not fetch a new instruction until the last instruction retires, which was a design decision and not fundamental (Section 5.2). Due to our pipeline, $I_{L1D} = 3$ is usually too small and new memory requests often occur when the L1 DCache is servicing an outstanding real/dummy request (we refer to this case as a *long L1 DCache request*). Long requests incur a potentially large performance penalty (i.e., a L1D$\$_{\text{hit}}$ + I_{L1D} cycle L1 DCache hit latency, where $L1D\$_{hit}$ is the cycle latency for an L1 DCache hit, without intervals). Thus, long requests impact performance more heavily as $I_{L1D} = 1 \rightarrow 3$. Figure 7-5 shows the long request rate for each point shown in Figure 7-4 (top). Notice that between $I_{L1D} = 3$ and $I_{L1D} = 4$ (e.g., for h264), both the performance overhead and percentage of long L1 DCache requests drops significantly.

Varying I_{L2} (Figure 7-4 (center)) has a dramatic impact on overall power. This is because $E_{L2,\mathsf{ascend}}$ (see Table 7.3) depends on the number of ways in the L2 cache—for our parameter setting, an L2 access costs ascend ~ 51 nJ, whereas the baseline system can perform the same access for between $1.5 - 3$ nJ (depending on whether an L1 eviction occurred). One way to decrease this overhead is to re-architect the L2 Cache and L2 access protocol. We assume lazy L1 evictions (which marginally increases the cost of an L2 access) and a read+write compound L2 access (Section 5.4.6). We could also reduce the number of L2 ways, or perform the write operation at a separate interval (i.e., if the L2 is inclusive and the program has a significant amount of read-only data, as explained in Section 5.4.2).

Without making any hardware changes, however, a simple yet effective solution is to just increase I_{L2} (causing the L2 cache to be accessed less frequently). By increasing I_{L2} from 1 to 50, overall power consumption drops by a factor of 3.5×, yet performance (on average) drops only by 22%. The memory bound benchmarks are corner cases—in partic-

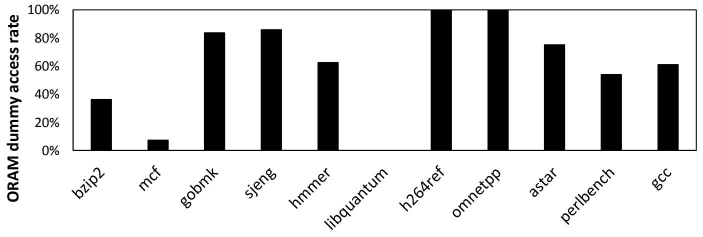


Figure 7-6: The percentage of dummy requests made to the ORAM, for the point shown in Figure 7-4 (bottom) that constrains $I_{ORAM} = 1$. Conceptually, this is the point that maximizes the dummy request rate to ORAM.

ular, increasing I_{L2} causes mcf to drop in performance by 66%, yet the same change seems to have no performance consequence for libquantum. This is due to the early completion ORAM optimization (Section 3.2.3). Not shown: libquantum has a 99% L2 cache miss rate and mcf has an 11% L2 miss rate. With early completion, benchmarks can continue making forward progress as soon as the read path operation (Step 3 in accessORAM() in Section 3.2.1) completes. Because libquantum (almost) always misses the L2 cache, it is always able to make a real request to the ORAM between when the Path ORAM writeback operation (Step 5) starts and the next call to accessORAM() is made—regardless of I_{L2}. This is important because it allows us to dramatically decrease the power consumption for libquantum "for free." mcf, on the other hand, has a lower L2 miss rate and therefore needs to access the L2 cache more times (i.e., do more work) in order to generate a real request to ORAM.

Varying I_{ORAM} (Figure 7-4 (bottom)) has the greatest impact on the memory bound benchmarks (mcf and libquantum in particular). We can tell that the drop in performance is due to I_{ORAM} being overset (i.e., is too large) for two reasons. First, as I_{ORAM} increases, memory bound benchmark performance decreases. Second, on average, those two benchmarks make dummy accesses to ORAM $\sim 3.6\%$ of the time (see Figure 7-6)—generally, the faster ORAM is accessed, the better. The early completion ORAM optimization yields a very small ORAM dummy rate, even for the point shown in the graph (where $I_{ORAM} = 1$). Notice that varying I_{ORAM} does not have a big impact on power. The lack of impact is because changing I_{ORAM} does not impact the values for other intervals (as opposed to I_{L1D}, discussed in Section 5.1.3).

7.2.3.3 Performance/Power for ascend (Speculative Policy)

Putting ideas from the previous section together, Figure 7-7 shows the performance/power overheads for ascend running each benchmark on both training and test inputs. This experiment re-uses the methodology from Sections 7.2.3-7.2.3.1, constrained to a fully speculative interval policy.

As with ascend_io, certain benchmarks perform better on the test input than the training input. Let us reconsider bzip2, whose L1D/L2 miss rates with train/test are 1.3%/38% and 0.5%/25%, respectively. Since miss rates for the test input are smaller than with the training input, each interval is set too low for test. Yet, the test input still performs better than the training input because the fully speculative policy allows forward progress to be made in higher levels (e.g., the pipeline) while lower levels (e.g., the L1 DCache) are servicing dummy requests. Overall, the fully speculative policy has an average performance and power overhead of 3.6× and 6.9×, respectively, relative to baseline.

7.2.3.4 Case Study: Memory Bound libquantum

We will now perform a more in-depth analysis of libquantum, a memory bound benchmark. For reference, relevent statistics for libquantum are shown in Table 7.6. libquantum is interesting because its % memory instructions and L1 DCache miss rate suggests that it is not memory bound, but its L2 miss rate is very high (99%), effectively making it memory bound in an Ascend setting.

Since libquantum's L2 miss rate is so high, it spends most of its time accessing ORAM (as stated in Section 7.2.3.2). For instance: nearly all settings for each public interval yielded a $\sim 0\%$ ORAM dummy request rate. Thus, intuition suggests that a hybrid interval

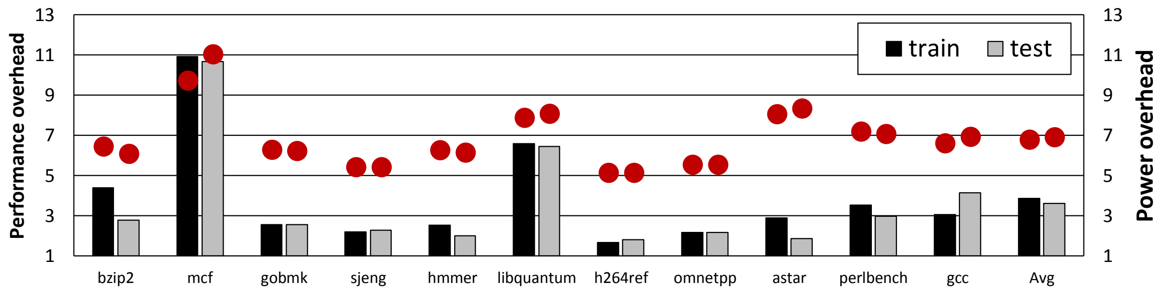


Figure 7-7: The best training point (in terms of power-performance product) and resulting test point given the same values for each interval. Power is shown as the red dot. All points use a fully speculative policy.

Table 7.6: Relevant benchmark statistics for libquantum, for the training and test inputs. % memory instructions refers to the dynamic instruction sequence per input. All statistics are taken over 3 billion real instructions.

	training	test
% memory instructions	18	28
L1 DCache miss rate (%)	4.1	2.9
L2 cache miss rate (%)	99	99

policy (Sections 5.6.4) that switches the caches and pipeline to the off state, while ORAM is being accessed, will reduce energy without impacting performance. Figure 7-8 (left) shows Pareto performance/power curves for a fully speculative, fully conservative (Section 5.6.3) and hybrid policy for libquantum only. For this experiment, we swept I_{L1D} between $1-5$, and I_{ORAM} between $1-50$, as done in Section 7.2.3.2, for each interval policy. We widened the search space for I_{L2} to be between $1-100$, to ensure that any result would not have artificially high energy consumption due to accessing the L2 too often (see the discussion of I_{L2} in Section 7.2.3.2).

The main result is that the hybrid policy causes the fully speculative policy's Pareto curve to shift down: for the training input, power consumption drops by 56% (relative to the fully speculative policy), while performance degrades negligibly ($\sim$ 1%). Thus, using the hybrid interval policy is a compelling choice for libquantum. Figure 7-8 (right) shows power/performance for libquantum when run with the test input. For this experiment: we manually selected a $\mathbb{P}$, constrained to the hybrid policy and using training inputs, whose performance was equal to the fully speculative points[6] and that minimized power consumption for this performance level. This was done to get the desired effect: a drop in power consumption for negligible loss in performance. We then, as usual, applied this setting of $\mathbb{P}$ to the test input; we see that the test input behaves very similarly to the training input. This result, when combined with the results from the last section, give us an average performance and power overhead of $3.6\times$ and $6.6\times$, respectively, relative to baseline.

The reason that libquantum gets reproducible power efficiency with the hybrid scheme is that its L2 miss rate is high and predictable. During each ORAM read path operation (Step 3 in accessORAM() in Section 3.2.1), the Ascend chip (aside from the ORAM interface) switches to the off state to conserve energy. Since the ORAM dummy rate is almost

[6]I.e., a point on the performance wall of IPC = .045 in Figure 7-8 (left).

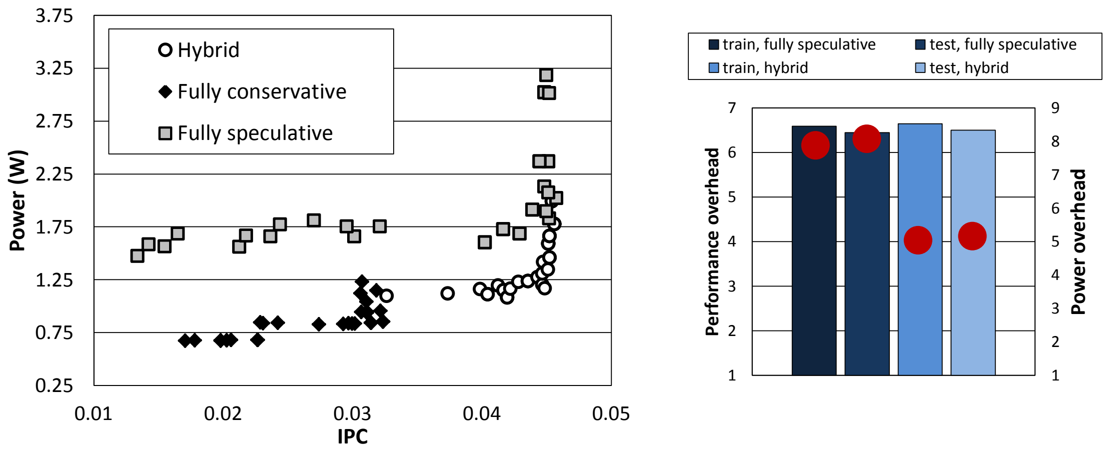


Figure 7-8: libquantum case study. (Left) Pareto curves for different interval policies. (Right) Power/performance improvements for the hybrid policy over the fully speculative policy.

0%, this does not severely impact performance. While the ORAM writeback operation (Step 3 in accessORAM()) is happening, Ascend is in the on state and the right strategy in terms of setting intervals is for the Ascend chip to perform as little work as possible (reducing energy) such that the next L2 access is made as close to, but not after, the ORAM writeback operation completes. Finally, the next ORAM read operation begins and the process repeats.

Notice that the fully conservative policy still achieves lower power consumptions than either the fully speculative or hybrid scheme. At the same time, performance suffers dramatically (40%) even for the point with the highest performance. This drop in performance is not necessarily fundamental and may be an artifact of constraining our design space exploration for each public interval value. This presents a problem: if the conservative policy's optimal setting for each public interval varies widely for each benchmark, it may require excessive amounts of profiling to find good interval settings for the conservative scheme. The next section will discuss heuristics to find good interval settings, without having to sweep a range of values.

7.2.4 Heuristics For Setting Public Parameters

Up to this point, we have relied on the server profiling each benchmark on training inputs by sweeping values for each public parameter in $\mathbb{P}$. This has the disadvantage that as $|\mathbb{P}|$ becomes large—which may be desirable in order to capture more fine-grained behavior found in programs (see Section 5.4.2 for some examples)—the amount of profiling that has to be done may increase exponentially. A natural question is whether the server can guess values for each public parameter, given a $|\mathbb{P}|$-independent amount of work that grows much slower with $|\mathbb{P}|$.

In this thesis, we designed each interval with a higher-level meaning in mind. For example, the I_{L2} is the number of times the L1 DCache must be accessed before making an L2 access (Section 5.1.3). A good guess for I_{L2}, therefore, might factor in the L1 DCache miss rate. The insight is that given program P and training input M, $P(M)$'s L1 DCache *real* miss rate is independent of the value of any parameter in $\mathbb{P}$. Thus, the server need only run $P(M)$ once, set $\mathbb{P}_{P,\mathsf{train}}$ based on statistics such as miss rate, and apply this setting to

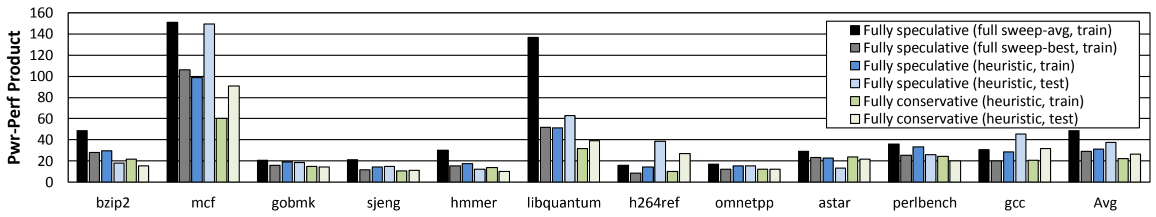


Figure 7-9: Power-performance product, varying interval policy, interval selection heuristic (not shown), and benchmark input.

the test input.

We now describe heuristics for setting each public interval based on these ideas. Suppose $Q(P(M))$ (Q for short) is the dynamic sequence of all instructions issued by $P(M)$. In all cases, M is the training input. Then each interval can be estimated by:

$$I_{L1D} = \left\lceil \frac{|Q| * \text{L1I\$}_{\text{hit}} + \sum_{i=1}^{|Q|} \mathsf{Exe}(Q_i)}{\sum_{i=1}^{|Q|} \mathsf{Mem}(Q_i)} \right\rceil$$

$$I_{L2} = \left\lceil \frac{1}{\mathsf{L1D}_{miss}(P(M))} \right\rceil$$

$$I_{ORAM} = \left\lceil \frac{1}{\mathsf{L2}_{miss}(P(M))} \right\rceil$$

where $\text{L1I\$}_{\text{hit}}$ is instruction cache hit latency (given in Table 7.2), $\mathsf{L1D}_{miss}()/\mathsf{L2}_{miss}()$ return the miss rates for the L1 DCache/L2 cache. $\mathsf{Exe}(Q_i)$ returns the execute stage cycle latency for the ith instruction (given for each instruction type in Table 7.2) and $\mathsf{Mem}(Q_i)$ returns 1 if the ith instruction is a memory instruction (or returns 0 otherwise). Intuitively, I_{L1D} is set to the average number of cycles between when consecutive memory instructions send requests to the L1 DCache (factoring out cache miss latency).

In practice, we find that for certain benchmarks, the heuristic for I_{L1D} oversets that interval by a large amount. For instance, the heuristic for I_{L1D} yields values that range between 5 and 14. Yet in previous sections, we found that certain benchmarks (e.g., the compute bound benchmarks) suffer large hits in performance if $I_{L1D} > 1$. The heuristic for I_{L1D} may be wrong for these benchmarks because of memory access time patterns that cannot be captured in intervals (e.g., when a function call is made, many memory instructions occur consecutively to push/pop the stack). Thus, in the following discussion we will evaluate two settings for intervals:

1. heuristic_full: Set I_{L1D}, I_{L2} and I_{ORAM} as specified above.

2. heuristic_partial: Set I_{L2} and I_{ORAM} as given above, and set $I_{L1D} = 1$ always.[7]

and choose the setting that yields the better power-performance product (PPP) on the training input.

Figure 7-9 shows PPP varying the public interval search heuristic, benchmark input and interval policy. To summarize labeling in the graph:

[7]This means that I_{L2} and I_{ORAM} might be underset. We found that adjusting I_{L2} and I_{ORAM} to compensate has little impact in power/performance, unless many values are swept—which somewhat defeats the purpose of using the heuristics. We note that increasing I_{L2}, to compensate for $I_{L1D} = 1$, causes large performance degradation.

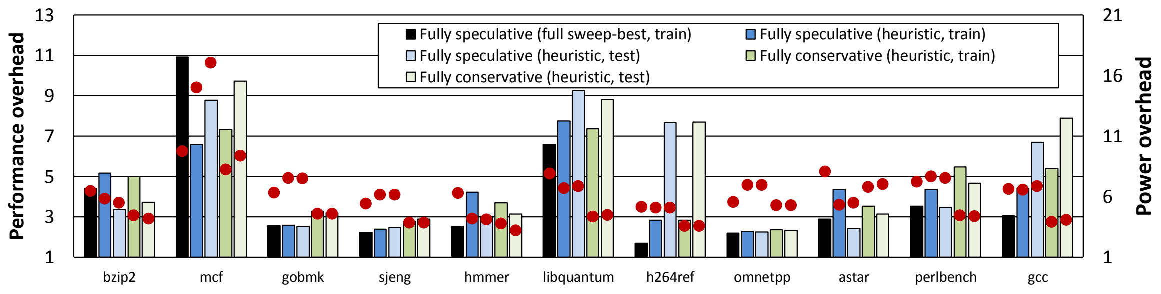


Figure 7-10: Power-performance breakdown (as before, power is shown as a red dot), varying interval policy, interval selection heuristic (not shown), and benchmark input.

1. Fully speculative (full sweep – best, train) means *fully sweep* the space of interval values given the training input (as done in Section 7.2.3.2) and the fully speculative interval policy. Choose the interval setting with the *best* (lowest) PPP and run that point on the *training* input.

2. Fully speculative (full sweep – avg, train) means the same as #1, except choose the interval setting in the sweep that yields the mean of PPP. This will be used for comparison purposes only.

3. Fully conservative (heuristic, test) means use heuristic_full and heuristic_partial to set each interval value (see previous paragraph) given the training input and a fully conservative interval policy. Record which heuristic yielded a lower PPP and use the parameter values chosen by *that* heuristic on the *test* input.

In all cases, public intervals are set based on the behavior of the training input. For comparison purposes, we show results for when both the training and test inputs are evaluated given those interval settings.[8]

7.2.4.1 Interval Policy Recommendations

The first observation in Figure 7-9 is that both the speculative and conservative policies, when evaluated on the training inputs, yield average PPPs that are on-par with Fully speculative (full sweep – best, train). In fact, the fully conservative policy yields a 30% improvement in PPP over Fully speculative (full sweep – best, train). This means that by using the two heuristic interval settings, we can achieve *better* quality of results than a baseline speculative scheme, *without* sweeping interval values. We show Fully speculative (full sweep – avg, train) to make a point about PPP variance: the average PPP is worse than the best by 68%, so our heuristics had the potential to perform quite badly. We note (not shown) that for the fully speculative policy, 5 out of 11 benchmarks yielded better PPPs with heuristic_partial, and 3 out of 11 chose heuristic_full given a fully conservative policy.

[8]Note that we won't use heuristics to set interval policies. Unlike intervals, which have natural heuristics such as instruction composition and miss rate, it is less clear how to guess which interval policy will yield the best PPP. One idea for future work might be to perform analysis on multiple training inputs, to see the extent to which program behavior changes based on input (i.e., if program behavior is input-independent, as is the case with software encryption algorithms, the fully conservative policy is a compelling choice). Fortunately, using heuristics to guess intervals is more important than guessing interval policy because the space of interval values far exceeds the space of interval policies.

Table 7.7: Recommended interval and interval policy settings given interval setting heuristics.

Benchmark	I_{L1D}	I_{L2}	I_{ORAM}	Interval policy
astar	10	15	38	Fully Speculative
bzip2	1	74	3	Fully Conservative
libq	14	25	2	“”
mcf	9	4	9	“”
gobmk	1	56	29	“”
sjeng	1	181	11	“”
hmmer	4	183	3	“”
gcc	1	189	7	“”
perlb	1	74	13	“”
h264	1	599	508	“”
omnet	1	52	2476	“”

The second observation in Figure 7-9 is that the better PPP occurs for both training and test inputs with the same interval policy. This means that if the server selects (for example) the fully conservative policy for mcf—because Fully conservative (heuristic, train) has a better PPP than Fully speculative (heuristic, train)—this decision correctly predicts that Fully conservative (heuristic, test) yields a better PPP than Fully speculative (heuristic, test). This is important because it means we can perform small additional amounts of offline profiling, on top of the interval selection heuristics, to set each interval policy. Based on analyzing PPP for the training input, the server should choose the parameter values given in Table 7.7 for each benchmark, when running the test input, which are summarized here.

1. The server should select the conservative policy for every benchmark except for astar (see Figure 7-9).

2. (Not shown) The server should select heuristic_full for 4 benchmarks (mcf, libquantum, hmmer, astar). The remaining benchmarks should use the heuristic_partial setting.

The exact settings for each benchmark is shown in Table 7.7. Putting these decisions together while evaluating the test inputs, our interval selection heuristics yield 5.2× performance and 4.7× power overhead, on average, relative to baseline (see Figure 7-10).

Figure 7-10 breaks down performance and power overhead explicitly for the data shown in Figure 7-9. Generally, the fully conservative policy sacrifices some amount of performance for a disproportionate savings in energy. For example, between speculative and conservative policies, mcf loses 10% performance but improves in power consumption by 82%. This means the interval selection heuristics are working and that the test input does not significantly deviate from training input behavior. (If the test input *did* deviate significantly, or if the heuristic generated wildly incorrect values, the frequent stalls imposed by the conservative interval policy would cause large performance degradations as seen with libquantum in Section 7.2.3.4). Note that for some benchmarks (e.g., hmmer), the conservative policy seems to perform better than the corresponding speculative point. This is an artifact of each policy choosing whichever heuristic setting—heuristic_full, etc—yields a better PPP on the training input. We note (not shown) that heuristic_partial yields better performance and worse energy efficiency than heuristic_full, regardless of interval policy.

Chapter 8

Related and Future Work

We now describe work related to performing secure computation on encrypted data and discuss future research directions for the Ascend setting.

8.1 Related Work

We break related work into the following categories: fully-homomorphic encryption (theoretical, more secure than Ascend) and tamper-resistant hardware (practical, less secure than Ascend). Lastly, we discuss other work done to secure either the I/O or power channels, individually.

8.1.1 Fully Homomorphic Encryption (FHE) Techniques

In [39], Craig Gentry presented the first fully homomorphic encryption (FHE) scheme [3, 38] that allows a server to receive encrypted data and perform, without access to the secret decryption key, arbitrarily-complex dynamically-chosen computations on that data while it remains encrypted. FHE evaluates programs as circuits and therefore implicitly assumes batch computation as does Ascend. FHE, however, assumes no physical security and is impervious to all physical side channels (I/O, power, EM, etc). FHE is currently theoretical: performance overheads are roughly seven orders of magnitude for straight-line code, and significantly more for general-purpose batch programs. Since FHE offers such a strong security guarantee, we will briefly discuss some work related to running general-purpose batch programs under FHE.

Several works [41, 53] have investigated how to run simple programs under FHE. In [41], a domain-specific language based on Haskell was designed to support simple FHE-based programs without data-dependent loops. [53] shows how information flow tagging can limit leakage due to control flow constructs.

Several works [51, 43] have investigated FHE for general-purpose batch programs. [51] described program interpreter-level techniques for efficiently mapping programs with complex control flow constructs to FHE. In that work, program execution can be optimized for the common-case; in exchange, the user gets a probabilistic guarantee that a program completes (which is similar to Ascend's notion of running for T time). Brenner et al. [43] describe a processor architecture—built out of software FHE circuits—for running encrypted programs. An important difference between Ascend and the work of Brenner et al. (aside from their reliance on FHE) is that for each instruction, Brenner et al. incurs overheads proportional to the size of the program data and instruction memories. This will incur orders of magnitude more overhead, on top of the overhead from FHE alone. Ascend uses ORAM and server-specified intervals to allow execution to be optimized on a program by program basis.

8.1.2 Secure Hardware and TCB Minimization

8.1.2.1 TPM-Related

The TPM [23, 9, 33] is a small chip soldered onto a motherboard and capable of performing a limited set of secure operations. The TPM is able to provide a proof (attestation) to the user that a particular sequence of steps were taken in launching the user's application. Both AMD (with SVM extensions [1]) and Intel (with TXT [60]) provide support for the TPM. The setup for both is similar: on a special instruction, the processor chip is flushed except for a specified peice of trusted setup code (typically a VMM) and the VMM launches the user application. The TPM's attestation, along with the processor chip's hardware support, proves to the user that (a) the program was run in isolation and (b) that the trusted setup code was used to launch the program. One representative project is Flicker [2], 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). In all cases, the user program, TPM, processor chip, and setup code is trusted.

8.1.2.2 Tamper-Resistant (Single-Chip) Processors

The eXecute Only Memory (XOM) architecture [18, 19, 12] is designed to mitigate both software and certain physical attacks. In XOM, security requires applications to run in secure compartments, where both instructions and data are encrypted and from which data can escape only on explicit request from the application itself. XOM assumes that operating systems are completely untrusted and potentially malicious. XOM needs to be augmented with protection against replay attacks on memory, and assumes trust in the user program. To the best of our knowledge, XOM was not built in hardware.

Aegis [20], a single-chip secure processor, was the first secure processor to include memory integrity verification and encryption on-chip so as to allow external memory to be untrusted. Two versions of Aegis were proposed, one with an untrusted OS and another with a trusted kernel. Only the latter was implemented in hardware [27]. Like XOM, Aegis must be protected against replay attacks and trusts the user program in all cases.

8.1.2.3 Untrusted Programs

Star-CPU [47] introduces an architectural template for proving no leakage through (in particular) the timing channel. In that work, the notion of "trusted constants" set at boot time is related to our notion of public server-specified parameters as both ideas make observable behavior data-independent. However, Ascend and Star-CPU use these constants to suppress fundamentally different sources of leakage: inter-thread leakage in the case of Star-CPU and leakage against an adversary watching the I/O and power pins in the case of Ascend. Furthermore, Ascend allows the server to change these 'constants' as a means to improve efficiency, based on program analysis performed offline. The Ascend chip is designed so that these changes do not impact security. Star-CPU does not discuss how these constants should/could be changed, and what impact this would have on the system's security. We applaud the efforts in [47] to provably verify their designs. Similar analysis for the power-obfuscated Ascend architecture discussed in Chapter 5 is an interesting direction for future work.

DataSafe [49] is another work that tolerates untrusted programs. In that work, private data is given special policies by the user and trusted hardware, a software hypervisor and software policies handlers manage data and polices to prevent leakage using information flow tracking ideas. DataSafe does not consider covert leakage channels—namely time, power and memory access pattern.

8.1.2.4 Differences to Ascend

The difference between Ascend and Flicker/Aegis/XOM is that Ascend only requires trust in the Ascend chip itself: *crucially* the user program is untrusted. Ascend only requires that the server run *some* program that performs useful computation for the user.[1] The user program may be malicious or buggy. In either case, hardware mechanisms within Ascend prevent the program from being able to leak *private data-dependent information* over the I/O and power channels.

In TPM-related works and Aegis/XOM, user programs are assumed to be trusted or will only be run if they are deemed "trustworthy." As programs grow in size, it becomes more difficult to formally verify intended/trustworthy behavior. For sufficiently large programs, the problem becomes intractable. Furthermore, programs are patched over time and must be re-verified after each patch; each patch forces the program to be re-verified, placing additional burden on the server and user.

Even if a program is deemed trustworthy, however, Ascend's threat model (Section 2.3) assumes a more powerful adversary than previous works. For instance, the I/O and power channel leakage is outside the TPM's threat model [23]. Thus, TPM-based systems may consider a program to be trusted even if that program *does* leak through its I/O or power signature. The same is true for XOM, Aegis and DataSafe. Star-CPU protects the timing channel in the case when multiple contexts are sharing the same processor. In that work, normal access control mechanisms prevent one context from *reading the data* of another thread—thus, on-chip data does not have to be encrypted but on-chip access patterns must be hidden (to prevent leakage through the cache footprint). We point out that hiding cache access patterns (their work) and hiding main memory access patterns (our work) are completely different problems. Further, Star-CPU does not consider the power channel.

8.1.3 Obfuscating the I/O Channel

The least-common-denominator primitive to obfuscate the I/O channel is Oblivious-RAM (ORAM), which completely hides memory access pattern (see Chapter 3.2 for citations). Built on ORAM-like ideas, HIDE [24] (and follow-on work [30]) adds architectural support for obfuscating memory access patterns through the idea of randomly shuffling memory locations between consecutive accesses. However, to have reasonable performance overheads, HIDE only applies this technique within small chunks of memory (usually 8 KB to 64 KB). In our threat model, obfuscation over small chunks breaks security because the server can engineer a curious program to perform inter-chunk accesses based on private data, and decipher all the encrypted data. Further, since no software in the Ascend model is trusted,

[1]If the server does not fulfill this requirement, we say that it is performing a denial of service attack.

we cannot rely on a trusted compiler to create a new program that groups memory requests into a single chunk—as is done in HIDE.

8.1.4 Obfuscating the Power Channel

In this thesis, we optimize PA resistant circuits (see Chapter 4 for citations) and use these primitives to construct a complete secure co-processor (Chapter 5) that does not leak at the microarchitecture level. To the best of our knowledge we are the first to propose a complete chip architecture that mitigates all architectural leakage, supports general-purpose batch programs, and does so while allowing the server to trade-off performance and power dynamically.

Another mitigation for PA analysis is algorithm blinding [42]. This technique is the algorithm-level counterpart to circuit bit masking techniques [21]. The idea is to mask the inputs to a secure circuit and then de-mask the outputs so that intermediate power traces do not leak useful information. This technique is effective in mitigating leakage for certain crypto algorithms (e.g., RSA [8]) whose inputs and outputs can be masked and de-masked. It is unclear how to perform masking for general-purpose batch programs.

A program-level countermeasure suitable for general-purpose programs is to write [11] or compile [26] the program so that every program path performs data-independent amounts of work. [26] discusses these transformations in terms of timing-attacks only; [11] suggests using similar ideas to mitigate SPA attacks for algorithms such as RSA (which performs square/multiply operations depending on key bits). Ascend can be viewed as a hardware counterpart to these techniques; to our knowledge, the software-based techniques have not explored how to improve performance through common-case program behavior, as done in Ascend and [51].

8.2 Future Work

We now discuss interesting directions for future work with Ascend.

8.2.1 Program Model

The batch programming model requires that the application run in a sandbox—in our case the Ascend processor plus its ORAM. This limits the amount of off-chip data that Ascend can access and also limits how Ascend can interact with its user.

To support very large working sets (Terabytes in size, as in Big Data applications), Ascend's initialization will become a system bottleneck and the overhead from ORAM will increase. First: by definition, the batch model needs to load any data that the program might touch at program start time (Section 2.2, Step 3). For Big Data applications, this itself would take a long time (extrapolating from Section 7.2.2: our highest-performance ORAM design has a bandwidth of $\sim$ 50 MB/s, if the ORAM is accessed continuously). Second: it is not clear how the overhead from ORAM will increase if one assumes large data sets. For instance, our results assume the ORAM can be stored completely in DRAM. An ORAM for Big Data applications would have to be stored in disk, incurring multiple disk accesses per ORAM access. [52] proposes such a system, but requires thousands of trusted co-processors to jointly access the larger ORAM which greatly expands the TCB beyond the Ascend proposal.

One extension that can tackle both of the above problems is to support *stream computation* (related to Private Information Retrieval (PIR) in the literature [6]). In that case, unencrypted public data is streamed into Ascend, which performs private filter/query operations. Only query "matches" are stored in ORAM. The insight is that in unstructured search, data is touched once. Thus a small ORAM may be sufficient to support arbitrarily large data streams.

Batch programs by definition do not interact with other programs or users. Despite this, multi-interactive protocols (Section 6.1) are a first-step towards interaction. In that case, the user and Ascend can interact at intervals (or every T_i cycles where each T_i is set by the user) to decide if the computation should be run for more time. Interacting at intervals can be adapted for other tasks—such as receiving key strokes from the user, if the program has a shell. Furthermore, multiple Ascend chips can theoretically interact if such an interaction passes encrypted values at strict intervals and all Ascend chips are currently owned by the same user (or a set of trusted users).

8.2.2 Operating Systems, Multiple Threads and Multi-Programming

We also constrained Ascend to support one single-threaded batch program at a time. Important extensions are to include multi-threaded programs and multi-program workloads. The important question is whether a single user controls all threads/programs on the Ascend chip, or whether the chip is shared between multiple (untrusted) users.

We believe that Ascend can support single user multi-threading and multi-programming without significant modification. Suppose the user wants to run programs $P_1, \ldots, P_n$ on private inputs $M_1, \ldots, M_n$, for some n. In that case, the server can load $P' = P_{vmm}(P_1, \ldots, P_n)$ into Ascend, where P_{vmm} is an untrusted VMM that can context switch between the different programs as it likes. The high-level idea is that as long as P' is also a batch program (i.e., does not interact beyond the Ascend+ORAM sandbox), it leaks no more information than any other batch program run within Ascend. As before, the entire result of running $P'(M_1, \ldots, M_n)$ is encrypted and returned to the user. The server never learns when data-dependent context switches happen, or anything else about the intermediate or final states of the VMM that the server could not deduce apriori.

Running multi-threaded applications with a single user can work similarly. Of course, with multi-threading support one may want to re-architect Ascend to have multiple cores. In that case, the power obfuscation problem becomes more difficult because multi-core cache coherence is more involved than choosing between inclusive/exclusive caches (Section 5.4.5).

Multi-threading/programming with multiple (untrusted) users creates new security challenges. This case is fundamentally different from the above cases because untrusted users will be able to decrypt the results of untrusted threads/programs and this adds leakage channels such as cache footprint/timing/etc. There is a large literature for these types of attacks, and the problem is discussed in [47].

8.2.3 Additional Side Channels

In this thesis, we only consider the I/O and power side channels. We showed these attack surfaces not only because of their prevalence but also to show how the concept of specifying the observable behavior of the processor apriori can be generalized.

Others side channels include passive attacks such as leakage through EM/RF emissions [16] (which is similar to the power channel) or invasive attacks such as triggering faults [10]. Given a primitive to obfuscate signals in the value domain for the EM channel (i.e., an EM counterpart to a power analysis resistant logic), we believe that the idea of data-independent parameters (intervals, producer/consumer policies) can be used to secure the EM/RF channels. We note that logics like WDDL try to make capacitive discharge throughout the chip data-independent, which is a source of EM flux that leads to EM attacks. Thus, these techniques may be sufficient to thwart EM attacks. We do not, however, claim to have addressed this source of leakage yet.

It may or may not be possible for the server to use invasive attacks (i.e., triggering faults) to coerce Ascend into leaking privacy. On the one hand, the entire Ascend chip is architected to perform data-independent amounts of work. Thus, if a fault attack were to flip a bit in program memory or change an interval, this would not leak privacy. In the former case, the program may crash but observable behavior should still be indistinguishable from normal program operation. On the other hand, if the adversary can change the symmetric key inside Ascend to a value of his/her choosing through a fault attack, security can be broken.

Introduced in Section 8.1.2.3, cache footprint-based attacks pose another direction for future work. We believe that our work and others (such as Star-CPU [47]) can be merged so that threads can safely and efficiently share hardware resources. For example, a program's working set over time can be determined apriori by a server using public inputs to that program. The server can then use that information to create a static schedule that determines how much cache/eviction buffer/pipeline/etc the thread should get when running the user's input. Here, the static schedule is performing the same job as the set of knobs $\mathbb{P}$ that we have described in this thesis. What separates the idea from traditional static scheduling is that our 'static' schedule can *change* over time, as long as it changes based on the server's apriori profiling. This direction leads to new questions. For example, if multiple threads share a core and each thread requests 50% of the cache for 1000 cycles, how can a scheduler decide who gets what resources?

8.2.4 More Advanced Microarchitecture

The microarchitecture we presented in Chapter 5 was not meant to be limiting. Ascend can support multiple cores, a more advanced pipeline and other conventional architectural structures in theory.

That said, it is useful to understand which structures are more and less efficient given the primitives that Ascend is built upon (ORAM, WDDL, `cell-1` SRAMs in this thesis). For instance:

1. Since ORAM access latency is orders of magnitude more than conventional DRAM, an architecture that supports multiple outstanding loads to main memory may not be as effective (i.e., ORAM latency dominates always).

2. Since all ways in an associative cache must be accessed on each cache access to prevent leakage through the power channel, cache associativity should be constrained for energy-consumption reasons (Section 5.4.3).

3. Since registers that have back-pressure add a pipeline stage (due to WDDL precharge), feed-forward designs are preferable.

On the other hand, certain structures do map efficiently to our primitives. For example, the re-order buffer (i.e., to support out-of-order execution) is already implemented as a fully-associative table, which obfuscates *which* address is being looked up by default.

8.2.5 Additional Public Parameters

A big idea used throughout the thesis was to specify Ascend's observable behavior through a set of parameters (we explored setting different circuits to intervals and levels of the memory hierarchy to speculative/conservative policies). These parameters are also not meant to be limiting—more parameters that perform different data-independent functions may be added to improve efficiency in different respects.

One aspect of these parameters that we have not yet explored is how they should be changed dynamically by the server while a program is running. In our evaluation, each parameter was set at program start time. This was not necessary: for instance, when a program changes phase it may be preferable to change memory access intervals. As before, any dynamic change to a parameter must be made solely based on offline analysis done by the server: mechanisms within Ascend will prevent the server from finding out how well its strategy performed on the user's data.

Another aspect left unexplored is parameter expressivity. Intervals are limited in that they require that an equal amount of time pass in between two accesses. In the SPEC benchmarks, other patterns occur with regularity. For example, when a program makes a function call it typically performs a sequence of store operations followed by a sequence of load operations. Intervals are ill-suited for this behavior because they implicitly assume that memory operations are not consecutive. This may be the reason that using the % of dynamic memory instructions statistic to set I_{L1D} (Section 7.2.3) performs poorly.

Chapter 9

Conclusion

This thesis has shown the viability of Ascend, a secure processor architecture that prevents privacy leakage over the I/O and power channels, and additionally guarantees integrity over the I/O channel, while only requiring trust in a single processor chip. Crucially, the program running on the processor is untrusted and may be malicious. To get efficiency and to trade-off performance and power without compromising security, Ascend can be parameterized on a program to program basis by the server, based on the server's apriori knowledge of each program that will be run.

Surprisingly, the performance/power overheads associated with our architectural mechanisms are only 3.6×/6.6× (when the server is given large amounts of offline time to sweep public parameter values) and 5.2×/4.7× (when the server uses heuristics to estimate public parameter values) while running SPEC benchmarks. Furthermore—when protecting the I/O channel only—performance/power overhead drops to only 2.6×/2.2×. This result makes Ascend practical, and capable of running real programs with roughly the same overhead as running programs in interpreted languages.

Bibliography

[1] Advanced micro devices. amd64 virtualization: Secure virtual machine architecture reference manual.

[2] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev.*

[3] R. Rivest, L. Adleman, and M.L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 1978.

[4] O. Goldreich. Towards a theory of software protection and simulation on oblivious rams. In *STOC*, 1987.

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

[6] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, pages 45–51, 1995.

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

[8] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.

[9] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[10] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *EUROCRYPT*, pages 37–51, 1997.

[11] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.

[12] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[13] Premkishore Shivakumar and Norman J. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, February 2001.

[14] Jose Renau. Sesc: Superescalar simulator. Technical report, university of illinois urbana-champaign ECE department, 2002.

[15] K. Tiri, M. Akmal, and I. Verbauwhede. A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the 28th ESSCIRC 2002.*, sept. 2002.

[16] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The em side-channel(s). In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 29–45, London, UK, UK, 2003. Springer-Verlag.

[17] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and Merkle Trees for Efficient Memory Integrity Verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, New-York, February 2003. IEEE.

[18] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[19] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 178–192, 2003.

[20] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *ICS (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[21] Daisuke Suzuki, Minoru Saeki, and Tetsuya Ichikawa. Random switching logic: A countermeasure against dpa based on transition probability, 2004. dice@iss.isl.melco.co.jp 12755 received 3 Dec 2004.

[22] K. Tiri and I. Verbauwhede. A logic level design methodology for a secure dpa resistant asic or fpga implementation. In *Proceedings of DATE, 2004.*, feb. 2004.

[23] Trusted Computing Group. TCG Specification Architecture Overview Revision 1.2. http://www.trustedcomputinggroup.com/home, 2004.

[24] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the 11th ASPLOS*, 2004.

[25] Stefan Mangard. Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints. In *Systems CHES 2005, 7th International Workshop*. Springer, 2005.

[26] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *In Cryptology ePrint Archive, Report 2005/368*, 2005.

[27] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32^{nd} *ISCA'05*, New-York, June 2005. ACM.

[28] Daisuke Suzuki, Minoru Saeki, and Tetsuya Ichikawa. Dpa leakage models for cmos logic circuits. In *Cryptographic Hardware and Embedded Systems - CHES 2005*, 2005.

[29] Zhimin Chen and Yujie Zhou. Dual-rail random switching logic: A countermeasure to reduce side channel leakage. In *CHES 2006*, 2006.

[30] Lan Gao, Jun Yang, Marek Chrobak, Youtao Zhang, San Nguyen, and Hsien-Hsin S. Lee. A low-cost memory remapping scheme for address bus protection. In *Proceedings of the 15th PACT*, PACT '06. ACM, 2006.

[31] E. Konur, Y. Ozelci, E. Arikan, and U. Eksi. Power analysis resistant sram. In *Automation Congress, 2006. WAC '06. World*, pages 1 –6, july 2006.

[32] Konrad J. Kulikowski, Mark G. Karpovsky, and Er Taubin. Power attacks on secure hardware based on early propagation of data. In *In 12th IEEE International On-Line Testing Symposium*, pages 10–12, 2006.

[33] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *Proceedings of the 1st STC'06*, November 2006.

[34] Darryl Gove. Cpu2006 working set size. *SIGARCH Comput. Archit. News*, 35(1):90–96, March 2007.

[35] Darryl Gove. Cpu2006 working set size. *SIGARCH Comput. Archit. News*, 35(1):90–96, March 2007.

[36] E. Arikan and A. Ataman. A new power analysis resistant sram cell. In *International Conference on ELECO 2009.*, nov. 2009.

[37] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE symposium on S&P, 2009*, may 2009.

[38] C. Gentry. A fully homomorphic encryption scheme. PhD thesis, Stanford University, 2009.

[39] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC'09*, pages 169–178, 2009.

[40] 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 *2010 IEEE Symposium on VLSIC*, june 2010.

[41] Alex Bain, John Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. A domain-specific language for computing on encrypted data. In *FSTTCS 2011*. LIPIcs, December 2011. Invited paper.

[42] Gerrit Bleumer. Blinding techniques. In *Encyclopedia of Cryptography and Security (Volume 2, 2nd Ed.)*, pages 150–152. 2011.

[43] Michael Brenner, Jan Wiebelitz, Gabriele von Voigt, and Matthew Smith. Secret program execution in the cloud applying homomorphic encryption. In *IEEE DEST 2011*, May 2011.

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

[45] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd CCSW '11*, pages 113–124, New York, NY, USA, 2011. ACM.

[46] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt*, pages 197–214, 2011.

[47] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceedings of the 38th ISCA*, ISCA '11, 2011.

[48] Carolyn Whitnall and Elisabeth Oswald. A comprehensive evaluation of mutual information analysis using a fair evaluation framework. In *CRYPTO 2011*, 2011.

[49] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the 19th Conference on Computer and Communications Security (CCS'12)*, 2012.

[50] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing*, pages 3–8, October 2012.

[51] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Towards an interpreter for efficient encrypted computation. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, CCSW '12, pages 83–94, 2012.

[52] Jacob R. Lorch, James W. Mickens, Bryan Parno, Mariana Raykova 0001, and Joshua Schiffman. Toward practical private access to data centers via parallel oram. *IACR Cryptology ePrint Archive*, 2012:133, 2012. informal publication.

[53] J.C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow control for programming on encrypted data. Cryptology ePrint Archive, Report 2012/205, 2012. http://eprint.iacr.org/.

[54] V. Rozic, W. Dehaene, and I. Verbauwhede. Design solutions for securing sram cell against power analysis. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, pages 122–127, 2012.

[55] Jaewoong Sim, Jaekyu Lee, Moinuddin K. Qureshi, and Hyesoon Kim. Flexclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 321–332, Piscataway, NJ, USA, 2012. IEEE Press.

[56] E. Stefanov and E. Shi. Path O-RAM: An Extremely Simple Oblivious RAM Protocol. Cornell University Library, arXiv:1202.5150v1, 2012. arxiv.org/abs/1202.5150.

[57] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[58] Carolyn Whitnall, Elisabeth Oswald, and François-Xavier Standaert. The myth of generic dpa...and the magic of learning. *IACR Cryptology ePrint Archive*, 2012:256, 2012.

[59] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*. 2013.

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