A Secure Processor Architecture for Encrypted Computation on Untrusted Programs

Christopher Fletcher
MIT CSAIL
cwfletch@mit.edu

Marten van Dijk
RSA Laboratories
marten.vandijk@rsa.com

Srinivas Devadas
MIT CSAIL
devadas@mit.edu

ABSTRACT

This paper considers encrypted computation where the user specifies encrypted inputs to an untrusted program, and the server computes on those encrypted inputs. To this end we propose a secure processor architecture, called *Ascend*, that guarantees privacy of data when arbitrary programs use the data running in a cloud-like environment (e.g., an untrusted server running an untrusted software stack).

The key idea to guarantee privacy is *obfuscated instruction execution*; Ascend does not disclose what instruction is being run at any given time, be it an arithmetic instruction or a memory instruction. Periodic accesses to external instruction and data memory are performed through an Oblivious RAM (ORAM) interface to prevent leakage through memory access patterns. We evaluate the processor architecture on SPEC benchmarks running on encrypted data and quantify overheads.

Categories and Subject Descriptors

C.1.0 [**Computer Systems Organization**]: Processor architectures—*General*; C.0 [**Computer Systems Organization**]: General—*Modeling of computer architecture*; E.3 [**Data**]: Data Encryption

Keywords

Secure processors, Encrypted computation

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 data is stored and computed upon in the cloud. Computation requires the data to be exposed to the cloud servers, which may be attacked by malicious applications, hypervisors, operating systems or insiders.

Encrypted computation has the potential to solve the data privacy problem. In encrypted computation, the user specifies encrypted inputs to a program, and the server computes on encrypted inputs to produce an encrypted result. This encrypted result is sent back to the user who decrypts it to get the actual result. In this paper, we consider cases where the program is supplied by the server, the user or a third party and can be either public or private (encrypted). The program is not trusted by the user in all cases. In our context, to be "trusted," a program must not be intentionally malicious and must be provably free of any bugs that have the potential to leak information about the program data. Data from the user is always considered private.

In the ideal scenario, no one other than the user sees decrypted data or knows the secret key used to encrypt the data. This ideal can be reached through the use of fully homomorphic encryption (FHE) techniques [3]; unfortunately, FHE approaches currently result in about 8-9 orders of magnitude slowdown [4], which severely limits their applicability.

Secure processors and coprocessors [18, 10, 16, 1] assume a secret key stored in hardware and can perform private execution efficiently; the user, however, has to trust the processor as well as the application/program and the operating system (OS) or kernel running on the processor. While there have been proposals (e.g., [10], [16]) to build processors with hardware support for context management so as to avoid having to trust the OS, these processors do not appear to have been built. Further, these proposals leak information through memory access patterns. Secure processors are currently used in niche applications such as smart cards, where specific trusted applications are run.

Secure coprocessors such as the Trusted Platform Module (TPM) [17] allow the processor to be conventional, but require trust in the OS to support private execution of large applications. Applications that use the TPM or similar trusted hardware without trusting the OS (e.g., [12], [9]) have been limited. Using the TPM along with Intel TXT [8] allows a user to only trust the processor chip, the TPM, the program being run and the connecting bus. An untrusted server still needs to be prevented from gleaning information about the encrypted inputs by running different programs on the input and inspecting memory contents or memory access patterns. TPM-based systems require the user to trust that the program run on the data will not expose the data; a malicious program may leak data through memory access patterns or the frequency of memory accesses.

1.1 Motivating Example

In virtually all trusted computing platforms, the user application is trusted. If the user supplies a program (possibly encrypted) along with the program data, it may be reasonable to assume that the program is not intentionally malicious (e.g., if the user wrote the program him/her self). Having the user supply the (encrypted) program and/or verifying that the program does not leak data is not always possible in a computation outsourcing setting, however. For example, the user may be paying for time to use a proprietary program whose binary instructions should not be revealed to the user. If the encrypted data is not tied to a particular trusted or veri-

STC'12, October 15, 2012, Raleigh, North Carolina, USA.

fied program, a semi-honest server may decide to run different programs on the user's encrypted data to satisfy its curiosity about the data. For example, the server may decide to run the program shown in Algorithm 1 on the user's encrypted data $\mathcal{M}$. Here, the server

Algorithm 1 A simple program that can compromise the user's encrypted data $\mathcal{M}$. & is the bitwise AND operation.

```
y = M[0]
while (y & 1) ?= 0 do
    issue a random load or store request from/to memory
end while
```

will be able to detect if the low-order bit of some word in the user's encrypted data equals 0 by monitoring how long the program takes to run (e.g., if the program finishes instantly or appears to be stuck in the loop) and whether the program produces a stream of memory requests.

Regardless of whether the program is encrypted or malicious by design, program bugs can also leak privacy. Writing and compiling programs that are provably secure in this sense (indistinguishable given arbitrary inputs) is a hard problem.

1.2 Our Solution

We propose a secure processor architecture called *Ascend*[1] that supports private computation of arbitrary programs with a semi-honest server. Security is independent of the program that uses the data and the operating system. We focus on the case where Ascend is a coprocessor inside a server and when we refer to the untrusted server, we mean the software stack/OS and anything outside the Ascend chip.

To be secure, Ascend obfuscates the instructions that it executes to make forward progress in the program, *which obfuscates Ascend's external input-output (I/O) and power pins.* Each pin carries a digital or analog signal at a given time and these signals change over time in program-dependent ways. To obfuscate *when* the value on each pin changes, Ascend must perform a program data-independent amount of work to evaluate each instruction. All processor circuits must fire on each instruction fetch to give off the impression that *any* instruction could have been executed and on/off-chip memories must be accessed only at *public* time intervals. To obfuscate the bits and memory access pattern on the I/O pins, external memory requests must be made using oblivious RAM (ORAM) techniques [6]. The adversary learns an estimate of the number of cycles required to complete the computation, which can be shown to be the least amount of leakage possible [2].

Ascend is marginally more complex than a conventional processor, in the sense that Ascend must implement an ISA and also make sure that the work it does is sufficiently obfuscated. Trusted ORAM client-side logic is built inside of Ascend (this mechanism can be viewed as a hardware memory controller primitive). Unlike XOM [10] or Aegis [16] Ascend neither trusts nor implements any part of the operating system, internally or otherwise.

2. FRAMEWORK

To start, we introduce a general framework for performing computation under encryption for arbitrary programs. We assume black box symmetric-key $encrypt(\dots)$ and $decrypt(\dots)$ functions, which take a plaintext/ciphertext as input and return the corresponding ciphertext/plaintext using randomized encryption or decryption.

[1]Architecture for Secure Computation on Encrypted Data.

The Ascend secure processor is modeled as a tamper-proof black box that has I/O pins, which it uses to make requests to the outside world. Ascend has an internal ORAM interface to an external RAM (where the external RAM is under the server's control). Ascend's ORAM interface is functionally a read/write controller to memory such that an observer learns nothing about the data being accessed or the sequence of program address requests made over time, despite the RAM being stored external to the processor. Time is measured in clock cycles.

2.1 Two-Interactive Protocols

We model general computation with a two-interactive protocol between a trusted user, a server and the Ascend chip. Suppose the user wants the server to evaluate deterministic algorithm $\mathcal{A}$ (made up of instructions) on inputs from the user, collectively denoted by x, and inputs from the server, collectively denoted by y. Formally, a two-interactive protocol Π for computing on $\mathcal{A}$ runs as follows (shown graphically in Figure 1):

1. The user shares a secret (symmetric) key securely with Ascend. We assume that Ascend is equipped with a private key and a certified public key.
2. The user encrypts its inputs x using the chosen symmetric key to form the ciphertext $encrypt(x)$ and then chooses a number of cycles S, which is the time/energy budget that the user is willing to pay the server to compute on $\mathcal{A}$. The user then transmits to the server the pair $(encrypt(x), S)$ together with algorithm $\mathcal{A}$ if the server is not providing $\mathcal{A}$.
3. (**Initialization**) After receiving the pair $(encrypt(x), S)$ and optionally $\mathcal{A}$, the server engages in an interactive protocol with Ascend to initialize ORAM memories that will be used to store $\mathcal{A}$, x and y in encrypted form. Once complete, the ORAM memory after 0 cycles is referred to as $\mathcal{M}_0$. Decrypted data x is not revealed to the server during this interaction. After the process is complete, Ascend will be able to make ORAM read/write requests to the external RAM to fetch instructions in $\mathcal{A}$ or data in x or y.
4. The server sends S to Ascend and Ascend spends a number of clock cycles and energy, corresponding to S, making forward progress in $\mathcal{A}$. During this period of time, Ascend may make ORAM requests to the server to request more instructions or data.
5. The result of the server-Ascend interactions is an ORAM $\mathcal{M}_S$, the program state after S cycles. The server can either send the ORAM back to the user as is, or start an interactive protocol with Ascend to "unload" the ORAM to form ciphertext $\mathcal{M}'_S$ (which is in some format that is more efficient for the user to decrypt).
6. The user decrypts $\mathcal{M}_S$ and checks whether S was sufficient to complete $\mathcal{A}(x, y)$. 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 Π outputs to the client the evaluation $\mathcal{A}(x, y)$ (if S was sufficient) or some *intermediate result*.

The disadvantage of only two interactions is that the user may receive an intermediate result (rather than the final one) indicating that the computation was not finished. The advantage is no additional unnecessary privacy leakage about the final result; i.e., the server does not gain additional understanding about the output of $\mathcal{A}$ evaluated on the unencrypted inputs besides what the server is already able to extract from the algorithm itself, the number and sizes of the encrypted inputs, and other a-priori knowledge. It can be shown that this leakage is optimal [2].

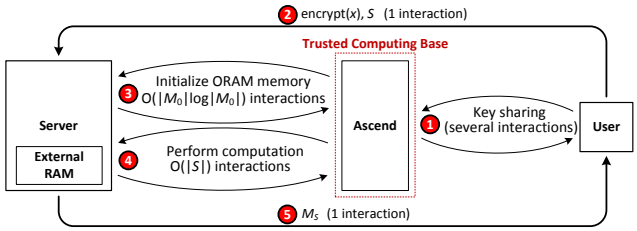

Figure 1: The two-interactive protocol between a user, server and Ascend. Numbers in the figure match the two-interactive protocol description from Section 2.1. The only trusted entity is the Ascend processor—neither the server nor any arrow (which would be implemented as a bus, channel, etc) is trusted.

2.2 Security Model

We assume the semi-honest model and that the server is "honest but curious." The server is honest in that it executes $\mathcal{A}$ under encryption for the required number of cycles and sends back the result exactly as specified (no deviations, malicious or otherwise). In particular, the server does not send back to the user the result produced by executing a different algorithm, or evaluating $\mathcal{A}$ on different inputs, or evaluating $\mathcal{A}$ on the user's input for less than S cycles. The server will try to finish the user's program as best it can, given the number of cycles specified by the user, in order to get additional business from that user in the future.

The server is curious in that it may try to learn as much as possible about the user's input from its view; namely $\mathcal{A}$, the encrypted inputs as given by the user, and *black box* access to the Ascend processor. The server can monitor Ascend's pins for timing/power and I/O behavior, or apply its own inputs when it pleases. For example, the server can run different programs on the user's inputs in order to try to learn something about the inputs by monitoring Ascend externally. When the server applies a public program to the encrypted data, the server has other a priori knowledge such as the program's structure and offline profiling information that it may have obtained by running the program on different inputs.

2.3 Ascend Security Level

The Ascend processor is a tamper-proof black box and is designed to meet the conditions for *oblivious computation* [6]. To make Ascend oblivious given untrusted $\mathcal{A}$, the following properties must be met:

1. The specific sequence of instructions needed to make forward progress in $\mathcal{A}$ must be *obfuscated*. That is, Ascend should appear to spend the same amount of time/energy/etc to evaluate each instruction, regardless of what instruction is being evaluated.
2. Both (a) the address sequence of external requests and (b) the times at which those requests are made must be indistinguishable for any $\mathcal{A}$ and $\mathcal{M}_0$. Ascend uses an ORAM interface to make external requests, thereby satisfying (a).

Observe that if both of these conditions are satisfied, the server's view of Ascend itself (condition 1) and Ascend's input/output behavior (condition 2) is independent of $\mathcal{A}$ and $\mathcal{M}_0$, which satisfies the properties for being oblivious.

Note that satisfying condition 1 perfectly is a circuit design and implementation problem, which is outside the scope of this paper. In this paper, we use a simplistic strategy: we force Ascend to "go through the motions" of each possible instruction to make one instruction's worth of forward progress in $\mathcal{A}$. Let PC' denote the dynamic program counter that changes based on data-dependent program conditions and assume that this value is stored inside the Ascend processor. At any given point in the execution of any given program, PC' points to exactly one instruction denoted $I(PC')$ (we are assuming sequential program semantics) in $\mathcal{A}$, which corresponds to one instruction type in the chosen ISA. To evaluate $I(PC')$, Ascend must speculatively evaluate each instruction in its ISA. If the current instruction it is evaluating matches the instruction type for $I(PC')$, the instruction successfully updates program state. Otherwise, no change in program state occurs but Ascend must still activate the circuits that it would have if the instruction were actually executed. We refer to this extra work as *dummy work*. For example, if Ascend has an internal data memory and one of the instructions in its ISA accesses the memory, Ascend must access the memory (either with a real or *dummy* request) for every instruction that it evaluates. To be secure, dummy work must be indistinguishable from real work.

Satisfying the second part (b) in condition 2 is done by making *predictable and periodic requests* to the external RAM that implements the ORAM. Conceptually, if every ORAM request takes a fixed number of clock cycles to complete then this condition can be satisfied if Ascend makes exactly one external request every T clock cycles. In actuality, Ascend will make a real request every T cycles if it has one to make, or a dummy request if it does not (for the same reason as in the previous paragraph). To maintain security, T is public, set by the server and cannot depend on $\mathcal{M}_0$. In a realistic setting, every ORAM request will take a variable number of cycles because of external bus traffic and physical NUMA (non-uniform memory architecture) constraints. To maintain the same level of security, it suffices for Ascend to make either a real or dummy request T cycles after the last request completed (e.g., arrived at Ascend's input pins). As before, a dummy ORAM request must be indistinguishable from a real ORAM request.

3. PROCESSOR ARCHITECTURE

In this section, we describe the ORAM interface and a performance-optimized Ascend processor design. As a rule, we only a make a performance optimization if it does not decrease the security level described in Section 2.3.

3.1 ORAM Interface

The Ascend processor has an internal ORAM interface to external RAM. The interface accepts a read/write request for a *block* of program data or instructions (using program addresses). A block in this setting is a region of consecutive (address, data) pairs and is analogous to a cache block in normal processors. As soon as a request is made, the ORAM interface will start a variable-latency interactive protocol with the outside (untrusted) world and either return the requested block (on a read) or signal that the write completed successfully. (To orient the reader, this interactive protocol takes thousands of clock cycles to complete.) The amount of time/energy that it takes for the ORAM interface to initiate the request (e.g., lookup its internal state and setup signals to Ascend's pins) is assumed to be independent of the request itself.

ORAM, pronounced oblivious RAM, has the property that its interface completely hides the data access pattern (which blocks were read/written) from the external RAM; from the perspective of the external RAM, read/write operations are indistinguishable from random requests. ORAM only considers hiding the data access *pattern* and not the times at which read/write operations are requested (which is discussed in Section 3.2). The ORAM interface between Ascend and external RAM is secure if, for any two data request sequences (produced by Ascend) of the same length, their access patterns to external RAM (produced by the interface) are computationally indistinguishable by anyone but Ascend. This guarantees that no information is leaked about the data accessed by

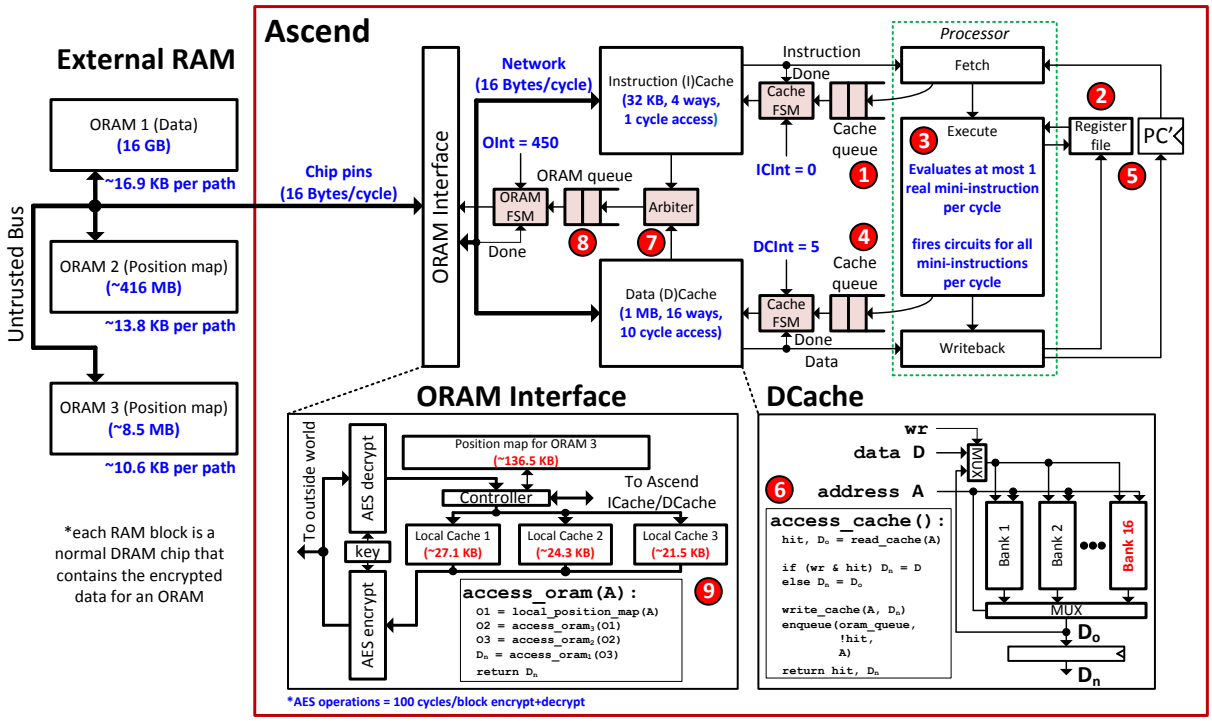

Figure 2: The Ascend processor architecture; annotated with the parameters and representative latencies that our performance model assumes in Section 4. Independent variables are **blue** and dependent variables are **red**. Pink structures and (DCINT, ICINT, OINT) are Ascend-specific and discussed in Section 3.2. ORAM parameters are derived in Figure 3.

Ascend, how it was accessed (read, write, sequential, random, etc.), and whether it was accessed before.

ORAM was introduced in [5, 11]. Recently two new breakthroughs were published in [13, 15]. Their authors also constructed a much more simple and practical ORAM construction called Path ORAM [14] which we use here. The external RAM is treated as a binary tree where each node is a bucket that can hold up to a small fixed number ($Z = 4$) of blocks. In order to obtain a capacity of $N = 2^{L+1}$ data blocks, the tree needs $L+1$ levels, its root at level 0 and its 2^L leafs at level L. Each leaf is labeled by L bits. Blocks have $(L+1)$-bit addresses. To manage the Path ORAM, the client (Ascend) maintains a *position map* and *local cache* (discussed below).

Initially, when no data has been written, all blocks in the tree are all-zero. The protocol uses randomized encryption (we use a parallelizable method based on 128-AES) before it writes data blocks back to external RAM such that only with very small probability the external RAM is able to learn whether two data encryptions correspond to the same data.

Citing from [14]:

> "We maintain the invariant that at any time, each data block is mapped to a uniformly random leaf bucket in the tree, and uncached blocks are always placed in some bucket along the path [from the root] to the mapped leaf. Whenever a block is read from the [external RAM], the entire path to the mapped leaf is [decrypted and] read into the local cache, the requested block is remapped to another leaf, and then the path is [re-encrypted and] written back to the [external RAM]. When the path is written back to the [external RAM], additional blocks in the cache may be evicted into the path as long as the invariant is preserved and there is remaining space in the buckets. ... [Ascend] stores a position map array $position[u]$ that consists of N integers, mapping each block u to one of the 2^L leafs [in the external RAM's] tree data structure. The mapping is random and hence multiple blocks may be mapped to the same leaf and there may exist leafs to which no blocks are mapped. The position map changes over time as blocks are accessed and remapped."

The position map is an array of NL bits. For our parameter settings this ($\sim$ 416 MB) is too large for Ascend's local memory. For this reason we extend the Path ORAM by storing the position map in a second ORAM and the position map of the second ORAM in a third ORAM (some care is needed; each node in each tree needs to store besides the address of the block also the label of the corresponding leaf-bucket). The local cache for each ORAM (typically 100 blocks plus the number of blocks along one path in the tree) is entirely stored in Ascend's local memory.

3.2 Ascend Processor

We architect the Ascend processor with emphasis on security first and performance second. To maintain security, we add architectural mechanisms to obfuscate program behavior (e.g., to perform dummy work and to make periodic memory requests to obfuscate ORAM usage; see Section 2.3). To increase performance, we extend the idea of making periodic requests to other processor operations (such as accessing internal data memory) to reduce the amount of total dummy work performed over time.

The Ascend processor (Figure 2) is made up of the following main components: (a) familiar register file (RF) and cache resources[2], (b) several security-specific structures that will be used to make different types of requests at periodic intervals, an ORAM interface (Section 3.1), and a fetch-execute pipeline capable of eval-

[2]We architect these structures somewhat differently for security reasons.

uating obfuscated instructions. Subsets of $\mathcal{M}$ and $\mathcal{A}$ are stored in on-chip *data (D)Cache* and *instruction (I)Cache* memory, respectively.

To increase performance, both cache *and* ORAM requests are made at periodic intervals set by the server. If $\mathcal{A}$ has greater demands on either the cache or ORAM, the Ascend processor stops making forward progress in $\mathcal{A}$ until the next interval completes. Program obfuscation happens at three levels:

Level 1: Instruction obfuscation. To fetch an instruction, PC' is added to the ICache queue (Figure 2, 1). When the cache request is serviced, $I(PC')$ is decoded and the maximum number of reads needed for an arbitrary instruction are made to the RF (2). If $I(PC')$ requires less than the maximum number of operands, some of the RF requests are dummy requests. Next, all arithmetic execution units (ALU, etc) are invoked (3), and the DCache queue enqueues a read/write request for memory instructions (4). Non-memory instructions go through the motions of adding a request to the DCache queue, but do not actually add the request (e.g., by de-asserting a queue write-enable flag). Finally (5), real or fake writes are made to the RF (depending on whether $I(PC')$ performs RF writeback). If no instruction can be executed (which may happen because of a cache miss, described below), a dummy instruction that performs all of the actions described above is executed instead.

Level 2: Cache obfuscation. A pending request in a cache queue is only serviced once every *cache interval* cycles. We refer to this interval as DCINT for the DCache and ICINT for the ICache—both intervals are *public/static* parameters that are set by the server. To block requests from accessing a cache's data arrays, a dedicated hardware structure called the *cache FSM* ticks a counter once per cycle from $0 \ldots$ *cache interval*-1 (during which time the FSM is in the PENDING state) and sends exactly one pending request to the cache once the counter hits its maximum value (at which time it transitions to the ACCESS state). Requests sent to the cache perform the `access_cache()` operation in Figure 2 (6) and add a new request to the ORAM queue in the event of a cache miss (8). As before, the system must go through the motions of adding a new request to the ORAM queue in the event of a cache hit. If the ICache and DCache share the ORAM queue, an arbiter must decide (based on a public policy) which gets access when/if simultaneous requests occur (7). Once the request is complete, which the cache signals with a *done* flag, the cache FSM transitions back to the PENDING state and the process repeats. If there is no pending request in the cache queue when the FSM transitions to the ACCESS state, a dummy request (which performs the same operation as `access_cache()` with a dummy address/data) is made to the cache. While either a real or dummy request is being serviced, the processor continues to fetch/execute (possibly dummy) obfuscated instructions.

Level 3: ORAM access obfuscation. Pending requests in the ORAM queue are only serviced by the ORAM interface *ORAM interval* (OINT) cycles after the ORAM interface completes its last request (9). Similar to the cache FSM/queue, an *ORAM FSM* and *ORAM queue* store and regulate when ORAM requests are made. Once the OINT cycle threshold is met, either a pending request or a dummy request is sent to the ORAM interface.

The processor assumes that it has access to cache, RF and FIFO queue resources—all of which must be architected to make a specific request look like an arbitrary request. Figure 2 illustrates our approach for the DCache resource. All accesses to the DCache (either reads or writes) perform *both* a read and a write (6). If the access was a read, the old cache block is first read out of the cache into a holding register and then written back to the cache unchanged. In systems with large caches, physical access latency and energy per access changes based on where in the cache is being accessed. To address this issue, we split the DCache into banks (which are analogous to ways in a normal cache) and access each bank on each access. Banks are *monolithic* in that an observer should not be able to distinguish between one address in the bank being accessed versus another. Note that since the ICache has the capacity of one bank and is read-only, we do not apply these ideas to that memory.

Core model	in order, single issue, 1 instruction/per cycle
L1 ICache arch. hit, miss latency (baseline/Ascend)	32 KB, 4 way (LRU) 1/1, 0/*oram_latency*
L1 DCache arch. (baseline/Ascend) hit, miss latency (baseline/Ascend)	32 KB/1 MB, 4/16 way (LRU) 2/10, 1/*oram_latency*
L2 Cache arch. (baseline only) hit, miss latency	1 MB, 16 way (LRU) 10, 100
On/Off-chip network bandwidth	16 B/cycle
ICINT/DCINT/OINT AES encrypt/decrypt latency	0/5/450 100 total

Table 1: Default parameterization for performance model (see Figure 2). Latencies are in clock cycles.

4. EVALUATION

We evaluate Ascend over the SPEC06int benchmarks. We chose SPEC for its memory intensive nature—the benchmarks have nominal working set sizes of $\sim$ 900 MB [7]. Since accessing the ORAM (e.g., missing in the last level cache) is Ascend's largest overhead, SPEC stresses our system to the extent possible.

In all experiments, we run for 3 billion instructions with a warmup period of between 1 and 20 billion instructions, depending on the benchmark, to get out of initialization code. All results are collected in a two-step process. First, a functional instruction trace (with memory addresses for memory instructions) is generated using the SESC simulator's *rabbit mode*[3]. The instruction trace is then fed into the performance model shown in Figure 2, implemented as an event-driven simulator. The performance model accounts for all of the parameters shown in the Figure (consolidated in Table 1). In particular, the simulation models (a) instruction latencies by type (all non-memory instructions are single cycle), (b) cache architecture (capacity, associativity), (c) when cache hits/misses occur and their associated latencies, (d) ORAM interface interactions, and (e) various data transportation latencies (such as chip pin bandwidth).

SESC runs the MIPS ISA which is composed of mostly 2 operand/1 destination instructions. (In Figure 2 2, two operands will be read from the RF for every instruction.) Some MIPS instructions (e.g., floating point divide) require multiple cycles to complete. We break these instructions into multiple *mini-instructions*, each of which is single-cycle. Thus, Ascend must evaluate every mini-instruction after each instruction fetch to obfuscate its pipeline.

We compare against a baseline (*base*) processor that uses the same performance model as Ascend but (a) does not encrypt data, (b) services cache and off-chip requests as soon as those requests are made and (c) does not obfuscate instructions. Since a single-level cache hierarchy does not do modern processors justice, *base* has a unified L2 cache that backs 32 KB L1 I/D caches (see Table 1).

[3]Rabbit mode is similar to fast-forward and does not model timing.

$$
\begin{aligned}
access\_oram(\mathbf{ORAM\ 1}) &= aes\_latency + 2 * \left(ram\_latency + \left\lceil \log_2 \left(\frac{working\_set}{block\_size} \right) \right\rceil * Z * \frac{block\_size + 32}{pin\_bandwidth} \right) \\
&= 100 + 2 * \left(50 + \left\lceil \log_2 \left(\frac{2^{34}}{2^7} \right) \right\rceil * 4 * \frac{2^7 + 32}{16} \right) = \mathbf{2360\ cycles} \\
capacity(\mathbf{ORAM\ 2}) &= \frac{working\_set}{block\_size} * \frac{\left\lceil \log_2 \left(\frac{working\_set}{block\_size} \right) \right\rceil - 1}{8} = \frac{2^{34}}{2^7} * \frac{\log_2 \left(\frac{2^{34}}{2^7} \right) - 1}{8} = \mathbf{416\ MBytes} \\
access\_oram(\mathbf{ORAM\ 2}) &= 100 + 2 * \left(50 + \left\lceil \log_2 \left(\frac{capacity(\mathbf{ORAM\ 2})}{2^7} \right) \right\rceil * 4 * \frac{2^7 + 32}{16} \right) = \mathbf{1960\ cycles} \\
capacity(\mathbf{ORAM\ 3}) &= \frac{capacity(\mathbf{ORAM\ 2})}{block\_size} * \frac{\left\lceil \log_2 \left(\frac{capacity(\mathbf{ORAM\ 2})}{block\_size} \right) \right\rceil - 1}{8} \approx \mathbf{8.5\ MBytes} \\
access\_oram(\mathbf{ORAM\ 3}) &= 100 + 2 * \left(50 + \left\lceil \log_2 \left(\frac{capacity(\mathbf{ORAM\ 3})}{2^7} \right) \right\rceil * 4 * \frac{2^7 + 32}{16} \right) \approx \mathbf{1560\ cycles} \\
position\_map(\mathbf{ORAM\ 3}) &= \frac{capacity(\mathbf{ORAM\ 3})}{block\_size} * \frac{\left\lceil \log_2 \left(\frac{capacity(\mathbf{ORAM\ 3})}{block\_size} \right) \right\rceil - 1}{8} \approx \mathbf{136.5\ KBytes} \\
local\_cache(\mathbf{ORAM\ 1}, \mathbf{ORAM\ 2}, \mathbf{ORAM\ 3}) &= 27.1 + 24.3 + 21.5 = \mathbf{72.9\ KBytes} \\
\mathbf{total\_latency\_per\_oram\_access} &\approx \mathbf{2360 + 1960 + 1560 = 5880\ cyles}
\end{aligned}
$$

Figure 3: Cycle latency derivation for a 16 GB ORAM, split into 3 ORAMs (pseudo-code is given as `access_oram()` in Figure 2). ORAM 3 contains program data while ORAMs 1 and 2 store the position maps for ORAMs 2 and 3, respectively. Different local caches have different capacities to store different length paths (Section 3.1).

Figure 4: Ascend slowdown compared to the baseline system.

Shown in Figure 4, Ascend attains between $\sim 12 - 13.5\times$ slowdown relative to *base*. The two design points that we evaluate vary the benchmark working set. The 2 GB point is aggressive (the SPEC workloads have $<$ 1 GB working sets) while the 16 GB point shows scalability. When the memory per app increases by a factor of eight, performance drops by a factor of $1.12\times$.

We explored various ORAM parameters (e.g., the block size for each ORAM and number of ORAMs) to maximize performance *given a 250 KB on-chip budget*. The data ORAM's capacity is set based on the working set size (2 or 16 GB). The capacities for the position map ORAMs are set based on the data ORAM and each ORAM's block size. We found that small block sizes (128 B per block for every ORAM) are generally the right strategy: despite larger blocks amortizing ORAM accesses because more data is fetched per access, locality in the SPEC benchmarks seems to be erratic. Thus, most of the data in larger block sizes (we experimented with sizes up to 4 KB) is not used.

The on-chip storage includes the local cache for each ORAM and the position map for the smallest ORAM. For the 2 GB working set, the overall ORAM cycle latency is 5080 and the on-chip storage requirement is 91.5 KB—the 16 GB working set's cycle latency is 5880 while its on-chip storage requirement is $72.9 + 136.5 = 209.4$ KB (derivation shown in Figure 3).

5. CONCLUSIONS

We have shown the viability of a secure processor architecture that does not require trust in anything other than a single processor chip to guarantee the privacy of data provided by the client. Surprisingly, the slowdown associated with the architectural mechanisms is only $13.5\times$ on average—roughly comparable to the slowdown of interpreted languages.

6. REFERENCES

[1] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. ARM white paper, July 2004.

[2] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Compilation techniques for efficient encrypted computation. Cryptology ePrint Archive, Report 2012/266, 2012.

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

[4] C. Gentry, S. Halevi, and N.P. Smart. Homomorphic Evaluation of the AES Circuit. IACR eprint archive, 2012.

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

[6] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

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

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

[9] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, 2009.

[10] 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.

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

[12] 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 ACM CCS Workshop on Scalable Trusted Computing (STC'06)*, November 2006.

[13] 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.

[14] 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.

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

[16] 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} *Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[17] Trusted Computing Group. Trusted Computing Platform Alliance (TCPA) Main Specification Version 1.1b. https://www.trustedcomputinggroup.org/specs/TPM/TCPA_Main_TCG_Architecture_v1_1b.pdf, 2003.

[18] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.