Efficient Private Information Retrieval Using Secure Hardware

Xiangyao Yu, Christopher W. Fletcher *, Ling Ren, Marten van Dijk and Srinivas Devadas
MIT CSAIL, Cambridge, MA, USA
{yxy, cwfletch, renling, marten, devadas}@mit.edu

ABSTRACT

Existing crypto-based Private Information Retrieval (PIR) schemes are either limited in the set of queries they can perform, or have prohibitively large performance overheads in making query comparisons. On the other hand, most previous tamper-resistant hardware schemes can support more general queries and have lower performance overheads, but need to trust that the query program will not leak information about the user query.

We introduce Stream-Ascend in this paper, a processor that executes complex stream queries securely even if the query matching program is not trusted and would leak the query when executed on conventional secure processors. Stream-Ascend is based on Ascend, a recently-proposed tamper-resistant processor for untrusted programs. Directly applying Ascend to the PIR setting would result in significant inefficiency; Stream-Ascend is architected to support a streaming model of computation where each record is processed exactly once. Stream-Ascend is significantly more efficient than Ascend because its performance depends only on the working set of the application, not the length of the stream as in Ascend. Simulation results show that the performance overhead of Stream-Ascend relative to an insecure and idealized baseline processor is only 32.6%, 11%, and 13.5% for a set of streaming benchmarks.

1. INTRODUCTION

Private Information Retrieval (PIR) [9] is an important technique that allows a user to retrieve data from an untrusted server without the server being able to tell what data the user is interested in. In one setting, a cloud server maintains a huge amount of unencrypted data and has made this data accessible to many users as a service. Remote users, running on computationally limited (mobile) devices, send queries to the server with individual and private query criteria. The server's job is to run a query program, which could be supplied by the user, a third party or by the server itself, and return to the user data records that "match" the user's query. At the same time, the server should not be able to discover anything about the user's query, which is an input to the query program. The query could be a combination of keywords to be searched for in webpages, or may be an image with an associated closeness metric to other images.

PIR is useful in many settings. Queries on certain types of patents may reveal the confidential research projects of a company. In a map database, frequent queries to a particular geographical region may indicate the drilling location of an oil company. In both settings, if the untrusted server somehow figures out the access pattern of the user's query, crucial information will be leaked which may result in the company incurring losses. To achieve PIR, the memory access pattern induced by the query when the query program is run needs to be hidden.

Due to the importance of PIR, it has been the subject of considerable research attention from the cryptography and security communities. Two main approaches to solve PIR have appeared in literature: *crypto-based* and *tamper-resistant hardware*. Crypto-based solutions do not require any secure hardware, instead, cryptography (e.g., Fully Homomorphic Encryption [30, 15]) is used to theoretically guarantee that whatever the server does, it cannot learn the user's access patterns. Crypto-based approaches either suffer from significant performance degradation or sacrifice generality of queries to achieve efficiency (cf. Section 2).

On the other hand, *tamper-resistant hardware* has been proposed as a solution to PIR (e.g., [18, 36]). In this approach, a certain piece of hardware is trusted by the user, e.g., a secure coprocessor. The user will directly set up a secure channel with the trusted coprocessor and run the queries on it. This approach usually has better performance than crypto-based approaches for general queries. The major limitation of prior proposals is that the query program running on the coprocessor has to be trusted to not leak information through its interaction with *main memory* (or disk). This assumption seldom holds, especially when the untrusted server specifies the query program.

Ascend [14] is a recently-proposed processor that is capable of running untrusted programs while still being secure against software attacks. Ascend uses a two-interactive protocol where the user specifies the time T that a program runs for; Ascend will always run for time T before it returns the final result. This protocol limits the information leaked to just the estimated program execution time.[1] Ascend uses *Oblivious RAM* (ORAM, introduced in Section 3.2) to obfuscate memory access patterns. However, performance overhead increases with the ORAM size, which in turn grows with the number of data records searched.

*Christopher Fletcher was supported by a National Science Foundation Graduate Research Fellowship, Grant No. 1122374, and a DoD National Defense Science and Engineering Graduate Fellowship. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract N66001-10-2-4089. The opinions in this paper don't necessarily represent DARPA or official US policy.

[1]The untrusted server will be able to estimate program execution time for the user data; this has be shown to be the optimal leakage [13].

In the literature, PIR typically follows a *stream model* of computation. That is, the server streams *each data record* to a query program which compares the data record to the user query. For each query, the program should process each data record exactly once, otherwise, the untrusted server knows that the query is only interested in a subset of the data and information is leaked.

In this paper, we make crucial modifications to the previously-proposed Ascend architecture to implement a practical PIR system – *Stream-Ascend*. Since Stream-Ascend preserves privacy even under untrusted programs (and by extension prevents attacks stemming from query-dependent memory access patterns), it significantly improves the generality of previous tamper-resistant hardware approaches which can only run trusted programs. Furthermore, by making the input data streaming rate public and static, Stream-Ascend does not leak the dynamism in the streaming rate. This aspect is more secure than even crypto PIR because in crypto PIR, the rate of queries to the server leaks. To remove the overhead of ORAM in Ascend (Section 3), we made a key observation that only the working set of the currently processed record needs to be stored in ORAM. This significantly reduces both ORAM capacity requirement and the number of ORAM accesses.

In particular, we make the following contributions:

- We present an architecture and execution model for Stream-Ascend. Our design makes minimal hardware modifications to Ascend. To the best of our knowledge, Stream-Ascend is the first tamper-resistant hardware-based PIR system that efficiently supports complex streaming queries while allowing untrusted query programs.
- By adopting a streaming model and exploiting the fact that a query's working set will likely fit in on-chip cache, the performance bottleneck of ORAM is largely avoided.[2] Simulation results show that Stream-Ascend only imposes less than 32.6% performance overhead relative to an insecure baseline system.
- We modify the two-interactive protocol in Ascend to better fit the PIR setting. Instead of specifying the program running time T, the protocol specifies the rate at which records are streamed into the processor. After the execution, Stream-Ascend returns the top M records that best match the query criteria. We also describe a simple, secure way of estimating the stream rate that results in records being dropped with low probability.

The rest of this paper is organized as follows: Section 2 describes related work. Section 3 presents the necessary details of the Ascend processor on which Stream-Ascend is based. Section 4 describes Stream-Ascend. Section 5 evaluates Stream-Ascend with respect to baseline systems. Section 6 concludes the paper.

2. RELATED WORK

The two primary areas of related work are crypto-PIR techniques and tamper-resistant hardware schemes. We also briefly describe work in securing databases.

2.1 Crypto-Based Approaches

Most work on crypto-based PIR is based on homomorphic encryption schemes [20, 8, 22, 5, 26, 7]. With homomorphic encryption, the server performs computation between the user's query and unencrypted records without ever decrypting the user's query. The query program is implemented as a circuit made up of homomorphic operations. The search process then follows the stream model: for every unstructured data record, the server sends the query program the user's encrypted query, the record itself, and an encrypted buffer used to store matches (the *keep buffer*). The output of this computation is a new encrypted (and possibly updated) keep buffer. The encryption scheme must make it impossible for the server to tell whether the new keep buffer was updated. For instance, with homomorphic encryption, the output of any homomorphic circuit is encrypted using a probabilistic encryption scheme which changes the keep buffer's bits regardless of whether it was logically updated.

Important issues for homomorphic encryption-based PIR schemes are how many different types of Boolean queries they can support and their performance. Several previous works have used additive homomorphic cryptosystems [27, 11] as an underlying primitive, thus supporting only OR queries on a set of keywords (e.g., [20, 8, 22, 5]). While some generalizations have been made (e.g., [26] uses [7] to do a single AND on two sets of OR keywords), they are still quite limited in the types of queries that they can support and cannot implement more complex applications such as image matching. Recently, the first Fully Homomorphic Encryption (FHE) [30] technique was introduced [15]. With FHE, a stream-style algorithm can now handle user queries of arbitrary complexity. Unfortunately, FHE currently incurs huge overheads, about a billion times slowdown for straight-line code, and this overhead increases significantly with more complex filtering algorithms.

2.2 Tamper-Resistant Hardware Approaches

Motivated by the limitations in homomorphic encryption-based PIR techniques, another candidate paradigm for performing general searches over large sources of unstructured data is tamper-resistant hardware [18, 21, 36, 14, 37, 33]. In this setting, the query program and private user query are run inside a secure hardware compartment (typically a tamper-resistant processor chip or board) on the server side that protects the user's private data while it is being computed upon.

Outside of [14], previous work assumes that the query program is such that an adversary learns nothing about the user's query from the traffic on the processor's pins. This requires that the offchip memory access sequence of a query is independent of the query program's input data. The user has to check that the query program is written in such a way so as to not leak any information. A malicious query program can easily leak information about the query through memory traffic in [21, 36], for example.

Tamper-resistant hardware must implement schemes to prevent information leakage from physical and software attacks. A physical attack usually requires physical access to the tamper-resistant hardware: e.g., an attacker can induce a hardware fault [6] to change program behavior by tampering with the hardware's power pins, or passively monitor the hardware's power consumption [19], or listen for EM/RF emissions [2]. For example, a malicious insider may monitor a tamper-resistant processor's power consumption to learn how frequently it is using its cache, which may indicate that the keep buffer has just been updated. Software attacks, on the other hand, are mounted by a malicious piece of software

[2]Security is not compromised if the working set spills out of cache, thanks to ORAM.

and the attacker need not be present. Information can be leaked when a malicious process shares physical resources with the user process [38] or when the user process itself is subverted through a code injection attack. For example, a malicious sniffer program may try to learn if a particular location in external memory (DRAM) has been updated by constantly polling that location. Software attacks are a more pressing concern in a cloud server setting: they can be launched without physical access to the server racks and against many user processes without much additional effort on the part of the attacker. Thus, we will focus on protecting against software-based attacks for the rest of the paper.

2.3 Secure Databases

Previous works on securing databases ([3, 4]) have proposed using trusted hardware to compute critical stages in query processing. [3] uses a trusted FPGA to do standard database operations (e.g., filter, join, aggregate etc.) over encrypted data. This line of work has two main limitations: First, the type of queries supported is very limited. Second, though the data a certain query operates on is encrypted, the type of the query is leaked to the untrusted server.

3. ASCEND

A recently proposed tamper-resistant hardware processor, called Ascend [14], is designed to protect against software-based attacks when running *untrusted batch programs*. Ascend is a single-chip coprocessor that runs on the server-side. In this section, we describe Ascend and discuss some of its drawbacks when used in the PIR setting.

3.1 Ascend Architecture

Ascend was designed for batch computation, where all the program data must be present in the main memory after initialization. The key idea to guarantee privacy is *obfuscated program execution* in hardware; to evaluate an arbitrary instruction, Ascend gives off the signature of having evaluated any possible instruction. In particular, for an arbitrary batch program P (the query program in our setting), any two encrypted inputs x and x' (the user's private query), and any two sets of public data y and y' (the server's data stream), Ascend guarantees that from the perspective of the chip's input/output (I/O) pins $P(x, y)$ is indistinguishable from $P(x', y')$, therefore satisfying the criterion for oblivious computation [17].

To get this level of security, Ascend (a) encrypts all data sent over its I/O pins using dedicated/internal crypto-processors,[3] (b) obfuscates the address going off-chip using Oblivious-RAM (ORAM), and (c) accesses ORAM at *fixed and public intervals* to obfuscate *when* an request is made.[4] ORAM (details explained in Section 3.2) is conceptually a DRAM whose contents are encrypted and shuffled data blocks. It hides program memory address patterns as well as the data read from or written to memory. If the program has no outstanding request to ORAM when the public interval arrives, a *non-blocking* dummy request is made instead.

3.2 Oblivious-RAM

Oblivious-RAM (ORAM) [16, 25, 17, 31, 35] is one of the most important building blocks in Ascend. ORAM prevents any leakage on a program's memory access pattern. That is, an adversary should not be able to tell (a) whether a given memory request is a read or write, (b) which location in memory is accessed, or (c) what data is read/written to that location.

In the original proposal, Ascend used a recent construction called Path ORAM [34] for its practical performance and simplicity. We note that Ascend can be built on top of other ORAMs, and we experiment with Path ORAM and the ORAM of [31] in Section 5. We defer Path ORAM's detailed operation to Appendix A, but summarize points relevant to this paper below.

Path ORAM is composed of two parts—an external (untrusted) memory and a trusted *ORAM interface*. The ORAM interface is trusted, and built internal to Ascend in the same way as a conventional memory controller. The ORAM interface maintains a data structure that maps each program data block to a set of random locations in external memory, and the data block is guaranteed to be in one of the these locations. The number of locations in the set grows poly logarithmically with the size of the memory. (For the rest of the paper, a *block* is synonymous to a processor cache block). Each data block is stored in the external memory as a 3-tuple (address, data, ORAM bookkeeping information) that is encrypted under the user's session key.

When the program makes an address request to the ORAM interface, the ORAM interface converts the program address into the sequence of random addresses, which are then accessed and re-randomized.[5] To orient the reader, a representative insecure system can access a 128-byte block of program data in DRAM in 100 cycles and only needs to touch the block of interest. Path ORAM, on the other hand, must touch hundreds of data blocks to retrieve the 128-byte block (assuming a Path ORAM optimized for secure processors as in [28]). One can approximate the set of touched blocks as a random set of blocks in DRAM—a representative cycle latency per access from [28] is 3752 processor cycles.

3.3 User-Ascend Interaction

Ascend uses a *two-interactive protocol* (shown in Figure 1 (left)) to securely interact with users. First, the user chooses a secret session (symmetric) key K, encrypts it with Ascend's public key and sends it to Ascend. Second, the user sends its encrypted private inputs $encrypt_K(x)$, a public running time T and a public number of bytes to include in the final result R. The server then sends $encrypt_K(x)$, T, R, a public access interval for the ORAM ($ORAM_{int}$[6]), an (untrusted) program P of the server's choice, and public data y to an Ascend chip (**initialization step**). Ascend then runs for T time (**execution step**) and produces $z_{final} = encrypt_K(P(x, y))$ if T was sufficient to complete P given x and y, else $z_{int} = encrypt_K(\text{“}did\ not\ finish''})$ where $|z_{final}| = |z_{int}| = R$ (**termination step**). During execution, the program P owns the entire Ascend chip—Ascend currently supports a single user running a single thread at a time.

3.4 Ascend Security Model

[3] [14] assumes AES.

[4] The proposal in [14] also introduces schemes to protect against power analysis attacks on the power pins, but since power analysis is a physical attack, we will not use those schemes in this paper.

[5] I.e., the set of random addresses must be re-randomized to make accessing the same location twice indistinguishable from any other access pattern. See Appendix A.

[6] I.e., Ascend makes an ORAM request once every $ORAM_{int}$ clock cycles after the prior request completes.

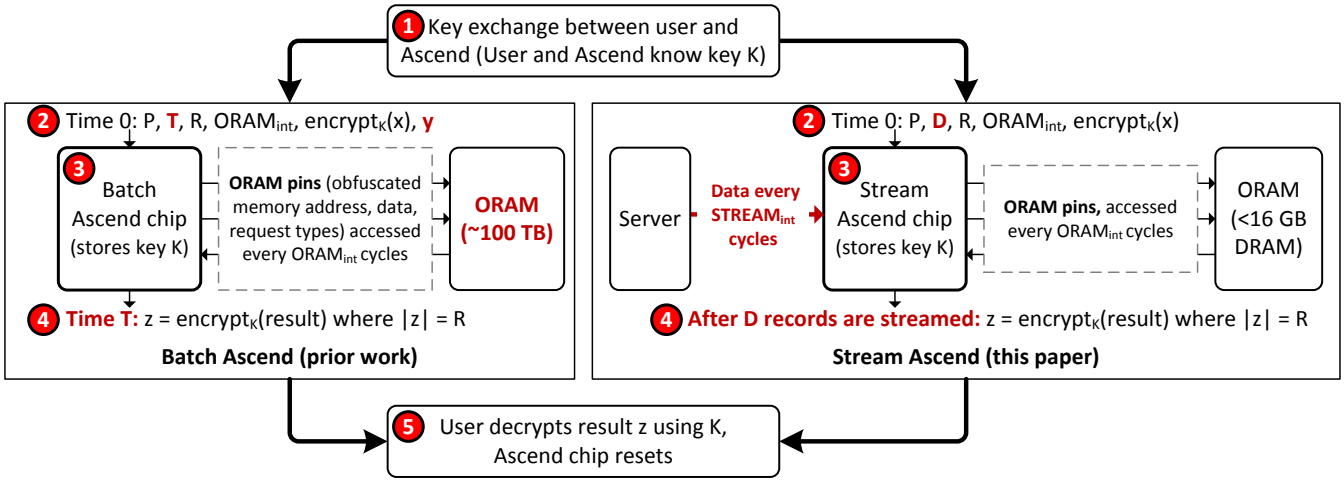


Figure 1: The two-interactive protocol for Ascend (left) and Stream-Ascend (right), assuming a stream computation over 100 Terabytes of unstructured data. Bold symbols indicate differences. Symbols are explained in Section 3.3.

Qualitatively, Ascend defeats software attacks for the following reasons. While running, a single thread owned by one user owns the entire Ascend chip. Thus, no attacks regarding resource sharing on-chip are possible (e.g., [38]). During the initialization step, the server may corrupt or send an Ascend chip incorrect $encrypt_K(x)$, T, R, $ORAM_{int}$, P, or y. Regardless, Ascend will run for exactly T time and output exactly R bytes. During the execution step, Ascend interacts solely with ORAM every $ORAM_{int}$ cycles, which is public and does not leak information. In each ORAM access, the address, data, and the operation (whether it is a load or store) are hidden due to the ORAM primitive (Section 3.2). Thus, sending Ascend a fake $encrypt_K(x), T, R, P$, or y does not reveal any information about the private input x.

Tampering with bits in ORAM. Since ORAM is stored in main memory, external to the Ascend chip, an adversary may try to change bits in the ORAM to influence program execution. Such attackers can be thwarted using integrity verification for external memory. Integrity trees to detect external memory bit tampering can be implemented as in Aegis [36]. [28] describes a more efficient way to integrate Merkle trees with Path ORAM.

3.5 Ascend Limitations for PIR

Ascend was shown to have reasonable overheads ($\approx 5\times$) for batch applications, i.e., SPEC benchmarks. However, Ascend cannot be directly applied to PIR applications, for reasons described in the next section.

3.5.1 ORAM scalability

To get reasonable performance overhead, Ascend limits its ORAM's capacity to fit in external DRAM. However, private queries in the PIR setting will be processing tens to hundreds of Terabytes of unstructured data, for potentially many users, per run. Since such large data sets do not fit in DRAM, ORAM would be implemented over both DRAM and disk. Such a huge "Oblivious disk" is impractical in an Ascend context for the following two reasons.

The first reason is large ORAM latency and low throughput. Take Path ORAM as an example. Suppose we need a 100 TB disk with a 128-byte block size. To access such a block in an insecure setting, a single request is made to disk. However, to access the same 128-byte block in Path ORAM, Ascend must touch ~ 660 blocks, plus some additional storage used for ORAM bookkeeping information. Though small chunks of blocks (e.g., 4 blocks) can be grouped together in Path ORAM—decreasing the effective number of disk accesses down to ~ 165, each access still turns into hundreds of accesses to random locations over many disks.

The second reason is storage overhead. All data in Ascend's ORAM must be encrypted with a single user's session key (Section 3.2). In the PIR setting, this means we have to copy terabytes of public data for each user, encrypted with each user's session key. Such data replication is not tenable for a large number of users. Even for a single user, upon receiving the first query, the server needs to allocate enough disk space for Oblivious disk required by the query, and initialize it with terabytes of public data. The initialization alone would take a considerable amount of time.

Previous work proposed parallelizing ORAM operations with coprocessors in data centers [23]. In their approach, a single ORAM is encrypted using a key not owned by any particular user and users interact with ORAM with per-user keys. Though their approach eliminates data replication, each user has to now trust all coprocessors as opposed to a single Ascend chip. And parallel operations only reduce latency; each ORAM access still requires 165 disk read/writes. In addition, having hundreds of coprocessors and hard disks working in parallel is itself expensive.

3.5.2 Two-Interactive Protocol

As described in Section 3.3, Ascend uses a two-interactive protocol to process users' queries: The user specifies execution time T, Ascend runs for exactly T time and returns final results (if the program terminates) or returns a *"did not finish"* flag. While this protocol works well for batch applications where intermediate results are usually meaningless, this is not true for PIR applications. In PIR, the server can always return the top M matching records among all the records it has already processed. The longer the program runs, the higher the quality of the M records, but intermediate results are meaningful. Ascend's two-interactive pro-

tocol fails to capture this feature since it always returns the *"did not finish"* flag if the program has not finished yet.

4. STREAM ASCEND

We propose Stream-Ascend to address the downside of Ascend on PIR applications. In this section, we present the key ideas of Stream-Ascend and illustrate how Stream-Ascend improves Batch-Ascend in the PIR setting. (We refer to the original Ascend proposal as Batch-Ascend in this section). Then, Stream-Ascend's architecture is discussed in detail.

4.1 Insights in Stream Ascend

Stream-Ascend uses three key insights to overcome the shortcomings of Batch-Ascend in the PIR setting, as is discussed below.

4.1.1 Stream Public Data at Intervals

A primary bottleneck in Batch-Ascend in the PIR setting is that it requires a multi-terabyte ORAM to hold the public data in order to obfuscate the access pattern to that data. One key observation is that in PIR, the access pattern to records is *public*: records are always scanned one-by-one. Such a public access pattern does not need to be hidden by ORAM. Furthermore, each record is only processed once and never accessed again. Based on these observations, we propose streaming all public *unencrypted* data into Ascend chip at a *public* rate which is controlled by the server. Ascend processes the stream of data records and only stores the top M matches in its ORAM. The ORAM only needs to accommodate the working set of the query program (which may depend on the record size) and the M matches. The ORAM will be *much* smaller than the whole dataset. If a record is not one of the M best matches so far, it is simply discarded after being processed. In this way, Stream-Ascend requires a significantly smaller ORAM than Batch-Ascend.

In Stream-Ascend, the user specifies the $STREAM_{int}$ parameter which is the *minimum* number of cycles between two words being streamed in by the server. The server is allowed to stream data into Stream-Ascend at any interval larger than $STREAM_{int}$. For security reasons, Stream-Ascend never sends any signal to the server indicating whether the stream is coming too quickly or slowly (i.e., no back-pressure mechanisms). The system will perform well if $STREAM_{int}$ is close to the true average processing time per word per record. If $STREAM_{int}$ is set too low, records may be lost because Ascend does not have enough time to process each record; if too high, Ascend will "wait" for the stream and performance will suffer. In all cases, Stream-Ascend does not leak any information apart from $STREAM_{int}$. We argue that this leaks the same amount of information as execution time T in Batch-Ascend. Since the public dataset size is public, the equivalent execution time can be calculated as $dataset\ size/STREAM_{int}$. Thus $STREAM_{int}$ and T are exposing exactly the same information in different ways.

Several design issues arise as data is streamed in at server controlled public intervals. For example, an efficient mechanism is required to determine $STREAM_{int}$ (cf. Section 4.6). Also, hardware support needs to be added to handle data streaming without extra information leakage (cf. Section 4.2).

4.1.2 Working Set Usually Fits in Cache

With the above optimization, Stream-Ascend successfully shrinks the ORAM size down to a practical number. However, ORAM latency even for practical-size ORAMs is quite large compared to an insecure DRAM (Section 3.2). We observe that using on-chip caches, we can virtually eliminate performance degradation due to ORAM latency, provided working set mostly fits in on-chip cache. The existence of an ORAM interface means that Stream-Ascend can handle spilling gracefully.

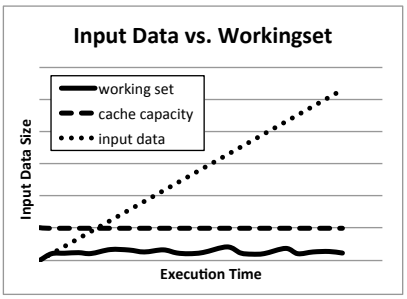


Figure 2: Input data size vs. working set size over time for the DocDist benchmark.

Figure 2 illustrates these observations for the DocDist benchmark. The total streamed-in data size increases linearly with time. But since we only process one record at a time, the working set of Stream-Ascend is limited. Records in the keep buffer are not part of the working set for the following reason. After processing one record, we only need to compare its score with the scores of the top M matching records in order to determine whether the processed record is kept or not. Thus, only scores and pointers to records need to be stored in cache. The working set is roughly M scores and pointers plus the memory required to process a single record. If the working set fits in on-chip cache, ORAM will be rarely accessed. In this case, the ORAM latency bottleneck is largely avoided.

4.1.3 Enhanced Two-Interactive Protocol

The improved two-interactive protocol for Stream-Ascend is shown in Figure 1 (right). Similar to the two-interactive protocol in Batch-Ascend, the user sends the query to the Ascend chip which processes data streamed in from the server (the streaming rate—$STREAM_{int}$ is discussed in Section 4.6). However, instead of sending back the final results or a *"did not finish"* flag after time T, Stream-Ascend sends to the user the intermediate results after streaming in D records. Intermediate results contain the top M matching records so far. These are meaningful results to the user though the quality may be worse than the final results when all data has been streamed in and processed.

4.2 Architecture Overview

Starting from Batch-Ascend, we designed Stream-Ascend with three goals listed below:

1. Minimize the number of ORAM accesses. Accessing a 4 GB ORAM costs thousands of cycles—much slower than accessing a conventional DRAM. Further, accessing ORAM at a strict public interval can up to double this access latency.[7] Therefore, we would like to minimize ORAM accesses as much as possible in Stream-Ascend.

[7]If the application suddenly needs ORAM right after a dummy request begins, for example.

2. Minimize record loss. A key difference between Batch-Ascend and Stream-Ascend over large data sets is that, for security reasons, Stream-Ascend must receive data from the server at a *user data-independent* rate (Section 4.1.1). Thus, we get a rate matching problem: the fastest input rate is constant yet the processing rate (how quickly Ascend processes each data record) varies. To avoid record loss, Stream-Ascend should buffer unprocessed records in cache and (only if necessary) in ORAM.
3. Minimize the hardware modifications that need to be made to Batch-Ascend.

The overall architecture of Stream-Ascend is shown in Figure 3. Two main hardware mechanisms are added to Batch-Ascend [14] to support streaming: a *Front End FIFO* and *multithreading*.

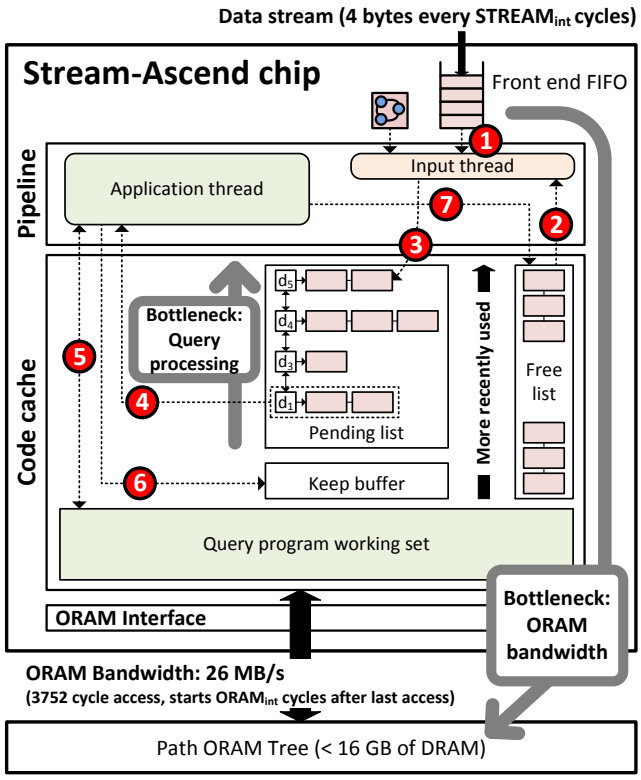


Figure 3: The Stream-Ascend architecture. Dotted arrows indicate to what structures each thread reads/writes to (see Section 4.4). ORAM bandwidth assumes the ORAM configuration in [14] (16 GB ORAM capacity, 128-byte cache lines, 1 GHz processor clock).

The **Front End FIFO** (FEF) is a dedicated hardware FIFO that receives data from the server. It serves as a synchronization buffer between Stream-Ascend's processing pipeline and the data stream.

We add **multithreading support** to Ascend so that it can concurrently move data from the FEF and run query programs. Stream-Ascend has two threads: the *input thread* and the *application thread*. The *Input thread* moves data from the FEF to a software data structure. The *Application thread* runs the user's private query program on the data records (that have been moved to memory by the input thread).

The application thread and the input thread communicate through three software structures: a free list, a pending list, and a keep buffer. The free list contains the empty memory blocks that can be used for storing data records. The pending list and keep buffer borrow blocks from the free list when they need them, and return blocks that are no longer used back to the free list. The input thread reads data from the FEF and puts it into the pending list, which is then read and processed by the application thread. After the application thread processes a record, if the record turns out to be one of top M matching records so far, it is stored in the keep buffer (which is also what the user gets from Ascend in the end, see Section 2.1). Otherwise, the record would be discarded and corresponding memory blocks are returned to the free list.[8] Initially, the free list contains all the available blocks, and the pending list and keep buffer are empty.

The pending list and the free list together form the *software buffer* of Stream-Ascend: data from the FEF is streamed into blocks in the free list (write into software buffer) and read out through the pending list (read from software buffer) by the query program. The free list's capacity is the software buffer size: if data records in the stream have a high variance in query processing time, the software buffer depth will have to be larger to avoid overflow.

4.3 The Front End FIFO

The FEF serves as a synchronization buffer between Stream-Ascend and the server. Every $STREAM_{int}$ cycles, 4 bytes (implementation-dependent but chosen to match the MIPS ISA word width) of data are streamed in from the server to the FEF via Ascend's external pins (Figure 3, 1). The FEF will always accept these bytes in the same way observed by the server, though internally Stream-Ascend may decide to keep or discard these bytes. The program running within Stream-Ascend can read the FEF at any time, by loading a special memory address (i.e., a memory map). But the server should never be able to find out how full the FEF is, or if it has overflowed.

To be implementable in hardware, the FEF should be very small ($\sim$ 1 KB in our evaluation). The FEF is not meant to handle significant noise in record processing time, but rather to tolerate small-scale noise that is inherent in the processor such as last-level cache misses.

The FEF can overflow and drop incoming data due to different reasons, e.g., $STREAM_{int}$ is set too low, the input thread does not get enough cycles on the pipeline, or it runs out of available data blocks (the free list becomes empty). When the FEF overflows, the entire (incomplete) record should be discarded. In this case, the FEF no longer stores the remaining portion of the damaged data record as it gets streamed in. It also sets a flag on the memory map to notify the input thread of the overflow.

4.4 Multi-threading

Stream-Ascend uses hardware multi-threading similar to conventional processors. Each thread has its own register file and program counter (PC), and shares the core pipeline, all the on-chip caches and the main memory. At any time, only one thread occupies the pipeline, and it can be swapped out for the other at cycle granularity (similar to fine/coarse-grained multi-threading; the operating system is not involved). We note that the server never sees the state of

[8] E.g., by re-assigning pointers between the keep buffer, pending list and free list.

any thread in plaintext at any point, which mitigates (for example) cache timing attacks [38].

4.4.1 The Input Thread

The input thread (when active) adds entries from the FEF to the pending list. It requests from the free list as many blocks as an entry needs (❷). When an entry is completely streamed in, the data blocks associated with the entry are attached to the pending list and the entry is marked as ready (❸).

In the case of the FEF overflowing, the input thread throws away any portion of the broken record it has buffered so far, and returns the associated blocks to the free list. Data in the FEF belonging to the broken records should also be thrown away.

4.4.2 The Application Thread

The application thread (when active) reads one entry from the pending list (❹) and runs the user's query on that entry (❺). After the processing finishes, the application thread either moves the entry to the keep buffer (if it decides to keep the record, ❻) or back to the free list (otherwise, ❼). If the keep buffer is already full, the lowest-scoring entry in the keep buffer is kicked out, and the corresponding memory blocks are returned to the free list.

4.4.3 Thread Swapping

Which thread owns the execution pipeline during any particular cycle is determined through both hardware and software mechanisms. These mechanisms are crucial to efficiency.

Hardware-triggered thread swapping is performed in two circumstances. First, any thread is swapped out in the case of a last-level cache miss. Last-level cache misses need to wait for an ORAM access to complete, which is typically thousands of cycles. Executing other threads instead of waiting can hide this latency to some extent.

Second, the input thread is swapped onto the pipeline when the FEF occupancy reaches a certain threshold and the freelist is not empty. This strategy is important to minimize FEF overflow with a small FEF. Without this strategy, for instance, if the application thread never incurs a last-level cache miss and always owns the pipeline, the FEF will eventually overflow.

Note that since the free list is a software structure, Stream-Ascend must be made aware of when it becomes empty. We assume a memory map-accessible register which is set by the Input thread (which generally takes memory blocks away from free list) and cleared by the Application thread (which returns blocks back to free list).

Software-triggered thread swapping puts the input thread or the application thread explicitly to sleep if they cannot make forward progress. The input thread goes to sleep if the free list runs out of memory blocks or if the FEF is empty. The application thread goes to sleep if the pending list is empty, which means there is no record to process.

4.5 Data Locality Optimizations

In Stream-Ascend, memory allocation is explicitly managed instead of using system calls (e.g., *malloc*() and $free()$). The consideration is that system calls are generally expensive and lack locality. In order to improve locality and reduce off-chip traffic, we organize both the free list and the pending list as stacks (as opposed to FIFOs). That is, when a new record is buffered and added to the pending list, it has a high probability of being processed by the query program before being evicted from the on-chip caches. As a comparison, if these lists are implemented in FIFOs, the first record inserted into the FIFO is more likely to have been evicted to the ORAM. In the worst case, all records would have to be pushed to and then pulled from ORAM. A stack-based implementation considerably reduces the number of ORAM accesses.

Since the free list and pending list are manipulated by both the input thread and application thread, stack push/pop operations can cause race conditions. In this paper, we do not assume hardware support for locks. Instead, we use lock-free stack designs. Specifically, we always push elements to the top of the stack, and pop the second top element from the stack. The second element has typically been pushed onto the stack recently and thus is likely to still be in on-chip cache.

4.6 Estimate $STREAM_{int}$

$STREAM_{int}$ is an important parameter in Stream-Ascend for rate matching. If data is streamed in faster than the processing speed, records will be dropped due to FEF overflow. If data is streamed in too slow, the processor will be idle most of the time waiting for input and performance overhead will increase. It is important to discover the threshold stream interval $THRESH_{int}$ when Stream-Ascend starts to drop records. As long as $STREAM_{int}$ is larger than $THRESH_{int}$, no records will be dropped. In order to estimate $THRESH_{int}$, we can set up a pre-computation phase to run Ascend on some sampled data records. The pre-computation phase works in the same way as batch-Ascend as described below.

Upon receiving the program from the user, the server first generates a set of sampled records from the entire input data. These records are fed into the Ascend chip and Ascend runs for a fixed and public T and computes the average time T_{exe} to process each word in the input. There are two major sources of overhead that may result in a larger cycles per word in the actual processing than T_{exe} in the pre-computation. One is the overhead of the input thread. The other is that data is streamed in at fixed intervals.

To deal with the first issue, Ascend can estimate the performance overhead of the input thread for each record (T_{input}). This overhead depends on the implementation of the input thread. In our implementation, T_{input} is approximately 240 cycles/word. With regard to the second issue, since a software buffer is used in Stream-Ascend between input and application threads, periodic input only weakly affects performance.

Ascend encrypts and sends $T_{total} = T_{exe} + T_{input}$ back to the user. To reduce the probability of record loss, the user can add a safety margin x and set $STREAM_{int} = T_{total} \times (1 + x)$. The user then sends $STREAM_{int}$ to the server. After such interaction, $STREAM_{int}$ is the only information revealed to the server. $T_{exe}, T_{input}, T_{total}$ are not revealed to the server.

4.7 Application Software Knobs

So far, we have assumed that Stream-Ascend takes a fixed amount of time to process a particular data record. When $STREAM_{int}$ is set too small, some records have to be dropped due to the rate matching problem. However, we can also shrink the amount of time to process a single record when the software buffer becomes too full by using software knobs to control the complexity of the program online. Such

Table 1: Microarchitecture for baseline and Stream-Ascend variants. On a cache miss, the processor incurs the cache hit plus miss latency.

Core model: in order, single issue	
Cycles per Arith/Mult/Div instr	1/4/12
Cycles per FP Arith/Mult/Div instr	2/4/10
Memory	
L1 I/D Cache	32 KB, 4-way
L1 I/D Cache hit+miss latencies	1+0/2+1
L2 Unified/Inclusive L2 Cache	1 MB, 16-way
L2 hit+miss latencies	$10+M_{latency}$
Cache block size	128 bytes
ORAM Capacity	4G
DRAM latency	100 cycles
Path ORAM latency	3752 cycles
Application: input data size	
Document matching	26.5 MB
DNA sequence matching	6 MB
Image Retrieval	15.5 MB

software knobs may be specified by the user and are changed dynamically depending on the status of Stream-Ascend (e.g., free list size).

In the CBIR application (cf. Section 5.2.3), for example, the program can tune the *number of octaves* or the *number of levels per octave* in order to control the complexity of the algorithm. The higher these parameters are, the larger the processing time, thereby requiring larger $STREAM_{int}$. Software knobs sacrifice the quality of matching results for some images, but may significantly reduce the record drop rate.

5. EVALUATION

In this section, we evaluate three applications (document matching, DNA sequence matching and content-based image retrieval) on Stream-Ascend and a baseline insecure coprocessor.

5.1 Methodology

All experiments are carried out with a cycle-level simulator based on the public domain SESC [29] simulator that uses the MIPS ISA. Instruction/memory address traces are first generated through SESC's rabbit (fast forward) mode and then fed into a timing model that represents a processor chip. Simulations are run until the entire data set is streamed through the simulator (which takes between 4 billion to 100 billion instructions depending on the benchmark and the dataset).

5.1.1 Comparison Points

We compare the following systems (all of which have the same on-chip microarchitecture, given in Table 1):

Stream-Ascend: The Stream-Ascend proposal described in Section 4. $STREAM_{int}$ and free list capacities are varied throughout the evaluation (depending on these values, Stream-Ascend may lose data records due to FEF overflow). ORAM capacity is set to 4 GB and access latency ($M_{latency}$ in Table 1) is set to 3752 cycles (using the model in [28]) and $ORAM_{int} = 1100$ cycles. As discussed in Section 4.1.1, only the working set for one record and the M top matching records need to be stored in the ORAM, which consumes a small amount of memory. We assume a large (and therefore slower) ORAM (4 GB) to be conservative: more ORAM capacity means more space for the free list.

Oracle-Ascend: An idealized Stream-Ascend design that implements the free/pending lists and keep buffer in magic hardware. Pending list capacity is infinite and initially contains the entire data stream, meaning that the Input thread is never active and that the Application thread can move through the stream at its own rate instead of the fixed $STREAM_{int}$ rate (and will therefore never incur record loss). Accessing a word in the pending list costs a single cycle always. ORAM parameters are the same as with Stream-Ascend. Compared to Stream-Ascend, this system removes the overhead of *input thread* and the overhead resulting from data streaming in at intervals.

Oracle-baseline: An *idealized* insecure processor. Oracle-Baseline is the same as Oracle-Ascend except that $M_{latency}$ is set to 108 cycles (conventional DRAM latency) and memory requests are serviced as soon as they are made (i.e., not set to intervals). This system models an insecure computation over unstructured data. Having the entire data stream present in magic memory models perfect stream prefetching (this is an option when security is not an issue).

5.2 Application Case Studies

We evaluate our system over three applications that process unstructured data given user search/filter critiera (ordered from least complex to most complex).

5.2.1 Document Matching

The first application (DocDist) compares documents for similarity. It takes a private set of document features f_u and a private distance metric from the user, and returns the documents whose features have the shortest distance to f_u. We show results for a corpus of several thousand wikinews pages [1] (which vary in length between 350 Bytes and 205 KB).

Docdist was designed to be minimalist and to be as sensitive to $STREAM_{int}$ as possible.

5.2.2 DNA Sequence Matching

The second application is DNA sequence matching (DNA), which takes the user's private query and returns the public DNA sequences that share the longest common substring with the user query.[9] We use DNA sequences from human and chimp chromosomes (6 million nucleotides in total, randomly broken into segments of length from 1K to 10K).

5.2.3 Content-Based Image Retrieval

Our third application is the content-based image retrieval (CBIR) application, which takes a (private) image specified by the user and returns a set of images that are most similar to the user image. The CBIR algorithm extracts SIFT features [24] for images, quantizes those features into a bag of words representation, and then compares each bag of words to a reference image (processed in the same way) for similarity [32, 10]. One example application is watermarking: for our evaluation we compare a secret 100×100 pixel watermark with 300 other images (varying in size between 6 KB and 30 KB) from the Caltech101 dataset [12]. We were constrained to this number and size of images due to simulation time constraints.

5.3 Performance Study

[9]The algorithm described here also works for general strings.

In this section, we evaluate Stream-Ascend under different parameter settings. For each experiment, the input data is at least several mega bytes (see Table 1). Though this is much smaller than the input data in a real streaming application (which can be hundreds of terabytes), we believe it is enough to provide interesting results because Ascend's on-chip cache size is only 1 mega byte: *much* smaller than input data size.

In our experiments, we set the number of returned records $M = 1$. We have run experiments with varying M; however, results are insensitive to M since the working set only increases slightly with increasing M. We vary record size in one experiment (cf. Section 5.3.4).

5.3.1 Software Buffer Size

Figure 4 shows the drop rate of Stream-Ascend with different software buffer (cf. Section 4.2) sizes and $STREAM_{int}$. Given a certain buffer size (which corresponds to a single curve in the figure), Stream-Ascend starts to drop records when $STREAM_{int}$ is smaller than $THRESH_{int}$.[10] For all benchmarks, $THRESH_{int}$ decreases for larger software buffers. This is because larger buffers can tolerate more variance in the input stream, so $STREAM_{int}$ can be set closer to the optimal value. Thre are diminishing returns after the software buffer is large enough.

When $STREAM_{int}$ is smaller than $THRESH_{int}$, drop rate increases almost linearly with respect to $STREAM_{int}$. Drop rate will be 100% when $STREAM_{int}$ goes down to 0. This corresponds to the case where data rushes into Stream-Ascend so fast that there is no time to process a single record.

For different applications, $THRESH_{int}$ is also different. For example, CBIR requires much more computation than DocDist for the same input data size, so $STREAM_{int}$ is also higher to match the CPU processing speed.

5.3.2 Infinite Software Buffer

To have a better understanding of the software buffer in Stream-Ascend, Figure 5 shows the performance of each application when the software buffer has a infinite size.[11] The y-axis shows the performance in terms of average number of cycles to process a single word, and the x-axis is $STREAM_{int}$. In this case, we won't drop records due to rate matching problems: there will always be space in the buffer for data to stream in.

If $STREAM_{int} > THRESH_{int}$, performance degrades linearly with respect to $STREAM_{int}$. In this region, the application thread is consistently waiting for the Input thread. So the input rate becomes the bottleneck of the system. Since each record is immediately processed after it is streamed in, the software buffer is always empty. Thus, even if the capacity of software buffer is finite, Stream-Ascend still drops no records.

If $STREAM_{int} < THRESH_{int}$, performance stays almost constant for a large region. When $STREAM_{int}$ is even smaller, input data accumulates in the software buffer which eventually becomes larger than the on-chip cache and is evicted to the ORAM. As a result, the application needs to access the ORAM more in order to pull this data back to the chip to process them. Performance is thus degraded since ORAM operations are expensive. This is obvious for DocDist in Figure 5(a). This effect is less pronounced with CBIR and DNA because those benchmarks are compute bound. That is, the time spent performing computation on-chip dominates the time spent waiting for ORAM, making performance insensitive to ORAM latency.

The observation above indicates that even if the total streaming data fits in ORAM, it is better not to set the $STREAM_{int}$ too small. The best performance will be achieved when input rate and processing rate match.

[10]Recall that $THRESH_{int}$ is the minimum $STREAM_{int}$ such that record loss is $> 0\%$; see Section 4.6.

[11]That is, the software buffer size is larger than the input data size. This is achieved by having a large free list (Section 4.2).

Benchmark	$STREAM_{int}$
Doc Dist	1580
DNA	40.9×10^3
Img	87.2×10^3

Table 2: $STREAM_{int}$ for each benchmark. Methods in Section 4.6 are used to compute $STREAM_{int}$ (10% safety margin assumed).

Benchmark	Oracle-Ascend	Stream-Ascend
Doc Dist	<0.1%	32.6%
DNA	<0.1%	11%
Img	2.6%	13.5%

Table 3: Performance degradation of Stream-Ascend and Oracle-Ascend with respect to Oracle-Baseline.

5.3.3 Determining $STREAM_{int}$

In a real implementation of Stream-Ascend with a finite software buffer, we need to keep $STREAM_{int}$ larger than $THRESH_{int}$ to achieve zero drop rate. Higher $STREAM_{int}$ will degrade the system performance linearly.

Based on the methods in Section 4.6, the $STREAM_{int}$ of each application is set to be the numbers in Table 2. The sampled data has the same distribution as the streaming data. And we add a 10% safety margin to each application to decrease the likelihood of dropping records.

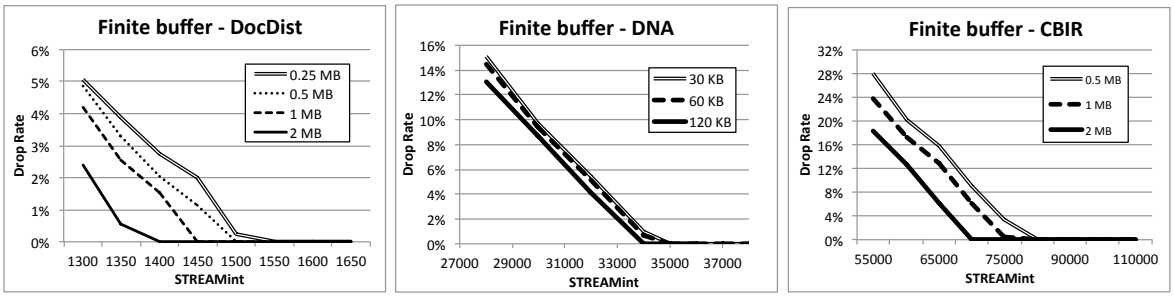
Under these $STREAM_{int}$ values, Table 3 shows the end-to-end performance of Stream-Ascend and Oracle-Ascend, in terms of slowdown with respect to Oracle-Baseline. The performance degradation of Oracle-Ascend shows the overhead of using ORAM. Since ORAM is rarely accessed with Oracle-Ascend (Oracle-Ascend rate matches perfectly), overhead due to ORAM is small for all benchmarks. Stream-Ascend's overhead comes from using ORAM, having a conservative $STREAM_{int}$ and from having a second thread (the input thread) running alongside the application. DocDist has the highest overhead because DocDist's $STREAM_{int}$ is much smaller than that of the other two benchmarks. The input thread therefore needs to perform more work per unit time, making its overhead more pronounced.

5.3.4 Sensitivity Study

An important insight of Stream-Ascend is that if data records all fit in the on-chip cache, then ORAM does not need to be frequently accessed. For all the experiments above, data records and working set are smaller than the cache size. In this section, we study the sensitivity of performance to record size and ORAM latency.

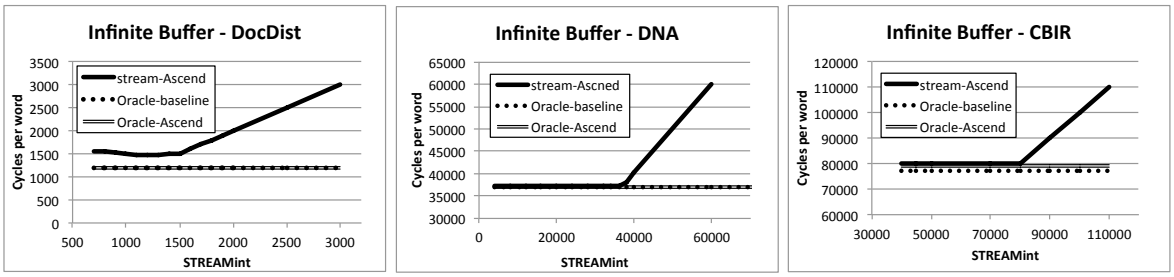
Record Size

Figure 6 shows the performance change by sweeping in-



(a) DocDist (b) DNA (c) CBIR

Figure 4: Drop rate vs. $STREAM_{int}$, sweeping software buffer size.



(a) DocDist (b) DNA (c) CBIR

Figure 5: Performance (number of words executed per cycle) vs. $STREAM_{int}$ with infinite software buffer size (drop rate = 0).

put record size in the DocDist and DNA benchmarks. When record size is smaller than on-chip cache capacity (1 MB), performance of both applications remains flat regardless of record size change. However, when records are larger than the on-chip cache size, ORAM needs to be frequently accessed for each record matching, which degrades performance. It turns out that the DNA benchmark is more compute bound than the DocDist benchmark. So DocDist is more sensitive to record size change.

For the CBIR benchmark, there is no clear correlation between input record (image) size and working set. In this case, the working set is determined by both the image size and the number of features in the images. Therefore CBIR is not shown in Figure 6.

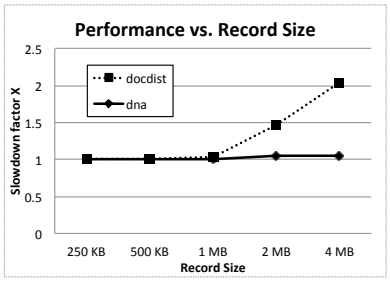


Figure 6: Performance of Stream-Ascend for different data record size, normalized to Oracle-Baseline. 1 MB cache size assumed.

ORAM Latency

Figure 7 shows the performance of the DocDist benchmark for two different ORAMs. The ORAM design of [31] (denoted as slow ORAM in the figure) is compared to Path ORAM. The access latency of the slow ORAM is 57708 cycles.

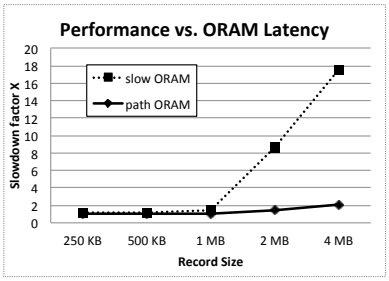


Figure 7: Performance of two different ORAM configurations running Docdist. Slow ORAM is described in [31].

When working set fits in the on-chip cache, Stream-Ascend completely removes the ORAM bottleneck. So performance does not degrade even if the ORAM is much slower. This fact makes Stream-Ascend very practical since it does not require much from the ORAM subsystem.

When the working set does not fit in cache, performance starts to degrade. However, Stream-Ascend is powerful enough that it can handle this case gracefully especially when Path ORAM is used.

6. CONCLUSION

We have demonstrated how to perform low-overhead PIR computation on streams of public data using a secure coprocessor that blocks software attacks. The proposed architecture, Stream-Ascend, is able to run complex user

queries with small performance overhead. In three typical benchmarks, we show that the total performance overhead from our system is less than 32.6% relative to an idealized baseline—indicating that private information retrieval is viable in certain large-scale data mining settings.

7. REFERENCES

[1] Wikimedia data dumps. http://meta.wikimedia.org/wiki/Database_dump.

[2] D. Agrawal, B. Archambeault, J. R. Rao, and P. 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.

[3] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. *Proc. of the 6th CIDR, Asilomar, CA*, 2013.

[4] S. Bajaj and R. Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 international conference on Management of data*, pages 205–216. ACM, 2011.

[5] J. Bethencourt, D. Song, and B. Waters. New techniques for private stream searching. Technical report, Carnegie Mellon University, March 2006.

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

[7] D. Boneh, E. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *TCC*, pages 325–341, 2005.

[8] Y. C. Chang. Single database private information retrieval with logarithmic communication. In *ACISP*, 2004.

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

[10] O. Chum, J. Philbin, and A. Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In M. Everingham, C. Needham, and R. Fraille, editors, *BMVC 2008: Proceedings of the 19th British Machine Vision Conference*, volume 1, pages 493–502, London, UK, 2008. BMVA.

[11] I. Damgard and M. Jurik. A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography*, pages 119–136, 2001.

[12] L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. In *IEEE. CVPR 2004*.

[13] C. Fletcher, M. van Dijk, and S. Devadas. Compilation techniques for efficient encrypted computation. Cryptology ePrint Archive, Report 2012/266, 2012.

[14] C. Fletcher, M. van Dijk, and S. 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, Oct. 2012.

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

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

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

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

[19] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.

[20] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, pages 364–373, 1997.

[21] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. 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.

[22] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *ISC*, pages 314–328, 2005.

[23] J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman. Toward practical private access to data centers via parallel oram. *IACR Cryptology ePrint Archive*, 2012:133, 2012. informal publication.

[24] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004.

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

[26] R. Ostrovsky and W. E. Skeith. Private searching on streaming data. In *Advances in Cryptology 96 CRYPTO 2005*, volume 3621 of *LNCS*, pages 223–240, 2005.

[27] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Eurocrypt*, pages 223–238, 1999.

[28] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. Cryptology ePrint Archive, Report 2012/76, 2013.

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

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

[31] 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.

[32] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*, ICCV '03, pages 1470–, Washington, DC, USA, 2003. IEEE Computer Society.

[33] S. W. Smith, D. Safford, and D. S. Ord. Practical private information retrieval with secure coprocessors, 2000.

[34] 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.

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

[36] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. 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.

[37] S. Wang, X. Ding, R. H. Deng, and F. Bao. Private information retrieval using trusted hardware. In *ESORICS*, pages 49–64, 2006.

[38] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

APPENDIX

A. PATH ORAM

Path ORAM organizes the physical (external) memory as a binary tree (Figure 8), where each node is a *bucket* that can hold up to Z data blocks (a block is a cache line in our setting). If a bucket has less than Z blocks, the remaining space is filled with dummy blocks. The root of the tree is referred to as level 0, and the leaves as level L. All the blocks are encrypted with randomized encryption. To access the external ORAM tree, Path ORAM has an *ORAM interface* which is implemented inside Ascend using trusted logic and is analogous to a normal processor's memory controller. The ORAM interface is made up of two main structures, a *position map* and a *local cache*.[12] The position map is a lookup table that associates 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 (we assume ~ 100) at a time.

At any time, each data block in the Path ORAM is mapped (randomly) to one of the 2^L leaves in the ORAM tree via the position map. Path ORAM maintains the following invariant: *if data block b is currently mapped to leaf l, then b must be stored either (a) on the path from the root to leaf l, or (b) in the local cache* (see Figure 8). The steps to access a block b in Path ORAM are as follows:

[12]"Local cache" is ORAM terminology, not to be confused with on-chip processor caches.

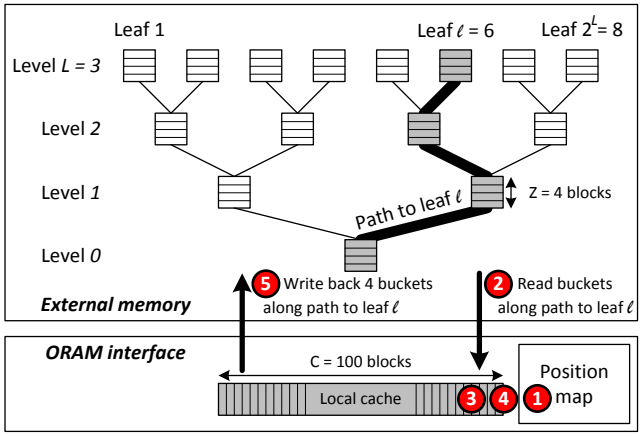


Figure 8: A Path ORAM for $L = 3$ levels. Numbers indicate the steps (from Section 3.2) to access a block mapped to leaf $l = 6$.

1. Look up the position map with block b's virtual address, yielding the corresponding leaf label l.
2. Read all the buckets along the path to leaf l. Decrypt all blocks within Ascend and add them to the local cache if they are real (i.e., not dummy) blocks.
3. Return b to Ascend's pipeline on a read or update b on a write.
4. Assign a new random leaf l' to b (update the position map).
5. Evict and encrypt as many blocks from the local cache to buckets from the root to leaf l. Fill any remaining space on the path with dummy blocks.

In Step 4, a block is randomly remapped to a new leaf whenever it is accessed. This is the key to Path ORAM's security: it guarantees that a random path is read and written on every access regardless of the requested address sequence. The path read and write (step 2 and 5) should be done in a data-independent way (e.g. from the root to the leaf).

The position map is usually too large for a processor's on-chip storage, so Batch-Ascend [14] implements a hierarchical Path ORAM. In a 2-level hierarchical Path ORAM, for instance, the original position map is stored in a second ORAM, and the second ORAM's position map is stored on chip. The above trick can be repeated, i.e., adding more levels of ORAMs to further reduce the final position map size at the expense of increased latency. [14] assumed a path ORAM latency of 5880 cycles. [28] proposed several optimizations to Path ORAM, and improved the performance to 3752 cycles.

In Path ORAM, there is a notion of *ORAM failure* which means the overflow of local cache. Though the lower bound of *ORAM failure rate* has not been published yet, it is not a problem in Ascend because of the *background eviction* technique. Interested readers can refer to [28] for more details.