View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

Generalized External Interaction with Tamper-Resistant Hardware with Bounded Information Leakage

Xiangyao Yu
Massachusetts Institute of Technology
yxy@mit.edu

Christopher W. Fletcher*
Massachusetts Institute of Technology
cwfletch@mit.edu

Ling Ren
Massachusetts Institute of Technology
renling@mit.edu

Marten van Dijk
University of Connecticut
vandijk@engr.uconn.edu

Srinivas Devadas
Massachusetts Institute of Technology
devadas@mit.edu

ABSTRACT

This paper investigates secure ways to interact with tamper-resistant hardware leaking a strictly bounded amount of information. Architectural support for the interaction mechanisms is studied and performance implications are evaluated.

The interaction mechanisms are built on top of a recently-proposed secure processor Ascend [11]. Ascend is chosen because unlike other tamper-resistant hardware systems, Ascend completely obfuscates pin traffic through the use of *Oblivious RAM* (ORAM) and *periodic* ORAM accesses. However, the original Ascend proposal, with the exception of main memory, can only communicate with the outside world at the beginning or end of program execution; no intermediate information transfer is allowed.

Our system, Stream-Ascend, is an extension of Ascend that enables intermediate interaction with the outside world. Stream-Ascend significantly improves the generality and efficiency of Ascend in supporting many applications that fit into a streaming model, while maintaining the same security level. Simulation results show that with smart scheduling algorithms, the performance overhead of Stream-Ascend relative to an insecure and idealized baseline processor is only 24.5%, 0.7%, and 3.9% for a set of streaming benchmarks in a large dataset processing application. Stream-Ascend is able to achieve a very high security level with small overheads for a large class of applications.

*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCSW'13, November 8, 2013, Berlin, Germany.
Copyright 2013 ACM 978-1-4503-2490-8/13/11 ...$15.00.
http://–enter the whole DOI string from rightsreview form confirmation.

Categories and Subject Descriptors

B.4.2 [**Input/Output Devices**]: Channels and controllers; C.2.0 [**General**]: Security and protection

Keywords

Ascend; tamper-resistant hardware; interaction; pin traffic; streaming

1. INTRODUCTION

Privacy in cloud computing is a huge security problem. For computation outsourcing, the user sends his/her jobs to the server who does the computation for the user. To be secure, the server should convince the user that his/her privacy is not leaked. In an ideal setting, the user's sensitive data and perhaps the sensitive program as well, should be completely protected such that no adversary (including the server) can learn any information about it.

One solution to achieve security is to use tamper-resistant hardware and secure processors. In this setting, the user sends his/her encrypted data (and program as well if the program is private) to the trusted hardware, inside which the data is decrypted and computed upon. The results of computation are encrypted and sent back to the user. Many such hardware platforms have been proposed, including Intel's TPM+TXT [15], which is based on TPM [35, 3, 26], eXecute Only Memory (XOM) [17, 18, 19] and Aegis [33, 34].

Among all previously-proposed tamper-resistant hardware, Ascend [11] is the first to completely obfuscate the traffic on the processor's pins. As a result, Ascend does not require any trust in the program or operating system that is running on the processor as in other tamper-resistant hardware proposals. Ascend achieves this level of security mainly through two techniques:

- Oblivious RAM (ORAM) is used to obfuscate main memory accesses, and
- Main memory accesses are made at public and fixed intervals, protecting against timing attacks.

To protect leakage through program *input* and *output*, Ascend adopts a *two-interactive protocol* where the user specifies encrypted input of fixed and public length to Ascend. Ascend will run the program for a fixed amount of time and also return an encrypted final result of fixed and public length.

The limitation of this protocol is that Ascend cannot accept inputs or produce outputs during program execution. This significantly limits the types of applications that can run on Ascend: both input and output must be transmitted one-time. For example, packet processing in routers has unceasing input coming from the network; a database application needs to read data from disks that cannot fit in the ORAM in Ascend; and many applications need to interact with the user during program execution. All of these applications cannot run on Ascend.

In this paper, we introduce a general model that allows any tamper-resistant hardware to interact with the outside world during program execution without leaking information through chip pins (cf., Section 3). Only small and user-controlled amount of leakage occurs before execution and does not grow over time. The interaction model is based on data streaming as in *private stream search* [23, 6] and is also applicable to database query processing and packet processing. Though the interaction model discussed in this paper is general and applicable to any tamper-resistant hardware, we will use Ascend as an example to explain the mechanisms, since Ascend is the only secure processor that successfully obfuscates chip pin traffic.

We make crucial modifications to Ascend and design a new system called *Stream-Ascend*. Stream-Ascend extends Ascend by allowing the processor to interact with the outside world during program execution while preserving the security of Ascend by making all the interactions data- and program-independent. We also make a key observation that the performance overhead of Ascend can be largely eliminated by taking advantage of the streaming computation model. That is, if the working set of the currently-processed record fits in the cache, Stream-Ascend can significantly reduce both the ORAM capacity requirement and the number of ORAM accesses.

In particular, we make the following contributions:

- A general model is proposed to securely interact with any tamper-resistant hardware. The model guarantees that privacy cannot be leaked through chip pins.
- Based on this model, we present an architecture and execution model for Stream-Ascend, which is built on top of Ascend [11]. Stream-Ascend significantly extends the generality and efficiency of Ascend. To the best of our knowledge, Stream-Ascend is the first tamper-resistant hardware-based 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. Simulation results using large dataset processing benchmarks show that Stream-Ascend only imposes less than 24.5% performance overhead relative to an insecure baseline system.

The rest of this paper is organized as follows: Section 2 presents the necessary details of the Ascend processor on which Stream-Ascend is based. Section 3 discusses the general model to securely interact with tamper-resistant hardware without leaking information through chip pins. Section 4 describes Stream-Ascend. Section 5 describes application scenarios. Section 6 evaluates Stream-Ascend with respect to baseline systems. Section 7 describes related work. Section 8 concludes the paper.

2. BACKGROUND

A recently proposed tamper-resistant hardware processor, called Ascend [11] is designed to protect against all software-based and some hardware-based attacks when running *untrusted batch programs*. Ascend is a single-chip coprocessor that runs on the server-side. In this paper, we consider an honest-but-curious server who may try to learn user's private information by observing the pin traffic of Ascend. In this section, we describe Ascend and discuss some of its drawbacks.

2.1 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, any two encrypted inputs x and x' (the user's private query), and any two sets of public input y and y' (server's public data), 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 [14].

To get this level of security, Ascend (a) encrypts all data sent over its I/O pins using dedicated/internal crypto-processors (e.g., AES) (b) obfuscates the address going off-chip using Oblivious-RAM (ORAM), and (c) accesses ORAM at *fixed and public intervals* to obfuscate *when* a request is made. ORAM (details explained in Section 2.2) 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, but not the timing of ORAM accesses. The timing channel is protected by periodic accesses.

2.2 Oblivious-RAM

Oblivious-RAM (ORAM) [13, 22, 14, 27, 31] is one of the most important building blocks in Ascend. ORAM prevents any leakage from 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 [32] 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 [27] in Section 6. In this paper, we treat ORAM as a black box which serves as a secure main memory. The key feature of ORAM is its high access latency: according to [24], each Path ORAM access is translated to multiple DRAM accesses which consume 3752 processor cycles for the ORAM configurations in Table 1. Readers can refer to [24, 32] for details of Path ORAM.

2.3 User-Ascend Interaction

Ascend uses a *two-interactive protocol* (shown in Figure 1) 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 $encrypt_K(x_{user})$, a public running time T and a public number of bytes to include in the final result S. At the same time, the server

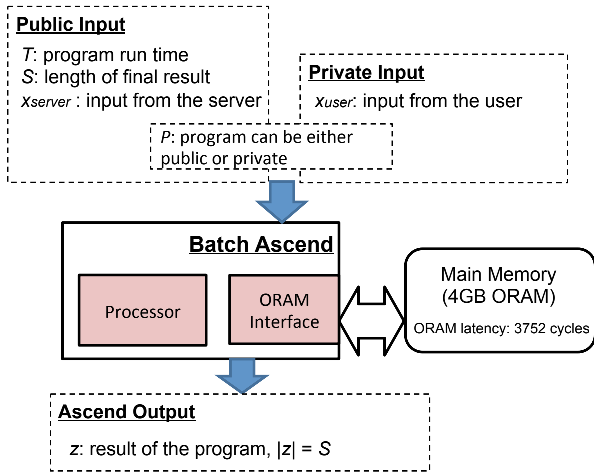


Figure 1: Ascend Architecture and user-Ascend interaction.

sends the public input x_{server} to Ascend. The program P can be provided by either the user or the server. Ascend then runs for T time (**execution step**) and produces $z_{final} = encrypt_K(P(x_{user}, x_{server}))$ if T was sufficient to complete P, else $z_{int} = encrypt_K(\text{“}did\ not\ finish''\text{)}$ where $|z_{final}| = |z_{int}| = S$ (**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.

2.4 Ascend Limitations

Ascend was shown to have reasonable overheads ($\approx 2\times$) for compute-bound batch applications, e.g., most of the SPEC benchmarks ([24]). However, the computation model of Ascend is still very limited.

2.4.1 Two-Interactive Protocol

As described in Section 2.3, Ascend uses a two-interactive protocol to process users' queries. All input to the program should be specified at the beginning of the execution and only the final results can be returned to the user. This model does not allow the use of Ascend in applications that require constant input/output from/to the outside world, e.g., network traffic, user input, etc.

For example, data filtering at a network router cannot be implemented using Ascend because, first, input data arrives constantly during execution instead of at one time, and second, the user needs output constantly.

For the rest of the paper, we will refer to the original Ascend proposal as Batch-Ascend.

2.4.2 ORAM scalability

For certain applications, instead of constantly requesting data during program execution, Ascend can conceivably load all the data it needs before any computation starts. In a database application for example, Ascend can load all the tables into its main memory (which is implemented as an ORAM) and obfuscate the access pattern. This usually requires an ORAM size on the order of terabytes which does not fit in DRAM. Thus, the ORAM would be implemented over both DRAM and disk. Such a huge “Oblivious disk” is impractical in an Ascend context due to ORAM scalability.

Take a Path ORAM on 100 TB disks as an example. Instead of accessing a single data block as in an insecure system, Path ORAM needs to touch $\sim$ 660 blocks. 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 $\sim$ 165, each access still turns into hundreds of accesses to random locations over many disks.

Previous work proposed parallelizing ORAM operations with coprocessors in data centers [20]. In their approach, multiple trusted coprocessors split a single ORAM access into parallel accesses in order to reduce access latency. However, the power consumption of each ORAM access is not reduced (actually, more power is consumed). Furthermore, parallelization requires user's trust in many coprocessors instead of one.

Recent work, namely ObliviStore [30], significantly improved ORAM performance in a cloud storage setting. However, with a 1 TB file system, a throughput of only several MB/s is achieved with SSDs, and less than 1 MB/s with HDDs.

3. INTERACTION WITH TAMPER-RESISTANT HARDWARE

Tamper-resistant hardware may interact with external equipment like memory, network interfaces, user input devices, etc. In order to not leak information through chip pins, these interactions must happen in a data-independent way such that an adversary cannot figure out what messages are being transferred or the pattern of message flow. The *one-time* input/output in Batch-Ascend is a secure way to interact with tamper-resistant hardware. However, the model can only support limited applications (c.f., Section 2.4.1).

As discussed in the previous section, Ascend obfuscates the access pattern over the memory channel by using Oblivious RAM. However, ORAM does not help much with network interfaces or user inputs which require different communication strategies.

3.1 Streaming Model

Streaming computation is a simple model that achieves a very high security level. In this model, all the data is streamed into the trusted hardware, the results are streamed out and the operations on each data record look indistinguishable to an adversary. Security of the system relies on the fact that all the data records exhibit data-independent behaviors. The streaming model is quite general and many applications fit into it. Examples are sequential scan in databases, network routers, data filtering, online news feeds, user input, etc. Though it is true that streaming limits the programming model to certain types of applications, this is the price to pay in order to fully obfuscate pin traffic. The streaming model has been widely adopted in *private stream search* [23, 6].

The methods of securely interacting with tamper-resistant hardware can be extended to a *stream of interaction*.

3.2 General Interaction Model

Figure 2 shows the basic structure of the interaction model. Similar to Batch-Ascend, the system is assumed to contain two parts: the *untrusted server* and the *tamper-resistant hardware*. The key security insight of our interaction model is the following: if the pin traffic of the tamper-resistant hardware is completely controlled by the server and

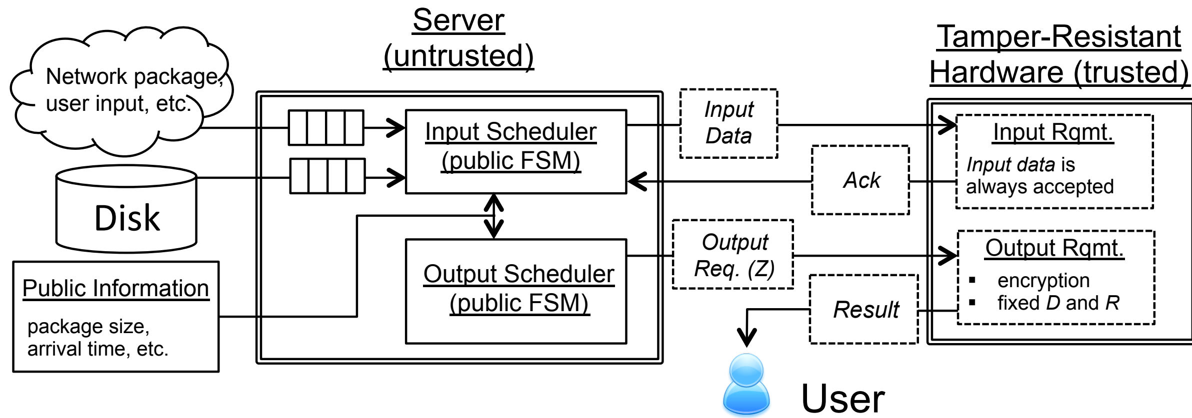


Figure 2: General model of interaction with tamper-resistant hardware.

the server does not know any private information of the user, then the pin traffic will not leak any privacy. We will show how this insight is realized by going through each block in the figure in the sequel.

3.2.1 *Untrusted Server*

The server can load data from data sources (e.g., network interfaces, disks, etc.) into the internal buffers and transfer these data to the trusted tamper-resistant hardware. For security reasons, how the data is sent to the trusted hardware must happen in data-independent ways. Specifically, an *input scheduler* is introduced to achieve this goal.

The *input scheduler* is a public *finite state machine (FSM)* which takes public information as input and generates a public scheduling which indicates when the data in the server's buffer is sent to the tamper-resistant hardware. The input to the *input scheduler* can be any public information observable by the server, including input arrival rate, input size, packet type (which may be public for certain applications), ORAM latency, network congestion, etc. The public *input scheduler*, which may be specified either by the user or by the server, collects these inputs and switches internal state accordingly. Based on its state the *input scheduler* will occasionally send data blocks to the trusted hardware which will consume the data blocks in a data-independent way.

Similarly, the *output scheduler* is also a public FSM specified either by the user or by the server. Outputs of the *output scheduler* are requests to the tamper-resistant hardware with a result size Z. Upon receiving a request, the tamper-resistant hardware will return a fixed amount of encrypted data of size Z in a fixed pattern back to the user.

The tamper-resistant hardware can only get input data from the *input scheduler* and can only output data upon receiving a request from the *output scheduler*. The schedulers completely determine the behavior of I/O pins of the trusted hardware.

3.2.2 *Tamper-Resistant Hardware*

For security reasons, upon receiving the *input data* and the *output request* from the schedulers, the response of the tamper-resistant hardware should not leak any data- or program-dependent information. The requirements on the hardware are listed below.

Input requirement: The trusted hardware should always accept (e.g., write to a hardware buffer) the input data in the same way as observed by an adversary, regardless of the content of the data or the internal state of the hardware.

Output requirement: Upon receiving the request from the output scheduler, the trusted hardware should return the output data in a fixed pattern. For simplicity, we assume that the hardware will start to return the data after time D upon receiving the request, and the data transfer rate should always be R bytes/s. D and R are determined by hardware and cannot be changed. Furthermore, all the output data should be probabilistically encrypted. If there are no results ready to be sent out, the hardware sends encrypted dummy results at the right time.

The hardware implementation that achieves the above requirement will be discussed in Section 4.

3.3 Security of Interaction

The interaction model discussed above is secure against attacks through observing chip pins. The key idea is that the pin traffic of the tamper-resistant hardware is completely controlled by the server. The behavior of the trusted hardware in terms of pin traffic will be completely public and predictable by an adversary, regardless of the internal state of the hardware. Adversaries learn nothing new observing the pin traffic.

It is true that the scheduler itself may leak some information about the application. If the server specifies the scheduling algorithm, no private information is leaked. But if the user specifies the scheduling algorithm, leakage may happen. The user may decide to use a certain scheduler rather than another one for performance considerations and this decision leaks some features about the data and program. However, this leakage is user-controlled and only one-time when the scheduler is specified and does not increase over time. The tradeoff between performance and security is made by the user.

In practice, the server can provide input/output schedulers that are general enough for all users as the default setting. The default setting achieves reasonable performance in general and does not leak private information (since it is provided by the server). Users are still allowed to define their own schedulers to further exploit performance, with the cost of more leakage.

4. STREAM ASCEND

The discussion in the previous section is general to all tamper-resistant hardware based systems. But the trusted hardware has been treated as a black box that magically achieves all the requirements. In this section, we will focus on a hardware implementation to support the interaction model. Specifically, the mechanisms are built on top of Ascend since it is the only secure hardware that completely obfuscates the pin traffic using ORAM and periodicity. The resulting system is called *Stream-Ascend*. The architecture of *Stream-Ascend* will be discussed in detail.

4.1 Two-Interactive Protocol

According to Section 2, the user communicates with Batch-Ascend using a *two-interactive protocol* where the user specifies the program runtime T and result size S.

With the general interaction model, however, the user should instead specify the *input scheduling algorithm* and the *output scheduling algorithm*. The original two-interactive protocol can also fit into our new interaction model with extremely simple FSMs. In this case, the input scheduling is to put data into Stream-Ascend at time 0. And the output scheduling is to request output of size S at time T.

4.2 Architecture Overview

In order to efficient support the interaction model and to meet the requirements listed in Section 3.2.2, three main hardware mechanisms are added to Batch-Ascend [11]: a *Front End FIFO (FEF)*, a *Back End FIFO (BEF)* and *multithreading*.

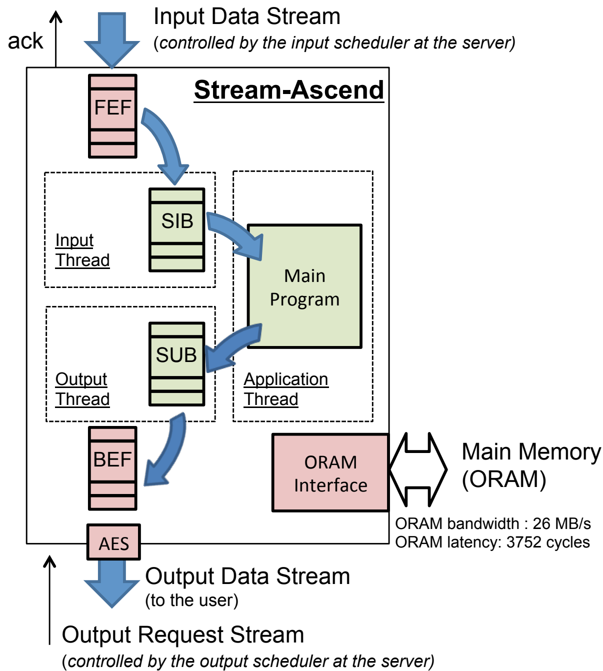


Figure 3: The Stream-Ascend architecture. Two main hardware structures (FEF and BEF) and multithreading support are added on top of Batch-Ascend. Blue arrows show the direction of data flow.

The overall architecture of Stream-Ascend is shown in Figure 3. Both the FEF and BEF are used to temporarily store the data blocks. FEF is used to receive data from streaming input, and the BEF is used to store messages that are ready to be sent back to the user. Both FIFOs serve as synchronization buffers between Stream-Ascend and the outside world in order to tolerate small mismatches between processing rate and streaming input/output rate.

In practice, however, the above mentioned mismatch can be large. We implement a software buffer for each FIFO as a backup in order to tolerate large mismatch. These software buffers are called *Software Input Buffer* (SIB) and *Software oUtput Buffer* (SUB).

In order to efficiently move data between the FEF/BEF and the corresponding software buffer, multithreading support is added to Stream-Ascend. Two threads—*input thread* and *output thread* are specifically assigned to data moving. We will introduce several context switching strategies in Section 4.4.3 to minimize the size of the FEF and BEF.

4.3 The Front/Back End FIFO

Both the FEF and BEF serve as synchronization buffers between Stream-Ascend and the input and output streams. They are key to meet the requirements in Section 3.2.2

The FEF will always accept the data coming from the server in the same way as seen by an adversary, though internally Stream-Ascend may decide to keep or discard the data. Inside Stream-Ascend, the input thread will constantly move data from the FEF (if not empty) to the SIB (if not full). But an adversary should never be able to find out how full the FEF is, or if it has overflowed.

In practice, the FEF may overflow if data is streamed in faster than the speed of processing. Both the FEF and SIB may accumulate over time and overflow eventually. When the FEF overflows, the entire (incomplete) record that causes the overflow 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, which will clean up the SIB in software.

BEF is similar to FEF in that an adversary cannot observe its interval states. But unlike the FEF, BEF will never overflow. If the BEF is full, the output thread stops moving data from the SUB into it. Upon receiving the output request from the *output scheduler* at the server, the BEF will output encrypted results to the user at a fixed pattern (c.f., Section 3.2.2). If the BEF is empty, encrypted dummy results will be returned.

Notice that all the operations mentioned above are internal to the Stream-Ascend. An adversary from the outside can only see that input data are consumed and output requests are satisfied. So no information is leaked.

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 another at cycle granularity (similar to fine/coarse-grained multi-threading; the operating system is not involved).

4.4.1 The Input/Output Thread

The input thread (when active) adds data from the FEF to the SIB. When an record is completely streamed in, the record is marked as ready for the application thread.

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

The output thread is similar to the input thread but the data flows in reverse order: results are first written to the SUB and then moved to the BEF by the output thread.

4.4.2 The Application Thread

The application thread reads an record from the SIB and runs the user's query on it. The ORAM may be accessed while the query is being processed.

The application will store the results of processing to SUB which is internal to Stream-Ascend. If the SUB is full, which means the output speed is slower than the processing speed, the application thread will simply drop the results of the current data record.

4.4.3 Context Switch

Which thread owns the execution pipeline during any particular cycle is determined through both hardware and software mechanisms. These mechanisms are crucial to make the size of the FEF and BEF small enough in practice.

Hardware-triggered context switch is performed in two circumstances. First, any thread is swapped out in the case of a last-level cache miss which triggers an ORAM access that takes at least thousands of cycles. Executing other threads instead of waiting can hide this latency to some extent.

Second, the input/output thread is swapped onto the pipeline when the FEF/BEF occupancy reaches a certain threshold and the SIB/SUB is not full/empty. This strategy is important to minimize hardware FIFO overflow with a small FIFO size. 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 and the BEF will drain out.

The emptiness/fullness of the SIB/SUB is signified using a memory map-accessible register implemented in hardware. The register is set by the thread at the upstream of the data flow and cleared by the thread at the downstream.

Software-triggered context switch puts a thread explicitly to sleep if it cannot make forward progress. The input thread goes to sleep if the SIB is full or if the FEF is empty. Similarly, the output thread goes to sleep if the SUB is empty or if the BEF is full. The application thread goes to sleep if the SIB is empty, which means there is no record to process. Notice that the application thread will *not* go to sleep if the SUB is full. Instead, the result of that record will simply be ignored and thrown away.

4.5 Other Optimizations

4.5.1 Data Locality Optimizations

As discussed in Section 2.2, ORAM may become a performance bottleneck in Stream-Ascend. Thus it is desirable to minimize off-chip traffic. We try to achieve this goal by explicitly managing memory allocation for maximum locality instead of using system calls (e.g., $malloc()$ and $free()$).

In order to improve locality and reduce off-chip traffic, we organize both the SIB and SUB as stacks (as opposed to FIFOs). That is, when a new record is buffered and added to the software buffer, 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 buffers 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 both the SIB and SUB are manipulated by two threads, stack push/pop operations can cause race conditions. In our design, we use a lock-free stack design. Specifically, we always push elements to the top of the stack, and pop the second from top element from the stack.

4.5.2 Working Set Usually Fits in Cache

In practice, the working set size in order to process a single data record is usually very small. For example, the maximum packet size in IPv4 is 64 KB and each twitter feed is at most 140 characters. For these applications, the working set usually fits into the on-chip caches, as illustrated in Figure 4.

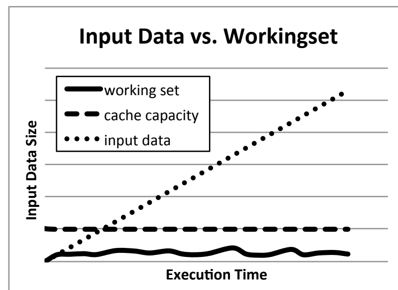


Figure 4: Input data size vs. working set size over time for the DocDist benchmark (Section 6.2.1).

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. If the working set fits in on-chip cache, ORAM will be rarely accessed. In this case, the ORAM latency bottleneck is largely avoided.

5. CASE STUDIES

In this section, we will analyze two typical application scenarios of Stream-Ascend: *data filtering* and *large dataset processing*. The scheduling algorithms discussed in this section are simple for illustrative purposes, and are not necessarily the most efficient.

5.1 Data Filtering

Data Filtering is a type of application that reads input data from an unceasing input stream (network packets, user input, online news feeds, etc.) and constantly sends outputs to the user. Applications that fit in this model include packet processing in network routers, email filtering, online news feeds processing, user inputs, etc.

The *input scheduler* of this application can be very simple: data records are fed into Stream-Ascend as soon as they arrive. This is obviously a secure scheduling since the output of the scheduler only depends on when data arrives, which is certainly public information.

For the *output scheduler*, the server may simply decide to send an output request to Stream-Ascend asking for a result of size S' after time T' when the data record is fed into Stream-Ascend. The assumptions here are that first, the result size is strictly smaller than S' for each record processing, and second, each record processing time is strictly

smaller than T'. These assumptions are true for many data filtering applications.

5.1.1 Parameter Determination

In the above example, no parameter needs to be determined for *input scheduler*. For *output scheduler*, however, we need to determine two parameters: S' and T'.

The output size S' is relatively easy for the user to learn, since the user knows the output format of the program.

The processing time T' is a little bit harder to determine. Though the user may know the code of the program, it is still difficult to know the execution time of the program on a particular hardware platform — Stream-Ascend.

We can set up a pre-computation phase using Batch-Ascend (which can be viewed as Stream-Ascend with special input/output schedulers) to handle this problem. In the pre-computation phase, the user sends the program with example inputs (e.g., worst-case input) to the Batch-Ascend, which runs the program with all the example inputs and returns the execution time back to the user after fixed time T. The user can determine T' based on the results.

5.2 Large Dataset Processing

Large dataset processing is a type of application that extracts useful information from a large database. Examples of this application range from very simple SQL queries to complicated queries like Content-Based Image Retrieval (CBIR). Stream-Ascend will request all the data records from the database in a streaming manner, process each of them and return the final results back to the client.

Notice that Stream-Ascend can only efficiently support SQL queries that fit in the streaming model, e.g., sequential scan of a table. Queries like *index access* might be very inefficient since the system needs to wait for the data records to be streamed in.

The *input scheduler* reads data from disks and feeds them to Stream-Ascend. For simplicity, we assume the *input scheduler* to be a simple *periodic* scheduler, which sends a word (this can be of arbitrary size and vary in size) to Stream-Ascend every $STREAM_{int}$ cycles. Note that the value of $STREAM_{int}$ is critical to the Quality-of-Service (QoS): if it is set too small, records are streamed in too fast and will be dropped due to FEF overflow (c.f., Section 4.3); if it is set too large, the processor will be idle waiting for input most of the time. The determination of $STREAM_{int}$ is discussed in Section 5.2.1.

We can choose the *output scheduler* to be simply requesting the final results of size S after the last data record in the database is sent to Stream-Ascend, plus a time delay L (to finish the last several records in the SIB and FEF). S is easy to learn by the user and L can be chosen conservatively since its overhead is only paid once for the whole execution.

5.2.1 $STREAM_{int}$ Determination

Similar to Section 5.1.1, pre-computation with Batch-Ascend is used to estimate the value of $STREAM_{int}$.

In the pre-computation phase, the server will first generate a set of sampled records from the entire input data. These records are fed into the Batch-Ascend to compute the average time T_{word} to process each word in the input. T_{word} is then returned to the user who determines $STREAM_{int}$ based on T_{word}.

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	108 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

6. EVALUATION

In this section, we focus on the *large dataset processing* domain, and evaluate three applications (document matching, DNA sequence matching and content-based image retrieval) on Stream-Ascend and a baseline insecure coprocessor. For simplicity, we focus on the scheduling algorithms discussed in Section 5.2. But most of the results here also apply to other scheduling algorithms.

6.1 Methodology

All experiments are carried out with a cycle-level simulator based on the public domain SESC [25] 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 dataset is streamed through the simulator (which takes between 4 billion to 100 billion instructions depending on the benchmark and the dataset).

6.1.1 Comparison Points

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

Stream-Ascend:

The Stream-Ascend proposal described in Section 4 with the scheduling algorithms discussed in Section 5.2. Perfect schedulers are assumed for end-to-end evaluation. $STREAM_{int}$ and SIB capacity are varied throughout the evaluation.

Oracle-Ascend: An idealized Stream-Ascend design that implements the SIB in magic hardware. Accessing a word in the magic SIB costs a single cycle always. Compared to Stream-Ascend, this system removes the overhead of *input thread*.

Oracle-Baseline: An *idealized* insecure processor with DRAM as main memory. Similar to Oracle-Ascend, it also has a magic SIB. Oracle-Baseline stands for the performance upper bound which is not achievable in practice.

6.2 Application Case Studies

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

6.2.1 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).

6.2.2 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 (the algorithm also works for general strings). We use DNA sequences from human and chimp chromosomes (6 million nucleotides in total, randomly broken into segments of length from 1K to 10K).

6.2.3 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 [21] 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 [28, 9]. 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 [10]. We were constrained to this number and size of images to keep simulation time reasonable.

6.3 Performance Study

In this section, we evaluate Stream-Ascend under different parameter settings. For each experiment, the input data is at least several megabytes (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 megabyte: *much* smaller than input data size.

6.3.1 Software Buffer Size

Figure 5 shows the drop rate of Stream-Ascend with different SIB (cf. Section 4.2) sizes and $STREAM_{int}$. Given a certain SIB size (which corresponds to a single curve in the figure), Stream-Ascend starts to drop records when $STREAM_{int}$ is smaller than a threshold value, which we define as $THRESH_{int}$. For all benchmarks, the $THRESH_{int}$ decreases for larger SIB. This is because larger buffers can tolerate more variance in the processing speed, so $STREAM_{int}$ can be set closer to the optimal value. There are diminishing returns after the software buffer is large enough.

When $STREAM_{int}$ is smaller than the $THRESH_{int}$, 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, the $THRESH_{int}$ 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.

Benchmark	Oracle-Ascend	Stream-Ascend (optimal scheduling)
Doc Dist	<0.1%	24.5%
DNA	<0.1%	0.7%
Img	2.6%	3.9%

Table 2: Performance degradation of Stream-Ascend and Oracle-Ascend with respect to Oracle-Baseline.

6.3.2 Infinite Software Buffer

To have a better understanding of the software buffer in Stream-Ascend, Figure 6 shows the performance of each application when the software buffer has an infinite size.[1] 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 will not drop records due to rate matching problems: there will always be space in the SIB 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 thread becomes the bottleneck of the system. Since each record is immediately processed after it is streamed in, the SIB is almost always empty. Thus, even if software buffer capacity is finite, Stream-Ascend still drops no records. This is the region in which Stream-Ascend should operate.

If $STREAM_{int} < THRESH_{int}$, performance stays almost constant for a large region. In this region, the application thread is the bottleneck of the system and the CPU is busy all the time.

$STREAM_{int} = THRESH_{int}$ is the sweet spot that we want to achieve. This corresponds to the perfect rate matching case where no records are dropped and the CPU is always busy processing. This may not be possible to achieve with a simple *periodic input scheduler* as we have been assuming (due to nonlinearity in record processing time). But more complicated scheduling algorithms may have performance closer to this optimal case.

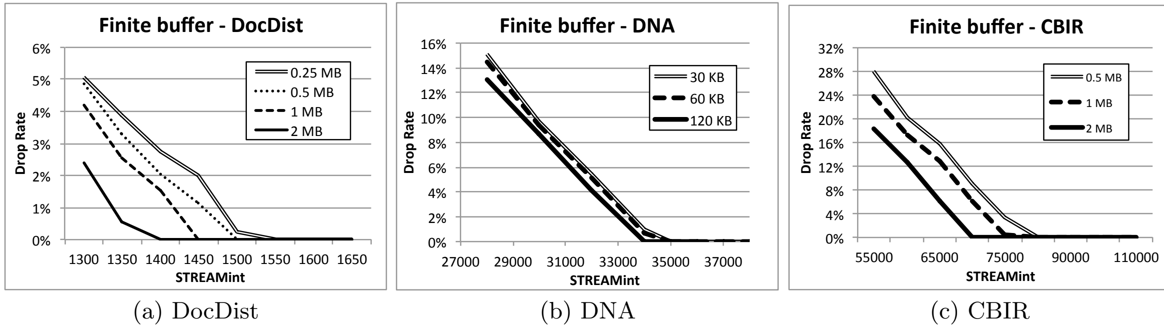
6.3.3 Overhead Analysis

Table 2 shows the end-to-end performance of Stream-Ascend and Oracle-Ascend, in terms of slowdown with respect to Oracle-Baseline. Instead of a *periodic input scheduler*, we assume an *optimal scheduler* which achieves *no record loss* and *100% CPU utilization* at the same time. How this scheduling can be achieved depends on the application and data running in the system.

The performance degradation of Oracle-Ascend compared to Oracle-Baseline shows the overhead of using ORAM. Since the working set mostly fits in cache, ORAM is rarely accessed with Oracle-Ascend (Oracle-Ascend rate matches perfectly), and overhead due to ORAM is small for all benchmarks.

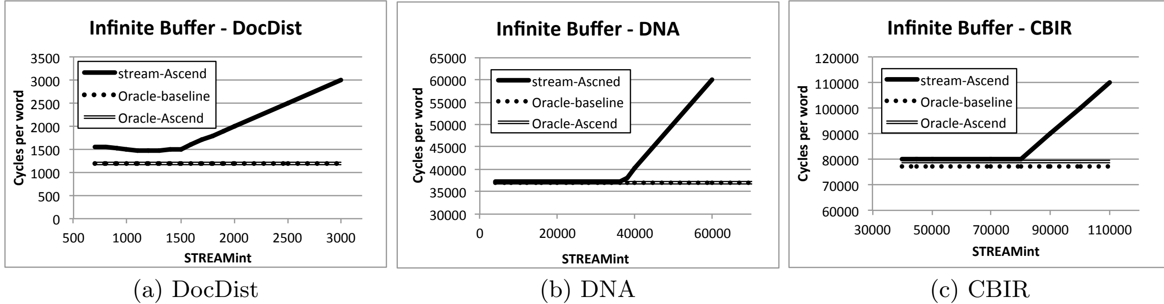
The overhead of Stream-Ascend comes from both using ORAM and having extra threads (*input thread* and *output thread*) running alongside the application thread. DocDist has the highest overhead due to the input thread because the

[1] That is, the software buffer size is larger than the input data size.



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

Figure 5: Drop rate vs. $STREAM_{int}$, sweeping software input buffer (SIB) size.



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

Figure 6: Performance (number of cycles to execute a single word) vs. $STREAM_{int}$ with infinite SIB size (drop rate = 0).

record processing program is so simple that the percentage of input thread's computation becomes larger.

6.3.4 *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 7 shows the performance change by sweeping input record size in the DocDist and DNA benchmarks, where the working set size is approximately the record size. When record size is smaller than on-chip cache capacity (1 MB), the 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 (accesses ORAM less frequently) 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 size. 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 7.

ORAM Latency

Figure 8 shows the performance of the DocDist benchmark for two different ORAMs. The ORAM design of [27] (denoted as slow ORAM in the figure) is compared to Path

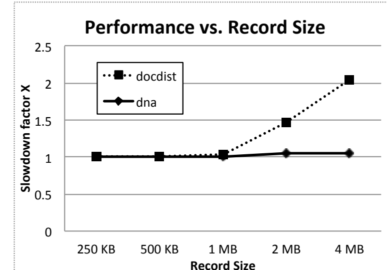


Figure 7: Performance of Stream-Ascend for different data record size, normalized to Oracle-Baseline. 1 MB cache size assumed.

ORAM. The access latency of the slow ORAM is 57708 cycles as compared to 3752 for Path ORAM.

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.

7. RELATED WORK

7.1 Private Information Retrieval (PIR)

Private Information Retrieval (PIR), first proposed in [8], considers a two party protocol where the *server* holds an n-

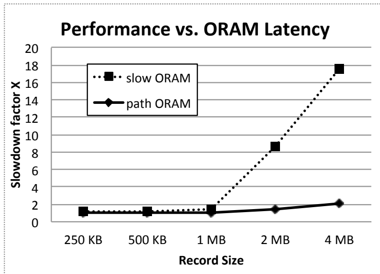


Figure 8: Performance of two different ORAM configurations running Docdist. Slow ORAM is described in [27].

bit string $B = b_1 b_2 ... b_n$ and the *user* specifies a secret input i. The user should only learn b_i while the server learns nothing about i.

Much previous work ([16, 7]) has investigated various aspects of PIR. Regardless, the PIR model is still very limited; it only supports requiring data items though indexes. Complicated query processing cannot be supported.

[16, 12] generalized PIR to support *keyword search* where the user provides a keyword and the server returns the data items that match this keyword. Still, this computation model is very limited.

7.2 Private Stream Searching

[23] is the first paper to propose *private stream searching.* [6] made optimizations to [23] and [5] evaluates the algorithm on real applications. Private stream searching reads in a stream of input data records, runs a query program on each of them and decides whether to keep the record (write to a buffer) or discard it. It employs a streaming model similar to Stream-Ascend. It also provides a more general computation model than PIR. However, existing schemes require the program to be expanded into a circuit so loops would be prohibitively expensive. This limits the generality of this model. Furthermore, existing work in private stream searching processes each record independently so aggregation of several data records are not possible (this can be achieved with Stream-Ascend).

7.3 Tamper-Resistant Hardware

Many tamper-resistant hardware schemes have been proposed and built in literature ([15, 19, 33, 11, 36, 29]). In this setting, the private user query is 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 Ascend ([11]), 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 off-chip memory access sequence of a query be independent of the query program's input data, and further, that the timing of memory requests is independent of the input data. The user has to check that the query program is written in such a way so as to not leak any information; most programs do not satisfy this property. A malicious query program can easily leak information about the query through memory traffic in [19, 33], for example.

The mechanisms proposed in this paper do not leak any information through pin traffic. Though the mechanism can be applied to any tamper-resistant hardware system, Ascend was chosen for demonstration since Ascend requires the least modification.

7.4 Secure Databases

Previous works on securing databases ([2, 4]) have proposed using trusted hardware to compute critical stages in query processing. [2] 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. A user cannot run complex data processing with the query. Second, though the data a certain query operates on is encrypted, the type of the query is leaked to the untrusted server.

Stream-Ascend, on the other hand, cannot efficiently support all kinds of database queries since not all of them fit in the streaming model (e.g., index access). But complicated data processing can be done in Stream-Ascend. Stream-Ascend is also able to hide the query type sent to the database which makes it more secure than previously-proposed secure databases.

Finally, we believe Stream-Ascend can be properly integrated with other secure databases to efficiently and securely support all types of queries. This is left for future work.

8. CONCLUSION

We comprehensively studied the secure ways to interact with a tamper-resistant hardware based system without leaking information through chip pins. The proposed mechanisms are implemented on top of Ascend [11]. The resulting architecture, Stream-Ascend, significantly generalized the types of applications that can run on Ascend with small performance overhead. In three typical benchmarks in the large dataset processing domain, we showed that with smart scheduling algorithms, the total performance overhead from our system is less than 24.5% relative to an idealized insecure baseline.

Acknowledgements

We thank Raluca Popa for her careful reading of an earlier version of this manuscript.

9. REFERENCES

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

[2] 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.

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

[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 constructions and practical applications for private stream searching (extended abstract). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 132–139, Washington, DC, USA, 2006. IEEE Computer Society.

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

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

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

[9] 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.

[10] 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*.

[11] 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.

[12] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography*, pages 303–324. Springer, 2005.

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

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

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

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

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

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

[19] 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* 9th *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[20] 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.

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

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

[23] 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.

[24] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the Int'l Symposium on Computer Architecture*, June 2013. Available at Cryptology ePrint Archive, Report 2012/76.

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

[26] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *Proceedings of the 1st STC'06*, Nov. 2006.

[27] 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.

[28] 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.

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

[30] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.

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

[32] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the ACM Computer and Communication Security Conference*, 2013.

[33] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17th *ICS (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[34] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32nd *ISCA'05*, New-York, June 2005. ACM.

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

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