Towards Tiny Trusted Third Parties

Dartmouth Computer Science Technical Report TR2005-547

Alexander Iliev Sean Smith

July 2005

Abstract

Many security protocols hypothesize the existence of a *trusted third party (TTP)* to ease handling of computation and data too sensitive for the other parties involved. Subsequent discussion usually dismisses these protocols as hypothetical or impractical, under the assumption that trusted third parties cannot exist. However, the last decade has seen the emergence of hardware-based devices that, to high assurance, can carry out computation unmolested; emerging research promises more. In theory, such devices can perform the role of a trusted third party in real-world problems.

In practice, we have found problems. The devices aspire to be general-purpose processors but are too small to accommodate real-world problem sizes. The small size forces programmers to hand-tune each algorithm anew, if possible, to fit inside the small space without losing security. This tuning heavily uses operations that general-purpose processors do not perform well. Furthermore, perhaps by trying to incorporate too much functionality, current devices are also too expensive to deploy widely.

Our current research attempts to overcome these barriers, by focusing on the effective use of *tiny* TTPs (*T3Ps*). To eliminate the programming obstacle, we used our experience building hardware TTP apps to design and prototype an efficient way to execute *arbitrary* programs on T3Ps while preserving the critical trust properties. To eliminate the performance and cost obstacles, we are currently examining the potential hardware design for a T3P optimized for these operations.

In previous papers, we reported our work on the programming obstacle. In this paper, we examine the potential hardware designs. We estimate that such a T3P could outperform existing devices by several orders of magnitude, while also having a gate-count of only 30K-60K, one to three orders of magnitude smaller than existing devices.

1 Introduction

Many distributed protocols consist of computation sensitive to two or more players, who are potentially adversarial and with different interests and motivations. To ensure that this computation possesses the desired security and privacy properties matching the interests of these participants, designers often posit the existence of a *trusted third* *party (TTP)*. If the main players trust this TTP not to cheat, then the desired security properties can be obtained simply by moving the sensitive computation to the TTP. This approach avoids the need for subtle and sometimes expensive cryptographic protocols to achieve the same security without a TTP.

However, many regard the use of TTPs in protocols as cheating, on the part of the designer. What organization or humans are sufficiently trustworthy? Even if an organization might be sufficiently trustworthy right now, what about the future? Consequently, although the addition of a TTP may make a design feasible from a computational perspective, it is considered akin to using magic or fairies [26]: not particularly appropriate for the real world.

Organizational TTPs Traditionally, researchers tend to imagine a TTP as some organization-based entity. In many applications, a third party can offer trust mostly on the basis of being in a different, presumably impartial, organization than the participants in the computation. Certification authorities usually fall into this category—a CA is trusted because, as an organization, it is impartial about the secure computations its certificates protect, and it must remain honest in order to maintain its reputation. However, there is no technological way in which a relying party can decide if a given certificate from a CA was actually generated properly.

This is presumably the kind of TTP that most people would not want to mediate their multiparty secure computations.

Hardware On the other hand, computing devices designed to be trustworthy for relying parties at remote locations have been designed and built. Such devices do actually offer a possible incarnation of a TTP which does not rely on organizational trust, but only on technological provisions, like strong physical security and self-authentication mechanisms. Such devices, often called *secure coprocessors* in the literature, provide secrecy and integrity for computation and data, against a wide variety of adversaries—even those with physical access.

Unfortunately, these trustworthy devices tend to be small in computational power and in memory. These limitations may even be inevitable; a TTP armored against an adversary with physical access might not have a large memory because of the difficulty of armoring a large device, nor run at high speed due to the difficulty of dissipating heat. Although limited in size and power, these devices tend to be large and expensive nonetheless; for example, the IBM 4758 housed an Intel 486-class processor and retailed for approximately $2,500.

These limitations make it hard to actually use hardware-based TTPs in the real world: how does one execute a large program with large data in a small box, in a way that does not reveal information about the execution to the adversary?

We have nonetheless developed and implemented algorithms for solving some specific large problems on hardware-based TTPs despite these size limitations. We have written a compiler that transforms arbitrary programs into operations—including the above algorithms—that can easily fit inside limited-power TTPs.

Throughout our prototyping work, we used the IBM 4758 TTP, and noticed that we overtaxed certain hardware capabilities, but barely used others; this experience suggests that building a *tiny TTP* optimized for these operations might both improve performance and lower cost.

This Paper This paper develops our vision for building tiny TTPs and using them to build practical solutions to real security problems.

- Section 2 presents some background in current hardware-based TTPs, particularly the IBM 4758.
- Section 3 reviews our experience using limited-power TTPs for *private information retrieval (PIR)*.
- Section 4 reviews how we compile arbitrary-size programs into circuits consisting of arithmetic and logic gates, and "array gates" (which essentially perform PIR), each of which we can handle with limited-power hardware-based TTPs. We have demonstrated that this solution is considerably more efficient than traditional scrambled-circuit-evaluation techniques for *secure multi-party computation (SMC)*.
- Section 5 discusses the basic hardware elements we envision for an optimized *tiny TTP*.
- Section 6 discusses how these elements might be assembled into a private execution engine.
- Section 7 estimates how the performance and gate-count for this T3P compares to prior hardware-based TTPs.
- Section 8 discusses related work.
- Section 9 concludes.

2 Current Hardware

The current incarnation of a hardware-based TTP is the *secure coprocessor*—a small general purpose computer armored to be secure against physical attack, such that code running on it has strong assurance of running unmolested and unobserved [33]. It also includes mechanisms, called *outbound authentication* (OA)[1] to prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [25]. The coprocessor is attached to a *host* computer, which provides storage, network and fast (but untrusted) CPU services to the HW TTP. The TTP is assumed to

[1]OA is more recently referred to as *attestation* in the context of the Trusted Computing Group.

M	size of database or array record
N	length of database or array
k	in square-root algorithm, how many accesses before a re-permute
g	number of record pairs each super-switch (g-switch) operates on
π	a (pseudo)random permutation on $[1..N]$

Table 1: Summary of parameters and notation used in this paper.

be trusted by clients (by virtue of all the above provisions), but the host is not trusted (not even its root user). The strongest adversary against the schemes presented here is the superuser on the host, who may also be equipped with a drill and logic analyzer.

IBM 4758 Secure Coprocessor

The 4758 is a commercially available device, validated to the highest level of software and physical security scrutiny then (around 1999) offered—FIPS 140-1 level 4 [28][2]. It has an Intel 486 processor at 99 MHz, 4MB of RAM and 4MB of FLASH memory. It connects to its host via the PCI bus.

The 4758 also has cryptographic acceleration hardware, and notably a "fast path" DES and TDES mode of operation, where data can be streamed from the host through the device's DES engine and back out without touching the internal RAM. The only operation possible in this mode however, is a single encrypt, decrypt or MAC computation. If anything more needs to be done, like decryption *and* MAC computation, or any data processing, the data needs to be read into the device's RAM, and can be processed only much more slowly than in the fast-path mode.

In production, the 4758 runs IBM's CP/Q++ embedded OS; however, experimental research devices can run a version of Linux (as does the follow-on product from IBM, the PCIXCC [1]). Linux has considerable advantages over CP/Q++ in terms of code portability and ease of development.

3 Using Limited-size Hardware TTPs for the PIR Problem

Our lab has designed and built a number of applications using hardware-based TTPs.

The solutions we have studied and implemented share some structure, which can be used to design more efficient hardware-based TTP architectures, much like graphics processing benefits from optimized hardware to deal with a small set of computationally intensive primitives. In this section we describe the problems and outline the structure of these solutions.

[2]FIPS 140-1 has been superseded by 140-2 since 2002, but the new standard does not provide any higher assurance levels [29].

3.1 Private Information Retrieval

One of the problems we have worked on is *private information retrieval (PIR)*: Boris has a database of items $X = \langle x_1, x_2, \ldots, x_N \rangle$ and Agnes selects an index $i \in [1..N]$. We want Agnes to learn x_i without Boris learning i.

After this problem had received much treatment in the theoretical literature (eg. [7, 8, 17]), Smith and Safford suggested the *practical PIR (PPIR)* variant: Agnes should need to do no more work than set up an SSL tunnel, ask for a record, and receive the response; no other parties should be involved; and the solution should provide reasonable performance on real-world dataset sizes [27].

Our work focuses on providing practical solutions with the assistance of a hardware TTP [13, 15]. The PPIR problem nicely embodies the challenge of a limited-size hardware TTP: if the TTP is big enough to store the entire database X, then the problem is easy; however, what if the TTP is too small?

We accepted this challenge, and assumed that the TTP is small: if the records are of size $\mathrm{O}(M)$, that the TTP can only fit $\mathrm{O}(M + \log N)$ bits—the minimum required to fit a constant number of records and indices. The whole database, and any additional working storage, resides on the TTP's host and is fully accessible to the adversary.

The basic algorithm we use was initially developed by Goldreich and Ostrovsky in their *oblivious RAM* work [11], and later also used by Asonov [2]. Figure 1 sketches the algorithm. First, the TTP generates a *shuffle*: a randomly permuted (and of course encrypted) version of the database. For example, suppose π is a random permutation on $\{1, .., N\}$ and E denotes encryption under a new random key. Then a shuffle of X can be denoted as $Y = \langle y_1, y_2, \ldots, y_N \rangle$, where each $y_i = E(x_{\pi^{-1}(i)})$. We denote such a shuffle under π as $\pi(X)$. During this process, the host's view of the TTP's I/O should be independent (or look independent when restricted to feasible computations) of the actual permutation π—the database permutation procedure must be *oblivious*.

After generating a shuffle, suppose the TTP has received and answered a sequence of requests $r_1, r_2, .., r_m$ from clients. How does the TTP service the next request, for r_{m+1}? The TTP could just fetch $\pi(X)[\pi(r_{m+1})]$ from the host. This would hide the actual identity of r_{m+1} from the host, who does not know π. However the host would still learn whether r_{m+1} is the same as or different than all the previous r_i's respectively. To hide these relationships, the TTP fetches all the previous r_i's. If r_{m+1} was not among them, the TTP then fetches it. If it was among them, the TTP fetches a random r_j which has not been fetched yet. What the host observes is that all the previous fetched records are fetched again, and then one more. It does not know which among all these is the actual current query.

Since the time to handle a request thus grows with the number of requests since the shuffle, each permuted database can only be used a limited number of times before a new one, using a new random permutation, needs to be generated.[3]

[3]This is a simplified exposition, we refer the reader to [13, 15] for the full details, and some extensions from the original prototype.

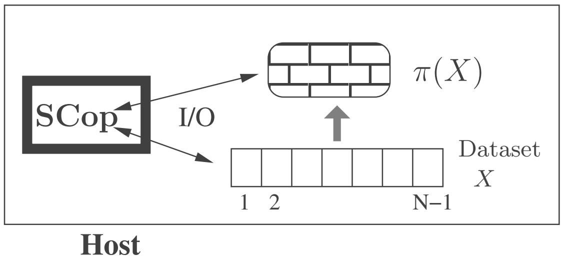

Figure 1: The TTP randomly encrypts and permutes the dataset X to give $\pi(X)$, reading in only a few items at a time, and hiding π from the host.

In subsequent work, we extended our solution to allow Agnes to change the value of x_i.

The most time-consuming part of this scheme is permuting the database obliviously while using only very little trusted storage. It can be achieved in $O(N \log^2 N)$ time using Batcher's bitonic sorting network [3]. Such networks are an important theme here–we elaborate on them in the next section. Using a Beneš network would reduce the time to $O(N \log N)$, but, to current knowledge, it requires $O(N \log N)$ trusted space to calculate the network's switch settings for a given π.

Since the permutation step is the most time consuming, it makes sense to try to optimize it, in the form of a *tiny TTP*, described in the following sections.

To give an idea of the performance of our prototype, Table 2 shows the approximate quality of service for different dataset sizes.

N	Response Time
1024	5.5
2048	5.5
4096	8.7
8192	18.5

Table 2: Maximum query response times (in seconds) attainable by current PPIR prototype, with different sizes of datasets. Record size is 850 bytes. As the dataset grows larger, the time to re-permute it periodically dominates the overhead of re-fetching records. We had set the number of queries before re-permutation to 200.

3.2 Oblivious Networks

A common theme in our work was the need for the TTP to perform operations *obliviously* on a sequence X of N items residing on the host. Operations include permuting with a given permutation π, sorting with a given sorting key $k(x_i)$, and merging two sorted subsequences. Our non-standard requirements are:

Obliviousness the adversary (bounded to feasible computations) does not learn anything about the parameters of the operation from observing the IO that the TTP performs. Equivalently, the IO sequence that the host observes is independent of the operation parameters[4].

Tiny secure memory The TTP has space to read in and operate on only a constant number of records and indices at a time.

From other fields of computer science (eg. communication routing) comes the construct of *oblivious networks* which can perform any of these tasks. These networks consists of small operators whose settings are operation-dependent (eg. depend on the permutation π), wired together in a fixed manner, independent of the operation.

Example: Permutation network A Beneš permutation network can perform any permutation π of N input items by passing them through $N \log N$ crossbar switches (the operators), each of which operates on two items, either crossing them or passing them straight. The crossbar settings differ for different π, but the connections between the switches is fixed for a given input size N. In Figure 2 we illustrate a small Beneš network.

A *bitonic sorting network* is similar to the Beneš network, but it sorts its input, and consists of $\frac{N}{4} \log^2 N$ *comparators*, each of which sorts two items. The comparators are arranged in $\frac{\log^2 N}{2}$ *stages.*

These networks provide a way for the TTP to perform sorting and permutation obliviously. For example, in the Beneš network case, to execute a switch the TTP reads in the two items involved, internally crosses them or not, and writes them out re-encrypted (using a new pseudorandom IV) so the host cannot tell if it was a cross or not. We call this implementation of a switch an *encrypted switch.*

In general, oblivious networks work for our problem because (1) the TTP can use cryptography to make each operator look the same (ie. independent of its setting) to the host, and (2) the network wiring is intrinsically independent of the inputs. Thus the TTP can emulate the whole network on any dataset while keeping the host's view independent of the dataset and the secret parameters (like π).

4 Using Limited-size Hardware TTPs for Arbitrary Problems

Suppose we wanted to execute a program with the security and privacy properties a TTP provides (as defined in Section 4.1). If a TTP were big enough, we could just run the entire program inside there. If the TTP is small, what can we do? In Section 3, we discussed how, with much work, we could fit the PIR problem for large datasets into

[4]For an adversary bounded to feasible computation, the requirement is that the adversary has a negligible success probability in distinguishing an actual TTP I/O pattern from a randomly generated one, informally speaking.

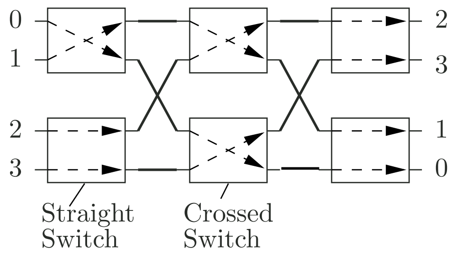

Figure 2: A Beneš permutation network with 4 inputs, performing the permutation $\langle 2, 3, 1, 0 \rangle$. The dashed lines represent switch settings, which depend on the permutation. The rest of the network only depends on the number of inputs.

a small TTP without sacrificing the security properties. Can we extend this result to arbitrary programs?

We addressed this more general problem by looking at secure multi-party computation (SMC) (also known as *secure function evaluation*—SFE). Restricted to two parties, the SFE problem asks how Agnes and Boris can evaluate a function f, with 2 inputs and 2 outputs, on their respective private inputs x_A and x_B, such that each only learns their own partial result.

An ideal solution to this problem is that Agnes and Boris send their inputs to an ideal TTP, which computes f and distributes the results. In practice most current work on SFE declares that such an ideal TTP does not exist, so Agnes and Boris need to achieve a similar result by themselves. Most protocols to achieve this make use of a *scrambled* or *blinded* circuit representation of f, as introduced by Yao [32]. One of the players blinds the circuit, and the other evaluates the blinded circuit, without knowing what actual values the blinded values he is computing represent. Recently a two-party SFE prototype, Fairplay, has been implemented. [21].

One major weakness of the blinded circuit approach to SFE is that the use of indirectly-addressed arrays is very expensive—accessing an N-element array requires $O(N)$ gates, and hence $O(N)$ time. Observing that array access in the SFE setting is very similar to PIR (i.e,. should leak no information about the value *or* index), we started a project to use a hardware TTP to accelerate SFE protocols. Our current progress is reported in [14]. We have built a new compiler for Fairplay's *Secure Function Definition Language (SFDL)*, which produces an arithmetic circuit augmented with *array access gates*.

Then we used a 4758 to evaluate the circuit. (We pretended the 4758 had more limited memory than it does, so that our results might be meaningful for future T3Ps.) The circuit is stored as an array of gates, sorted in topological order. Each gate has a corresponding *value slot*, which is where the gate's output value is stored (encrypted). The TTP writes the players' input values into the value slots of *input gates*, then it evaluates the circuit gates one by one, filling in their value slots. Array access gates are handled using PPIR, so that the adversary on the TTP's host (which is where the array is stored) learns nothing about the array access pattern.

This early prototype shows a large improvement in performance over Fairplay (running on 2.7 GHz Intel Xeon processors with 4GB RAM and gigabit ethernet connection) for array-intensive code, and similar performance as Fairplay for scalar code.

4.1 Properties of a "TTP application"

The security requirements of an SFE protocol are defined in Section 4—the inputs and outputs of the computation have clear "owners", and only the owner of a datum should learn it. This definition covers many useful scenarios, but we see several useful extensions in defining what a TTP application does and what the security requirements are.

- **Servers** The standard SFE scenario specifies a one-time computation. Our PIR prototype is structured as a server which responds to queries interactively. It cannot be structured as a one-time computation, because of the expensive preprocessing (permutation) of the dataset, which needs to be amortized across many queries. This structure can be accomodated by specifying a circuit which the TTP executes in response to a stimulus, like a PIR query.

  The security requirement would be similar to that of one-time SFE—inputs and outputs are only learned by their owners.

- **Less sensitive data** Our TTP could support the notion of less sensitive data, as specified in the function f and its circuit form [5]. Such data may not need to be accessed obliviously, or even need to be encrypted.

5 Optimizing Hardware Bottlenecks

As we have discussed in the previous sections, the bottlenecks of our TTP-based PPIR and SFE schemes are the permutation and sorting and merging networks, and their essential component is the encrypted switch. Logically, the switch does the following:

- read in two encrypted records, check integrity, labeling, etc.;
- depending on the control bit, switch them or not;
- send them back out again, with new labelling, and the contents re-encrypted.

In practice, re-encryption consists of decrypting and then encrypting with a pseudorandom IV.

Also as mentioned before, we used the 4758 for our prototypes. Although the 4758 had support for quickly streaming data through its internal symmetric crypto engine (see

[5] In this work we do not discuss the problems associated with writing secure code, in SFDL. If the SFDL programmer introduces a way for sensitive data to flow to an output where it should not, the TTP circuit evaluator has no way to detect this error. Sufficiently complex functions may be verified with explicit information flow control tools (see [24] for an overview).

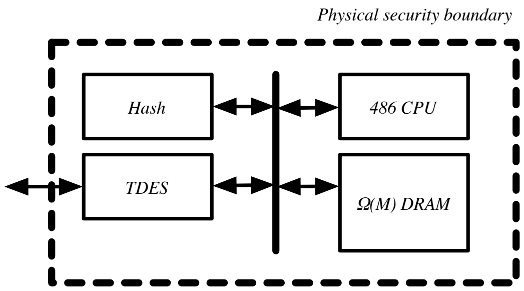

Figure 3: In our previous prototypes, we used the IBM 4758 platform, which let us bring records in or out through the TDES engine (which was much faster than using an OpenSSL software implementation of TDES on the 486 CPU), but required storing the records internally in RAM for integrity checking and switching.

[20] for a discussion), it only had one engine, and had no support for simultaneous encryption and integrity checking, not to mention conditional switching without involving the CPU and main RAM. These limitations forced us to implement encrypted switches by bringing both of the two records into the internal memory, then running integrity checking on each, then writing each out again. This usage of the 4758's internal CPU and RAM slowed performance; requiring storage for two plaintext records led to an $O(M)$ internal memory requirement. Figure 3 sketches this architecture.

What we needed for an encrypted switch did not match the hardware provisions of the 4758—its I/O channels and TDES engine could not be utilized in the fast-path mode, the CPU was not fully used, and we did not use much RAM. This situation suggests designing a new type of hardware, optimized for our purposes.

5.1 The Symmetric Engine

A basic building block of this hardware is an engine for fast symmetric cryptography. Our 4758 prototype work used the TDES cipher; new-generation hardware should probably use AES.

The designs here are mostly concerned with streaming data through the device, but internally most of the streaming data will be chunked into AES blocks, and processed as those blocks.

Besides streaming the data through the cipher, we also want the engine to check the integrity of the record. We see several ways of doing that.

- Encrypt in CBC mode, compute a CBC-MAC on the ciphertext, and concatenate the MAC to the ciphertext. This is the standard and generically secure composition[6] of encryption and MAC [16].

[6] *Generically secure composition* here means that if one has any secure encryption scheme and any secure authentication scheme (eg. MAC), then combining/composing them in the order of encrypt-then-autheticate is guaranteed to be secure.

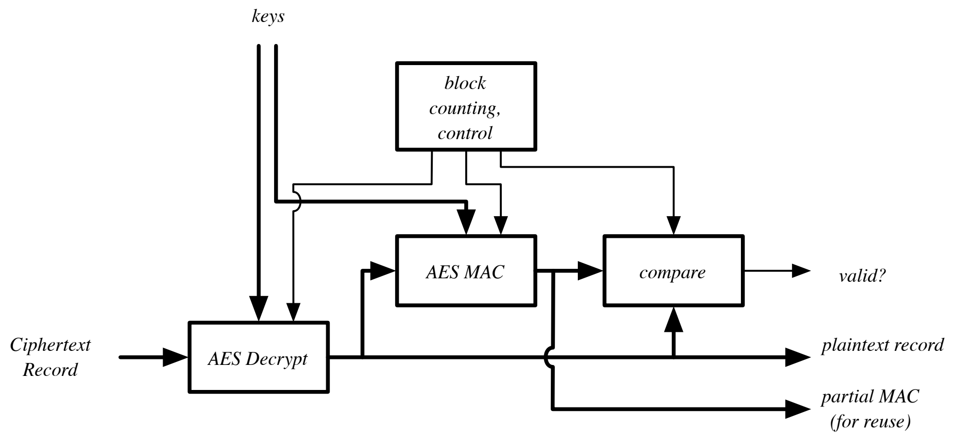

Figure 4: This sketch shows an example input-line symmetric crypto engine, a building block of our T3Ps. Block by block, we bring the data in, decrypt it, and calculate its MAC.

- Compute a CBC-MAC on the plaintext, concatenate the MAC to the plaintext, and encrypt in CBC mode. This order of MAC and encryption is not generically secure, but *is* secure when using CBC mode for encryption. It has the advantage that most of the MAC work can be re-used across re-encryptions.
- The newer GCM mode of *authenticated encryption with associated data* (AEAD) fits our needs precisely, as it efficiently handles integrity for an unencrypted header, it allows both MAC and decryption to be computed in arbitrary block order, it was designed with performance in mind, and it carries no patents [22]. It is however less vetted at this point (though proved secure).

Figure 4 sketches an example implementation of the CBC-mac-based approach.

We also need a similar engine to *add* the integrity check and encrypt.

5.2 A Simple Switch

We can then use four symmetric engines (two inbound, and two outbound) to build an encrypted switch, as sketched in Figure 5. We do not need to store the complete records inside the switch, but instead stream the two records through the switch, crossing the streams if needed. Thus we reduce the internal memory requirement from O(M) to O(B) (where B is the block size of the cipher).

5.3 A g-in, g-out Switch

In previous 4758 work, the limiting factor in streaming data from the host, through the TDES engine, and back out again was the speed of I/O, not the crypto hardware itself [20]. This observation suggests that, at first approximation, the time complexity

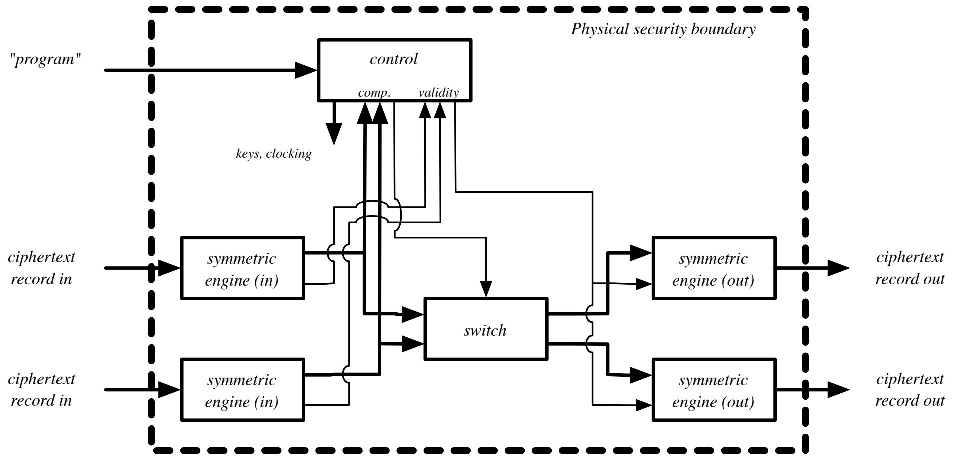

Figure 5: The sketch shows an example 2-switch T3P. Block by block, we bring in two records, decrypt them, potentially swap them, then encrypt them again and send them out.

associated with executing a network of encrypted switches follows from the number of switches, and less so from the size of each switch.

Thus the question naturally arises: can we decompose a network into a smaller number of larger switches?

Without loss of generality, let us consider the basic reverse butterfly network on N lines, where N is a power of 2. (Our re-shuffle engine uses a bitonic merging network and a Beneš network, but both of these are built from butterfly networks and their reversals.) This network is N-lines wide, and $\log N$ lines deep.

Let g be a power of 2. To save rounds with the host, we could bring in g lines into a larger switch instead of just 2, and do the work of $\frac{g}{2}$ switches at once without interacting with the host.

However, we can exploit the structure of these networks to coalesce not just $g/2$ switches within one column, but $(g/2)\log g$ switches across $\log g$ columns, into a g-in, g-out super-switch, referred to as a g-switch (in the limit, if $g = N$, we get just one "N-switch"—the whole network).

Previous work on doing FFTs on parallel machines also uses this property of butterfly networks to increase locality [9].

We can see this decomposition by looking at the butterfly network. For binary strings x and y, let $x \parallel y$ denote their concatenation, and let $\langle x \rangle$ denote the integer whose binary representation is x. For N a power of 2, the network can be recursively defined as:

- a *stage* consisting of $\frac{N}{2}$ switches on $\langle 0x \rangle$ and $\langle 1x \rangle$ (for each of $N/2$ x's)
- followed by a reverse butterfly network on the $\frac{N}{2}$ lines $\langle 0x \rangle$, and another on the $\frac{N}{2}$ lines $\langle 1x \rangle$.

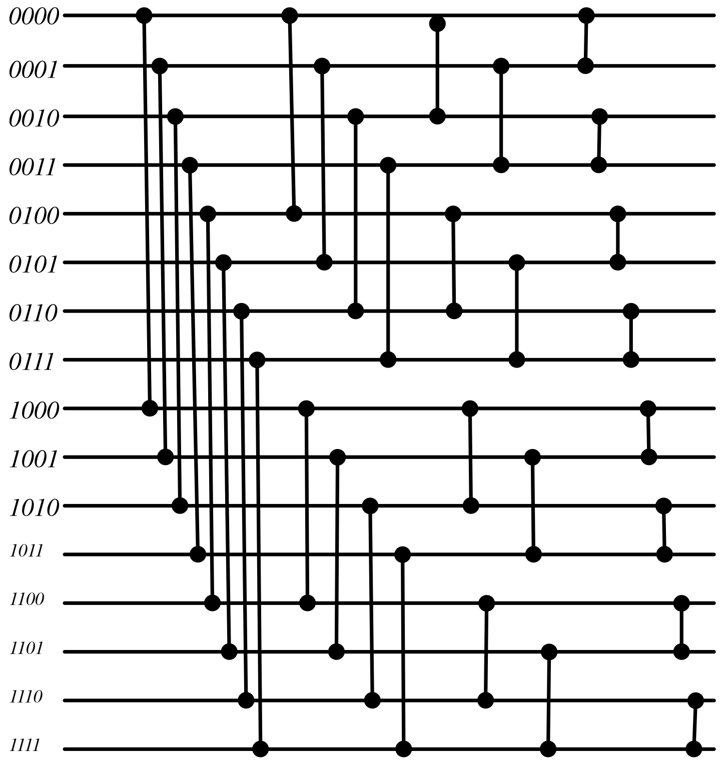

Figure 6: This sketch shows an example reverse butterfly network for $N = 16$ inputs. The vertical lines denote switches on the two endpoint lines, and we write the input line indices in binary. This network has 32 2-input switches, arranged in 4 stages.

Figure 6 shows the network for $N = 16$.

Lemma. *In the first s stages of a reverse butterfly network, two lines are compared only if their indices are of the form $\langle x \parallel y\rangle$ and $\langle x' \parallel y\rangle$, where x and x' are length-s binary sequences differing in exactly one bit.*

Proof. For $s = 1$, this is clearly true, by the definition of the network.

For $s > 1$, suppose it's true for $s - 1$. The network consists of a stage, followed by two isomorphic subnetworks.

The leading stage only compares $\langle 0 \parallel x\rangle$ to $\langle 1 \parallel x\rangle$, ie. the indices differ in the first bit only.

By inductive assumption, we know that the first $s - 1$ stages in the two subnets only do these types of comparisons:

- $\langle 0 \parallel x \parallel y\rangle$ to $\langle 0 \parallel x' \parallel y\rangle$, and
- $\langle 1 \parallel x \parallel y\rangle$ to $\langle 1 \parallel x' \parallel y\rangle$

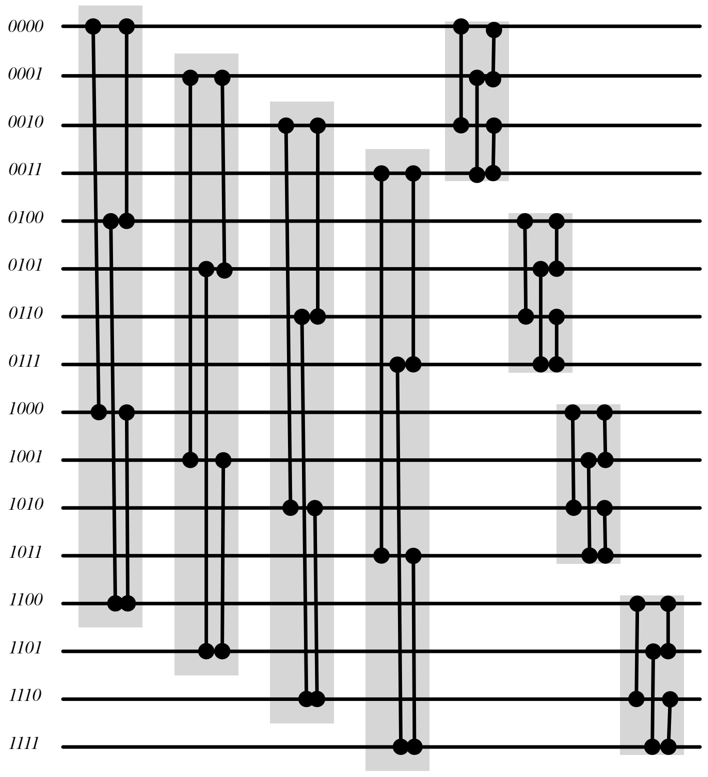

Figure 7: This sketch shows how the the $N = 16$ reverse butterfly network can instead be implemented as 8 4-switches, each performing the action of 4 2-switches.

(where x and x' are binary sequences of length $s - 1$ differing in exactly one bit).

This covers all switches in the first s stages of the overall network. □

Let $g = 2^s$, and, for the first s stages of the network, designate a g-switch G_y to cover all g lines which have their last $n - s$ bits equal to y. By the lemma above, there are no switches between any G_y's, so all the G_y's are disjoint, as needed.

g-switches are recursively assigned in the 2^s sub-networks (which are of course disjoint) after the first s stages.

If s does not divide n, the last stage of super-switches will need to be smaller than g.

Figure 7 shows an example decomposition into 4-switches.

5.4 Circuit Gates

Since our T3P will be used to evaluate arithmetic circuits with array-gates, it will need an arithmetic logic unit (ALU) to evaluate the arithmetic and logic gates, and a controller to control PIR into arrays for array gates.

Most of the logic and arithmetic operations are on integers, so we need to decide what the native word size will be, and require the compiler or user to deal with larger integers itself.

The ALU will be pretty similar to the switch—1, 2 or 3 inputs come in and one comes out. The same crypto and index checks will need to be done as for the switch, thus the same crypto cores and mostly the same control logic can be used. This ALU will be used much less than the switch, at least in array-intensive code. Thus, it can probably be a fairly simple implementation, without pipelining and other optimizations.

6 Building a Privacy Engine

The next step is considering how to put together one or more g-switches into a private execution engine.

We put one or more g-switches on a module along with some control and I/O (PCI etc.), and attach this package to a host computer. However, this raises some design questions:

- Where is the physical security boundary—around one switch, some set of switches if is are more than one, or the entire module?
- What size switch (g) should we use?
- How many should we use?
- How do we execute a network (such as a run of a Beneš or bitonic merge network) using these switches, while preserving privacy?

As our subsequent analysis shows, if we have spare gates and I/O speed, it may be better to grow g rather than add more switches.

6.1 Using the Encrypted Switch to Execute a Network

Coordination of the switches within a run of a network requires considering how to set up the encryption keys, IVs, and record headers to ensure that the switches can detect if the host is feeding them the wrong ciphertexts, but the host cannot determine which ciphertext output of a switch matches which ciphertext input. Since key scheduling for a block cipher incurs a non-trivial cost, we'll sketch an example approach that uses the same key within each network run.

The cryptographic assumption we use here is that using a strong block cipher in CBC mode with (pseudo)random IVs is IND-CPA secure, meaning that an adversary cannot distinguish the ciphertexts of two plaintexts picked by him. This is proven in [5].

We'll start with some parameters:

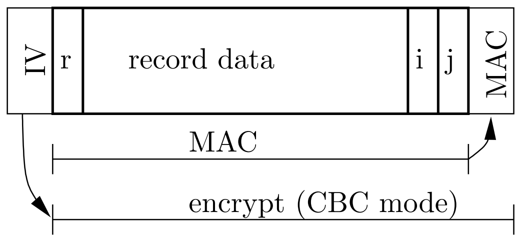

Figure 8: Structure of an encrypted record during an oblivious network execution. The record's number is r, its next destination switch is i, and its input line of that switch is j. Note that i and j change every time the record passes through a switch, but the rest of the plaintext stays fixed, so we do not need to re-compute most of the MAC after every switch.

- Let k be the key for this run.
- Let $0, ..., G-1$ be a topological ordering of the switches.
- Let *Succ* be a successor function that takes (i_0, j_0) to (i_1, j_1) when output line j_0 from switch i_0 goes to input line j_1 at switch i_1.

Each g-switch i needs to be able to compute *Succ* and a PRNG (or have access to a module that can). Special hardware will probably be needed to seed the PRNG with sufficient randomness when needed. Switch i also needs to know i and k. If we're doing a permutation instead of a merger of sorted lists, then switch i also needs to know its program $P(i)$: the local permutation it is to carry out.

Ordinary switch execution Switch i reads in the IV for the record on each input line, If we're sorting, the switch needs to read in the record number from each header and then set up its local permutation; if we're permuting, the switch needs to read in its programming and set up its local permutation. The switches uses its PRNG to set up IVs at each output line; for each line j, the switch sets up a header consisting of $Succ(i, j)$ and the record number mapping to that line. The switch then, in lockstep, brings the rest of the two input ciphertexts through the input symmetric engines, and pumps the output plaintexts out through the output symmetric engines. At the end, the switch checks whether the integrity checks on any of the input lines have failed, and and whether the footer for each line j is indeed marked with (i, j).

In Figure 8 we show a diagram of an encrypted record during execution of a network.

Failure Handling To avoid revealing information to an adversarial host, the switch should ensure that all failure cases are detected as soon as they are introduced by the adversary. This is ensured by the integrity protection and record labelling described above. Within a single switch execution, a detected error could be reported immediately or at the end of that switch operation (presumably by aborting the network execution).

Input and Output Processing The above simple sketch applies to input records which were already encrypted with the key k and marked with the appropriate switch lines, and left output records in a similar state. The preparation of the encrypted records is part of the overall SFE procedure outlined in Section 6.4.

6.2 Physical Security for the Switches

One physical design for a complete TTP is to have a single physically shielded compartment containing one or more switches and their controller. This is the easiest option to work with in designing how the controller and switches communicate.

Alternatively, the switches and controller could be protected separately, with untrusted interconnects. In this case, the components will need to set up secure channels between each other, as well as with the outside. This suggests the need for public-key cryptography (PKC) capability for all the components.

Clever use of longer-lived symmetric keys can reduce the need for PKC, if that becomes too onerous. For example, in the basic model of the 4758 TTP, each device leaves the factory with a private key only it knows, and a certificate testifying that the device knowing that private key is a genuine untampered 4758. An alternative model is to have a master device and several slave devices leaving the factory as a family; each slave has a device-specific secret known only by itself and its master; the master has PKC ability and a private key. In our TTP design, the master component would of course be the controller, and it would maintain shared secrets with the switches.

Besides the trusted controller (which, for now, we also place in a physically secure module), we also need a request engine with $O(k \log N)$ storage, to do the fetching of the k records. (In previous work, we considered but dismissed as impractical the recursive approach, where request engine does PIR on the "touched" records instead of bringing them all in each time. However, this new hardware setting may change things. We plan to revisit this issue.)

6.3 Multiplexing the Symmetric Core

The straightforward way of building a g-switch is to follow the schema of Figure 5. For example, if we're using AES for symmetric cryptography and a CBC-MAC for integrity, then we use two AES cores for each input line, and two more for each output line. However, AES cores are rather large, and this approach forces us to have $2g$ of them.

Alternatively, suppose we restrict a core to doing just block permutations with a fixed key for a network execution. (That is, we do the CBC chaining manually, outside the core.) We can then multiplex the core in as needed. For example, the switch can wait for the host to write a block of data to input line j. Then, while the host is writing the next line, the switch can do the following:

- save a copy of the block for the next chaining,

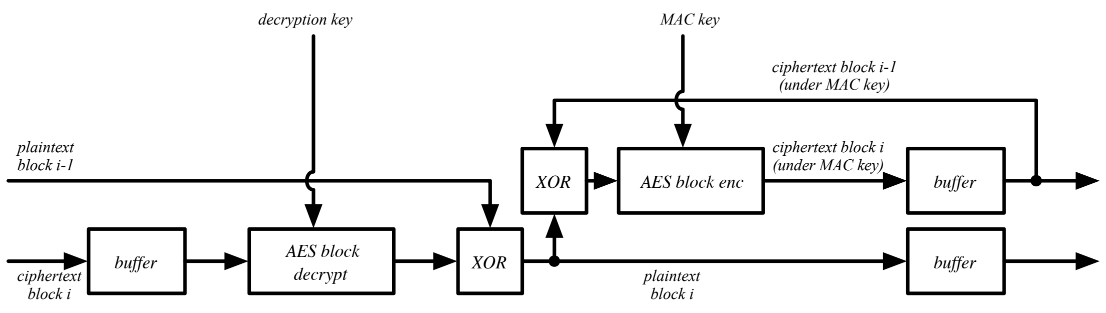

Figure 9: This sketches the symmetric processing for an input line (using CBC-MAC for integrity checking), with the AES block permutations isolated, so we can multiplex them.

- feed it through the cipher
- XOR in the chaining block
- save a copy for output
- XOR in a saved block, for CBC chaining for the MAC
- feed it through the MAC cipher
- save the output

This might be simpler if we had a separate core for the decryption and for the MAC (particularly if we decide to do hashing instead). Figure 9 and Figure 10 sketch this approach. (However, we have many design options here with regard to what portion of the operation we multiplex.)

If we are clever about record format (e.g., putting the sanity-check (i, j) headers at the end instead of the beginning), then most of the MAC work can be re-used for the output.

This analysis suggests that we can we make do with two cores, and replicate them only if they cannot keep up with the input speed.

Eventually, we may want to look at larger buffers on the input and output lines as well.

6.4 A whole scenario walkthrough

We have so far focused on the most performance-critical component in a tiny TTP, the encrypted switch, and how it is used to perform an oblivious network. However there are many things that the TTP needs to do around the switch too. Here we outline a more holisitc view of a whole SFE session. We assume the TTP is on a third computer, but it could also be co-located with one of the players.

- One player generates the circuit, and both players sign it after they agree it's correct.

- Both players establish secure sessions with the TTP. During this, the TTP gets their public keys too.
- Each player sends its input to the TTP, which writes scalars into the appropriate circuit value slots (see Section 4), and writes arrays encrypted with an initial key onto the host.
- The TTP permutes the arrays.
- The TTP starts the circuit evaluation. It computes a hash of the circuit gates during the process, and checks the players' signatures on the circuit at the end.
- It stores each output value encrypted on the host and distributes outputs at the end.

Each of those steps can be initiated by the host, or the TTP could run through all of them, blocking while waiting for input.

6.5 Concurrent PPIR sessions

The TTP will almost certainly have to deal with multiple arrays, or copies of the same array due to conditionals, which means it has to flush and restore PPIR contexts (keys, current shuffle number, current access number, etc.). This should not be a large overhead, as the TTP will at least do one retrieval ($O(k)$) with a PPIR context, not to mention re-permutes and multiple adjacent accesses to one array.

7 Preliminary Performance Evaluation

7.1 Time

To evaluate the performance of our proposed T3P, let's consider using it to obliviously merge two sorted lists of total length N. For purposes of this back-of-the-envelope calculation, we will assume that the PCI bus is an original generation PCI version, transferring 32 bits at 33MHz, for a bandwidth of 132MB/sec. The AES cores in the device will be sufficient to process the data at that rate. Note that the host will have to use its RAM to store all or some of the database, as the fastest current hard disks transfer data at only about 100MB/sec.

7.1.1 Reshuffling

As described before, merging N items takes $\frac{N \log N}{g \log g}$ g-switches. Executing one g-switch requires reading and writing gM bytes over the PCI bus. At 132 MB/sec, this corresponds to $\frac{gM}{132{,}000{,}000}$ seconds per switch, which gives us $\frac{MN \log N}{\log g \cdot 132{,}000{,}000}$ seconds for the whole merge.

N	Minutes on the 4758	Minutes using a T3P
512	1.7	0.000099
1024	3.8	0.00022
2048	8.3	0.00048
4096	19	0.0011
8192	44	0.0023

Table 3: Comparing the measured PIR re-shuffle time on the 4758 to the projected re-shuffle using our T3P approach, for $M = 850$ and $g = 32$.

In our previous work with the 4758, we measured the performance to do a TDES-based reshuffle when $M = 850$ bytes, for various N [13]. Reshuffling time is dominated by a merge of two sorted lists. Table 3 compares these figures to what our new approach should yield, for $g = 32$.

7.1.2 Request Engine

Our PPIR algorithm requires the trusted hardware, when handling a request, to read in all the records touched since the last shuffle. Our previous analysis [13] also showed that, until N got too large, this cost dominated the request handling time, rather than the amortized cost of the shuffle. For example, re-reading 256 850-byte records took 5.5 seconds on the 4758. If we did this with a special-purpose switch, this would take 0.002 seconds instead.

7.2 Gates

A quick look at the technical literature reveals a Helion AES engine that takes $6K$ gates and can exceed 500Mbs, which corresponds to 62MBs. To saturate the PCI bus speed, we would need four sets of two cores, for $48K$ gates. (If we were happy with 31Mhz, then we could do $24K$ gates instead.) We would also need latches and switching gates, but the AES cores would dominate.

It is interesting to note that the number of cores does not increase with g.

In order to accommodate public-key cryptography for establishing secure sessions with users, and authenticating the device, we plan to embed a modular arithmetic engine in the secured part of the device. Smart-card scale FPGA implementations of RSA exists using less than 3K gates.

For comparison, the current AEGIS [30] design uses 300K gates.

The largest components on the IBM 4758 are the 486 CPU (about 1.5 million transistors, or 400K gates) and the 16KB SRAM (about 260K gates). If we (conservatively) equate one gate to 4 bits of DRAM, the 4MB DRAM is about 8M gates. The crypto hardware adds even more.

8 Related Work

Secure hardware Current research (e.g., [18, 19, 30]) and product efforts (e.g., [12]) explore the notion of a secure computing environment built around the CPU alone. In contrast, the TCPA/TCG approach [31] attempts to secure an entire desktop, but not against physically present adversaries.

The XOM project [18, 19] investigated how to design a desktop-oriented processor architecture and operating system such that only the processor needs to be trusted, and not the OS and the RAM. The adversary's goal is to copy software which is run on the machine. They leave open the implications of the adversary observing a program's memory access pattern. In comparison, we are considering a very different use scenario—a specialized module, the tiny TTP, which does not have to provide for a full-blown multi-tasking OS, but which needs to hide *all* information about the software it is executing, and presumably be able to withstand a higher level of physical attack than a client-oriented system like XOM.

The AEGIS project [30] investigates the use of an innovative way for a processor to wield a secret key, by using a *Physical Random Function* based on random delays in silicon gates. It also assumes an untrusted RAM, but leaves open the consequences of exposing the RAM access pattern.

Cryptographically weak devices *Remotely keyed encryption* schemes seek to enable high-bandwidth encryption (on a host machine) using long-term keys held in low-bandwidth devices like smart cards [6]. This work shares the theme of enabling large computations using a small trusted space, but is otherwise quite different as it has no obliviousness requirements, and an adversary controlling the host can decrypt ciphertext until he is removed.

In a similar space, Modadugu et al. have developed a prototype using an untrusted host to help a Palm Pilot with the computation of generating RSA keys [23].

Less trusted and less functional TTPs The construct of *commodity servers* was introduced by Beaver to provide a trade-off between a heavyweight secure computation without *any* third parties, and a computation which fully trusts a third party [4]. Commodity servers assist a computation by providing random but potentially structured input to the players. The servers do not learn the player's actual inputs.

A similar scheme for PIR protocols appears in [10].

TCG The *Trusted Computing Group* (formerly known as the *Trusted Computing Platform Association*) has specified an architecture that takes a different approach: shrinking the physical security boundary to a single chip (the *Trusted Platform Module, TPM*) and then integrating this chip into the boot process of a larger, exposed system.

9 Conclusions and Future Work

We are currently proceeding to implement our T3P design on an FPGA. By this we will be able to confirm the performance gains we hope the design will achieve, and be in a position to move further with a small form-factor ASIC implementation suitable for physical shielding.

We plan to continue with examination both of the software and hardware aspects of this work. For example, on the software end, we plan to examine the SFE circuits for areas other than array access where special-purpose hardware may improve efficiency. On the hardware end, we plan to explore potential performance improvements from adding additional hardware (such as DRAM) to the T3P PCI card, but outside the trusted boundary.

We are confident that this project will enable practical solutions to secure multiparty computation problems which are beyond the scope of current approaches. This will directly increase the range of tractable SMC problems, for example enabling PIR on databases of millions of records. Also it should enable more complex problems, which incur a high cost even without privacy properties, to be solved with strong privacy guarantees.

Acknowledgments This work was supported in part by Cisco Corporation, the NSF (CAREER, CCR-0209144 and EIA-9802068), Internet2/AT&T, Sun, Intel, and by the Office for Domestic Preparedness, U.S. Dept. of Homeland Security (2000-DT-CX-K001). The views and conclusions do not necessarily represent those of the sponsors.

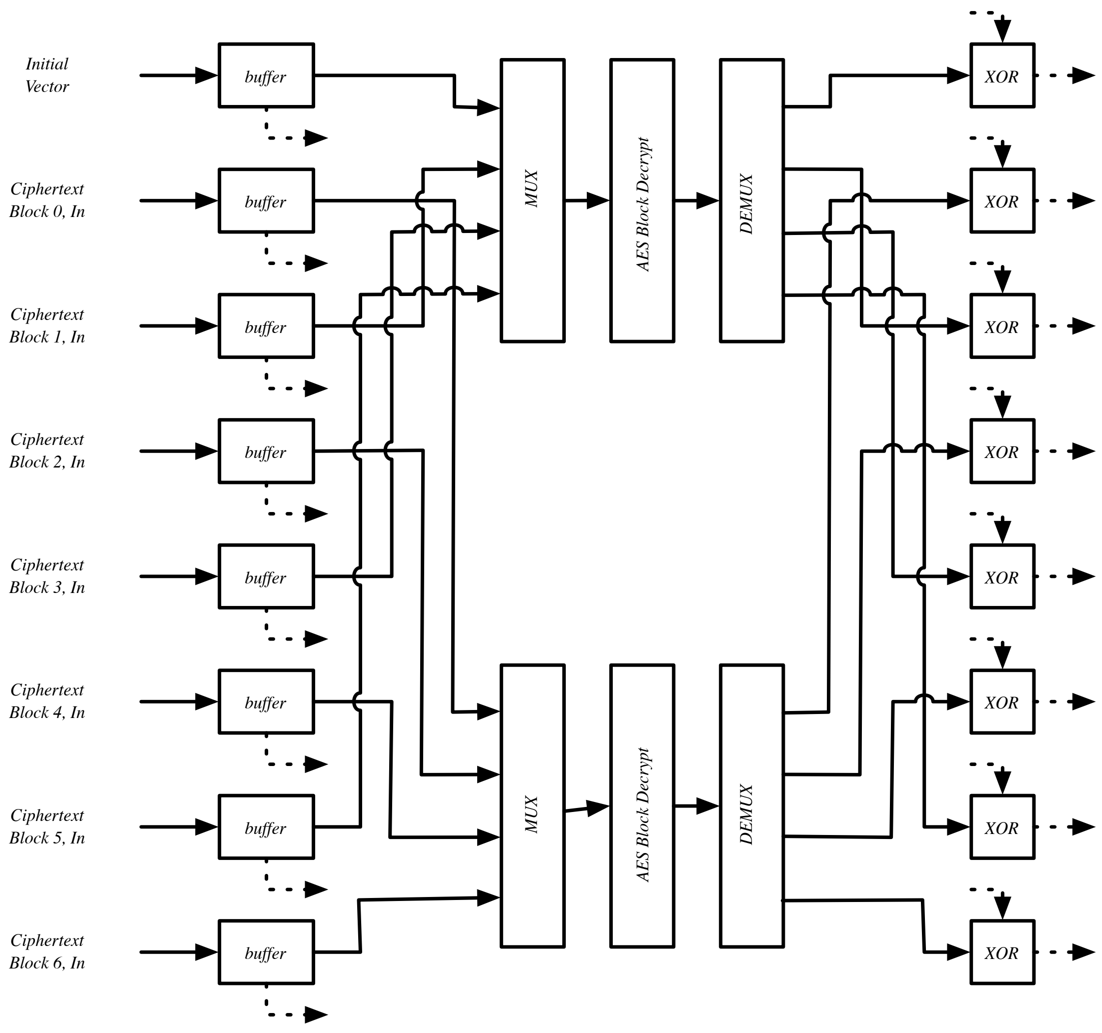

Figure 10: When the AES cores are sufficiently fast, we can reduce gate count while retaining performance by multiplexing the use of AES cores. In this sketch, we multiplexing two cores for the decryption step on the input lines. We might use this approach, for example, when the nth decryption finished by the time the host has finished writing the next block for line $n+2$. (We might save additional space by re-using logic across the multiplexers and de-multiplexers, respectively.)

References

[1] T. Arnold and L. van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48:475–487, May 2004.

[2] Dmitri Asnonov. *Querying Databases Privately: A New Approach to Private Information Retrieval*. Springer-Verlag LNCS 3128, 2004.

[3] Kenneth Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, Atlantic City, NJ, USA, April 1968. Thomson Book Company.

[4] Donald Beaver. Server-assisted cryptography. In *NSPW '98, Proceedings of the 1998 Workshop on New Security Paradigms*, Charlottsville, VA, USA, September 1998. ACM.

[5] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. http://www-cse.ucsd.edu/users/mihir/cse207/classnotes.html, May 2004. URL live in Aug 2005.

[6] Matt Blaze, Joan Feigenbaum, and Moni Naor. A formal treatment of remotely keyed encryption (extended abstract). In *Advances in Cryptology - EUROCRYPT '98: International Conference on the Theory and Application of Cryptographic Techniques*, volume 1403 of *LNCS*, pages 251–265, Espoo, Finland, May 1998. Springer-Verlag.

[7] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Eurocrypt 1999*, Prague, Czech Republic, 1999. Springer-Verlag. LNCS 1592.

[8] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–982, November 1998.

[9] Thomas H. Cormen and David M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing*, 24(1):5–20, January 1998. Also available as Dartmouth College Computer Science Technical Report PCS-TR96-294, August 1996.

[10] Yael Gertner, Shafi Goldwasser, and Tal Malkin. A random server model for private information retrieval, or how to achieve information theoretic PIR avoiding database replication. In *RANDOM '98: Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 200–217, London, UK, 1998. Springer-Verlag.

[11] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[12] David Grawrock. LaGrande architecture. http://www.intel.com/technology/security/downloads/scms18-LT_arch.htm, September 2003. IDF Fall 2003 presentation.

[13] Alexander Iliev and Sean Smith. Private information storage with logarithmic-space secure hardware. In *I-NetSec 04: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*, pages 201–216, Toulouse, France, August 2004. IFIP, Kluwer Academic Publishers. Workshop under 19th IFIP International Information Security Conference–SEC2004.

[14] Alexander Iliev and Sean Smith. More efficient secure function evaluation using tiny trusted third parties. Technical Report TR2005-551, Dartmouth College, Computer Science, Hanover, NH, USA, July 2005. http://www.cs.dartmouth.edu/reports/abstracts/TR2005-551/.

[15] Alexander Iliev and Sean Smith. Protecting client privacy with trusted computing at the server: Two case studies. *IEEE Security and Privacy*, 3(2), March 2005.

[16] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 310–331, London, UK, 2001. Springer-Verlag.

[17] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[18] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, November 2000.

[19] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178–192, Bolton Landing, NY, USA, 2003. ACM Press.

[20] Mark Lindemann and Sean W. Smith. Improving des coprocessor throughput for short operations. In *10th USENIX Security Symposium*, Washington, D.C, August 2001.

[21] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In Matt Blaze, editor, *13th USENIX Security Symposium*, pages 287–302. USENIX, August 2004.

[22] David A. McGrew and John Viega. The security and performance of the Galois/counter mode (GCM) of operation. In *INDOCRYPT*, volume 3348 of *LNCS*, pages 343–355. Springer, 2004.

[23] N. Modadugu, D. Boneh, and M. Kim. Generating RSA keys on the PalmPilot with the help of an untrusted server. In *RSA Data Security Conference and Expo*, 2000.

[24] A. Sabelfeld and A.C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[25] Sean Smith. Outbound authentication for programmable secure coprocessors. In *7th European Symposium on Research in Computer Science*, October 2002.

[26] S.W. Smith. Fairy Dust, Secrets and the Real Worl. *IEEE Security and Privacy*, 1:89–93, January/February 2003.

[27] S.W. Smith and D. Safford. Practical Server Privacy Using Secure Coprocessors. *IBM Systems Journal*, 40:683–695, 2001.

[28] National Institute Of Standards and Technology. Security requirements for cryptographic modules. http://csrc.nist.gov/publications/fips/fips140-1/fips1401.pdf, Jan 1994. FIPS PUB 140-1; URL current in June 2005.

[29] National Institute Of Standards and Technology. Security requirements for cryptographic modules. http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf, May 2001. FIPS PUB 140-2; URL current in June 2005.

[30] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, Madison, WI, USA, June 2005.

[31] Trusted Computing Group. https://www.trustedcomputinggroup.org/home, May 2005.

[32] A. C. Yao. How to generate and exchange secrets. In *Proc. of the 27th Symposium on the Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.

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