Faerieplay on Tiny Trusted Third Parties (Work in Progress)*

Alexander Iliev and Sean W. Smith

Department of Computer Science, Dartmouth College, Hanover, NH, USA
{sasho, sws}@cs.dartmouth.edu

Oct 31, 2006

Abstract

Many security protocols refer to a *trusted third party (TTP)* as an ideal way of handling computation and data with conflicting stakeholders. Subsequent discussion usually dismisses a TTP as hypothetical or impractical. However, the last decade has seen the emergence of hardware-based devices like the IBM 4758 that, to high assurance, can carry out computation unmolested; TPM-based systems like Intel's Lagrande also provide secure platforms; emerging research in trusted computing promises more.

In theory, such devices can perform the role of a TTP in real-world problems. In practice, all existing devices have problems. TPM-based systems are not secure against physical attack. The 4758 aspires to be general-purpose but is too small to accommodate real-world problem sizes. The small size forces programmers to hand-tune each algorithm anew, to fit inside the small space without losing security. This tuning heavily uses operations that general-purpose processors do not perform well. Furthermore, current devices are 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 designed and prototyped an efficient system, called Faerieplay, to execute *arbitrary* programs on T3Ps while preserving critical trust properties. To eliminate the performance and cost obstacles, we are currently examining the potential hardware design for a T3P optimized for bottleneck operations. We estimate that such a T3P could outperform the 4758 by several orders of magnitude, while also having a gate-count of only 30K-60K, one to three orders of magnitude smaller than the 4758 or hardened CPU systems like AEGIS. We are currently proceeding with a proof-of-concept prototype on a Xilinx FPGA.

1 Introduction

Many distributed protocols consist of computation on private, sensitive data belonging 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 satisfying the interests of these participants, designers often posit the existence of a *trusted third party (TTP)*. If the participants trust this TTP not to cheat, then the desired security properties can be obtained simply by moving sensitive computation to the 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 [17]: not particularly appropriate for the real world.

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, eg. [23], 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 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 expensive; 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 previously developed and implemented algorithms for solving some specific large problems on hardware-based TTPs despite these size limitations. More recently, we have generalized this, producing a compiler and runtime (which uses these algorithms) to run arbitrary programs securely inside small TTPs.

Throughout our prototyping work, we used the IBM 4758 TTP, and noticed that its hardware optimizations did not align with the algorithms we needed; this experience suggests that building what we call a *tiny TTP* optimized for these operations might both improve performance and lower cost.

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

*Please see our preliminary design report for more complete details, definitions and proofs [10].

Contributions Currently, one has several options for a hardware-based TTP, each with substantial problems. Our tiny TTP design provides substantive improvements to each such option. In particular:

- If one is using an IBM 4758 or 4764 as a TTP, and using a scheme like Faerieplay [8] or Oblivious RAM [4] to allow secure computation on larger data sets than the TTP's RAM, our design provides a lower-cost and higher-performance approach.

- If one is using a TPM-based system like Intel Lagrande[1] for sensitive computation, a tiny TTP can provide a higher assurance environment, with comprehensive protection against physical tampering.

Paper Outline In Section 2 we present some background on current hardware-based TTPs, particularly the IBM 4758. In Section 3 we review our experience using TTPs for *private information retrieval (PIR)* and also general *secure multi-party computation (SMC)*, allowing for constrained resources in the TTP, such as limited internal memory. In Section 4 we discuss the shortcomings of the 4758 for running these systems, and we introduce the basic hardware elements we envision for an optimized *tiny TTP*. In Section 5 we discuss how these elements might be assembled into a private execution engine. In Section 6 we estimate how the performance and size (gate-count) for this T3P compares to prior hardware-based TTPs. Here we also present some results from a full-fledged SMC prototype running on the 4758. In Section 8 we outline our current progress on a proof-of-concept tiny TTP on an FPGA, and our plans for completing this initial implementation.

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 [23]. It also includes mechanisms, called *outbound authentication* (OA)[2] to prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [16]. 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 be trusted by clients (by virtue of all the above provisions), but the host is not trusted (not even its root user). Thus, whenever we write "the host", it should be understood that from a security perspective this is equivalent to "the adversary".

IBM 4758 and 4764 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 [19][3]. 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 encrypt-and-MAC, 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.

The successor to the 4758 is the 4764, or PCIXCC [1]. It has 64MB of RAM, and a 266MHz PowerPC processor, and costs around $8,500. We believe it also cannot serve as an efficient TTP, as it is very costly, and does not provide hardware acceleration needed for oblivious computing. It may be adequate if the user can afford it, and the problem fits in the 64MB, but if not, and untrusted external RAM has to be used, the overhead will be large as with the 4758, and the device's hardware will be underutilized.

3 TTP Experience

We have designed and built several applications making essential use of a trustworthy TTP. 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.

3.1 Private Information Retrieval

As an example, let's consider the problem of *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. Many schemes were devised to solve this problem without any trusted parties (but usually assuming other settings like multiple non-colluding servers), but their performance and somewhat unrealistic settings appeared to limit their practical potential. Smith and Safford suggested the *practical PIR (PPIR)* variant of PIR: 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 [18].

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? Our work focuses on providing practical solutions to this problem. We assume that the TTP is very small, eg. for PPIR with N records of size M each, the TTP can only fit $O(M + \log N)$ 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 [4],

[1] Lagrande and similar client-side systems are not claimed to be secure against physical attack.

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

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

and later also used by Asonov [2]. First, the TTP generates a randomly permuted (and of course encrypted) version of the database. The TTP has to do this in a way that the host cannot learn anything about the permutation, as elaborated in Section 3.3. Then, the TTP can answer queries by just reading records from the permuted database, having applied π to the requested index to get the permuted index. The TTP also needs to hide relationships among queries, and the easiest way to achieve this is to re-read previously read records every time a new query arrives. This makes queries increase in cost each time, so after some number, the TTP generates a new randomly permuted database, and begins afresh. This is a very brief exposition, and we refer the reader to [7, 9] for the full details, and some extensions from the original prototype. A variation on our last PIR/W design (which allows private update of the database) is found in [21]—it gives a faster algorithm for transitioning between PIR sessions, at the expense of requiring more space in the TTP.

3.2 Arbitrary Multi-Party Problems

The general problem of securely computing a function with multiple adversarial stakeholders is known as *Secure Multi-party Computation* (SMC). With just two parties, the name Secure Function Evaluation (SFE) is also used. 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. This problem largely captures the notion of a "TTP application"—one which needs a TTP (or a complex special-purpose scheme to provide the same properties) to satisfy its security properties. We elaborate on extended models of a TTP app, like long-lived servers, in our Faerieplay report [8].

Current solutions to the SMC problem do not employ a TTP, but instead use a protocol on a combinatorial circuit form of the function f, eg. [22].

One unsolved problem with this approach to SMC is that the use of indirectly addressed arrays is very expensive in a combinatorial circuit—each access is linear time in the array size. Thus, in order to provide efficient indirect array handling to SMC, we combined the circuit-based solution to SMC with using a TTP to evaluate *array gates* efficiently and securely, calling the system Faerieplay[4][8]. The TTP uses a form of PIR to access the array stored on the host, without leaking information to the host.

Faerieplay provides a more efficient solution to the general SMC problem than current implementations like Fairplay[13]. Seen another way, it provides a general framework for deploying TTP applications, taking care of code and data too large to fit in the TTP.

3.3 Oblivious Networks with Encrypted Switches

A common theme in our work was the need for the TTP to perform operations *obliviously* on a sequence 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; such operations were the major bottleneck in our PPIR system. In all cases our obliviousness requirement is that the adversary (bounded by feasible computations) does not learn anything about the parameters of the operation from observing the IO that the TTP performs, and the memory restriction allows the TTP to read in and operate on only a few items at a time.

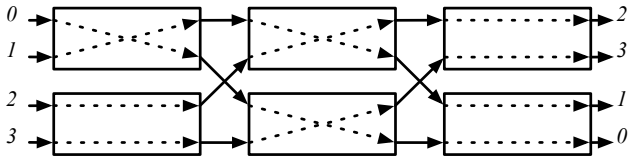

Figure 1: A Beneš permutation network with 4 inputs, performing the permutation $\langle 2, 3, 1, 0\rangle$. The dashed lines represent switch settings, which depend on the permutation. The rest of the network only depends on the number of inputs.

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 $\mathrm{O}(N \log N)$ 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 1 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 are useful 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). 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 (eg. 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*.

4 Optimizing Hardware Bottlenecks

As we have discussed in the previous sections, the bottlenecks of our TTP-based PPIR and Faerieplay schemes are the permutation and sorting and merging networks, and their essential component is the encrypted switch. However, what we needed for a fast encrypted switch did not match the hardware provisions of the 4758—we could not make full use of its I/O channels and fast-path TDES capability, the CPU was not fully

[4]The name Faerieplay refers to the perceived equivalence between TTPs in protocols and fairies, and is also in recognition of the inspiration we got from the Fairplay project

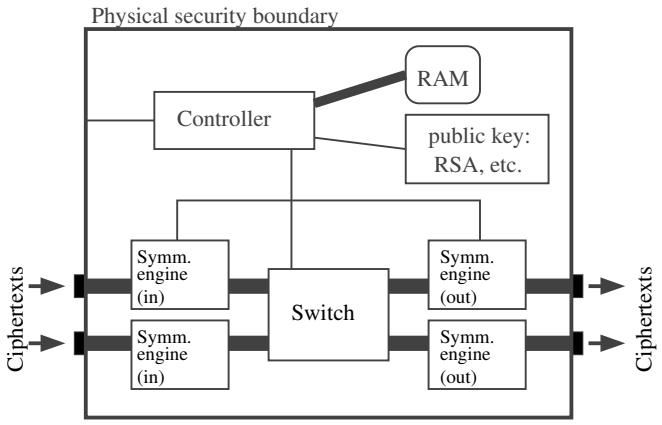

Figure 2: The outline of fast streaming through an encrypted switch in a tiny TTP. Block by block, it brings in two records, decrypts them, swaps them depending on the switch setting, then encrypts them again and sends them out.

used as no CPU-bound processing is involved, and we did not use all the RAM. We think that the slow performance, on an expensive device, suggests designing a new type of hardware, optimized for our purposes.

We note that while the designs presented here target only the bottleneck operation of a tiny TTP—oblivious networks—the full device is general purpose, ie. it can be programmed to compute any computable function. If it implements Faerieplay, the tiny TTP will be programmed with a circuit (compiled from standard source code), and if it implements Oblivious RAM it will be programmed with a standard RAM program.

In Figure 2 we show the main hardware optimization idea—a fast streaming capability to switch and re-encrypt two streams of data. 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 $\mathrm{O}(M)$ (the record size) to $\mathrm{O}(B)$ (where B is the block size of the cipher).

4.1 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. We define the symmetric engine to include MAC computation, so it performs *authenticated encryption*. For efficiency reasons this may mean that each engine will contain two AES cores, one for cipher and one for MAC. We are also considering the use of newer symmetric modes like Galois/Counter Mode (GCM) [14].

4.2 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 [12] This observation suggests that, at first approximation, the time complexity associated with executing a network of encrypted switches depends more on the number of switches, and less on the size of each switch, ie. a smaller number of larger switches can yield a performance benefit.

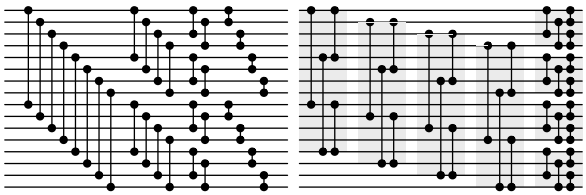

Figure 3: This sketch shows an example reverse butterfly network for $N = 16$ inputs. The vertical lines denote switches on the two endpoint lines. On the left, we show the original network of 32 2-input switches, arranged in 4 stages. On the right, we show it as eight 4-switches, each performing the action of four 2-switches.

Thus the question naturally arises: can we decompose a network into a smaller number of larger switches? It turns out that this is possible in all oblivious networks which we need to use. In particular, we use bitonic merging and sorting networks, and Beneš permutation networks; all of these are constructed with one or more *butterfly networks* and their reverses. In Figure 3 we illustrate how a reverse butterfly network on N lines, which has $(N/2)\log N$ switches, can be constructed with $\frac{N \log N}{g \log g}$ g-switches, each with g inputs and outputs (in the limit, if $g = N$, we get just one "N-switch"—the whole network.) In our longer report[10] we formally state and prove this property of the butterfly network. Using $g > 2$ has the benefit, in addition to reducing the number of I/O rounds, of reducing the total amount of I/O by a factor of $\log g$ too.

5 Building a Whole Privacy Engine

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, and the host cannot determine which ciphertext output of a switch matches which ciphertext input.

Switch execution The switch module will execute all the switches in a network in some topological order (eg. stage by stage). While keeping a small constant amount of state, the switch can keep track of which records it needs to work on next, and thus can check if the host is providing the correct records.

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, but could also be handled with shared symmetric keys.

Sharing Symmetric Cores In order to ensure that the speed with which the TTP's I/O bus streams data is similar to the

speed of the AES cores, so all major components work at maximum capacity, it may be necessary to share the AES cores. We can facilitate this by using an AES core which just does block-AES, without any chaining mode. Then the chaining hardware (which is simple) can be replicated, but the large AES core shared across several AES chains.

A whole Faerieplay scenario walk-through 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 holistic view of a whole SMC of some function f, using the Faerieplay scheme. 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, 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 Performance Evaluation

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 no current hard disks transfer data at 132MB/sec.

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.

In our previous work with the 4758, we measured the performance to do a TDES-based reshuffle when $M = 850$ bytes, for various N [7]. Reshuffling time is dominated by a merge of two sorted lists. Table 1 compares these figures to what our new approach should yield, for $g = 32$.

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

V	7	15	63
Time in mins.	4.1	8.9	53

Table 2: Running times for Dijkstra graph search, on a random graph of V vertices and average out-degree 5, running on Faerieplay with a 4758 acting as a tiny TTP (limited to $O(M + \log N)$ space).

Current 4758 Prototype Our current Faerieplay prototype uses the 4758 as a (slow) tiny TTP. The example program we have been working with is a graph search using Dijkstra's algorithm. If Alice owns a graph, and Bob has a source and destination, the program allows Bob to learn a path and/or distance between his two points, without revealing the graph to him, and without Alice learning the two points. We show some performance results in Table 2. If we can get the expected speedup on a Tiny TTP over the 4758, it should make this program practical on graphs of tens of thousands of nodes.

We have also developed an oblivious RAM prototype, to compare its performance with Faerieplay. As we expected, Faerieplay is faster, because it focuses the high-overhead oblivious access to just where it is needed—indirect arrays. We stress that both techniques for TTP-assisted secure computation can benefit similarly from our new tiny TTP design.

These threads of work will be presented together in [6].

Size (gates) There are a variety of AES hardware options, eg. a commercial Helion AES core with $6K$ gates and yielding 500Mb/s or 62MB/s[5]. To saturate the PCI bus speed of 132 MB/s, we would need four sets of two cores, for $48K$ gates. There will also be some gates for the control and actual switch, but the AES cores would dominate.

OpenCores (`www.opencores.org`) lists an AES core (`aes_core`), freely available, of 38K gates attaining about 1 Gb/s, thus requiring 4 instances in our design for a total of 150K gates.

Smart-card scale FPGA implementations of RSA exists using less than 3K gates. Our T3P will use RSA infrequently, so a relatively slow implementation should be adequate.

For comparison, the current AEGIS [20] 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.

[5] See `http://heliontech.com/aes_std.htm`. This is the variant without key-expansion, but since in our design we minimize key changes, it should be feasible to use one key-expansion module for several AES modules.

7 Related Work

Secure hardware Hardware modules like the TPM are intended to protect data—keys and software measurements—as opposed to computation. Also, they do not protect against physical tampering attacks, which are an important part of our threat model.

Current research (e.g., [11, 20]) and product efforts (e.g., [5]) explore the notion of a secure computing environment built around a security-enhanced CPU, with security provisions extending to the whole system by means of partitioning, and memory encryption and checking. These systems (if implemented) offer varying degrees of protection against a physically present and dedicated adversary with a drill and bus and RAM probes, but none of them address this threat directly.

The XOM project [11] investigated how to design a secure 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.

The AEGIS project [20] 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 [3]. 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 generating RSA keys [15].

8 Continuing Work and Conclusions

In order to confirm the performance gains we hope the T3P design will achieve, we are currently proceeding to implement some of it on an FPGA. We are working with a Xilinx Virtex 2 Pro FPGA, with 18.5K 4-input Lookup Tables (LUTs), on a PCI board made by Avnet. The initial proof-of-concept implementation will be a simple switch with AES decryption and encryption around it, and communicating to the host via PCI. This should demonstrate the speedup attainable in the bottleneck process of the whole Faerieplay system.

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.

Acknowledgements This research was supported in part by NSF grant CNS-0524695, and the Bureau of Justice Assistance grant 2005-DD-BX-1091. The views and conclusions do not necessarily reflect those of the sponsors.

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] Matt Blaze, Joan Feigenbaum, and Moni Naor. A formal treatment of remotely keyed encryption (extended abstract). In *EUROCRYPT '98*, volume 1403 of *LNCS*, pages 251–265, Espoo, Finland, May 1998. Springer-Verlag.

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

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

[6] Alexander Iliev. *Using Tiny Trusted Third Parties to Enhance Secure Two-Party Computations*. PhD thesis, Dartmouth College, Hanover, NH, USA, November 2006. to appear.

[7] 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.

[8] 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/`.

[9] Alexander Iliev and Sean Smith. Protecting client privacy with trusted computing at the server: Two case studies. *IEEE Security and Privacy*, 3(2):20–28, March 2005.

[10] Alexander Iliev and Sean Smith. Towards tiny trusted third parties. Technical Report TR2005-547, Dartmouth College, NH, USA, July 2005. `http://www.cs.dartmouth.edu/reports/abstracts/TR2005-547/`.

[11] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS 2000*, pages 168–177, November 2000.

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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 *ISCA 2005*, Madison, WI, USA, June 2005.

[21] S. Wang, X. Ding, R. Deng, and F. Bao. Private information retrieval using trusted hardware. In *ESORICS 2006*, September 2006. LNCS 4189.

[22] A. C. Yao. How to generate and exchange secrets. In *FOCS 1986*, pages 162–167. IEEE, 1986.

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