Cryptography as a Network Service

Tom Berson
Xerox PARC[‡]
berson@parc.xerox.com

Drew Dean
Xerox PARC
ddean@parc.xerox.com

Matt Franklin[*]
U.C. Davis
franklin@cs.ucdavis.edu

Diana Smetters
Xerox PARC
smetters@parc.xerox.com

Michael Spreitzer[†]
IBM Research
mspreitz@us.ibm.com

Abstract

Cryptography is a powerful tool for building secure distributed systems, albeit at substantial computational cost. This is especially true for public key cryptography. Conventional wisdom dictates that cryptography must be done locally in order to be secure. We argue for an approach in which public key cryptographic operations are provided as a network service. This service operates over long-lived associations secured by symmetric cryptography. This network architecture amortizes the cost of special-purpose cryptographic hardware across many users. We describe the implementation of such a system, and performance results we obtained.

1. Introduction

The received wisdom about cryptography is that it is difficult and expensive. This has led to system designs and engineering tradeoffs that aim to minimize the use of cryptography, and especially of public-key operations, because they are computationally intensive.

But this received wisdom is no longer true. A great many influences have already made cryptography easy and cheap, and will continue to make it easier and cheaper. These diverse influences include the professionalization of cryptographers, the creation of textbooks and of courses, the steady growth of computational power delivered by the operation of Moore's law, the algorithmic advances made by cryptographic researchers and engineers, the rise of e-commerce and wireless infrastructures which have a seemingly endless appetite for cryptographic services, the entry of many young people into the field, and the easing of government export controls. We envisage a near future where cryptographic operations will be as pervasive, cheap, and unremarkable, as IP protocol operations have become today.

We are following a paradigm of building future gadgets and services now, no matter how uneconomical that may be, in order to study the impact of those gadgets and services on people, on organizations, and on systems. We use this approach to identify and explore scientific, technical, economic and social consequences of this possible future.

We begin our exploration of the cryptographic future by making a canonical "expensive" cryptographic operation – modular exponentiation – fast, cheap, and ubiquitous. Instead of providing everyone with their own expensive hardware cryptographic accelerator, we have taken the novel approach of providing cryptographic operations as a network service. We have designed and built a fast "cryptoserver" for public-key operations. The cryptoserver is equipped with a number of hardware cryptographic accelerators, which as a result may be shared among a large number of clients. This sharing allows the cost of such acceleration hardware to be amortized over a large number of client machines, and allows even clients who perform only a moderate number of cryptographic operations to benefit from hardware speedups before such time as accelerators are cheap enough to be present in every machine. Use of such a server benefits clients in two ways: the cryptoserver may have access to cryptographic hardware capable of performing single cryptographic operations faster than they can be performed by the client, and second offloading cryptographic operations from the client CPU to these remote accelerators can free the client for other operations. Such load reduction can be quite significant, especially given that most cryptographic accelerators on the market today are highly parallel multiprocessors capable of processing many requests at once. Such benefits are available to any individual machine equipped with its own local cryptographic accelerator; pooling same in a server allows many client machines to benefit from such accelerators in situations where it would not be possible to so equip each individual client machine.

We have built interfaces to the cryptoserver and made it available to our colleagues via our in-house network.

[*]Work done while employed at Xerox PARC; Present address: Dept. of Computer Science, UC Davis, 1 Shields Ave., Davis, CA, 95616.

[†]Work done while employed at Xerox PARC; Present address: IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

[‡]Address: 3333 Coyote Hill Road, Palo Alto, CA, 94304

The approach we have taken to allow such a service to operate securely and efficiently is to trade expensive cryptographic operations for cheap ones, and to bootstrap many expensive cryptographic operations performed frequently on the cryptoserver by means of few expensive operations performed rarely on the client. Clients secure their communications with the cryptoserver using symmetric encryption (3DES). The keys used to provide this secure channel are set up as part of a (relatively) long-lived security association between client and cryptoserver – for the cost of one possibly expensive key exchange on the client and (inexpensive) symmetric encryption, the client gets back many cryptographic operations performed cheaply on the server.

This cryptoserver is something of a time machine: it allows us to do now what we expect everyone will be able to do in the future. To our delight, we discovered that the cryptoserver is useful and economical even in the present. Under reasonable assumptions, the amortized cost (including network bandwidth, hardware, development, *etc.*) of a 1024 bit modular exponentiation is 10^{-4} cents.

The rest of the paper is organized as follows: in Section 2 we discuss our trust model, in Section 3, we present our overall approach to sharing cryptographic services and discuss potential applications for the cryptoserver, in Section 4, we discuss our implementation, in Section 5, we present performance results, in Section 6, we discuss related work, in Section 7 we discuss future directions, and Section 8 concludes.

2. Trust Model

Outsourcing cryptography inherently raises questions about the trustworthiness of the computation. In many common cases, however, this is a non-issue because the cryptoserver and its clients are within the same security perimeter. Examples of this include a cryptoserver deployed in an enterprise intranet or a cryptoserver supporting SSL/TLS for a group of web servers all operated by a single company. There are a large number of applications such as these, where all of the machines sharing one cryptoserver already trust one another explicitly or implicitly (given that they are all controlled by the same individuals). In such cases, sharing key material with a (dedicated) cryptoserver does not imply any further loss of security. For these and similar cases, using a cryptoserver is just as trustworthy as performing the computation locally, under the assumption that the communication between the two is secure. In addition to the security provided by network topology (*e.g.,* firewalls) [18], we use symmetric cryptography to secure the link between the client and cryptoserver. We presently use two key 3DES, which current research [20] suggests has equivalent security to modular exponentiation with > 4000 bit moduli. We feel that this choice will be ample for some time to come. The choice of wire encryption algorithm is independent of the system architecture as a whole; it would be simple to substitute another appropriate cipher such as AES.

When the cryptoserver is under different administrative control, the trust relationship becomes more complicated. If the client is unwilling to trust the cryptoserver to perform the computation correctly, then there are a few precautions that could be taken. If the client's public key is low-exponent RSA, then verifying the computation is much less costly than performing it directly. This verification could be performed by the client itself, or shipped off as a request to a second cryptoserver. Alternatively, the original request could be shipped off to two or more independent cryptoservers as a way to perform quality checks on the output. More sophisticated strategies for quality checks on distributed computations appear in [24] and [25]. A client unwilling to trust a single cryptoserver with knowledge of its private key may use standard secret-sharing techniques to separate its key into multiple parts, and send each part to a different cryptoserver. The client then (relatively inexpensively) can recombine the results returned by all the servers to produce a valid decryption. Only if all the cryptoservers so used combine their information will they be able to recover the client's private key [5, 12]. Even more effective techniques can be used in the case of discrete log-based cryptosystems, where a fixed base is raised to many different random exponents. A "free" source of modular exponentiation can be used to generate such values efficiently [6]. Lastly, the client may be unwilling to trust the cryptoserver with knowledge of the data, *i.e.,* the message to be signed or the ciphertext to be decrypted. This can be accommodated by a straightforward application of "blinding" [8]. Note that blinding is particularly efficient for RSA signing operations, such as those used in the SSL/TLS key exchange.

3. Applications

We can identify three main types of applications for our fast cryptoservers: simple decryption/signature, bulk decryption/signature, and sophisticated cryptographic protocols.

In the first main type of application, the client needs to perform a single cryptographic operation. Instead of performing it locally, the client sends a request to the cryptoserver to have it done remotely. The second main type of application is similar to the first, except that many requests are sent to the cryptoserver simultaneously. These can certainly be treated as independent simple requests. However, there are a number of reasons to treat this type of application separately. One is that there are often amortized efficiency gains that can be realized by specialized cryptographic hardware when processing a number of re-

lated cryptographic operations, *i.e.,* different keys processing identical data, or identical keys processing different data. See [23] for a thorough survey of some of these methods. Second, we believe that bulk encryption and signature using one or more fast cryptoservers will turn out to be quite effective in certain (bandwidth-unconstrained) multicast scenarios.

The third main type of application exploits the same underlying cryptographic operations that are performed for encryption and signature, but in order to achieve more sophisticated security goals. For example, there are a number of security protocols for which the degree of privacy scales nearly linearly with the computational burden. A fast cryptoserver can perform the necessary computations on the user's behalf. The cryptoserver could be owned by a security service, or it could be an independent service that sub-contracted the privacy-preserving computation. The cryptoserver needs to perform only ordinary public key operations computations that are exactly what it computes when it signs and decrypts.

One example in this class is "Private Information Retrieval" [10]. The querier to a database hides his query among a plausible set of m possible queries, where m is a tunable parameter. This can be done in a naïve way by having the database return all m replies. There are more sophisticated approaches that greatly reduce the communication between the querier and the database. The first protocol for private information retrieval was due to Chor, Goldreich, Kushilevitz, and Sudan. A newer solution [7] requires the querier and the database to each perform about m modular exponentiations, and reduces the communication to one round of messages of size $\log m$. This example has many practical applications, such as allowing a user to get a real-time stock quote without revealing which stock he is interested in, or query a patent database without leaking the query to a competitor. It is possible to combine the approach of [7] with random self-reduction techniques to hide all information about the queries from the fast cryptoserver as well [15]. In fact, the calls to the fast cryptoserver can be made independent of the actual queries that will be made. When the computations are data-independent pre-computations, we say that the computations are "generic".

A second example is group authentication. A participant proves to be one of a plausible set of m possible participants without revealing which, where m is a tunable parameter. There is also a non-interactive version of this called a group signature. Existing cryptographic solutions require the participant and the verifier to each perform about m modular exponentiations, and exchange one round of communication of size m. This example has many practical applications, including to enhance the value of recommendation systems [16]. Early work on this kind of protocol includes [11, 17, 9]. The group authentication and group signature schemes can be computed in a way that is data-dependent and blindable. That is, sensitive information can be concealed from the fast cryptoserver, but cannot be performed as a generic precomputation

This third class of applications, those that go beyond simple signing and decryption, are far and away the most interesting. Those cryptographic protocols that have gained current market acceptance (*e.g.,* SSL/TLS) have been designed specifically to reduce the number of modular exponentiations due to their high cost. The applications where use of a cryptoserver will provide the most leverage are those few current protocols where computing large numbers of modular exponentiations is a major bottleneck (*e.g.,* in clusters of secure web servers). The cryptoserver also will make it attractive to develop new protocols unfettered by the need to keep modular exponentiations to a bare minimum. Such protocols have the added advantage that many of them may require the modular exponentiation of non-secret data; thus limiting the trust requirements for use of networked cryptoservers.

4. Implementation

The goal of our implementation is to efficiently export the computational resources of one or more cryptographic accelerators to the network in a secure fashion. See Figure 1.

4.1. Hardware Architecture

We built our cryptoserver out of generally available hardware, restricting our development efforts to custom software. Our initial prototype uses a Sun Ultra-10 workstation running Solaris 7 with one Atalla AXL200 accelerator and one nCipher nFast 300 PCI accelerator. The AXL200 is advertised at 236 1024-bit private key RSA operations per second and has a maximum throughput of 265 1024-bit private key RSA operations per second according to Atalla. The nFast is rated at 300 1024-bit RSA private key operations per second, assuming the use of Chinese remaindering. Our design cleanly factors the interface to the cryptographic accelerators from the rest of the software, so that multiple heterogeneous accelerator boards can be supported concurrently. Our software hides, as much as possible, the differences in the various accelerator hardware. We chose the Sun workstation because several cryptographic accelerator vendors support Solaris on SPARC hardware.

A production cryptoserver would differ from the present configuration in three major aspects:

1. It would likely be a multiprocessor, to more efficiently handle independent cryptographic requests in

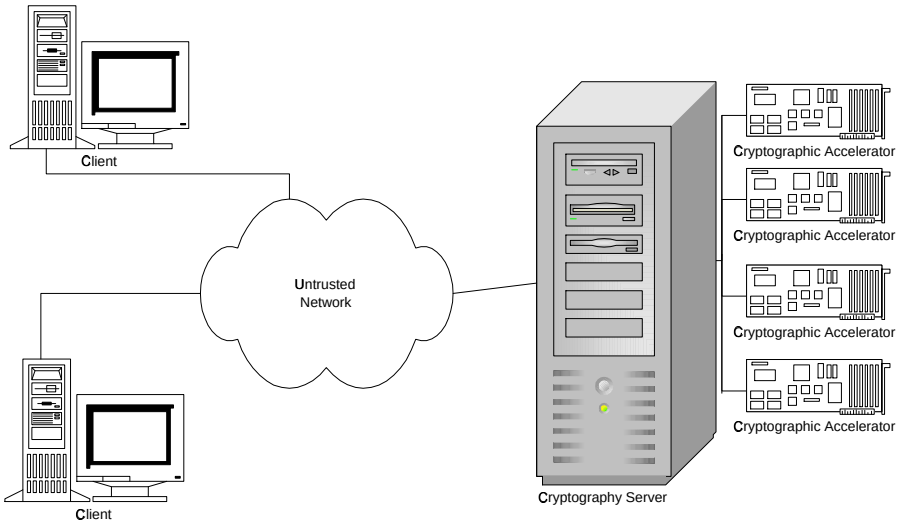

Figure 1. Cryptoserver Network Architecture

parallel.

2. Symmetric cryptographic acceleration hardware would be useful to speed encryption of client – cryptoserver communications, particularly if DES or triple-DES is desired. Faster block ciphers can be effectively accommodated in software.

3. Multiple public key accelerators would be used, for a dramatic throughput increase.

While these enhancements would undoubtingly increase performance, our present system, with very modest hardware, already performs acceptably well. Our software has been architected with scalability in mind, as discussed below.

4.2. Software Architecture

Our architecture was designed to meet a number of important goals:

1. It must scale effectively with the following factors:
   (a) the number of client requests
   (b) the amount of cryptographic hardware available (it should keep all useful cryptographic capacity busy at all times given sufficient requests)
   (c) the number of individual clients (security associations) being managed and used

2. Individual client requests must have very low overhead, both in terms of network operations and security association management; infrequent operations (security association negotiation) may be more expensive to compensate.

3. It must be easy to incorporate additional cryptographic acceleration capacity, and to take maximal advantage of different types of cryptographic accelerators.

4. It must be easy to plug in different mechanisms for negotiating security associations and protecting requests on the network.

There are many possible choices for the software to meet the goals described above. We explain our choice of a communications substrate for the implementation. We describe the interface that the cryptoserver presents to its clients. Finally, we explain our choice of software architecture.

4.2.1. Middleware

Our architecture places several fundamental requirements on the messaging middleware we choose. In order to build a server that will scale to support a number of highly parallel cryptographic coprocessors, the middleware layer must cope well with a multi-threaded server. Our basic design rests on the idea of being able to leverage one potentially expensive cryptographic operation on the client into many on the server, by using that one expensive client operation to establish a long-term security association between client and server. The middleware platform we use must support the establishment and maintenance of such associations, preferably via a choice of protocols. Finally, the middleware platform must allow for either a connectionless transport (*e.g.*, UDP), or if it relies on a connection-oriented transport layer (*e.g.*, TCP), it must be able to either maintain long-term security associations across mul-

tiple sequential TCP connections from the same client machine or manage a large number of dormant TCP connections from clients who may not be constantly active but want to maintain their security associations.

We chose to implement our system on top of Sun's Transport Independent Remote Procedure Call (TI-RPC) middleware. We chose TI-RPC because it supported:

1. multithreaded applications, for a simple path to building a scalable service;

2. through RPCSEC_GSS [13] it supports the Generic Security Service (GSS) API [21], so we could use multiple authentication and encryption technologies and negotiate long-term security associations

3. and it supported datagram (*e.g.,* UDP) transports.

We preferred to use UDP because datagrams are a much more natural match for the RPC paradigm than connection-oriented transports (such as TCP), and because a UDP-based solution obviated the need for complicated connection pool management logic, as we desire to scale the number of clients past the number of socket descriptors available in a single UNIX process (typically on the order of 1024). UDP also minimizes transport-related overhead for clients who may make infrequent calls to the cryptoserver. Other middleware choices, most notably CORBA, had one or more substantial gaps in supporting multithreaded applications over encrypted datagram transports.

A major advantage of this choice of middleware is the availability of RPCSEC_GSS, an interface to the GSS-API. GSSAPI is a pluggable security API, allowing a consistent interface to a variety of different authentication and encryption technologies. It is one of the few security technologies to explicitly support negotiation of security associations over connectionless transports (though we could negotiate security associations out-of-band over a connected transport), and which is capable of securing communications over such transports. Most importantly, RPCSEC_GSS naturally supports the most central idea in this paper: that of leveraging long-term security associations to secure RPC-based requests with a minimum of per-request overhead.

Another substantial advantage of TI-RPC is that it is compatible on the wire with Sun's ONC RPC, a widely deployed RPC protocol that is at the heart of NFS. As such, ONC RPC implementations are available on a wide variety of platforms. Sun recently made TI-RPC source code available under a liberal license, and as NFS v4 will be built on TI-RPC, we hope it will be widely ported over the next few years.

4.2.2. Security Association Negotiation

There are a variety of approaches to generating security associations between client and cryptoserver. The very simplest is a pure key exchange (*e.g.,* Diffie-Hellman) in order to produce a shared symmetric key used to encrypt further communication between client and server. This is the approach we took in our initial implementation, as we are not measuring key exchange performance. In a production server, we anticipate that session keys will be generated every 1–24 hours per client in actual use. If one uses a little care to make sure that keys expire uniformly across an hour, even with 10,000 clients and 1 hour session expirations, this implies 3 key agreements (*i.e.,* modular exponentiations) per second, or 1% of the capacity of a single AXL200. For a potential client, the value of using a cryptoserver will be determined by the cost of negotiating a security association in combination with the number of modular exponentiations across which that association can be amortized (which will be determined by the lifetime of the association and the rate of operations performed by the client). Even a client who performs only a very small number of cryptographic operations may find it worthwhile to use such a server in order to be able to choose when to perform that costly cryptography – at the time of security association negotiation, not at the time when the cryptographic operations themselves are required (which may be a time when the client is subject to many other demands).

To produce the performance numbers given below, we used the 192-bit Diffie-Hellman key agreement mechanism available with the distribution of TI-RPC. While this provides woefully inadequate security for production use [19], it is sufficient for the demonstration presented here.

For our symmetric cipher, we are using triple-DES. The performance of our overall system depends on the choice of symmetric cipher. Using triple DES is a very conservative choice as almost all other ciphers will offer better performance. The choice of symmetric cipher is determined by the GSS mechanism used. It would therefore be simple change to use of AES instead. Separating out the wire encryption in this way also allows us to consider use of bulk symmetric cryptographic accelerator hardware to handle communication on the server.

A variety of more interesting approaches to negotiating security associations are open to us. In the minimal case, a client would like to have assurance that the machine it is communicating with is indeed a trustworthy cryptoserver. Therefore, the server must be able to authenticate itself to the client. If the service is freely available, the client need not authenticate itself to the server (and indeed may want to remain anonymous). In order to provide this base level of functionality, we intend to use a public-key based GSS

```
program QCS_RPC_PROG {
  version QCS_RPC_VERS {
    QCS_value_res RPCMODEXP(QCS_mod_exp_coef, QCS_bignum) = 1;
    QCS_val_array_res RPCMODEXPARRAY(QCS_mod_exp_coef, QCS_bignum_array) = 2;
    QCS_value_res RPCRSAPCRTEXP(QCS_rsa_private_key, QCS_bignum) = 3;
    QCS_val_array_res RPCRSACRTARRAY(QCS_rsa_private_key,
                                     QCS_bignum_array) = 4;
    QCS_val_array_res RPCMULTIMODEXPARRAY(QCS_mod_exp_coef_array,
                                          QCS_bignum_array) = 5;
    int RPCGETMAXMODULUSLEN(void) = 7;
  } = 1;
} = 0x20000105;
```

Figure 2. RPC Interface to the cryptoserver

mechanism (such as SPKM [1]) and use a Certification Authority trusted by all clients to certify cryptoservers as such.

Further variants on authentication mechanisms would use client authentication to control access to a cryptoserver. PKI-based or Kerberos-based authentication mechanisms could be used to identify clients authorized to use the cryptoserver. Forms of digital cash could be used to allow clients to pay for cryptographic operations by both number of operations or quality of service (*e.g.*, speed, latency, etc) – clients could set up an account as part of security association negotiation, or could include payment tokens on a per-request basis.

4.2.3. Client Interface to the Cryptoserver

The client interface to the cryptoserver is designed to both allow sophisticated clients to most effectively use one or more such servers while minimizing network-related overhead, and to make it easy to incorporate cryptoserver support into legacy client packages such as OpenSSL and Microsoft's CryptoAPI with no changes required by programs which then in turn use those packages. The interface is also designed to allow requests to be passed through to the cryptographic hardware with a minimum of copying, and to be broken up in a variety of ways to most efficiently use any sort of accelerator hardware we may incorporate in the cryptoserver. We have packaged this interface as a C API implemented in a shared C++ library.

The interface to the cryptoserver is written in Sun's `rpcgen` RPC specification language. In Figure 2, we show a slimmed down version of specification; non-essential details about data types and benchmarking support have been removed. RPCMODEXP is a simple modular exponentiation. RPCMODEXPARRAY provides a more efficient way to encrypt multiple values with the same key. RPCRSAPCRTEXP and RPCRSACRTARRAY are the corresponding calls, but using Chinese remaindered exponentiation. Finally, RPCMULTIMODEXPARRAY provides a more communication efficient mechanism for raising multiple bases to multiple powers (modulo the corresponding moduli). RPCMODEXPARRAY and RPCRSACRTARRAY are particularly useful for non-RSA operations where there are no security issues with different keys and shared plaintext.

4.2.4. Server Program

We implemented the cryptoserver in C++. From a security point of view, we would prefer to implement the server in a safe language, such as Java. Unfortunately, there are four problems with this:

1. Lack of suitable middleware;
2. Poor performance of present Java compilers;
3. Difficulty of efficient interfacing to vendor libraries written in C;
4. Lack of a complete Unix system call interface.

These restrictions narrowed the choice of language to C or C++.

The server architecture can be seen in Figure 3. A configurable number of threads are responsible for decrypting and decoding incoming requests. As each request is decrypted, it is placed into a work item, which is moved onto the work queue. A pool of worker threads are preallocated to take requests from the work queue and hand them to a cryptographic accelerator for processing. Once the accelerator has finished processing that request, the worker thread moves the request onto the reply queue, and waits for the next request to appear on the work queue. There is a pool of replier threads (again, configurable in size) that takes work items off the reply queue, encrypts them, and returns the encrypted results to the requester. Empty work

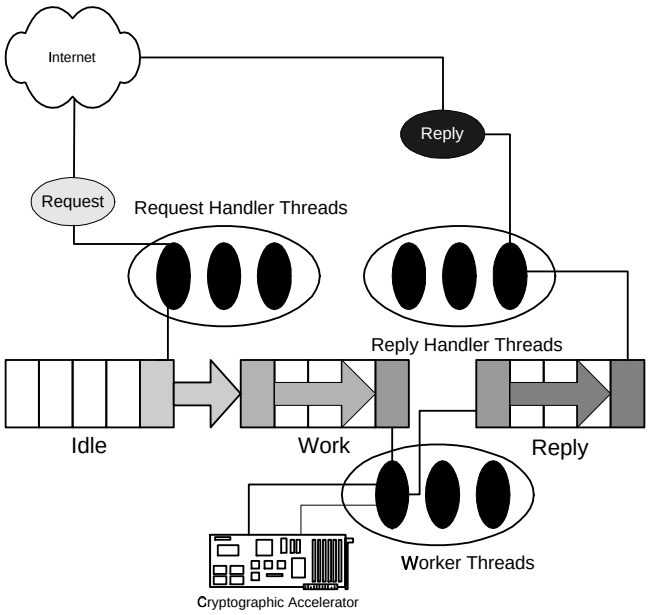

Figure 3. Cryptoserver Software Architecture

items are then placed in an idle queue to allow for object reuse.

Our software includes one additional important optimization: the RPCs that take array arguments are broken into multiple work items as they are placed on the work queue. This enables the separate operations to occur in parallel given our current cryptographic hardware. Of course, the RPC cannot return until all the results are available. This is implemented by making one work item be canonical per RPC request, and not moving the canonical work item onto the reply queue until all the operations are finished. The only time such automatic parallelization might be a disadvantage is if a hardware accelerator was able to take advantage of repeated use of the same key in order to speed up a group of operations. None of the accelerators we have worked with to date take advantage of any such options, but as the latencies of the individual cryptographic processors on these accelerator boards go down (see below), and the fraction of time spent in host-side preprocessing and copying of data goes up, such optimizations may become very important. Our architecture is flexible enough to take advantage of these optimizations when they become important.

4.2.5. Hardware Interface

In order to simplify the server, we use "shim" libraries to normalize the interface the server sees to each type of cryptographic hardware accelerator present in the system. Each individual hardware "shim" is responsible for any initialization required by the hardware it manages, and can provide information to the server about the capabilities of that hardware (*e.g.*, supported modulus lengths, whether the hardware driver supports features like negative numbers directly or the shim is providing that feature, etc.). Such information could be used by the server for more sophisticated scheduling of work items on particular accelerators. These shim libraries normalize byte order, handle support for negative numbers, exponents larger than the modulus, etc. Many hardware accelerators (and indeed cryptographic software packages) have intermittent or no support for such inputs, as they don't occur when using standard RSA. However, as technologies like the cryptoserver reduce the cost of modular exponentiations, cryptographic algorithms and protocols considered too costly and complex for practical use will be used, and these algorithms do not respect these narrow limits.

Normalizing the appearance of different types of hardware in this way allows the server to transparently support a heterogeneous collection of accelerators. It also lets us remove from the server any task-specific logic, and to compensate for any differences between the features supported by the accelerator drivers and the interface we present to cryptoserver clients.

Although we can normalize the software interface to each board, we cannot normalize the hardware itself. Each board contains a number of cryptographic coprocessors, and has a different degree of intrinsic parallelism. Because the coprocessors chosen to populate each type of board are different, each board has a characteristic latency

(time required for a single operation). This means that a single thread making sequential requests to a single board will see an exponentiation rate determined by that board's latency. The speed numbers quoted for each type of board represent throughput; the rate each coprocessor is capable of, multiplied by the number of processors on the board. Only a process sufficiently parallel to make full use of all the processors on a board will see the board's rated performance.

Each coprocessor also tackles the problem of modular exponentiation using a different algorithm and internal data format. The data formats will determine how much copying and re-ordering of input data must be done before a request can be sent to a coprocessor. The algorithm will determine how much preprocessing must be done in a host library to prepare the request for a coprocessor, and also how changes in input parameters will affect changes in coprocessor latency (and hence throughput). As an example, the choice of algorithm will control how the performance of the coprocessors changes as a function of exponent length, and hence how optimizations such as Chinese remaindering affect a board's performance.

In order to hide the complexity of scheduling multiple requests onto these parallel devices, cryptographic accelerator vendors provide a certain amount of host software. This usually consists of both libraries that take care of any necessary host-based pre- and post-processing of requests (for data formats, Montgomery reduction, etc), and programs/drivers that can manage requests from multiple client programs at once and schedule them for execution by the hardware coprocessors.

The cryptoserver must distribute requests across all of these board-specific schedulers. In order to maintain as much hardware independence as possible, we currently implement a very simple algorithm. We independently configure the number of worker threads the server uses to manage each cryptographic accelerator (or each pool of accelerators of a single type; frequently accelerator vendors provide one local interface to all of their accelerators that are present on a machine, and manage distributing requests across boards). This depends both on the inherent parallelism of each accelerator board, and on the architecture of the driver and any vendor libraries we use to interface with it. Each of these worker threads, when free, pulls a work item off the work queue and presents that item to the accelerator associated with that thread. Given our interest in experimenting with a variety of accelerator types, this simple and flexible scheduling algorithm makes sense. A high-volume production cryptoserver might benefit from a more sophisticated algorithm. If such a server is implemented using only a single type of cryptographic accelerator, this simple algorithm will end up relying on the scheduling algorithm implemented by the underlying vendor libraries, which is likely to have been optimally tuned for that class of device. A more interesting option would be to implement such a server using a collection of boards with different characteristics chosen to optimize performance across a variety of conditions (bursts of requests, high constant load, occasional single requests). Such a server would benefit from a more sophisticated scheduling algorithm.

To maximize parallelism, we must allocate at least as many worker threads to a board as there are cryptographic coprocessors on that board. It turns out that as you scale up the number of threads simultaneously making requests of a single board, performance improves sharply until the number of threads matches the number of coprocessors. As you increase the number of threads beyond this point, performance continues to improve slowly for a short period, and then plateaus. Given that at maximum throughput, at any given moment the number of threads blocked waiting for a response to return from a coprocessor should be equal to the number of coprocessors on a board, there is some room for further processing (and plenty of host CPU left) as additional threads can be preparing future requests for processing by the board (both copying data and doing any mathematical pre-processing), and doing any required post-processing of returned requests (again, both copying of data and mathematical post-processing, such as combining Chinese remaindered results). Allocating more threads than can be fully used by both pre- and post-processing and waiting on accelerators incurs little cost except at startup time; any such surplus threads simply remain blocked waiting for work. We therefore allocate a number of threads for each board slightly larger than the minimum necessary, based on the intrinsic parallelism of each board and experimental tests of how board performance scales beyond this minimum number.

At startup, the server takes an argument giving it either a single shim library name and number of threads (if it is to be run with a single board type for benchmarking purposes), or the name of a configuration file listing multiple shim libraries and corresponding thread counts. If there are multiple boards of a particular type present, the shim library handles that transparently; the number of threads allocated to that library must simply be scaled up to match.

5. Performance

Recall that our initial implementation of the cryptoserver is built around a Sun Ultra-10 workstation, containing a 440 MHz UltraSparc IIi processor, with one Atalla AXL200 cryptographic accelerator and one nCipher nFast 300 PCI. The Ultra-10 scores 18.1 on SPECint 95, the AXL200 has a maximum throughput of 265 1024-bit RSA operations per second (without Chinese remaindering), and the nCipher has a maximum throughput of

300 1024-bit RSA operations per second (93 ops/sec without Chinese remaindering). For benchmarking purposes, we ran cryptoserver clients on a dual processor, 250 MHz, 512 MB Sun UltraSparc (sans the Atalla board), connected via switched 100 Mbit/s Ethernet.

5.1. Microbenchmarks

In all our multithreaded benchmarks, all threads perform any necessary initialization code, and then line up waiting for a signal. The last thread to finish its initialization gets the start time, and signals all threads to start. As each thread finishes, it increments a counter. The last thread to finish gets the stop time, and measures operation rate as (operations per thread * thread count)/total time. If each thread measures its own elapsed time, the sheer amount of time spent on gettimeofday system calls starts to affect the overall measured value. Similarly, it becomes difficult to accurately measure the time taken by a single operation. We calculate average rates below by measuring the total time taken to have each thread perform 1000 1024-bit operations. Each such measurement of (number of threads * 1000 operations per thread) operations is considered one "block". Final rate values are generated by averaging the times for 1-8 blocks, and dividing by the total number of operations performed (block counts for each group of measurements are noted with those measurements). Throughput rates are given in operations/sec. As the variance of these averaged block times is low, and are noisy and difficult to interpret if presented as rates, we do not show variance data directly. All latencies are reported in milliseconds.

To measure the speed of each board individually when accessed locally, we wrote a multithreaded microbenchmark that repeatedly performs a modular exponentiations using our "shim" library for each board, and compares the result to the correct value. Each "shim" library (see Section 4.2.5) talks directly to a local accelerator via the board vendor's user libraries. This measurement provides an estimate of the baseline speed of the accelerator board, and an independent confirmation of the speed rating by the manufacturer. Each number below is in 1024-bit operations/sec, measured by having each thread perform one block of 1000 operations (numbers rounded to 2 decimal places).

Based on measurements like those above, latency information given below, and information from the vendors, we know that the Atalla board has 26 cryptographic coprocessors, while the nCipher board is using around 10. As noted above, performance continues to improve slowly as the thread count is increased beyond the number of coprocessors. We therefore ran the tests below with 30 threads devoted to each accelerator board. (In all cases, 5 additional threads were devoted to processing requests and, and another 5 to replies.) We do not present numbers for the Atalla board alone using the Chinese Remainder Theorem (CRT). The time taken by the Atalla board to process a modular exponentiation rises linearly with the number of bits in the exponent. There is therefore no overall advantage to using CRT on a loaded Atalla board. We do use CRT support on the Atalla board to allow requests to be parallelized at low load levels, and to support 2048-bit moduli. In order to simplify presentation of results, such support was disabled for the tests presented here. The normal Atalla libraries simply ignore CRT coefficients and process the private exponent directly.

Board	Threads	no CRT	w/CRT
AXL200	1	10.85	
AXL200	25	265.98	
nFast 300	1	11.67	37.00
nFast 300	10	92.40	288.92
nFast 300	30	93.28	297.74
nFast 300	32	93.30	299.97

Table 1. Local Accelerator Throughput (ops/sec)

We report below the latencies corresponding to the single-threaded measurements reported above. We then report results for the same computation performed by a multithreaded client of the cryptoserver. Measurements on the cryptoserver client are averages of 8 blocks (3 for nCipher with CRT) of 1000 operations per thread, the variances are quite small. The cryptoserver was run in 3 configurations: with the Atalla board alone, with the nCipher board alone, and with both boards. This allows more direct comparison to the local latencies measured on the single boards. In the ideal case, the throughput for the 2-board configuration should be the sum of the throughputs for each board used alone. In the presentation of the single-threaded measurements below, we present measurements for the first two cryptoserver configurations alone (a single-threaded client accessing a server controlling both boards will simply see the performance of that board that got to its requests first) We report numbers both with and without securing the wire with triple DES.

The table below lists the performance of the cryptoserver using a multithreaded client application. Results are divided according to whether the server was managing just the nCipher board, just the Atalla board, or both boards. Results are also divided according to whether wire encryption was turned on, and the type of request placed by the client (with or without CRT, a single request per RPC or a multiple request – a batch of 3 requests per RPC). The number of client threads was chosen in an attempt to maximize throughput.

Without wire encryption, our software delivers the full throughput of the accelerators. Adding triple-DES incurs

Machine/Board	Threads	Latency (ms)	Throughput (ops/s)	Latency w/CRT	Throughput w/CRT
Local accelerator:					
AXL200	1	92.19	10.85		
nFast 300	1	85.68	11.67	27.03	37.00
Remote Cryptoserver:					
AXL200 only (insecure)	1	93.08	10.74		
AXL200 only (secure)	1	94.12	10.62		
nFast 300 only (insecure)	1	86.78	11.52	28.27	35.37
nFast 300 only (secure)	1	88.12	11.35	29.86	33.49

Table 2. Single-Threaded Performance

a 2% reduction in throughput. When the accelerator is managing a single board alone, the SPARC Ultra-10 used as a server has no trouble keeping that board fully loaded. Even then, the load on both client and server is very low. When the cryptoserver manages both boards together, the demands of managing input and output, as well as the host side pre- and post-processing required by each board, begin to outstrip our current server host. While processing 1024-bit RSA requests using CRT, we cannot keep up with the total throughput of both boards (approximately 565 ops/sec) without either turning off wire encryption or batching requests in groups of 3. Attempts to do so saturated the cryptoserver's CPU (note that at the same time, the load on the client machine was still extremely low). Operating at these high host loads also increased the variability of the result somewhat, to the degree that the difference between secure and non-secure trials began to blur. This suggests that to scale beyond these two boards will require a faster or more parallel server host, offloading of the wire encryption to a symmetric cryptographic coprocessor, or both.

In spite of this difficulty, in general the penalty for using the cryptoserver is less than 3%. The fact that the client load remained extremely low, even when that one client machine was pushing both boards at full speed, reinforces the value of the cryptoserver merely in its role to offload processing, without beginning to consider speed benefits offered by increased parallelism and (possibly) faster single cryptographic operations.

5.2. TLS Performance

Microbenchmarks help us characterize the details of cryptoserver performance, but leave many unanswered questions as to how the cryptoserver performs with real world tasks. We decided to benchmark the cryptoserver with a client supporting the TLS protocol to get a better understanding of how a cryptoserver might accelerate a secure Web server. We wrote a small, multithreaded server that uses OpenSSL to respond to HTTP HEAD requests with a fixed string. We had a benchmark client written by

Cryptoserver Configuration	Threads	Throughput
AXL200 only (insecure)	26	265.73
AXL200 (secure)	26	265.58
nFast 300 only (insecure)	26	93.20
nFast 300 (secure)	26	93.21
nFast 300 (insecure, CRT)	26	299.41
nFast 300 (secure, CRT)	26	299.34
Both (insecure)	70	354.01
Both (secure)	70	354.17
Both (insecure, multi)	30	355.50
Both (secure, multi)	30	355.55
Both (insecure, CRT)	70	560.60
Both (insecure, CRT, multi)	30	563.94
Both (secure, CRT, multi)	30	562.25

Table 3. Multi-Threaded Performance

Dan Boneh, Michael Malkin, and Tom Wu that generates HTTP HEAD requests over a TLS connection.

While these benchmarks programs are somewhat artificial, they are small enough that we can easily understand their behavior. The client opens a TLS connection (specifically not resuming a prior connection), and sends a 19 byte HTTP HEAD request. The server replies with a 107 byte answer, and closes the connection. This is nearly the worst case for TLS, as we are exchanging very little data per RSA operation. However, since RSA performance is exactly what we are trying to characterize, this is exactly what we want.

The experimental setup is the same as before. For simplicity of presentation, we only measured the AXL200 board. The results are shown in Table 4. In all cases,

Cryptoserver Configuration	Throughput (connections/s)
Local AXL200	187.38
Remote AXL200 (secure)	244.68
Remote AXL200 (insecure)	261.64

Table 4. TLS Performance

the TLS benchmark server ran with 30 threads and the TLS benchmark client ran with 40 threads, *i.e.,* it could request up to 40 concurrent RSA operations on the TLS server. With the TLS server running locally, the server's CPU saturated. By using a remote cryptoserver, throughput actually increased, as we are able to take advantage of the available parallelism. The insecure connection to the cryptoserver enabled the client to use $> 98\%$ of an AXL200's maximum throughput; with a secure connection, the client exceeded 92% of an AXL200's maximum throughput.

6. Related Work

Network-attached cryptographic acceleration has only been used in special cases so far. Rainbow Technologies has sold products in their CryptoSwift EN line for several years, but its network connectivity is not secured. This makes it only appropriate for use on trustworthy networks. In contrast, our approach is suitable for deployment on any network with suitable availability – 1000 1024-bit RSA operations per second requires approximately 4 Mbit/s of bandwidth; hardly a problem with common 100 Mbit/s Ethernet infrastructure.

Note that in our approach the client must trust the cryptoserver with knowledge of his private key, and thus our approach is quite different from the harder and generally unsolved problems of "server-aided cryptography," "remotely-keyed encryption" [2, 3, 22] or "computing with encrypted data" [14]. We do this for practical reasons, as we seek performance levels as close to available hardware as possible. If truly practical server-aided cryptographic techniques become available, then we are ideally positioned to accommodate them.

7. Future Work

Viewing cryptography as a network service changes our perspective about the costs of cryptography. Cryptography is no longer computationally prohibitive; it is only an RPC away. We are building applications using abundant public key operations, including secure communication services for dynamic coalitions, private database retrieval, and others. We also plan to build cryptoserver clients implementing standard cryptographic APIs such as PKCS #11, Microsoft's CryptoAPI, and the Java Cryptography Environment. This will allow legacy applications to seamlessly take advantage of the cryptoserver. We may also examine other choices of implementation platform and middleware, increase the flexibility and usability of the server.

A remaining challenge is to see how well our software architecture scales. While the software was designed with scalability in mind (*e.g.,* using a fixed thread pool, accommodating inter- and intra-request parallelism, multiple request and reply handler threads to spread the symmetric cryptographic load, etc.), the proof will be actually running thousands of modular exponentiations per second on a suitable machine. This challenge will only increase as the speed of accelerators increases and their latency for single operations goes down.

In our implementation, each request includes the client's private key. Alternatively, if the cryptoserver already knows the client's private key, then the request may include an authentication token that demonstrates who the request is coming from and that the request is fresh. While this would require a more trustworthy cryptoserver, it would reduce the network bandwidth required by nearly half. There are several cryptographic accelerator products on the market that will maintain secure local storage of one or more private keys, and control access to them. It would be a simple matter to provide shared network access to such an accelerator in the manner presented above.

The cryptoserver offers interesting options for those paranoid about their cryptographic operations. As our server supports a heterogeneous collection of hardware accelerators running concurrently, it would be a fairly simple modification to use one accelerator to check the results delivered by another. By using different accelerators, a single accelerator could not produce a doctored result along with a doctored "inverse." The tradeoff between paranoia and throughput could be easily managed by checking a user-selectable fraction of results. By selecting hardware accelerators designed and manufactured in disjoint countries, no single government would be in a position to compromise a RSA operation. Such a system would be highly resistant to many attacks, including fault injection [4].

A similar level of paranoia is available to clients, as discussed in Section 3: it is easy for the client code to issue RPCs to more than one server. One might, for example, use servers operated by different organizations, or servers physically in multiple countries, to cross-verify results. This ability to use multiple cryptoservers also makes it very easy for clients to protect themselves against malicious servers through the use of threshold cryptographic techniques as discussed in Section 2.

8. Conclusion

We have demonstrated that public key cryptography can be provided as a service over untrusted networks. This architecture has many advantages: it offloads work from clients, it allows greater utilization of cryptographic accelerators by sharing them among many clients, and it has acceptably small performance overhead. In addition, it enables new security applications that were previously considered too costly. Our implementation consists of cus-

tomized software on top of generally available hardware. Benchmark data indicate that our approach is fast and effective. Hardware trends and other factors indicate that our approach will be increasingly attractive over time.

Acknowledgments

We thank Teresa Lunt for useful discussion about this work. We thank Larry Hines at Atalla for providing additional information about the AXL200. We thank Jessica Nelson at nCipher for technical assistance with the nFast 300. We thank the anonymous referees for helpful comments on an earlier version of this paper.

References

[1] C. Adams. RFC 2025: The simple public-key GSS-API mechanism (SPKM), Oct. 1996.

[2] M. Blaze. High-bandwidth encryption with low-bandwidth smartcards. In *Proceedings of the Fast Software Encryption Workshop*, number 1039 in Lecure Notes in Computer Science, pages 33–40. Springer-Verlag, 1996.

[3] M. Blaze, J. Feigenbaum, and M. Naor. A formal treatment of remotely keyed encryption. In K. Nyberg, editor, *Proceedings of EUROCRYPT'98*, number 1403 in Lecure Notes in Computer Science, pages 251–265. Springer-Verlag, 1998.

[4] D. Boneh, R. DeMillo, and R. Lipton. On the importance of checking cryptographic protocols for faults. In *Proceedings of Eurocrypt '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 1997.

[5] C. Boyd. Digital multisignatures. In H. Beker and F. Piper, editors, *Cryptography and Coding*, Institute of Mathematics and Its Applications (IMA), pages 241–246. Clarendon Press, 1989.

[6] V. Boyko, M. Peinado, and R. Venkatesan. Speeding up discrete log and factoring based schemes via precomputation. In K. Nyberg, editor, *Advances in Cryptology – EUROCRYPT '98*, number 1403 in LNCS, pages 221–235, Espoo, Finland, 1998. Springer-Verlag.

[7] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of EUROCRYPT '99*, pages 402–414, 1999.

[8] D. Chaum. Blind signatures for untraceable payments. In R. L. Rivest, A. Sherman, and D. Chaum, editors, *Proc. CRYPTO 82*, pages 199–203, New York, 1983. Plenum Press.

[9] D. Chaum and E. van Heyst. Group signatures. In D. W. Davies, editor, *Advances in Cryptology—EUROCRYPT 91*, volume 547 of *Lecture Notes in Computer Science*, pages 257–265. Springer-Verlag, 8–11 Apr. 1991.

[10] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the Association for Computing Machinery*, 45(6):965–981, Nov. 1998.

[11] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Y. G. Desmedt, editor, *Proc. CRYPTO 95*, pages 174–187. Springer, 1994. Lecture Notes in Computer Science No. 839.

[12] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Proc. CRYPTO 89*, pages 307–315. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[13] M. Eisler, A. Chiu, and L. Ling. RFC 2203: RPCSEC_GSS protocol specification, Sept. 1997.

[14] J. Feigenbaum. Encrypting problem instances: Or...can you take advantage of someone without having to trust him? In H. C. Williams, editor, *Proc. CRYPTO 85*, pages 477–488. Springer, 1986. Lecture Notes in Computer Science No. 218.

[15] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 151–160, New York, May 23–26 1998. ACM Press.

[16] B. Huberman, T. Hogg, and M. Franklin. Enhancing privacy and trust in electronic communities. In *Proceedings of the ACM Conference on Electronic Commerce*, 1999.

[17] M. Jakobsson, K. Sako, and R. Impagliazzo. Designated verifier proofs and their applications. In U. Maurer, editor, *Advances in Cryptology—EUROCRYPT 96*, volume 1070 of *Lecture Notes in Computer Science*, pages 143–154. Springer-Verlag, 12–16 May 1996.

[18] P. Kakkar, C. A. Gunter, and M. Abadi. Reasoning about security for active networks. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 118–129, Cambridge, UK, July 2000.

[19] B. A. LaMacchia and A. M. Odlyzko. Computation of discrete logarithms in prime fields. *Designs, Codes, and Cryptography*, 1:47–62, 1991.

[20] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. In *Proceedings of the Public Key Cryptography Conference 2000*, Jan. 2000. Available from `http://www.cryptosavvy.com`.

[21] J. Linn. RFC 2743: Generic security service application program interface, version 2, update 1, Jan. 2000.

[22] S. Lucks. Accelerated remotely keyed encryption. In L. Knudsen, editor, *Proceedings of the Fast Software Encryption Workshop*, number 1636 in Lecure Notes in Computer Science, pages 112–123. Springer-Verlag, 1999.

[23] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.

[24] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *Proceedings of the Network and Distributed Systems Security Symposium*. Internet Society, 1999.

[25] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet — the popcorn project. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 592–601, 1998.