Improving DES Coprocessor Throughput for Short Operations

Mark Lindemann
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598-0704
mjl@us.ibm.com

Sean W. Smith*
Dept. of CS/Institute for Security and Technology Studies
Dartmouth College
Hanover, NH 03755
sws@cs.dartmouth.edu, http://www.cs.dartmouth.edu/~sws/

Abstract

Over the last several years, our research team built a commercially-offered secure coprocessor that, besides other features, offers high-speed DES: over 20 megabytes/second. However, it obtains these speeds only on operations with large data lengths. For DES operations on short data (e.g., 8-80 bytes), our commercial offering was benchmarked at less than 2 kilobytes/second. The programmability of our device enabled us to investigate this issue, identify and address a series of bottlenecks that were not initially apparent, and ultimately bring our short-DES performance close to 3 megabytes/second. This paper reports the results of this real-world systems exercise in hardware cryptographic acceleration—and demonstrates the importance of, when designing specialty hardware, not overlooking the *software* aspects governing how a device can be used.

1 Introduction

What *is* "fast DES?" The challenge of *meaningfully* quantifying cryptographic performance has been a long-standing issue.

Over the past several years, our team has worked on producing, as a commercial offering, a cryptographic embedded system: a high-performance, programmable secure coprocessor platform [9], which could take on different personalities depending on the application program installed. This device featured hardware crypto support for modular math and DES in the original version, with outer-CBC TDES and SHA-1 added in the Model 2. Our initial commercial target was an application program [1] that turned the platform into a secure cryptographic accelerator.

Besides the physical and logical security of the device, our team prided itself on the fast DES (and, in the Model 2, outer-CBC TDES) that our device provided. Measured from an application program on the host (in order to give a more accurate figure), our initial device performed DES at about 20 megabytes/second; the follow-on does outer-CBC TDES at close to this rate. We note, however, that we were focused on *secure coprocessing*, and wanted fast DES in contexts where the keys and decisions were under the control of the trusted third party inside the box, not the less secure host. Two potential examples of such scenarios include re-encryption of a hardware-protected Kerberos database [3], and information servers that ensure privacy even against root [8].

However, these figures were for *bulk* performance: operations consisting of CBC encryption or decryption of input data that is itself megabytes long. For operations on short data, our device was *several orders of magnitude slower.* When an external colleague—who required large numbers of DES operations on inputs each 8-80 bytes—benchmarked our commercial offering, he only measured about 1.5 kilobytes/second. [5]

The programmability of our device enabled us to investigate this issue, and we assumed that our intimate knowledge of the internals would enable us to immedi-

*This work was supported in part by the U.S. Department of Justice, contract 2000-DT-CX-K001.

ately identify and rectify the bottleneck. This assumption turned out to be incorrect. In this paper, we report the lengthy sequence of experiments that followed. We finally improved short-DES performance by three orders of magnitude over the initial benchmark, but have been continually surprised at where the bottlenecks really were.

We offer this contribution as a real-world systems exercise in cryptographic acceleration. It demonstrates the value of programmability in a cryptographic accelerator—because without this flexibility, we would not have achieved the three orders of magnitude speedup. More importantly, it demonstrates the importance of considering *how a system will actually be used*, and *how the control data will be routed*, when designing specialty cryptographic hardware. Far too often, the hardware design process leaves these issues for post-facto software experimenters (like ourselves) to discover. Consequently, our work also offers some potential lessons for future design of hardware intended to accelerate high-latency operations on small data lengths, as well as for the future design process.

2 System Background

Our device is a multi-chip embedded module, packaged in a PCI card. In addition to cryptographic hardware, and circuitry for tamper detection and response, we have a *general-purpose computing environment*: a 486-class CPU, executing software loaded from internal ROM and FLASH. Two generations of the device exist commercially; the older Model 1 and the newer Model 2. We did our experiments on the Model 2 (since that is all we had); discussions of principles that apply to both models do not specify a model number.

2.1 Software

The multiple-layer software architecture consists of foundational security control (Layer 0 and Layer 1), supervisor-level system software (Layer 2), and user-level application software (Layer 3). (See Figure 1.)

Our Layer 2 component [2] was designed to support application development. Within Layer 2, a kernel provides standard OS abstractions of multiple tasks and multiple address spaces; these abstractions support independent *managers:* components within Layer 2 which

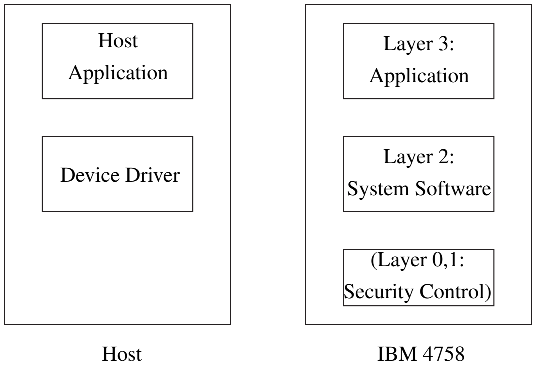

Figure 1 The software architecture of the coprocessor. The host software on the left runs on the host system; the card software on the right runs on the 486 inside the coprocessor.

handle cryptographic hardware and other I/O on the bottom, and provide higher-level APIs to the Layer 3 application on top.

Typically, this Layer 3 application provides the abstraction of its own API to host-side application. Figure 2 through Figure 4 shows the interaction of software components during applications such as standard DES acceleration:

(Figure 2) When it wants to use a service provided by the card-side application, the host-side application issues a call to the host-side device driver. The device driver then opens an `sccRequest` to the Layer 2 system software on the device. Layer 2 then informs the Layer 3 application resident on the device of the existence of this request, and some of the parameters the host sent along with it.

(Figure 3) The Layer 3 application then handles the host application's request for service; in this example, it directs Layer 2 to transfer data and perform the necessary crypto operations.

(Figure 4) The Layer 3 application then directs Layer 2 to close out the `sccRequest` and send the results back to the host.

2.2 Hardware

One of the many goals of our device was fast cryptography. As part of this goal, we included a FIFO/state machine structure that can transport data quickly into and out of an algorithm engine. Figure 5 shows how this pro-

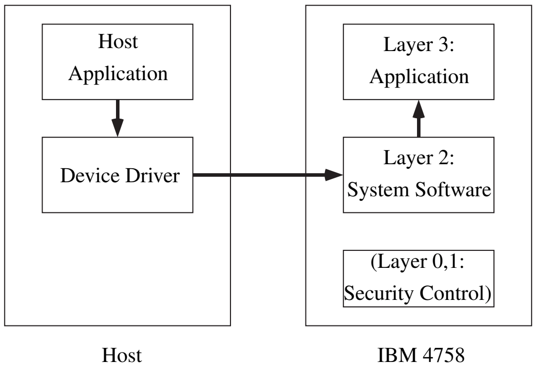

Figure 2 The host application opens an `sccRequest` to the application layer in the card.

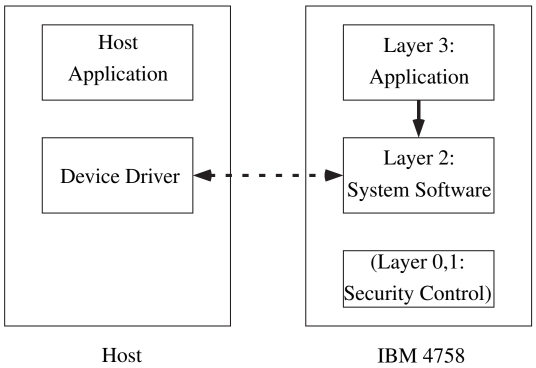

Figure 3 For standard external-external DES, the application layer asks Layer 2 to perform the operation; Layer 2 then directs the the data transfer.

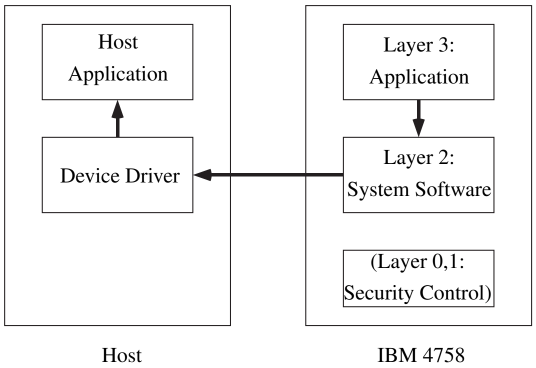

Figure 4 The application layer closes out the `sccRequest`, and sends the output back to the host application.

prietary FIFO structure works with the DES/TDES engine. (In our Model 2 hardware, this FIFO structure also supports fast SHA-1; in principle, this structure could be applied to any algorithm engine.)

For both input and output, we have two pairs of FIFOs—a *PCI FIFO* and an *internal FIFO*, for fast external and internal data transfer, respectively. We also have a *DMA controller*, for CPU-free transfer into and out of internal DRAM. These components enable the device CPU to arrange to do fast data transfer through the various on-board devices, without the active involvement of the CPU after the initial configuration. For example, to support fast bulk DES when the source and destination are both outside the device, the internal CPU can configure these components to support an external-to-external data path (PCI Input FIFO to Internal Input FIFO to DES, then back through the output FIFOs), load the relevant operational parameters (e.g., key, IV, mode) into the DES engine, and then let the the hardware move data through on its own.

Besides external-to-external DES, other common configuration paths include internal-to-internal bulk DES (Output DMA to Internal Input FIFO to DES, then back), and DMA transfer (e.g., PCI Input FIFO to Internal Input FIFO to Input DMA and vice versa). (Additionally, the DES hardware can be configured in bypass mode, but the commercial Layer 2 software does not use it.)

As an artifact of the hardware design, we have one principal constraint: *both* internal FIFO-DES paths must be selected (*bulk mode*), or *neither* must be selected (*non-bulk mode*).

However, changing between these modes resets the Internal FIFOs, and during non-bulk mode, the CPU has no way to restrain the Internal Input FIFO from filling to capacity.

Examples Figure 6 through Figure 10 show some examples of how the FIFO hardware supports card applications.

- (Figure 6) When the host application opens up an `sccRequest` to the card application, the card typically brings the input data into a DRAM buffer via DMA.

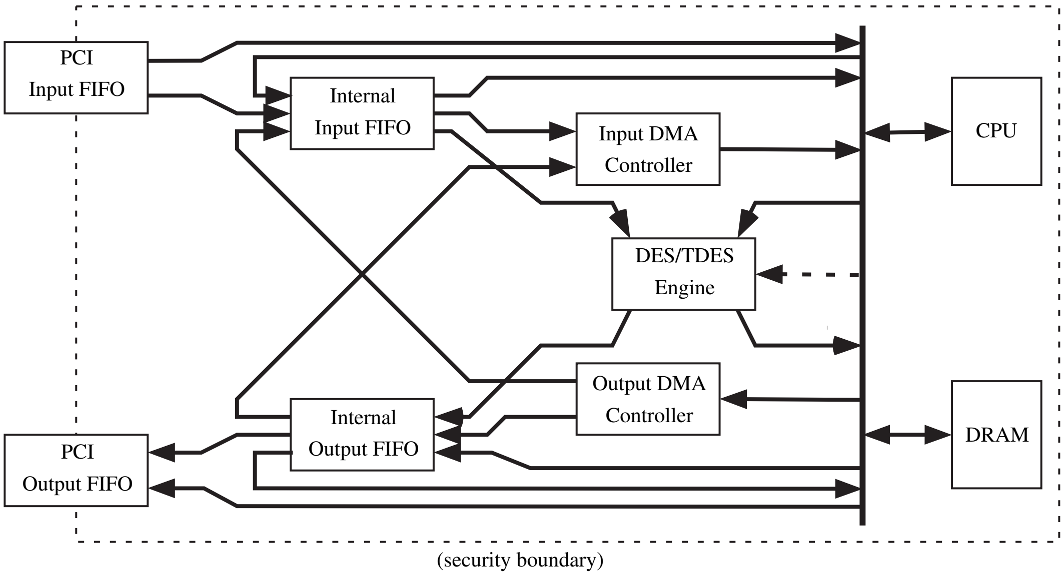

Figure 5 The FIFO structure supporting DES/TDES, within the coprocessor.

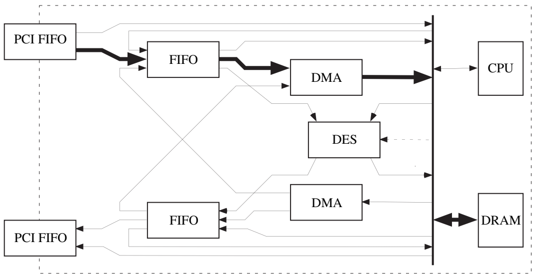

Figure 6 The bold arrows show how the internal CPU can configure the FIFOs to bring data into the card via DMA.

- (Figure 7) For a DES request, the card may then transfer the operational parameters from DRAM into the DES chip.
- (Figure 8) If the DES request is for external-external DES, the card will then configure the FIFOs to bring the data in from the host, through the DES chip (operating with the parameters we just loaded), then back to the host.
- (Figure 9) If the DES request is for internal-internal DES (but is too short to justify DMA), the card may just manually push the bytes through.
- (Figure 10) When the `sccRequest` is complete, the card may send the results back out to the host via DMA.

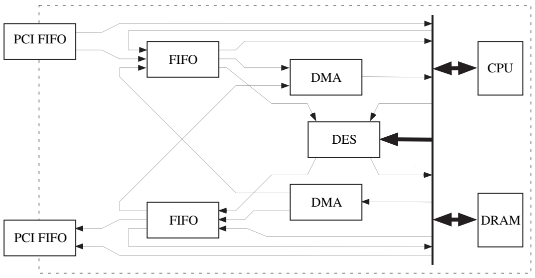

Figure 7 The bold arrows show how the internal CPU can load operational parameters into the DES chip from DRAM.

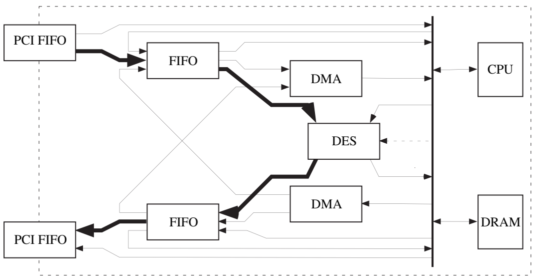

Figure 8 The bold arrows show how the internal CPU can configure the FIFOs to stream data from the host, through the DES chip, then back out to the host.

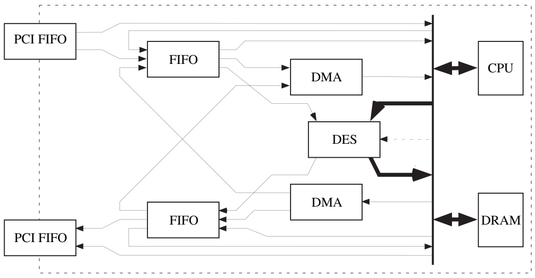

Figure 9 The bold arrows show how the internal CPU can also drive data from DRAM through DES via programmed I/O.

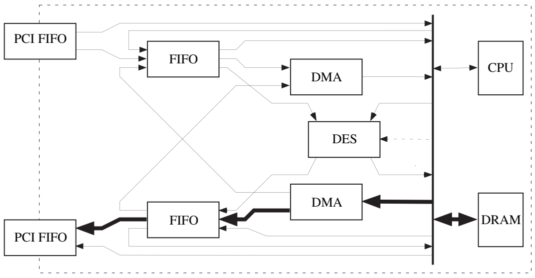

Figure 10 The bold arrows show how the internal CPU can configure the FIFOs to send data from DRAM back the host via DMA.

3 The Experiment Sequence

This unfunded "skunkworks" project had several goals: to try to see why the huge gap existed between what a colleague (using slower Model 1 hardware) measured for short-DES and what we measured for longer bulk DES; to try to improve the performance, if possible; and to explore migration of these changes (if the performance improves significantly) back into our commercial Layer 2 software (e.g., via some new "short-DES" API it provides to Layer 3).

But as a side-effect, we had a constraint: due to funding limitations (that is, zero funding) and the long-term goal of product improvement, we had to minimize the number of components we modified. For example, modifying the host device driver, even just to enable accurate latency measurements, was not feasible; and any solution we considered needed to be a small enough delta that a reasonable chance existed of moving it into the real product.

Since the colleague's database application (as well as the general nature of the problems to which we apply our secure coprocessing technology) required no exposure of key material, we did not measure host-only DES.

3.1 The Gauntlet is Thrown

Our colleague prompted this work when he demonstrated just how poorly our device performed for his application. Thus, to start our investigation, we needed to nail down the nature of the "DES" performance that he benchmarked at approximately 1.5 kilobytes/second.

This figure was measured from the host-side application program (recall Figure 1), using commercial Model 1 hardware with the IBM *Common Cryptographic Architecture (CCA)* application in Layer 3. (CCA also inserts a middle layer between the host application and the host device driver).

The DES operations were CBC-encrypt and CBC-decrypt, with data sizes distributed uniformly at random between 8 and 80 bytes. The IVs and keys changed with each operation; the keys were TDES-encrypted with a master key stored inside the device. Encrypted keys, IVs, and other operational parameters were sent in with each operation, but were not counted as part of the data throughput. Although the keys may change with each operation, the total number of keys (in our colleague's application, and in others we surveyed) was still fairly small, relative to the number of requests.

Experiment 1: Establishing a Baseline

Idea. We first needed to establish a baseline implementation that reproduced our colleague's set-up, but in a setting that we could instrument and modify. Our colleague used commercial Model 1 hardware and CCA; in our lab, we had neither, but we did have Model 2 prototypes. So, we did our best to simulate our colleague's configuration.

Experiment. We built a host application that generated sequences of short-DES requests (cipherkey, IV, data); we built a card-side application that: caught each request; unpacked the key; sent the data, key, and IV down to the DES engine; then sent the results back to the host. Figure 11 shows this operation.

Results. With this faster hardware (and lighter-weight software) than our colleague's set-up, we measured 9-12 kilobytes/second (with the speed decreasing, oddly, as the number of operations increased).

We chose keys randomly over a small set of cipherkeys. However, caching keys inside the card (to reduce the extra TDES key-decryption step) did not make a significant performance improvement in this test.

Experiment 2: Reducing Host-Card Interaction

Idea. Within our group, well-established folklore taught that each host-card interaction took a huge amount of time. Consequently, we first hypothesized that the reason short DES was so much slower than longer DES was because of the much greater number of host-card interactions (one set per each 44 bytes of data, on average) that our short-DES implementation required.

Experiment. We re-wrote the host-side application to batch a large sequence of short-DES requests into one `sccRequest`, and then re-wrote the card-side application to: receive this sequence in one step; process each request; and send the concatenated output back to the host in one step. Figure 12 shows this operation.

Results. We tried a several data formats here. Speeds ranged from 18 to 23 kilobytes/second (and now up to 40 kilobytes/second with key caching). This approach was an improvement, but still far below the apparent potential—host-card interaction was not the killer bottleneck.

Experiment 3: Batching into One Chip Operation

Idea. Another piece of well-established folklore taught that resetting the DES chip (to begin an operation) was expensive, but the operation itself was cheap. Until now, we had been resetting the chip for each operation (again, once per 44 bytes, on average).

Our next step was to see how fast things would go if we eliminated these resets.

Experiment. For purposes of this experiment, we generated a sequence of short-DES operation requests that all used one key, one direction ("decrypt" or "encrypt"), and IVs of zero (although the IVs could have been arbitrary). Our card-side application now received the operation sequence and sent it all down to the Layer 2 software. In Layer 2, we rewrote the DES Manager (the component controlling the DES hardware) to set up the chip with the key and an IV of zero, and to start pumping the data through the chip. However, at the end of each operation, our modified Manager did the proper XOR to break the chaining. (E.g., for encryption, the software manually XOR'd the last block of ciphertext from the previous operation with the first block of plaintext for the next operation, in order to cancel out the XOR that the chip would do.)

Results. Much to our surprise, we now measured as high as 360 kilobytes/second. Was DES-chip reset the killer bottleneck?

Distrusting folklore, we modified the experiment to reset the DES chip forh each operation anyway, and the top-end speed dropped slightly, to 320 kilobytes/second. So, it wasn't the elimination of chip resets that was saving time here.

Experiment 4: Batching into Multiple Chip Operations

Idea. How many Layer 3-Layer-2 context switches are necessary to handle the host's batched operation request?

Besides reducing the number of chip resets, the one-reset experiment of Experiment 3 also reduced the context switches from $O(n)$ to $O(1)$ (where n is the number of operations in the batch). The good performance of the multi-reset variant suggested that perhaps these context switches were a significant bottleneck.

Experiment. We went back to the multi-key, non-zero-IV set-up of Experiment 2, except now the card-side application sends the batched requests down to a modified DES manager, which then processes each one (with a chip reset and new key and IV each time). Figure 13 shows this operation.

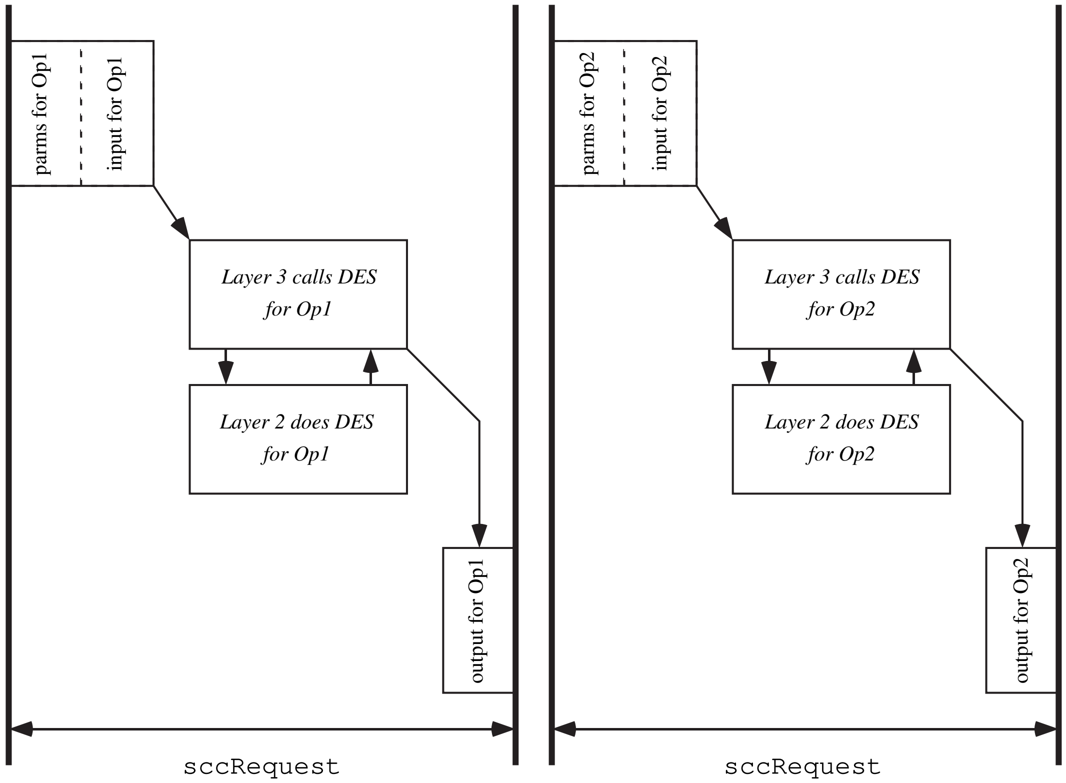

Figure 11 Experiment 1: the application handles each operation as a separate `sccRequest`, with PIO DES.

Results. Speeds ranged from 30 - 290 kilobytes/second.

However, something was still amiss. Each short DES operation requires a minimum number of I/O operations: to set up the DES chip, to get and set up the IV, to get and set up the keys, and then to either drive the data through the chip, or let the FIFO state machine pump it through.

Extrapolating from this back-of-the-envelope sketch to an estimated speed is tricky, due to the complex nature of contemporary CPUs. However, the sketch suggested that multi-megabyte speeds should be possible.

Experiment 5: Reducing Data Transfers

Idea. From our above analysis of what's "minimally necessary" for short-DES, we realized that we were wasting a lot of time with parameter and data transport. In practice, each byte of cipherkey, IV, and data was being handled many times. The bytes came in via FIFOs and DMA into DRAM with the initial `sccRequest` buffer transfer; the CPU was then taking the bytes out of DRAM and putting them into the DES chip; the CPU then took the data out of the DES chip and put it back into DRAM; the CPU then sent the data back to the host through the FIFOs.

However, in theory, each parameter (key, IV, and direction) should require only one transfer: the CPU reads it from the FIFO, then acts. If we let the FIFO state machine pump the data bytes through DES in bulk mode, then the CPU never need handle the data bytes at all.

Experiment. Our next sequence of experiments focused on trying to reduce the number of transfers down to this minimal level.

To simplify things (and since we were starting to try to converge to a "fast short-DES" API), we decided to eliminate key unpacking as a built-in part of the API—since each application has their own way of doing unpacking anyway, and the cost impact was small (for operation sequences distributed over a small number of keys, as we had assumed). Instead, we assumed that, within each application, some "initialization" step would conclude with a plaintext key-table resident in device DRAM. We also decided to standardize operation

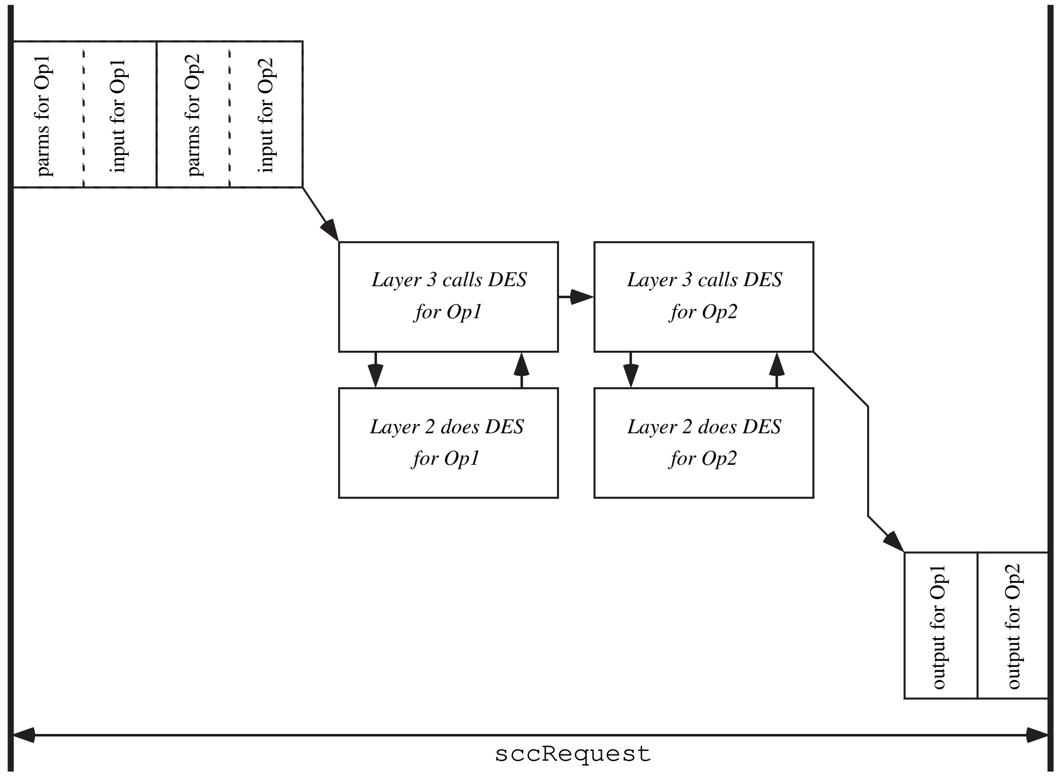

Figure 12 Exp 2: we reduced host-card interaction by batching all the operations into a single `sccRequest`.

lengths to 40 bytes (which, in theory, should mean that the speeds our colleague would see will be 10% higher than our measurements).

We rewrote our host application to generate sequences of requests that each include an index into the internal key-table, instead of a cipherkey. Our card-side application now calls the modified DES Manager (and makes the key table available to it), rather than immediately bringing the request sequence from the PCI Input FIFO into DRAM. For each operation, the modified DES Manager then: resets the DES chip; reads the IV and loads it into the chip; reads (and sanity checks) the key index, looks up the key, and loads it into the chip; reads the data length for this operation; then sets up the state machine to crank that number of bytes through the input FIFOs into the DES chip then back out the output FIFOs.

Figure 14 shows this operation.

Results. Speeds now ranged up to 1400 kilobytes/second.

Experiment 6: Using Memory Mapped I/O

The approach of Experiment 5 showed a major improvement, but performance was still lagging behind what we projected as possible.

Idea. Upon further investigation, we discovered that, in our device, I/O operation speed is not limited by the CPU speed but by the internal ISA bus (effective transfer speed of 8 megabytes/second) When we calculated the number of fetch-and-store transfers necessary for each operation (irrespective of the data length), the slow ISA speed was the bottleneck.

Consequent discussions with the hardware engineers revealed that every I/O register we needed to access—except for the PCI FIFOs—was available from a location that was also memory-mapped—and memory-mapped I/O operations should not be subject to the ISA speed limitations.

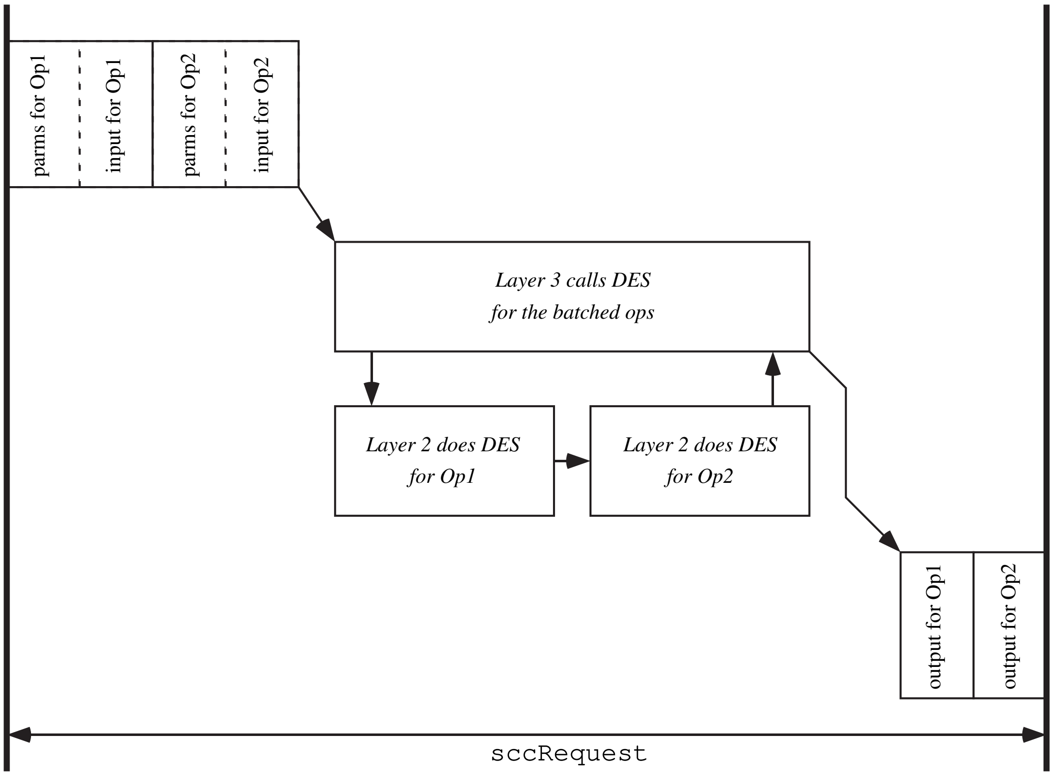

Figure 13 Exp 4: We reduced internal context switches by batching all the operations into a single call to a modified DES Manager in Layer 2.

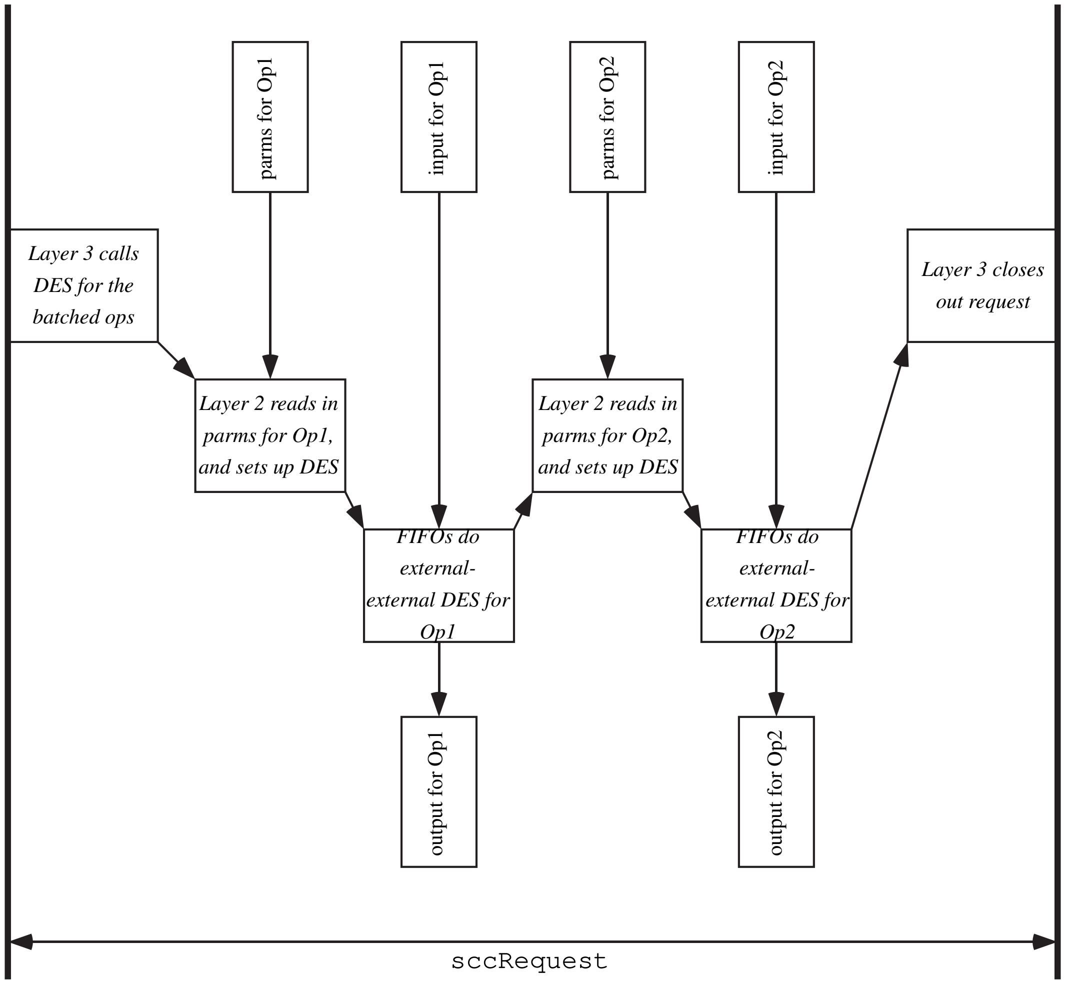

Figure 14 Experiment 5, Experiment 6: We reduce unnecessary data transfers by having the modified DES Manager, for each operation, read in the parameters and configure the FIFOs to do DES directly from and back to the host.

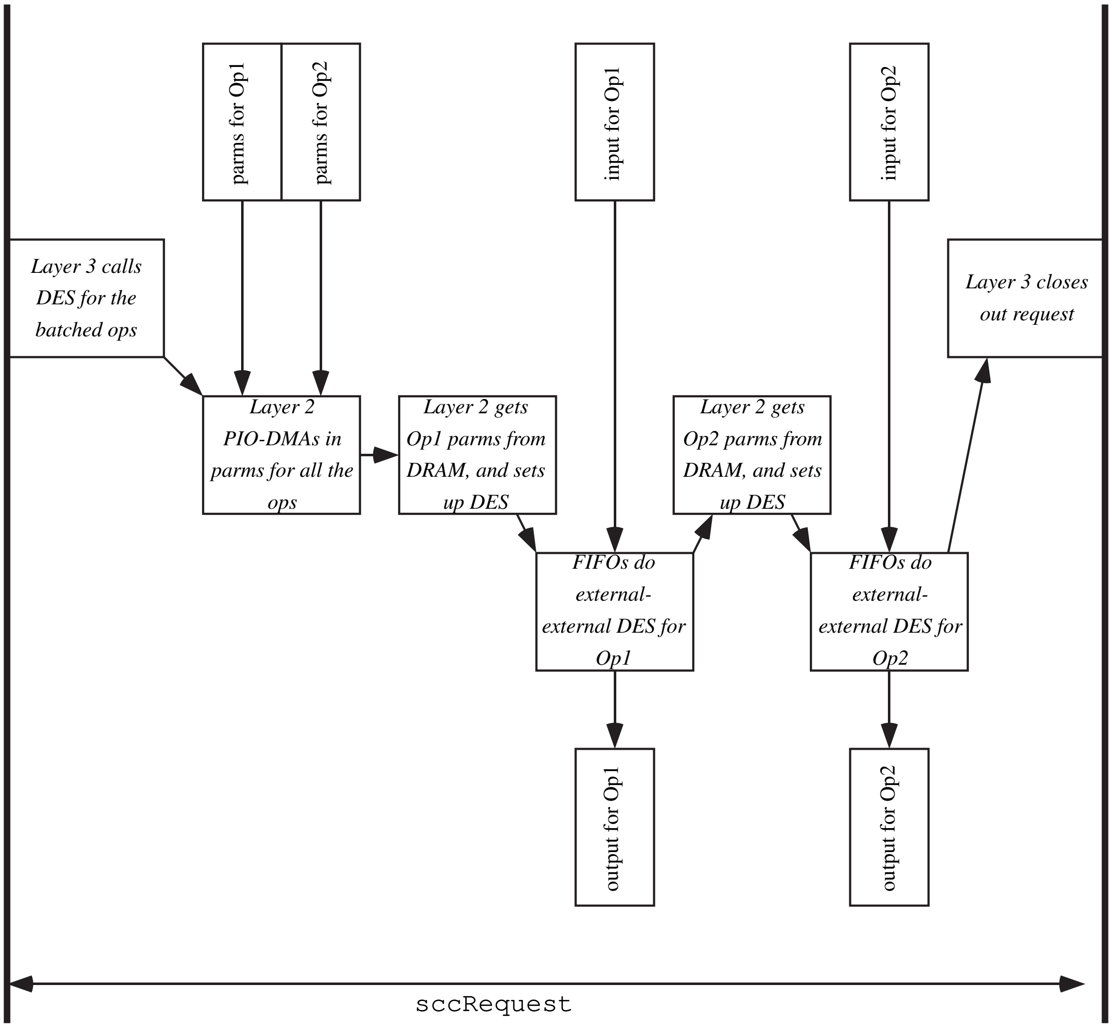

Figure 15 Experiment 7, Experiment 8: We reduce slow ISA I/Os by batching the parameters for all the operations into one block, and bringing them via PIO DMA.

Experiment. First, we proved the ISA-bottleneck hypothesis by doubling the number of ISA I/O instructions and observing an appropriate halving of the throughput.

Then, we re-worked the modified DES manager of Experiment 5 to use memory-mapped I/O instead of ISA I/O wherever possible. As an unexpected consequence, we discovered a hardware bug—certain state machine polling intermittently caused spurious FIFO reads. (Again, Figure 14 shows this operation.)

Results. Modifying our software again to work around this bug, we measured speeds up to 2500 kilobytes/second.

Experiment 7: Batching Operation Parameters

Idea. The approach of Experiment 6 still requires reading the the per-operation parameters via slow ISA I/O from the PCI Input FIFO. (Reading them via memory-mapped I/O from the Internal Input FIFO is not possible, since we would lose flow control in non-bulk mode.)

However, if we batched the parameters together, we could read them via memory-mapped operations, then change the FIFO configuration, and process the data.

Experiment. In our most recent experiment, we rewrote the host application to batch all the per-operation parameters into one group, prepended to the input data. The modified DES manager then: sets up the Internal FIFOs and the state-machine to read the batched parameters, by-passing the DES chip; reads the batched parameters via memory-mapped operations from the Internal Output FIFO into DRAM; reconfigures the FIFOs; using the buffered parameters, sets up the state-machine and the DES chip to pump each operation's data from the input FIFOs, through DES, then back out the output FIFOs. Figure 15 shows this operation.

Results. With this final approach, we measured speeds approaching 5000 kilobytes/second.

(As a control, we tried this batched-parameters approach using DMA and a separate request buffer, but obtained speeds slightly slower than Experiment 6.)

Experiment 8: Checking the Results

Idea. The results of Experiment 7 pleased us. However, colleagues disrupted this pleasure by pointing out that a recent errata sheet for our DES chip noted that using memory-mapped access for the IV and data length registers may cause incorrect results.

We were tempted to dismiss this news, since the external colleague had merely asked for *fast* cryptography; he said nothing about *correctness*. But we investigated nonetheless.

Experiment. First, we did a known-answer DES test on the implementation of Experiment 7—and it failed. So, we revised that implementation to ensure that the IV and data length registers were access via the slower ISA method. (Again, Figure 15 shows this operation.)

Results. With this final approach, we measured speeds approaching 3000 kilobytes/second.

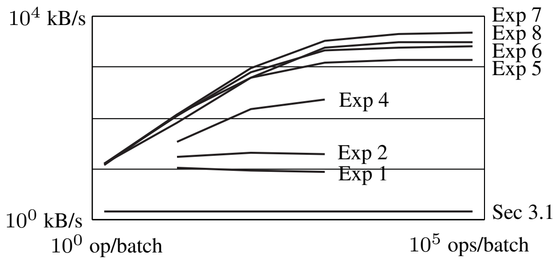

Figure 16 Summary of our short-DES experiments (preliminary figures, on an NT platform)

4 Analysis

4.1 Performance

Figure 16 summarizes the results of our experiment sequence.

On a coarse level, the short-DES speed can be modelled by:

$$\frac{C_1 \cdot Bats + C_2 \cdot Bats \cdot Ops + C_3 \cdot Bats \cdot Ops \cdot DLen}{Bats \cdot Ops \cdot DLen}$$

where *Bats* is the number of host-card batches, *Ops* is the number of operations per batch, *DLen* is the average data length per operation, and C_1, C_2, C_3 are unknown constants, representing the per-batch, per-operation, and per-byte overhead (respectively).

4.1.1 Improving Per-Batch Overhead

The curve of the top traces in Figure 16 suggests that, for fewer than 1000 operations, our speed is still being dominated by the per-batch overhead C_1. To reduce this cost, we are planning another round of hand-tuning the code.

In theory, we could eliminate the per-batch overhead C_1 entirely by modifying the host device driver-Layer 2 interaction to enable indefinite `sccRequests`, with some additional polling or signalling to indicate when more data is ready for transfer. However, our experiments were constrained by the limited resources of our own time, and the constraint that (should the results prove commercially viable) it would be possible to migrate our changes into the commercial offering with a minimum number of component changes. Both of these constraints have prevented us from exploring changes to the device driver protocol at this time.

4.1.2 Improving Per-Operation Overhead

The limitation of short DES puts an upper bound on *DLen*, which suggests a minimum $C_2/DLen$ component that we can never overcome.

API Approaches. For future work, we have been considering various ways to reduce the per-operation overhead C_2 by minimizing the number of per-operation parameter transfers. For example:

- The host application might, within a batch of operations, interleave "parameter blocks" that assert things like "the next N operations all use this key." This eliminates bringing in (and reading) the key index each time.
- The host application itself might process the IVs before or after transmitting the data to the card, as appropriate. (This is not a security issue if the host application already is trusted to provide the IVs.) This eliminates bringing in the IVs, and (since the DES chip has a default IV of zeros after reset) eliminates loading the IVs as well.

However, these approaches have two significant drawbacks. One is the fact that the "short-DES API" (that might eventually emerge in production code) would look less and less like standard DES. Another is that these variations make it much more complicated to benchmark performance meaningfully. How much work should the host application be expected to do? (Remember that the host CPU is probably capable of much greater computational power than the coprocessor CPU.) How do we quantify the "typical request sequences" for which these approaches are tuned, in a manner that enables a potential end user to make meaningful performance predictions?

Hardware Approaches. Another avenue (albeit a long-term one) for reducing per-operation overhead would be to re-design the FIFOs and the state machine.

In hindsight, we can now see that the current hardware has the potential for a fundamental improvement. Currently, the acceleration hardware provides a way to move the *data* very quickly through the engine, but not the *operational parameters*. If the DES engine (or whatever other algorithm engine is being driven this way) expected its data-input to include parameters (e.g., "do the next 40 bytes with key #7 and this IV") interleaved with data, then the per-operation overhead C_2 could approach the per-byte overhead C_3.

The state machine (or whatever system is driving the data through the engine) would need to handle the fact that the number of output bytes may be less than the number of input bytes (since those include the parameters). We also need a way for the CPU to control or restrict the class of engine operations over which the parameters, possibly chosen externally, are allowed to range. For example:

- The external entity may be allowed only to choose certain types of encryption operations (restriction on type).
- The CPU may wish to insert *indirection* on the parameters the external entity chooses and the parameters the engine sees (e.g., the external entity provides an index into an internal table, as we did with keys in the experiments).

The issues of Section 4.2 also apply here.

4.1.3 Improving Per-Byte Overhead

Well-established folklore teaches that the per-byte overhead C_3 is small. Consequently, we doubt C_3 can be improved much, nor that it is significant.

4.2 API Design Issues

Cryptographic APIs, once defined, may appear obvious. But as noted earlier, an implicit goal of this work was, if we were able to substantially improve short-DES performance, to produce a prototype of a new feature that could be migrated into the current commercial offering.

How to design a short-DES API that could provide this superior performance, be usable by a wide range of applications, and be reasonably easy to implement and maintain, raises a number of interesting challenges, including:

- **Key Unpacking.** What is the most general way to handle, in a Layer 2 API, the loading of keys from outside? Each application has its own method (and we haven't even discussed the implications of things such as FIPS 140-1 [6]).
- **Operation Restrictions.** One of the benefits of secure crypto coprocessors is increased security for sensitive operations and data, as well as cryptographic acceleration (which is not necessarily associated with secure coprocessors). Many applications that could use high-speed short DES might want to greatly restrict the modes or keys or IVs or other such parameters that an untrusted host-side entity could choose. How do we handle this in an API?
- **Algorithm Mix.** These techniques could also speed up TDES, SHA-1, DES-MAC, and other algorithms. Which would application programmers require? Would they require operations for different algorithms within the same batch? If so, how do we handle items such as key tables for different algorithms? (For example, allowing the user to choose single-DES operations using parts of TDES keys is risky.) What about variations such as decryption with one key and re-encryption with another, without the plaintext ever leaving the secure boundary? (This last option could speed Kerberos server implementations. [3])
- **Operation Sequences.** As speculated earlier, having the host sort operations in various ways could help speed performance—for some approaches. What's a reasonable balance between full flexibility and manageable implementation?
- **Source/Destination.** These experiments all dealt with operations whose parameters and input were coming from the outside, and whose output was going back to the outside. However, each of these three elements (parameters, input, and output) could also come from inside—and if we start thinking about various types of parameters, the option space grows considerably beyond even this 2^3. What's reasonable?
- **IVs.** Our colleague wanted to choose his own IVs. Some applications would require random IVs (which our device could generate itself); for other applications, the plaintext key (a sensitive item) is re-used for the IV. How do we handle this?
- **Dependent Operations.** Plausible scenarios can constructed for having later operations in a batch use data that resulted from earlier operations—for example, key-unwrapping operations could themselves be included in the same batch as the operations which use these keys. How do we handle this?

As we explore the design space, we are faced with another conundrum. If optimizing performance requires coding a tight CPU loop, then either

- our tight loop will be slowed by n tests (for the n options), or
- or we must implement 2^n different loops—one for each possible set of option choices.

(We briefly considered even having the DES Manager write the loop each time through.) This conundrum faded, however, when it became apparent that CPU speed was not the primary bottleneck.

5 Conclusions

From this experience, we learned many things.

- Meaningful benchmarks of symmetric crypto performance should include data lengths.

- Neglecting to consider how *operational parameters* can be efficiently sent into a cryptographic system can greatly hinder performance—and reduce the benefits of engineering a high-speed data path.
- Neglecting to consider how software can actually use new cryptographic hardware designs can reduce the benefits of these new designs.
- Complex accelerator architectures can hide bottlenecks that are not initially apparent.
- But with a programmable device, software experiments can identify these bottlenecks and overcome many of them.

In the hindsight, an appropriately specified goal ("fast short DES") could have led to an appropriate software and hardware model (e.g., based on standard principles of performance analysis [7]), and thus enabled examination of these issues before the hardware design had even begun. However, one of the contributions of our work is providing this hindsight: in the pressure of product development, hardware tends to be frozen early; and our field tends to introduce a separation between software design and hardware design that prevents a full examination of the interactions.

In future work, we plan to finalize a proposed short-DES API and and attempt to migrate it into the commercial offering (where it can then actually speed real customer applications); we also hope to examine other cryptographic services our device offers, to see if similar techniques will improve performance there. It also would be interesting to explore performance tradeoffs between host-only and coprocessor-enhanced DES for short operations, and then re-examine the security tradeoffs in light of this information.

Furthermore, we hope to use some of our experience in accelerating DES variants to build high-performance prototypes of alternative cryptographic coprocessor applications (such as root-secure private information servers [8], noted earlier, and authenticated encryption [4]).

Availability

Contact Ron Perez (ronpz@us.ibm.com) at IBM T.J. Watson Research Center for current information on the external availability of the experimental code discussed in this paper.

Acknowledgments

The authors gratefully acknowledge helpful discussions with Enzo Condorelli, Joan Dyer, Juan Gonzalez, Ulf Mattsson, Elaine Palmer, and Ron Perez; and also appreciate the helpful comments of the referees.

References

[1] D.G. Abraham, G.M. Dolan, G.P. Double, J.V. Stevens. "Transaction Security Systems." *IBM Systems Journal.* 30: 206-229. 1991.

[2] J. Dyer, R. Perez, S.W. Smith, M. Lindemann. "Application Support Architecture for a High-Performance, Programmable Secure Coprocessor." *22nd National Information Systems Security Conference.* October 1999.

[3] N. Itoi. "Secure Coprocessor Integration with Kerberos V5." *USENIX Security Symposium 2000.*

[4] C. S. Jutla. *Encryption Modes with Almost Free Message Integrity.* Draft Research Report, IBM T.J. Watson Research Center, July 2000.

[5] U. Mattsson, Personal communication, Protegrity Inc. Publication *Performance Report on Secure Coprocessors*, 1999.

[6] National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules.* Federal Information Processing Standards Publication 140-1, 1994.

[7] C. Smith. *Performance Engineering of Software Systems.* Addison-Wesley, 1990.

[8] S.W. Smith, D. Safford. "Practical Server Privacy Using Secure Coprocessors." *IBM Systems Journal*, to appear.

[9] S.W. Smith, S.H. Weingart. "Building a High-Performance, Programmable Secure Coprocessor." *Computer Networks (Special Issue on Computer Network Security.)* 31: 831-860. April 1999.