Architectural Support for Fast Symmetric-Key Cryptography

Jerome Burke John McDonald Todd Austin

Advanced Computer Architecture Laboratory
University of Michigan
{jaburke,johngm,austin}@eecs.umich.edu

Abstract

The emergence of the Internet as a trusted medium for commerce and communication has made cryptography an essential component of modern information systems. Cryptography provides the mechanisms necessary to implement accountability, accuracy, and confidentiality in communication. As demands for secure communication bandwidth grow, efficient cryptographic processing will become increasingly vital to good system performance.

In this paper, we explore techniques to improve the performance of symmetric key cipher algorithms. Eight popular strong encryption algorithms are examined in detail. Analysis reveals the algorithms are computationally complex and contain little parallelism. Overall throughput on a high-end microprocessor is quite poor, a 600 Mhz processor is incapable of saturating a T3 communication line with 3DES (triple DES) encrypted data.

We introduce new instructions that improve the efficiency of the analyzed algorithms. Our approach adds instruction set support for fast substitutions, general permutations, rotates, and modular arithmetic. Performance analysis of the optimized ciphers shows an overall speedup of 59% over a baseline machine with rotate instructions and 74% speedup over a baseline without rotates. Even higher speedups are demonstrated with optimized substitutions (SBOXes) and additional functional unit resources. Our analyses of the original and optimized algorithms suggest future directions for the design of high-performance programmable cryptographic processors.

1 Introduction

In an increasingly connected world, cryptography has become an essential component of modern information systems. Cryptography provides the mechanisms necessary to provide accountability, accuracy and confidentiality in inherently public communication mediums such as the Internet. Today, cryptographic processing is primarily reserved for electronic commerce transactions and secure e-mail, however, the adoption of virtual private networks (VPNs) [12] and secure IP (IPSEC) [3] will subject more of all communication to cryptographic processing. As secure communication bandwidth demands continue to grow, so too will the importance of efficient cryptographic processing.

Cryptography is the art of using mathematics to encrypt and decrypt data. There are many cryptography algorithms (or ciphers) in use today, some good, and some not so good. The quality of a cipher is judged by its ability to prevent an unrelated party from determining the original content of an encrypted message. Figure 1 illustrates the two forms of cryptography most commonly used in information systems today. The simplest ciphers are known as *symmetric-key ciphers*. Communicating parties share a common private key which is used to transform the message from *plaintext* to *ciphertext*. The ciphertext is communicated to the other party, and then the process is reversed using the same private key.

The primary obstacle in making private key symmetric ciphers useful is distribution of private keys. To securely share a private key, communicating parties would first have to be holding a shared private key! Public key cryptography solves this conundrum by implementing encryption with two keys, a well-known public key and a private key. Only the receiver knows the private key value. The receiver's public key, on the other hand, is widely published by trusted sources. As shown in Figure 1, to encrypt a message using a public key cipher, it is first converted to ciphertext using the public key. The resulting ciphertext can only be decrypted using the receiver's private key. Secure communication now proceeds without any insecure exchanges of private information. The process may also be reversed to produce what is known as a *digital signature*. Digital signatures *authenticate* the sender, *i.e.*, verify the identity of the sender. Since only the person holding the private key knows its value, only that person can create a digital signature that others can decrypt with the public key (assuming the private key has not been compromised).

It would seem that the additional benefits of public key encryption would obviate the need for private key encryption, however, the high cost of employing public key encryption requires that it be used sparingly. Strong public key ciphers are computationally very expensive, running

ASPLOS 2000 Cambridge, MA Nov. 12-15 , 2000

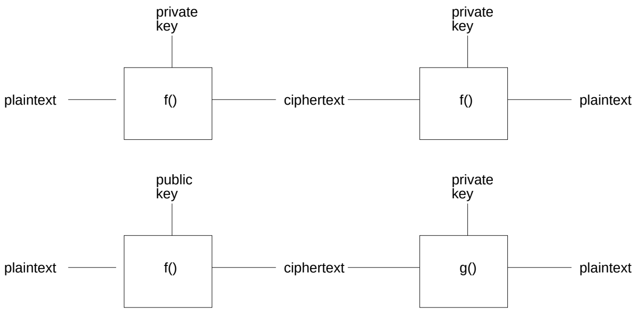

Figure 1. Private Key and Public Key Cryptography.

usually 1000 times slower than comparable private key ciphers [23]. Public key encryption requires exponentiation and modular multiplication of large multi-precision numbers of 1024 bits in length or more. As a result, most secure information systems only use public key encryption at the start of a session to authenticate communicating parties and to effect a secure exchange of private keys. The remainder of the session employs efficient private key algorithms, using the private keys exchanged during authentication.

An example of a system that uses this session management strategy is the Secure Sockets Layer (SSL) protocol [29]. SSL extends TCP/IP to support secure encrypted connections with authentication of senders and receivers. The protocol is used by web servers and browsers to establish secure HTTP connections. Figure 2 illustrates the relative costs of private and public key cryptography for a web server. The numbers shown were gathered by Intel for a heavily loaded web server running on an iA32 platform [22]. The results show the total fraction of time spent in the public key cipher code, private key cipher code, and other parts of the web server and operating system. Results are shown for increasing session length, where a session includes an initial public key authentication and private key exchange, and a transfer of a single message of the size listed.

Clearly, for very short sessions fast public key cipher processing is crucial for high transaction throughput. For sessions lengths on the order of a single web page object (approximately 21k bytes [2]), private key cipher processing dominates web server run time. For a 32k session length, private key processing overheads rise to 48% of overall run time. Since a session will likely see a user visit many web pages and web pages with many objects, private key cipher performance will quickly dominate the performance of SSL sessions. To further reduce the cost of public key authentication, SSL allows the use of a session cache, where authenticated private keys are held and can later be reused when users reconnect to view other web pages.

In this paper, we focus our attention on improving the performance of private key symmetric ciphers. We first examine the execution of eight widely known strong symmetric-key ciphers. We analyze their performance on detailed microarchitectural models, where we are able to clearly show their performance and the bottlenecks that slow their progress. Armed with these insights, we propose architectural extensions that streamline cipher kernel processing. The new instructions speed modular arithmetic, substitutions, general bit permutations, and rotates. We recode the cipher using these new instructions and then examine their performance on microarchitectural models with varying levels of support for fast cryptography. Our approach is a general one, the instructions are shown to be useful across a broad array of cipher algorithms.

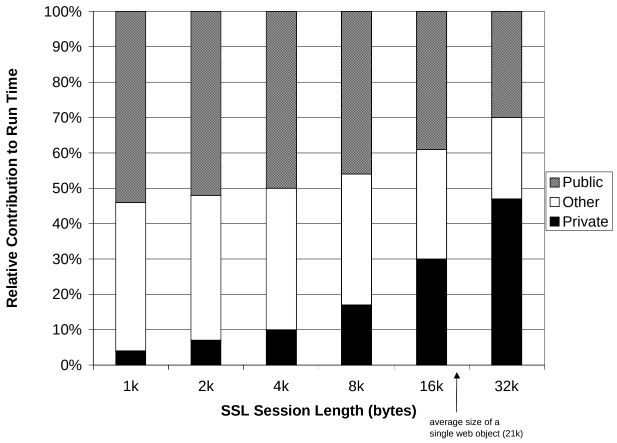

Figure 2. SSL Characterization by Session Length.

2 Private Key Symmetric Ciphers

Figure 3 shows the kernel of the Twofish cipher, developed by Counterpane Systems [26]. It is a candidate for the Advanced Encryption Standard (AES) [1], the US government's effort to develop a new strong encryption standard. Twofish is a particularly good example to look at because it captures many of the operations that ciphers employ. The code shown in Figure 3 is the encryption kernel, run on one 128-bit block of data to encrypt it into a 128-bit ciphertext block using a 128-bit key value. The Twofish decryption kernel is nearly identical except the order of the operations is reversed and inverted (*e.g.*, rotate left becomes rotate right).

The cipher algorithm first reads the input data, XOR's it with the 128-bit intermediate vector (IV) and key, and then enters the encryption loop. The encryption loop executes 16 iterations (or rounds as they are called in cryptography literature) to produce 128 bits of ciphertext. The ciphertext is then once again XOR'ed with the key and stored to the intermediate vector and the output buffer.

Within the kernel loop, the cipher algorithm employs a series of reversible operations to implement a process called *diffusion*. Diffusion works to randomly impress upon each of the output bits some information from each of the input bits. The direction of the diffusion process is set by the

```
x[0] = input[0] ^ key[0] ^ IV[0];
x[1] = input[1] ^ key[1] ^ IV[1];
x[2] = input[2] ^ key[2] ^ IV[2];
x[3] = input[3] ^ key[3] ^ IV[3];

for (ii=15, jj=0; ii >= 0; ii--, jj^=2) {
  t0 = (sbox1[x[jj][0]] ^ sbox2[x[jj][1]]
        ^ sbox3[x[jj][2]] ^ sbox4[x[jj][3]]);
  t1 = (sbox1[x[jj^1][3]] ^ sbox2[x[jj^1][0]]
        ^ sbox3[x[jj^1][1]] ^ sbox4[x[jj^1][2]]);

  x[jj^3] = x[jj^3] <<< 1;
  x[jj^2] = x[jj^2]^(t0+t1+key[ii<<1]);
  x[jj^3] = x[jj^3]^(t0+(t1<<1)+key[(ii<<1)+1]);
  x[jj^2] = x[jj^2] >>> 1;
}

output[0] = IV[0] = x[2] ^ key[0];
output[1] = IV[0] = x[3] ^ key[1];
output[2] = IV[0] = x[0] ^ key[2];
output[3] = IV[0] = x[1] ^ key[3];
```

Figure 3. The Twofish Cipher Kernel. All variables are 32-bit integers. Rotates are indicated by <<< and >>>.

private key. The more seemingly random and complete the diffusion process is, the more difficult it is to recreate the plaintext without the key value. With large keys and good diffusion, the ciphertext is extremely resistant to attackers. Quantitatively, a strong encryption algorithm is one where any change in the input results in a random perturbation of each output bit with probability 50%. Moreover, any change to the key value should have an equally dramatic effect on the ciphertext produced.

The process of diffusion has two important implications to the underlying machine architecture. First, diffusing input bits is computationally expensive on modern microprocessors. Most algorithms run their kernel loop at least 16 times on each block of data encrypted, successively mixing the data more and more on each round. The second implication is that cipher kernels have little parallelism. Parallelism in the cipher (especially coarse-grained parallelism) would imply that some aspect of the computation does not affect later ciphertext results, which would in turn imply that the cipher algorithm was not a strong one! While we did find a small level of ILP in cipher kernels, the process of making the kernels run fast primarily entails improving their execution efficiency on the underlying microarchitecture.

The intermediate vector IV ensures that the diffusion process propagates to all remaining ciphertext in the communication stream. The ciphertext value of encrypted block i is first XOR'ed with plaintext block $i + 1$ before it is encrypted. The end results is that the cipher kernel execution is a very long recurrence with virtually no parallelism.

To ensure that the ciphertext can be decrypted back to the plaintext, the cipher kernel must employ a series of key-parameterized reversible operations. The Twofish algorithm demonstrates a number of these:

Rotates Rotates are easily reversible (simply rotate the same distance in the opposite direction). Rotates also have good diffusion properties, impressing each bit onto another bit of the output.

Modular Addition Modular arithmetic, if based on a power-of-two base, is cheap, fast, and has relatively good diffusion properties. Moreover, it is easily inverted using modular subtraction or modular addition with the two's-complement of the addend. XOR operations, which are modular additions in base 2, are easily reversible by XOR'ing the same value onto the resulting ciphertext.

Substitutions Table-based substitutions can be used to quickly implement any key-parameterized function. An SBOX is a table of values indexed with plaintext (usually a byte) that produces the result of the key-parameterized function. SBOX's are easily reversible by inverting the table, *i.e.*, indices become values and values become indices.

In addition, other algorithms often employ two other mechanisms, modular multiplication and XBOXs.

Modular Multiplication Modular multiplication has been shown to have particularly good diffusion properties [18], and the operation can be easily reversed with modular multiplication of the modular inverse of the multiplicand. If the multiplicand is part of the key, all divides (which are typically much more expensive) can be confined to the cipher setup code. If the modulus of the operation is a power-of-two (as in RC6), it can be efficiently implemented using existing multiply instructions. A few algorithms (most notably IDEA) use a modulus of a $2^N + 1$ prime number. This further improves diffusion properties of the operation at the expense of more computation. Techniques have been developed to efficiently implement $2^N + 1$ prime modulus operations using only two additional (and parallel) adds plus one multiply [18].

General Permutations General permutations map N bits onto N bits with an arbitrary exchange of individual bit values. While trivial to implement in hardware with a wire network (called an XBOX), these permutations are quite expensive to implement in software. Consequently, newer ciphers strictly avoid permutations. We still consider them, however, as they are used in DES [11], the US encryption standard put into practice in the early 1970's and still in wide use today.

3 Experimental Framework

3.1 Cipher Benchmarks

We analyzed the eight private key symmetric ciphers listed in Table 1. The table lists for each algorithm the key size used for the experiments, the block size encrypted by each application of the cipher kernel, the number of rounds (iterations) executed within the cipher kernel, the author of the algorithm, and popular applications that use the cipher. Each of the algorithms use at least 128 bits of key data, and each is generally considered a strong algorithm, having undergone review and aggressive cryptanalysis. Four of the ciphers, *i.e.*, 3DES [11], Blowfish [8],

Cipher	Key Size	Blk Size	Rnds/ Blk	Author	Example Application
3DES	186	64	48	CryptSoft	SSL, SSH
Blowfish	128	64	16	CryptSoft	Norton Utilities
IDEA	128	64	8	Ascom	PGP, SSH
Mars	128	128	16	IBM	AES Candidate
RC4	128	8	1	CryptSoft	SSL
RC6	128	128	18	RSA Security	AES Candidate
Rijndael	128	128	10	Rijmen	AES Candidate
Twofish	128	128	16	Counterpane	AES Candidate

Table 1. Private Key Symmetric Ciphers Analyzed.

IDEA [18], and RC4 [25], are algorithms used in popular software packages. 3DES runs the US DES standard encryption algorithm [11] serially three times with three 56-bit keys. This is the mode of operation specified in the Secure Sockets Layer (SSL) protocol specification [29]. The remaining algorithms, *i.e.*, Mars [6], RC6 [24], Rijndael [10], and Twofish [26], are second round candidates for the Advanced Encryption Standard (AES) [1]. The National Institute of Standards and Technologies (NIST) is coordinating the multi-year AES competition that will ultimately lead to the selection of a new US encryption standard (to replace DES). However, many of the AES algorithms will likely emerge as high-quality encryption algorithms that will see use in popular applications and protocols.

Our baseline implementation of each algorithm is quite efficient. We obtained optimized implementations of each of the AES finalists from the inventors of the algorithms. 3DES, Blowfish, and RC4 were all developed by Eric Young of CryptSoft [9]. CryptSoft's code is quite efficient, as a result, it has found its way into many popular software systems including SSH, OpenSSL, FreeBSD, and the Mozilla web browser. The optimized IDEA implementation was provided by Ascom, inventors of the algorithm.

Operation of the algorithms can be tailored significantly, *e.g.*, number of rounds, block size, key size. We configured each algorithm as suggested by the inventors to maintain good strength and performance. 3DES was configured as per the SSL specification [29]. All ciphers were run in chaining-block-cipher (CBC) mode. In this mode, the value of cipher text block i is XOR'ed with plaintext block $i+1$ before it is encrypted. Nearly all applications use CBC mode as it produces ciphertext that is more resistant to attacks.

All baseline codes were compiled with the Compaq Alpha CC compiler (version 5.9) with full optimization and EV6 architecture optimizations (*e.g.*, byte and word loads). All hand coded versions of the algorithms were based on assembly outputs from the Compaq CC compiler with the same optimizations. All analyzed codes (baseline and optimized) were validated by running the optimized encryption kernel with the original decryption kernel (and vice versa).

3.2 Performance Analysis Tools

Performance analysis was performed with the SimpleScalar Tool Set version 3.0 for the Alpha architecture [5]. The SimpleScalar tool set includes detailed microarchitecture simulators that can be tailored to reveal the bottlenecks within a program. We used the SimpleView visualization framework to optimize the performance of the cipher kernels. The SimpleView viewer displays graphically the stalls experienced by instruction as they pass though the modeled pipeline, making it fairly straightforward to identify the bottlenecks that slowed cipher kernels.

We analyzed program performance on the detailed timing simulator (sim-outorder). The timing simulator executes user-level instructions, performing a detailed timing simulation of an aggressive 4-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. Our baseline simulation configuration models a future generation out-of-order processor microarchitecture. The processor has a large window of execution; it can fetch and issue up to 4 instructions per cycle. It has a 256 entry re-order buffer with a 64 entry load/store buffer. Loads can only execute when all prior store addresses are known. There is an 8 cycle minimum branch misprediction penalty. The baseline processor has 4 integer ALU units, 2-load/store units, 2-FP adders, 1-integer MULT/DIV, and 1-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 7 cycles, Integer DIV 12 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

The processor we simulated has 32k 2-way set-associative instruction and data caches. Both caches have block sizes of 32 bytes. The data cache is write-back, write-allocate, and is non-blocking with 2 ports. The data cache is pipelined to allow up to 2 new requests each cycle. There is a unified second-level 512k 4-way set-associative cache with 32 byte blocks, with a 12 cycle hit latency. If there is a second-level cache miss it takes a total of 120 cycles to make the round trip access to main memory. We model the bus latency to main memory with a 10 cycle bus occupancy per request. Address translation is implemented a 32 entry 8-way associative instruction TLB and a 32 entry 8-way associative data TLB, each with a 30 cycle miss penalty.

4 Cipher Kernel Analysis

4.1 Cipher Throughput

Figure 4 shows the encryption performance of each algorithm.[1] The performance of each algorithm is expressed as a rate, bytes encrypted per 1000 cycles. We selected this metric because it has a useful real interpretation: on a 1 GHz processor, the same value is the rate in megabytes/sec that the microarchitecture would be able to encrypt data. For each algorithm, we show the performance in instructions per byte encrypted (the 1 CPI machine), performance

[1] Because of the symmetry between the encryption and decryption algorithms, performance was comparable for these codes for all experiments. For the sake of brevity we only present encryption performance numbers.

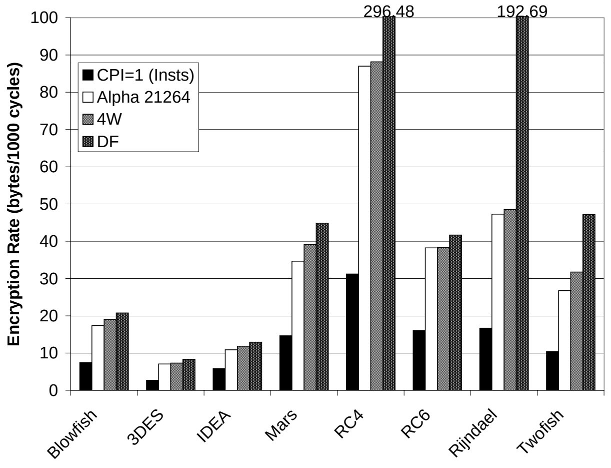

Figure 4. Cipher Encryption Performance.

on a real 600Mhz Alpha 21264 workstation, performance on the modeled baseline microarchitecture (4W), and the upper bound dataflow performance of the algorithm (DF). Dataflow performance was measured on the performance simulator using perfect branch prediction, infinite window size, unlimited fetch bandwidth, perfect memory address disambiguation (*e.g.*, loads never wait for unrelated stores even if their addresses have not yet been computed), and unlimited functional unit resources. The dataflow experiments represent the maximum performance that could be attained on any machine short of speeding up individual operation or employing value speculation [19].

We felt it was important to validate the performance of the baseline SimpleScalar model so we ran the identical code on a 600 Mhz Alpha 21264 workstation (running Tru64 Unix) with a 1GB main memory. We loosely based our simulator baseline on the Alpha 21264 microprocessor, setting the resource configuration and their latencies to values published by Compaq [17]. The correlation in absolute performance was quite close, all except Mars and Twofish were within 10% of the actual machine tests. Mars was 11% faster, Twofish was 15% faster. It's difficult to assess why these discrepancies exists as many of the details of the Alpha 21264 microarchitecture have not been disclosed. We feel they are likely due to the 21264's clustered microarchitecture, which was not modeled by SimpleScalar. The clustered microarchitecture has slightly higher forwarding latency for some instructions; this would account for the slightly better performance of the algorithms when running on the simulator. However, performance trends were indeed captured, thus we are fairly confident that our performance models are representative of real hardware.

As shown in Figure 4, the baseline 4-wide superscalar processor model performance (4W) varied dramatically. The worst performance was given by 3DES. 3DES is computationally very complex, and it contains operations (*e.g.*, general permutation) that do not map well to a general-purpose microprocessor. It's interesting to note that a 1 GHz processor running the 3DES kernel (a common encryption mode for secure web transports) would produce a maximum throughput of 7.32 MBytes/s, not even enough throughput to saturate a trailing edge 100 Mbs Ethernet link, and barely enough to saturate a low-cost T3 communication line. IDEA also turned in a poor performance, the primary bottleneck in this cipher is numerous 7-cycle multiplies.

The AES standard candidates have much better performance with Rijndael leading the pack at 48.51 bytes/1000 cycles. The best performance overall was delivered by RC4 at 88.16 bytes/1000 cycles, more than 10 times the performance of 3DES. RC4 benefits from significantly more parallelism than the other algorithms. The algorithm is essentially a key-based random number generator that XOR's a random sequence onto the input stream. The iterations of the random number generator are (mostly) independent, allowing its execution to fully saturate a wide machine, resulting in very high-bandwidth encryption.

The last experiment (DF) shows the scalability of cipher kernel performance. In these experiments, the original code is executed on a dataflow machine. This machine has infinite fetch, decode, execute, and retirement bandwidth. In addition, it is never slowed by branch misprediction, cache misses or ambiguous store address dependencies. This configuration gives the absolute top performance that the code could achieve (short of reducing instruction latencies or applying value speculation to break data dependencies). We found these results quite surprising, Blowfish, IDEA, and RC6 are running within 10% of dataflow machine performance. There is slightly more headroom for 3DES, Mars and Twofish, with potential speedups of 12%, 13%, and 32%, respectively. RC4 and Rijndael are the outliers, these codes have ample parallelism and could be sped up significantly.

4.2 Bottleneck Analysis

Figure 5 analyzes the bottlenecks in the cipher algorithms. Results are only shown for the algorithms that were not running at dataflow performance in Figure 4. For each cipher, the impact of various stall conditions is shown. Performance for each bar is shown relative to the performance of the dataflow machine running the cipher kernel. In each experiment, a single bottleneck (*e.g.*, branch mispredictions) is re-inserted into the processor model, and the resulting performance impact indicates the extent to which that bottleneck is fully exposed in execution (independent of all other bottlenecks). Note that if a bottleneck affects the dataflow machine, it may not help the performance of the baseline machine if it is removed. *All* of the exposed bottlenecks may have to be removed before performance improvements are seen in the baseline machine. The most important aspect of these analyses is that bottlenecks that do not affect the dataflow machine will not (and cannot) affect the performance of the baseline machine.

Six bottlenecks are analyzed. The *Alias* bar shows the impact of stalling loads in the pipeline until all earlier store addresses have been resolved (*i.e.*, no address aliases).

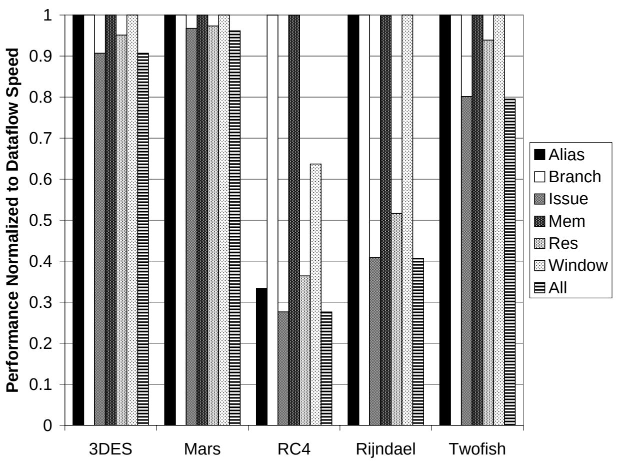

Figure 5. Analysis of Bottlenecks in Cipher Kernels.

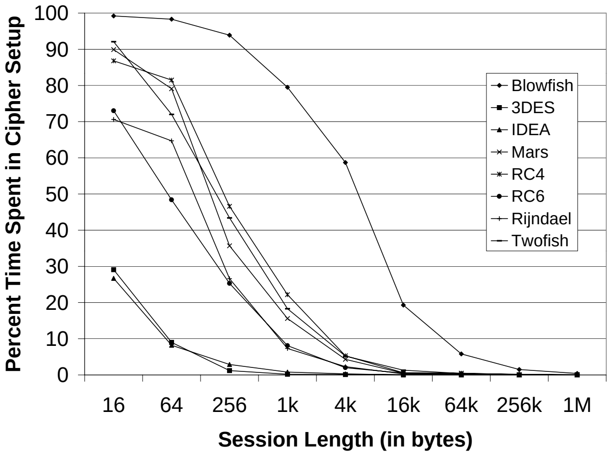

Figure 6. Setup Cost as a Function of Session Length.

Branch shows the effects of mispredictions, *Issue* shows the impact of reducing issue width. *Mem* shows the impact of introducing a realistic memory system, and *Res* gives the impact of limited functional unit resources. *Window* shows the impact of a limited-size instruction window, and *All* shows performance of the machine with all bottlenecks enabled.

It's interesting to note that branch mispredictions and data memory performance do not impair the performance of any ciphers. This observation is in stark contrast to other benchmarks commonly studied in the computer architecture literature. Branch mispredictions are not a problem for these codes as branches are quite predictable, usually found in kernel loops. Cache misses rarely occur as the algorithms read one memory value and then compute with it for often 100's of cycles. In addition, our memory system has a next-line prefetch capability which eliminates virtually all data cache misses in the cipher kernel.

A limited window size also has little affect on any of the codes, except RC4. RC4 has significant parallelism between rounds of the kernel cipher. The other algorithms do not, especially when configured in chaining block mode (as they were for these experiments). While there is some parallelism, it is fairly local for all the algorithms except RC4. A similar trend can be seen for address aliases. All the algorithms (except RC4) only access memory to update intermediate vectors and read input data, as a result, all loads are dependent on the previous stores. Having perfect alias detection does little for these codes as they still end up waiting on all previous stores. This is not the case for RC4, however, as this code performs many stores to an internal table used for key-based random number generation. Depending on the input stream, stores in the previous round of the algorithm could be dependent, however, the probability of this is $1/256$ (assuming good diffusion). In the dynamically scheduled microarchitecture, these unknown (and mostly independent) stores stall later loads and reduce the overall throughput of the algorithm. As shown in Figure 5, introducing these stalls has a significant affect on dataflow machine performance. The common bottlenecks across all ciphers are resource supply and issue width, with Rijndael and RC4 having the largest impacts.

When attacking a bottleneck, it is important to accurately discern what part of the code is creating the bottleneck. We can best spend optimization resources by focusing on this part of the algorithm. Each of the ciphers is composed of three major components: setup, encryption kernel, and decryption kernel. The setup code processes the key to create various SBox's and key-based permutations required by the cipher kernel. In addition, many of the algorithms precompute lookup tables (based on the key) that speed processing in the cipher kernel. The encryption and decryption kernels process one block of data.

Figure 6 shows the relative cost of cipher setup compared to the encryption kernels. (The decryption kernels were omitted as their cost is identical to encryption). Costs are measured in run time for varied session lengths. Since setup code is called only once per session, longer sessions better amortize setup cost. As shown in the figure, setup costs for 3DES and IDEA are quite small, even for 16 byte sessions. 3DES's setup times are small due to its costly encryption kernel. IDEA, on the other hand, is designed to have very low-cost startup. The next group, Mars, RC4, RC6, Rijndael, and Twofish all have moderately sized setup costs, with overheads dropping well below 10% at session lengths of 4k or greater. The clear outlier is Blowfish which only sees setup costs below 10% for sessions longer than 64k bytes. Blowfish runs the encryption kernel on the 128-bit key 520 times before commencing encryption of the input data! This amounts to the cost of encrypting 8k bytes of input data, requiring much longer sessions to amortize setup.

Given the dominance of cipher kernels in overall performance (even for Blowfish), we focused our attention on the

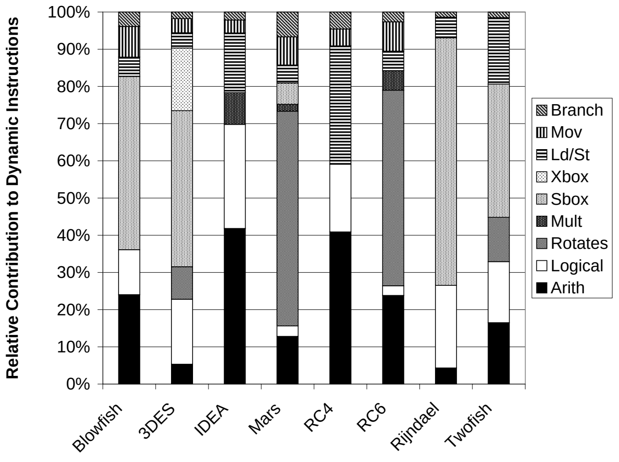

Figure 7. Characterization of Cipher Kernel Operations.

cipher kernels for the remainder of our analyses. For all remaining experiments, we use a session length of 4k bytes.

4.3 Cipher Kernel Characterization

The results of the previous section suggest two ways to improve cipher kernel performance: add more resources for kernel processing, or reduce the latency of cipher kernel operations.

Figure 7 details the operations executed in each cipher kernel. We classified each of the instructions in the cipher (by hand) into categories that describe their kernel operation. We then ran the ciphers and counted the number of dynamic occurrences of each instruction. Each portion of the stacked bars in the graph represents the fraction of all dynamic instructions contributing to that category. The operations were broken down into the following categories:

Arithmetic : Primarily addition used to implement encryption operations and address arithmetic.

Logic : Primarily XOR operations used in ciphers.

Rotates : Rotates are used in nearly all of kernels.

Multiplies : Integer multiplies used in a few of the kernels.

Substitutions : Table-based substitutions (called S-boxes) used to create reversible key-parameterized general functions within cipher kernels.

Permutes : Reversible general bit permutations used within a few of the cipher kernels.

Loads/Stores : Memory operations (minus those used for substitutions).

Control : Branches and jumps.

As shown in Figure 7, the algorithms can be broadly divided into two categories: those that rely on arithmetic computation, and those that rely on substitutions. IDEA and RC6 are computational algorithms relying heavily on multiplies to diffuse input data bits. Blowfish, 3DES, Rijndael and Twofish, on the other hand, rely heavily on SBox's to translate input data to ciphertext. The former groups will benefit from more computing resources (especially multiplies) and from faster operations (*e.g.*, rotates). The latter group will benefit from increased memory bandwidth and faster memory accesses for SBox translations.

Besides improving the efficiency of kernel operations, it is possible to speed up an algorithm using value prediction. Value predictors produce values that break dependencies between instructions. As dependencies are removed, instruction level parallelism and program performance increases. For example, if a value predictor could predict the input values to the cipher kernel rounds, it would be possible for kernel rounds to execute in parallel, resulting in significantly more cipher throughput. To test this possibility, we instrumented our microarchitecture model with an infinite-sized last value predictor [19] and used it to predict the results of all instructions in each cipher kernel. The most predictable dependence edge in any of the cipher kernels was predicted correctly only 6.3% of the time! Clearly, diffusion works to transform data in unpredictable ways, eliminating the possibility that value speculation might be useful in improving cipher performance.

5 Architectural Extensions

Figure 8 lists the additions we made to the instruction set to support fast execution of symmetric private key ciphers. A number of important considerations drove the design of these instructions. First, all instructions are limited to two register input operands and one register output. While having three register inputs would provide significantly more opportunity to combine low-latency operations, we felt the resulting impact of a third input operand was too great to consider for a simple instruction set enhancement. Adding a third register operand would increase the number of read ports on the register file by 50%, which would subsequently increase its cycle time. In addition, more bandwidth would be required from the register renamer as well; slowing the renamer down could potentially slow the entire pipeline.

Second, we carefully considered the impact on cycle time each new instruction could potentially create. Baseline functional units and modified functional units were specified in structural Verilog, synthesized using Cascade EPOCH synthesis tools, and timing analyses were performed using SPICE for a 0.25u MOSIS TSMC process. Finally, we worked to develop a small set of canonical operations that could be broadly applied to many cipher algorithms.

Figure 8 lists the additions we made to the Alpha instruction set to support fast execution of symmetric private key ciphers. Rotates (ROL and ROR) are fully supported, for 64 and 32-bit data types. The Alpha architecture does not sup-

MULMOD <srca>,<srcb>,<dest>

```
REG[<dest>] <− (REG(<srca>) * REG[<srcb>]) % 0x10001
```

ROL <srca>,<srcb>,<dest>

```
REG[<dest>] <− REG[<srca>] <<< (REG[<srcb>] & 0x3f)
```

ROR <srca>,<srcb>,<dest>

```
REG[<dest>] <− REG[<srca>] >>> (REG[<srcb>] & 0x3f)
```

ROLX <src>,#<rot>,<dest>

```
REG[<dest>] <− (REG(<src>) <<< #<rot>) ^ REG[<dest>]
```

RORX <src>,#<rot>,<dest>

```
REG[<dest>] <− (REG(<src>) >>> #<rot>) ^ REG[<dest>]
```

SBOX.#<tt>.#<bb>.<aliased> <table>, <index>, <dest>

```
REG[<dest>] <− MEM32[(REG[<table>]&~0x3ff)
                       |  (((REG[<index>]>><bb>*8)&0xff)<<2)]
```

The SBOX instruction extracts byte #<bb> (0..3) from register <index> and concatenates the resulting 8−bit value with register <table> to produce a 32−bit aligned Sbox address. The 32−bit value at the Sbox address is loaded (zero−extended) into <dest>.

NOTE: stores are not visible within the Sbox until an SBOXSYNC instruction is executed, unless the <aliased> flag is indicated. The table designator (#<tt>) may be any value, however, performance of the SBOX instruction may be improved if each Sbox table has associated with it a different table (#<tt>) value.

SBOXSYNC.<tt>

Sbox #<tt> is synchronized with memory. Once this instruction is executed, stores to the Sbox memory since the last SBOXSYNC will become visible to later SBOX instructions.

XBOX.#<bbb> <srca>,<srcb>,<dest>

```
REG[<dest>] = 0
for (i=#<bbb>*8, j=0; i < #<bbb>+8; i++, j++)
  REG[<dest>].bit[i] <− (REG[<srca>] >> (REG[<srcb>] >> j*6)) & 1
```

The XBOX instruction performs a partial general permutation of register <srca>, given the bit permutation map in register <srcb>. The result of the permutation is placed in register <dest>.

Figure 8. Architectural Support for Symmetric Ciphers.

port rotate instructions, as a result, the addition of rotates saves 4 instructions (and 3 execution cycles). Nearly all the ciphers (except IDEA and Rijndael) have a fairly frequent usage of rotates.

The ROLX and RORX instructions support a constant rotate of a register input, followed by an XOR with another register input, and the result replaces the second register input. This was the only consistent opportunity we found to combine operations; all other combinable operations required at least three input operands. While this instruction does require three inputs, the third is a constant (within the instruction), as a result, there is no impact to the register files or renamers. The ROLX and RORX instructions are useful in speeding up Mars and RC6. Timing analyses indicated that rotates easily fit in the cycle time of a same-sized ALU.

The MULMOD computes the modular multiplication of two register values modulo the value 0x10001. This is a fairly fast operation, we use the algorithm detailed in [18]. The implementation requires a 16-bit multiply, followed by two 16-bit (parallel) additions and then two levels of MUX'ing. Timing analyses indicate the operation can complete in just over three cycles, we conservatively estimate that the operation can complete in 4 ALU cycles.

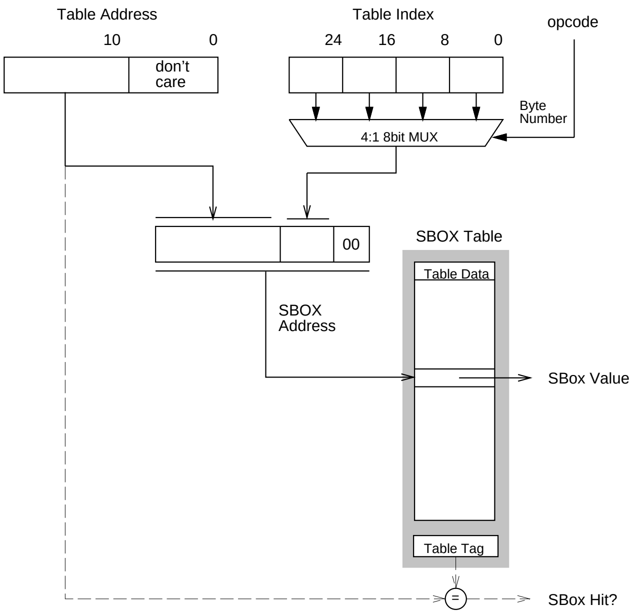

Figure 9. SBOX Instruction Semantics.

The SBOX instruction speeds the accessing of substitution tables. The instruction restricts SBOX's to 256-entry tables with 32-bit contents. As shown in Figure 9, an SBox is accessed by concatenating the upper bits of the table virtual address with eight bits extracted from the index register. The access returns from memory a 32-bit table value. The cipher algorithms studied all use 256 entry SBox's, with either 32 or 8 bit entries. We implemented 8 bit entries by zeroing the upper 24 bits of each SBox table entry. Other SBox table orientations could be efficiently implemented with these instructions as well. Smaller SBOX's could replicate SBox entries, thereby creating a don't care bit in SBox byte index. Larger SBoxes could be implemented by striping the table across multiple architectural tables and selecting the correct value based on the upper bits of the larger table index.

As shown in Figure 9, the SBOX instruction eliminates address generation (which takes a full cycle on the baseline machine). This is accomplished by restricting that all SBox tables be aligned to a 1k byte boundary. SBox address calculation then simplifies to zero-latency bit concatenation. To speed most SBOX operations, stores to SBox storage are not visible by later SBOX instructions until an SBOXSYNC instruction is executed. This optimization eliminates the need for SBOX instructions to snoop on store values in the processor core. Moreover, SBox implementations are possible that have separate storage that need not be kept coherent with cache memory. It was fairly straightforward to identify the locations to place SBOXSYNC instructions - always at the end of key setup routines which generated the key-based

	4W	4W+	8W+	DF
Fetch Speed	1 block/cycle		2 blocks/cycle	inf
Window Size	128		256	inf
Issue Width	4		8	inf
IALU resources	4 (1 cycle)		8	inf
IMULT/MULTMOD	1-64 (7 cycles)/2-32 (4 cycles)		2-64/4-32	inf
D-Cache Ports	2 (2 cycles)		4	inf
SBox Caches	0	4 single port (1 cycle)	4 dual ported	inf
Rotator/XBOX	2 (1 cycle)	4	8	inf

Table 2. Microarchitecture Models.

SBox entries. Notably, RC4 stores into its SBox table. To support this algorithm, we added an alias bit to the SBOX instruction, which if set allows later SBOX instructions to observe earlier store values. We implemented this form of the SBOX instruction by treating it as a load with optimized address generation.

We explored two SBox implementations: a simple cache-based implementation and a dedicated SBox cache. The simple implementation produces the SBox storage address and then sends the memory request to a data cache memory port. If the SBOX aliased bit is not set, SBOX instructions may execute in any order. As a result, these SBOX instructions need not enter the memory ordering buffer (the device that implements out-of-order load/store execution). The SBOX instructions simply enter the cache pipeline when a free port is available. With this implementation, SBOX instructions complete in 2 cycles, much faster than the 4 cycles required to implement SBox accesses with load instructions.

Our more aggressive SBox implementation adds four SBox caches to the microarchitecture. SBox caches have a single tag (the table base address), making them a one line sector cache [13]. Each SBox cache sector is 32-bytes in length (one data cache line). As shown in Figure 9, SBOX addresses are sent to the specified SBOX cache. The table indicator in the SBOX instruction allows the programmer to "schedule" the SBOX caches, specifying which cache contains a particular table. As a result, the underlying implementation need not implement a 4-ported 4k byte cache, but rather four faster single-ported 1k byte SBox caches. The instruction scheduler directs SBOX instructions to the correct SBOX cache based on the instruction opcode table specifier. The SBOX cache is virtually tagged, thus TLB resources are only required on misses. When the virtual tag does not match, the SBOX cache is flushed and the touched sector is fetched from the data cache. When the SBOXSYNC instruction is executed, all sector valid bits are cleared forcing subsequent SBOX instructions to re-fetch SBOX data from the data cache. On a task switch, the SBox cache is flushed by invalidating its tag. No writeback is necessary as SBox caches are read-only.

The XBOX instruction implements a portion of a full 64-bit permutation. The operation takes two input registers. One register is the operand to permute; the other register is a permutation map that describes where each input operand bit is written in the destination. The permutation map contains eight 6-bit indices, each indicates which bit from the input operand will be written in the output. The XBOX instruction opcode indicates which of the eight bytes in the destination register are permuted. The 32-bit permutations in the 3DES algorithm can be completed in 7 instructions (and executed in 3 cycles), a significant improvement over the baseline code which requires 39 instructions.

6 Performance Analysis

We hand coded optimized versions of each cipher kernel, and then examined their performance on four microarchitectures, ranging in cost and performance. Table 2 lists the four microarchitectures studied. The 4W microarchitecture is a typical high-performance four-issue microarchitecture with moderately sized memory system and resources. It is roughly modeled after the Alpha 21264 microarchitecture. In the 4W model, there are four ALUs, two data cache ports, and two Rotate/XBOX units that implement rotates and general permutations. SBOX instructions access the cache memory, thus they must compete with loads and stores for cache access ports. The 4W model also supports optimized multiplication. Word-sized (32-bit) multiplies have an early out after 4 cycles. In addition, modular 16-bit multiplies (modulo 0x10001) are implemented in hardware in 4 cycles. The 4W model can initiate one 64-bit multiply, two 32-bit multiplies, or two 16-bit modular multiplies per cycle. This resource configuration could be implemented inexpensively by mapping the shorter multipliers onto the 64-bit multiplier hardware. For example, a Wallace tree based multiplier can be converted to multiple shorter precision multipliers by simply isolating portions of the Wallace tree with MUXes.

The 4W+ microarchitecture improves the performance of SBOX instructions, reducing their latency and providing more bandwidth to SBox values without interfering with cache memory accesses. We added four SBox caches, each a 1k single-line sector cache. This configuration support four simultaneous accesses to four different SBox tables in a single cycle (four times the bandwidth of the 4W configuration). When a reference is made to an SBOX the tag is checked to see if it contains the referenced table, and then the valid bit of the sector is checked. If the sector is valid, the 32-bit referenced value is returned, otherwise, the SBox sector data (one data cache line) is demand fetched from the data cache. SBox storage does not observe stores to

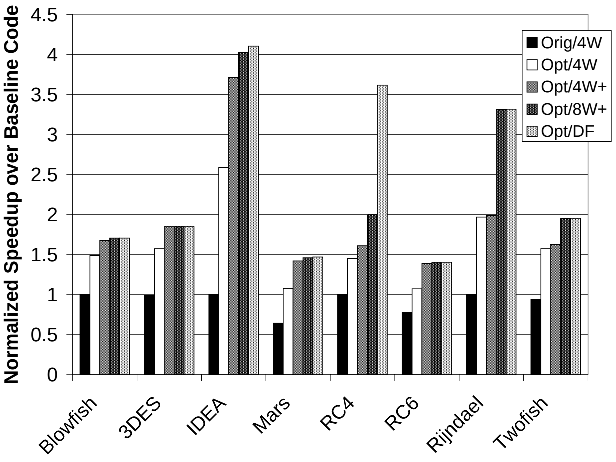

Figure 10. Relative Performance of the Optimized Kernels.

the data cache until an SBOXSYNC instruction is executed (which invalidates the SBOX tag). The 4W+ configuration also adds two more Rotate/XBOX units, thereby doubling the number of rotates and XBOX instructions that can be executed in a single cycle.

The 8W+ model provides approximately twice the execution bandwidth of the 4W+ model. This model doubles the issue width to eight, doubles the number of execution resources, and adds two more data cache ports. To accommodate the additional execution resources, the front-end performance is scaled to fetch two blocks per cycle, and the instruction window size is doubled to 256 instructions to expose more ILP. We don't claim that it would be wise to construct such a machine, we examine this configuration simply to demonstrate the headroom in performance if more resources were made available for cipher kernel execution. Finally, model DF is our dataflow model, described in the previous section. The maximum achievable performance of the re-coded kernels is given by these experiments.

Figure 10 shows the performance of the optimized codes running on the four processor models. All performance measurements are shown as speedups (in total cycles to process a 4k session) normalized to the performance of the original code *with rotates* running on the baseline microarchitecture with rotate instructions. Many architectures have fast rotates, so we felt that normalizing to this target was a more fair assessment of our instruction set extensions. For machines without rotate instructions, *e.g.*, Alpha-based processors, speedups are even more impressive!

The first bar, labeled Orig/4W, shows the performance of the original code (without rotate instructions) compared to the performance of the cipher kernels with rotate instructions. This experiment shows the impact on performance that an architecture will experience if it does not support rotates. Without a rotate instruction, rotates are synthesized using three instructions for a rotate by a constant amount (2 cycles to execute), and four instructions for a rotate by a variable amount (3 cycles to execute). The algorithms that most heavily use rotates, namely Mars and RC6, saw significant slowdowns of 40% and 24%, respectively. The "lowest hanging fruit" for architects to gather here are rotates. These simple instructions have little impact on the cost or cycle time of a machine and provide good speedups on three of the ciphers. Intel processors based on the P6 microarchitecture (PII, PIII) have good rotate performance. Measurements on a PIII found that the machine could sustain one rotate per cycle continuously. However, Intel recently announced that shifts and rotates on the Willamette microarchitecture would be at least twice as expensive as addition [32].

The second bar in Figure 10, labeled Opt/4W, give the performance improvement of the fully optimized (hand coded) cipher kernels running on the 4W model. Even on the less expensive microarchitecture, speedups for these new kernels is quite impressive. The kernels saw an average performance improvement of 59%, with IDEA seeing the best overall improvement of 159%. IDEA benefited from the faster and higher bandwidth modular multiplication support, an operation it uses frequently. Rijndael also saw very good speedups, with performance almost doubling. Rijndael benefited mostly from reduced latency for SBox accesses. With SBox support in hardware, these accesses reduce from three instructions to one, and speedup from five cycles to two.

Blowfish, 3DES, RC4, and Twofish all saw speedups near 50%. Like Rijndael, Blowfish, RC4 and Twofish benefited mostly from improved SBox access latency. 3DES saw benefits from improved SBox access latency and fast XBOX permutations. The outlier in these experiments was RC6. RC6 received most of its benefits with rotates; on the 4W microarchitecture it does benefit from fast modular multiplication, but only slightly.

The third bar in Figure 10, labeled Opt/4W+, gives the performance improvements with additional SBOX resources and rotator/XBOX units. Earlier results suggested that Rijndael and Twofish could benefit from more rotator and SBox resources, however, in these experiments they have both saturated the machine issue resources and thus cannot leverage more resources. Both ciphers are running at nearly 4 IPC in the 4W machine, additional resources are only useful if the microarchitecture can issue more than 4 instruction per cycle.

Finally, the fourth (Opt/8W+) and fifth (Opt/DF) bars show the optimized program performance with double the execution resources and infinite resources. As with the original code experiments, many of the cipher kernels are running near dataflow speed. Blowfish, 3DES, Mars, RC6 could not be sped up any more without reducing the latency of the individual operations. IDEA could benefit only marginally from addition resources. RC4, Rijndael, and Twofish have plenty of ILP to exploit, more resources improved their performance. In all cases except RC4, doubling the execution bandwidth exposes all available parallelism, permitting the ciphers to run at dataflow speed. RC4, on the other hand, still has a large supply of untapped ILP.

6.1 Discussion

Optimizing the architecture and software kernels has a powerful effect on the performance of the kernels, most are more than 50% faster after optimization. An important question to consider in light of these speedups is whether or not these optimizations affect the strength of the ciphers. The strength of an algorithm is defined as its resistance to attacks. There are two primary approaches to attacking a cipher, *brute-force attacks* and *cryptanalytic attacks*.

Brute-force attacks try all different key values until an intelligible message is found. Our optimizations certainly make this less costly for a given machine. Using the proposed optimizations, a single processor can now test roughly 50% or more keys in a given period of time. This performance improvement does not, however, make the algorithm more susceptible to attack since an attacker could just as easily use more machines to speed up an attack. Ciphers can be hardened to brute-force attacks by using longer keys, all the algorithms studied in this paper use at least 128-bit keys which are widely regarded as unsearchable key spaces for modern computing systems.

Cryptanalytic attacks apply a mix of logic, higher mathematics, linguistics, and brute-force attacks to essentially “derive” the input given information that is extracted from the ciphertext. This process is an attack on the cipher algorithm itself - a successful attack can compromise a cipher algorithm, rendering it useless. Since we do not modify the function of the cipher algorithms in any way, we do not affect their susceptibility to cryptanalytic attacks.

7 Related Work

Most published work on cryptographic hardware has focused on public key ciphers. The most expensive component of these algorithms is modular multiplication of multiprecision (1024-bits or more) operands. Most high performance algorithms are based on the Montgomery method [21]. There have been a number of proposals on how to speed this computation in hardware [31, 15, 30, 4], and Intel has demonstrated that the Merced iA64 processor has particularly good performance for this algorithm [22].

A number of algorithm-specific hardware implementations that have been described. IBM’s original DES proposal described a hardware implementation [11]. Shiva [28], IBM [16], Chrysalis-ITS [7], and Hi/FN [14] all offer high speed hardware implementations of the DES and 3DES algorithms. Published performance numbers for 3DES on these designs range from 8 MB/s to 58 MB/s. Our 3DES algorithm on a 1 GHz processor would achieve a performance of 12 MB/s - clearly there remains value in a hardware design for a specific algorithm. The details published on the IBM implementations [33, 34] are particularly interesting as they highlight other challenges that arise when developing a cryptographic processor including random number generation and key protection.

Hardware implementations have also been described for IDEA [18], Twofish [26], and Blowfish [24]. The FPGA research community has also shown that public key cipher algorithm performance can be improved using FPGA-based implementations [20]. While our approach cannot attain the peak performance that algorithm-specific hardware implementations can attain, our approach does provides the advantage of both performance and flexibility. Using a canonical set of symmetric cipher operations we speed the processing of many algorithms, possibly offering performance improvements for yet-to-be-developed algorithms. Given the wide variety of algorithms in use today, and the need for servers and clients to support different ciphers for different applications (or even connections), we feel there are benefits to providing instruction set support.

We are aware of only one previous proposal to add instruction set support for private key symmetric cryptography. Shi and Lee proposed adding an instruction (GRP) that supports efficient software implementations of general bit permutations [27]. Their approach is more efficient than our proposal. For a 32-bit operand they can perform any permutation in 5 instructions, our approach requires 7 instructions. Their approach also scales more favorably to larger operands. We are currently enhancing our tools to use Shi and Lee’s GRP instruction, however, we expect the performance impacts of this change to be small as none of our cipher algorithms have general permutation within their kernel loops. 3DES is the only algorithm that uses general permutations (for the initial and final permutations).

8 Conclusions and Future Work

As the Internet moves to the forefront as a trusted medium for commerce and communication, cryptography has become a integral part of modern information systems. We showed that even on very high-end microprocessors, common cryptographic algorithms (such as 3DES) cannot produce the throughput necessary to fully saturate a single T3 communication line - large web sites will often service many of these lines simultaneously. Furthermore, as the Internet and its applications move toward more security in communication, cryptographic bottlenecks will continue to grow.

We present detailed analyses of eight popular private key symmetric ciphers. We find that their performance and setup times vary widely. Analysis of their bottlenecks reveals that many of the algorithms run at near dataflow speed, those that do not simply require more resources for additional performance. Given these analyses, we proposed new instructions that speed the common operations of symmetric ciphers. Instruction set support is added for substitutions, permutations, rotates, and modular multiplication. We then examine their performance on microarchitecture models of varying cost and performance. Performance analysis of the optimized benchmarks revealed an 59% speedup over machines with rotate instructions, and a 74% speedup over machines without rotates.

Our analyses of the original and optimized algorithms suggest that there is more opportunity to improve the performance of cryptographic processing. Currently, we are exploring other optimizations for the eight kernels presented in this paper, including optimization of their setup routines and faster permutations. Looking further out, we are also exploring the possibility of hardware-cipher co-designs. Are there efficient functions that could be mapped to a processor pipeline what would at the same time be fast and have good diffusion properties? If so, it would make the case for a hardware-cipher co-design.

In this paper, we proposed techniques to add fast cryptography support to a general purpose processor. We are now exploring the implications of a design where the primary purpose of processor is cryptographic processing. Cryptographic processors would have to deliver orders of magnitude more performance to meet the bandwidth demands of secure servers and virtual private network (VPN) routers. This is quite an interesting design space: the processor must have general capabilities so that it can support a wide array of (possibly yet-to-be-invented) cryptographic ciphers, but it need not support the generality of all programs. SPEC performance is irrelevant for these processors, they need only execute cipher kernels quickly (both private and public key algorithms). Optimizations we are considering include fine-grained multi-threaded microarchitectures to extract inter-session parallelism, four operand instructions to permit increased operation combining, and microarchitecture-based loop optimizations. We are currently exploring these designs in depth and will report on their design and evaluation in a future paper.

Acknowledgements

We are grateful to Phil Yeh and the anonymous referees for their comments on earlier versions of this work.

References

[1] *Advanced Encryption Standard (AES) Development Effort*. US Government, http://csrc.nist.gov/encryption/aes/.

[2] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. *Proceedings of the ACM SIGMETRICS '96 Conference*, April 1996.

[3] R. Atkinson. Security architecture for the internet protocol. *IETF Draft Architecture ipsec-arch-sec00*, 1996.

[4] T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. *Proceedings. 14th IEEE Symposium on Computer Arithmetic*, pages 70–77, 1999.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[6] C. Burnwick and et al. *The Mars Encryption Algorithm*. IBM, http://csrc.nist.gov/encryption/aes/round2/AESAlgs/MARS, 1999.

[7] *Chrysalis-ITS Corporation*. http://www.chrysalis-its.com.

[8] *Counterpane Systems*. http://www.counterpane.com.

[9] *CryptSoft Technologies*. http://www.cryptsoft.com, 2000.

[10] J. Daemen and V. Rijmen. *AES Proposal: Rijndael*. http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Rijndael, 1999.

[11] D.W. Davies and W.L. Price. *Security for Computer Networks*. Wiley, 1989.

[12] P. Ferguson and G. Huston. What is a VPN. *http://www.employees.org/ ferguson/vpn.pdf*, 1998.

[13] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Palo Alto, Calif.: Morgan Kaufmann, 1990.

[14] *Hi/Fn Corporation*. http://www.hifn.com.

[15] J.-H. Hong and C.-W. Wu. Radix-4 modular multiplication and exponentiation algorithms for the rsa public-key cryptosystem. *Design Automation Conference (ASP-DAC 2000)*, pages 565–570, 2000.

[16] *S/390 and OS/390 Cryptography*. http://www.s390.ibm.com/security/cryptography.html.

[17] J. Keller. A superscalar alpha processor with out-of-order execution. *9th Annual Microprocessor Forum*, 1996.

[18] Xuejia Lai. *On the Design and Security of Block Ciphers*. Hartung-Gorre Verlag, 1992.

[19] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

[20] U. Meyer-Base and R. Watzel. A comparison of DES and LFSR based FPGA implementable cryptography algorithms. *3rd International Symposium on Communication Theory and Applications*, pages 291–298, 1995.

[21] P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[22] Stephen Moore. *Enhancing Security Performance Through IA-64 Architecture*. Intel Corporation, http://developer.intel.com/design/security/rsa2000/itanium.pdf, 1999.

[23] *An Introduction to Cryptography*. Network Associates, Inc., http://www.pgpi.org/doc/pgpintro/, 1999.

[24] R. L. Rivest and et al. *The RC6 Block Cipher*. RSA Security, http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RC6.

[25] *RSA Security*. http://www.rsa.com.

[26] B. Schneier and et al. *Twofish: A 128-Bit Block Cipher*. Counterpane Systems, http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Twofish, 1998.

[27] Z. Shi and R. B. Lee. Bit permutation instructions for accelerating software cryptography. *Proc. of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 138–148, 2000.

[28] *Shiva Corporation*. http://www.shiva.com.

[29] *The SSL Protocol, version 3.0*. Netscape, Inc., http://home.netscape.com/eng/ssl3/draft302.txt, 1999.

[30] C.-Y. Su, S.-A. Hwang, P.-S. Chen, and C.-W. Wu. An improved montgomery's algorithm for high-speed rsa public-key cryptosystem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(2):280–284, June 1999.

[31] W.-C. Tsai, C.B. Shung, and S.-J. Wang. Two systolic architectures for modular multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):103–107, February 2000.

[32] *Optimizing Software for the Willamette Architecture*. http://developer.intel.com.

[33] P. C. Yeh and R. M. Smith Sr. ESA/390 integrated cryptographic facility: An overview. *IBM Systems Journal*, 30(2), 1991.

[34] P. C. Yeh and R. M. Smith Sr. S/390 CMOS cryptographic coprocessor architecture: Overview and design considerations. *IBM Journal of Research and Development*, 43(5/6), September 1999.