

Tomorrow's Cryptography:
Parallel Computation via Multiple Processors, Vector
Processing, and Multi-Cored Chips

Eric C. Seidel, advisor Joseph N. Gregg PhD*

December 30, 2002

Abstract. This paper summarizes my research during my independent study on cryptography in the fall term of 2002. Here I state the growing need for better cryptography, introduce consumer hardware architectures of near future, and identify the growing discrepancy between the hardware on which current cryptographic standards were designed and the hardware the future consumer will be using. I note then the need for a new "modern" cryptography based on the presence of parallel processing capabilities in forthcoming consumer machines and the lack of support of such capabilities in some current and all legacy crypto algorithms. I list approaches used in past research to parallelize cryptographic algorithms. I then summarize various current algorithms and potential implementation changes to ready them for tomorrow's machines. I conclude with some brief discussion of newer cryptographic algorithms, particularly AES and AES finalists and how they will fare on the machines of the future.

*Eric.C.Seidel@lawrence.edu; Joseph.N.Gregg@lawrence.edu

Contents

1	The future of crypto	3
2	Parallel crypto of today	6
3	The importance of data-level changes	8
4	Making data-level changes	9
4.1	Hashing Algorithms	12
4.1.1	MD5 - Message Digest 5	13
4.1.2	SHA-1, Secure Hash Algorithm - Revision 1	14
4.1.3	RIPEND-160	15
4.1.4	Tiger	16
4.2	Block Cyphers (Secret Key Cryptography)	17
4.2.1	DES - Data Encryption Standard	18
4.2.2	3DES - "Tripple-Des"	20
4.2.3	Serpent	21
4.2.4	Twofish	22
4.2.5	Rijndael - the American Encryption Standard	22
4.2.6	RC6	23
4.3	Public-Key Cryptography	24
4.3.1	RSA - Prime Factorization	26
5	Final Thoughts	27

1 The future of crypto

From bank accounts, to medical records, personal emails, and more, increasingly more and more sensitive data is stored digitally. With the continued growth of the Internet, more and more of this data resides in places which themselves may not be secure from intruders, and much of this data is transferred daily from place to place across connections inherently insecure. To solve these problems of digital data security, we have cryptography. Most cryptography however has historically been used by governments, larger business and computer geeks and not by the average consumer. But, needs are shifting, and consumers are increasingly using encrypted emails, encrypted file systems, and soon smart cards: the most drastic change to consumer crypto. Today, the most commonly seen form of crypto to the consumer is the DES¹ encryption used between the customer's local ATM his/her bank, or the SSL/TLS² protocol securing the credit card transaction as they purchase over the internet. Now, and in the future, the consumer will be using his/her home computer, and eventually his/her personal smart card, for more data security. Both the load on the consumer's PC and the load on the central servers of the internet, in terms of cryptographic computations will increase. This growth will demand a new level of cryptography, a consumer level cryptography, one designed to work on modern architectures, and one which will be fast, cheap, secure, and transparent.

In addition to the stress caused on our cryptographic world by explosive usage, when one looks at the computers in use today and those which will be in use in the future, and then compares them both to the computers for which our current cryptographic standards

¹Data Encryption Standard

²Secure Socket Layer/Transport Layer Security

were designed, one finds great discrepancies. This discrepancy will cause future computing stress due to inefficient cryptographic implementations on future hardware. Prior to the solicitation for the American Encryption Standard (AES), the cryptographic world was based on a single, 32, 16, or even 8 bit processor. But increasingly the computer world of today, and most definitely that of tomorrow, is not one of the 32-bit desktop, but rather one of multi-cored chips³, multiple processor machines, and larger 64 or even 128-bit processors, many with a Vector Processing Unit (VPU)⁴. This a large change in the machinery of the consumer, and cryptography must be make ready for this change.

It is important to note that parallel-ready cryptographic algorithms have a vast number of uses, ranging from high-end, high-load servers, down to the averaged consumer's desktop, and many systems in between. Currently, high end servers inefficiently perform crypto using per-process or per-connection based parallelism⁵, using algorithms not designed to run efficiently on their hardware. Provided algorithms designed (or modified) to run on newer hardware, the same servers could serve many more clients, or serve with the same efficiency a higher percentage of clients using encryption.

Making crypto ready for tomorrow's architectures also allows greater speed of cryptography at the consumer's end. Already we are seeing interesting fast applications of AES such

³An IntelTMtechnology. By placing multiple processor cores on the same piece of silicon Intel can drastically reduce the cost of having more than one processor. They reduce cost associated with the amount of silicon used and the cost of all the additional architecture (buses, memory, caches, etc.) associated with a completely separate processor. Itanium (Intel's new 64-bit RISC processor) based, multi-cored chips are scheduled to ship by 2005.

⁴In contrast to scalar processing, a Vector Processing Unit (VPU) works on "vectors" of data, and performs the same operation (add, multiply, AND, OR, etc.) over a set of processor words, just as would be performed on a single processor word, except now multiple words are operated on all in a single cycle. This method of parallelization is commonly referred to as Single Instruction, Multiple Data (SIMD) computing.

⁵Described later in greater detail: each processor receives a single connection or process for which that processor processes all instructions, including encryption and decryption for the connection.

as Apple Computer's encrypted disk image technology: an encrypted virtual disk that can be read nearly as fast as unencrypted disk access due to an efficient AES implementation on their hardware⁶. Such technologies will become the norm, not the exception. With fast enough cryptographic algorithms, all data written out (to storage media, networks etc) could be done so in an encrypted fashion. Currently the number of simultaneous secure connections the average consumer has open at one time is small. But, that number will increase with things such as encrypted chat, secure video streaming, secure email, VPNs, and secure connections to handheld computers, cell phones, and other devices. Until cryptographic algorithms are ready to be performed at great speeds on the computers of the future, many of these applications listed remain very slow and difficult if not impossible on today's systems with current crypto algorithms.

This leaves us then with an interesting problem. We have a world increasingly in need of greater crypto speed, and yet one which is at the same time radically changing hardware architecture, and yet is still using 28 year old crypto algorithms. We are entering into a world in which parallel ready software is a must, and it is thus time that our cryptography software be brought into the 21st century. Some, perhaps much, of this shift to better, more flexible crypto has already begun by the influx of new algorithms via the AES solicitation. But recognizing that not all systems will be as quick to change, and that often new cryptographic algorithms take years, if not decades to be accepted, it is an important question to explore as to what if any changes, and amendments can we make to existing cryptographic standards to bring them into the future.

⁶I personally watched a technology demonstration in which a high data-rate movie was played directly from such an encrypted disk.

2 Parallel crypto of today

There are a number of approaches already made to adapt our aging cryptographic algorithms to newer hardware, and it is those that I will first look at here [13]. I will classify methods of parallelization based on at what “level” of the computer system parallelism is applied. I will be referring specifically to network cryptography, but analogous examples exist in single user (local) cryptography. The levels of classification I will use are: per-connection (per-process), per-packet (per-file), and inter-packet (data level) parallelism [14].

A per-connection parallelism method is one whereby each connection to the server, or from the client, is given its own thread or process that runs exclusively on one processor. This is the most common method of parallelization, and requires no modification to the existing algorithm and often no modification to the existing server software. By running multiple instances of the server process one can often achieve this level of parallelism. This method is limited in its ability to speed up a single connection, offering high-speed, single-connection cryptography no benefits (i.e. writing/reading to/from local media). This level of parallelism also does not address the question of running older algorithms on newer processors. By simply running a single algorithm instance per processor, this method makes no attempt to make the old algorithm run any more efficiently on each newer processors than the algorithm did on the previous architecture. This is the most common method of parallelism found in cryptography today, and the one which is most often the selling angle for multi-processor machines which run cryptographic applications. The per-connection parallelization method makes no attempt to fully utilize modern architectures and will not be discussed further in this paper.

Per-packet parallelism is a method in which connections disperse their packet processing load over multiple processors, wherein each packet is treated individually. One example would be a group of processors (or threads) handling the actual logic for all connections, then placing each prepared packet in a buffer, where it is handled (encrypted) by a group of processors. In this design a single processor might handle packets from various connections, or a single connection might use packets encrypted by various processors. This is similar to the design of specialized cryptographic hardware, where the cryptography portion of an application is offloaded to (a) specialized processor(s). Many current algorithms lend themselves well to this kind of parallelization, but surprisingly I encountered found no implementations in software of this per-packet parallelism.

Intra-packet parallelization is the most difficult type of parallelism to introduce post-facto, dependent heavily on algorithm design. This type of parallel processing is one which has historically not been addressed as much as other methods, but is part of the main focus of this paper. An example of this method would be a block-cypher such as DES running in ECB (Electronic Code Book) mode, whereby each block of the message is computed independent of the others and could be computed in parallel. This level of parallelization requires changes to the implementation of the cryptographic algorithm itself, depending no longer on the flexibility of the hardware or operating system on which it is run. The various methods by which this type of parallelism can be achieved will be discussed below.

3 The importance of data-level changes

Moving beyond the per-process, per-connections models, and down to the data levels allow us to fully exploit some of the growing technologies on the market today. At least one CPU – Motorola’s G4 – already ships with a VPU (the AltiVec Engine), making vector processing power available to the consumer. Intel has promised to begin shipping a multi-cored version of its new Itanium processor by the year 2005 and we will undoubtedly see other parallel architectures continue to enter the consumer market. In order to fully exploit these technologies, we can no longer depend on the flexibility of operating systems, or the seeming unending megahertz climb, but must reconsider our cryptographic algorithms to utilize these future architectures.

I list here two of many important reasons which support a reassessment of cryptography at this level:

1. Many cryptographic algorithms are processor inefficient on modern hardware. (One example is DES, which at most times uses only 4 or 6 bits of any processor register, when running on a 32-bit processor, that’s only 12% - 16% efficiency! When running on a 64-bit processor we see an efficiency of half that – ignoring any potential RISC⁷ optimizations.
2. Cryptographic algorithms in general make no accommodations for parallelization, thus

⁷Reduced Instruction Set Computing – allows a processor to perform several small (4,8, or 16 bit) operations in parallel in a single clock cycle. This is in contrast to Complex Instruction Set Computing used on current 32-bit Intel processors which only allows a single large (32-bit) operation per clock cycle. Instructions which on a complex instruction set processor exist as single instruction, often require more than one instruction on a RISC processor. Those same sets of instructions on the RISC processor can often still be performed over a single cycle in parallel. If the set of RISC instructions are dependent on one another, than they can be interleaved with other small (i 64 bit) operations in the pipeline.

neglecting possible gains on long term computations under multi-processor environments.

These reasons and more lead cryptographers to seek changes to our legacy algorithms at the data level. It is how we go about making these changes that I will now explore.

4 Making data-level changes

Parallelization at the data-level can allow algorithm speedup in the following ways:

1. By performing the same calculation on a larger amount of data. Performing the same calculation on large amounts of data concurrently is the most common technique discussed in this paper and is the technique used by Vector Processing Units and SIMD architectures. Multi-cored chips and true multiple processor architectures also can use this type of parallelism. Utilizing the advantages in this type of computing is important for cryptography because it is these SIMD or VPU architectures which are the most common form of parallelism available on modern computers.
2. By performing two distinct parts of a single algorithm at once. This is only possible in true multiple processor environments, allowing multiple individual processors to handle separate parts of an algorithm. A common technique of this type is pipelining: sending data from one processor to the next down an assembly chain of sorts. Allowing the computation of n sequential steps of the algorithm in parallel over a single clock cycle on n processors. An example of this is to let each processor do a single cryptographic round on data passed to it from a high data-rate network stream. If each processor

is able to complete a single round of the cypher in time t , we can add n rounds of encryption to our final cyphertext by adding n processors [4]. By doubling the number of processors we can in effect double the security of the data stream with no effect on data-rate. Other techniques of this type often require specific algorithm design modifications and introduce processor scheduling concerns, and therefore remain less common.

3. By making a single complex calculation faster (e.g. BigNum exponentiation) – distributing it's load over multiple processors. This is actually a layer below algorithm design, and depends on the implementations of the library from which the algorithm's implementation draws. This is useful in areas of cryptography where math intensive operations are performed over large data sets. A good example of such a area is Public Key Cryptography. Here math speed gains can be exploited from any VPU or set of processors as long as one has the knowledge and/or the vendor supplied math libraries to take advantage of the parallel processing power.

There are a couple common techniques and pitfalls in applying parallel processing to various cryptographic algorithms which I will mention here:

- *Hardware in Software* - Sometimes when moving from a system designed for a single smaller processor to an architecture including larger processors (or parallelism of any form) it is useful to take a step backwards before proceeding forwards. Such was work of Eli Biham, when he noticed that speed gains could be achieved for DES by implementing the hardware (logic gate) version of DES in software running on 1-bit or larger processors. Biham noticed that by viewing a larger than 1 bit processor as

an array of 1-bit SIMD processors, and processing the algorithm according to the logic gate implementation substantial speed could be gained. This approach is commonly referred to as the “BitSlice” implementation and is described in greater detail below in reference to DES. BitSlice ideas also can have applications in other algorithms.

- *SIMD on any processor* - Another technique when moving from a smaller processor (or single processor) to a larger (or multiple) processor(s), is to view the larger processor as an array of SIMD processors the size of the original smaller processor. This allows packet-level (file-level) parallelization of an algorithm, by computing two or more instances of the same algorithm at the same time across multiple packets or files. This implementation is only efficient under certain algorithmic design constraints and fails under such circumstances as when value based lookups are necessary. In cases where parts of the vector must be treated differently based on their 32-bit (or smaller) value, the implementation fails. Using lookups as an example, there are workarounds, but those workarounds often lose much efficiency. Lookups could also be translated into much larger entire-vector based lookups, but the tables required for such would be enormous. This implementation also runs into similar difficulties with other operations such as rotations, seeing as that rotations performed on larger words than the original intended will require significant intra-word adjustments^{8, 9}. This method of

⁸On an n -bit processor emulating r q -bit processors where $r \cdot q = n$. This can be accomplished in at most 5 operations, assuming rotation over n bits costs 1 operation and the mask words M and M^{-1} are pre-computed. Reduce the rotation to its smallest equivalent right rotation e , ROTATE the larger n bit word e places, AND the result R with a “mask” word M consisting of r sets of e 1’s followed by $q - e$ 0’s. , ROTATE the result R' $n - e$ places to form R'' , AND R'' with the inverse mask M^{-1} to form U , OR U and R'' for your final result.

⁹Without full knowledge of the bitwise implementation of PLUS and MULT operations, I must make this “SIMD on any processor” assertion with reservation. Without more exact knowledge I am unable to state the cost of performing r simultaneous q bit MULT or PLUS operations on an n bit processor. It may be

SIMD on any processor is very effective, but only in special cases and depends heavily on the processor on which it's implemented.

- *The problem of chaining* - Many cryptographic algorithms, in order to achieve increased security, or simply by their fundamental design constraints (e.g. hashing), involve chaining of information from one cypherblock to the next, introducing “recursive dependency”¹⁰ into the algorithm. This dependency makes applying block level parallelism to the algorithm impossible and will be seen in many algorithms.

With these methods and my further comments to techniques and pitfalls in mind, I then made a systematic review of various algorithm types attempting to apply these principles to each. The following are the results of my review:

4.1 Hashing Algorithms

Hashing algorithms take in a large block of data (normally a file, or a network packet) and compute a unique “hash” value, much shorter than the original data. This “hash” can be then passed around with (or separate from) the original data, and be used to verify the integrity of the data set. Hashing functions are often used in conjunction with Public Key Cryptography to produce “signed hashes” – short secure representations of the larger data.

The hash is first computed, and then “signed” to prevent a man-in-the-middle from simply re-computing a hash for the altered data. Signing only the hashes saves both parties from

much greater than I have assumed here. RISC processor may also have no problems with these.

¹⁰Lacking a better word, I will refer to the round-to-round and block-to-block dependency of various functions in various algorithms as “recursive dependency.” Hinting to the dependency introduced by applying the function to the same (or parts of the same) data in a recursive fashion.

the enormous expense of signing, or verifying a signature over a large block of data, but still offers similar integrity and authenticity verification, due to the uniqueness of the hash.

The speedup of hash functions is important to allow greater speed and security for those wishing to hash and sign each packet, or when hashing extremely large chunks of data, or sets of chunks of data (e.g. storing and verifying hashes for all executables on a public server). Hash functions are already in general quite fast, but many do not support modern architectures well, and thus waste many unnecessary CPU cycles. Making hashing even faster on modern processors would open the doors to potentially more new uses, and make current uses more convenient.

4.1.1 MD5 - Message Digest 5

MD5 is a 128-bit hash function operating on 32-bit words. MD5 was designed by Ronald R. Rivest and is the successor to MD4 (also designed by Rivest). MD4 however is no longer considered useful for security after several successful collision attacks in years past and some wonder if MD5 isn't reaching the end of its useful days[1]. MD5 involves a sequence of XORs ¹¹ and 32-bit addition on 32-bit blocks of data producing a series of four 32-bit chaining variables. These chaining variables are carried through the entire hashing process, and compose the final 128-bit signature. MD5 like nearly all hash functions is based off of these chaining variables which introduce recursive dependancy and prevent any direct parallel implementation.

¹¹I refer to the following common binary operations throughout:

		1	0			1	0			1	0
AND	1	1	0		1	1	1		1	0	1
	0	0	0		0	1	0		0	1	0

MD5 could potentially allow the SIMD implementation described in the previous section. This would allow packet-level (file-level) parallel computation on a single 64-bit or larger processor (grouping 32-bit data blocks as vectors of data blocks and performing the same MD5 calculations based on those vectors). Computing MD5 over multiple buffers in parallel on an SIMD architecture supporting 32-bit adds, could theoretically cost exactly the same as running a single buffer on the same $n \cdot 32$ bit chip, and would thus offer an n fold speedup over a normal implementation on that same chip. MD5 allows for possible SIMD architecture packet-level optimizations, but shows no promise for other parallelization techniques.

4.1.2 SHA-1, Secure Hash Algorithm - Revision 1

SHA-1 is 32-bit dependent 160-bit hash function operating on 32-bit words. SHA-1 was designed by the NSA as the successor to SHA-0 which was replaced due to an undisclosed vulnerability resulting in a collision of the hash function at under 2^{80} blocks¹². SHA-1 is quite popular (used commonly in SSL and distributed on nearly every *nix¹³ distribution) and is regarded as very secure. SHA-1 provides a slightly more complex non-linear function f , and a larger sized hash than MD5. SHA-1 relies on five 32-bit chaining variables, introducing a recursive dependency. This recursive dependency again preempts any attempt to process a group of blocks from a single packet in parallel. SHA-1, like MD5, can be implemented in an SIMD fashion on 64-bit or larger chips to achieve packet level parallelism, but I know of no 64-bit native implementation or other multiprocessor or larger processor optimizations. SHA-1

¹²To the best of my knowledge, this vulnerability has still not been discovered. However SHA-0 is no longer commonly in use.

¹³*nix is used to denote any of a variety of UNIXTMlike operating systems, including the BSDs, Solaris, Linux, and most recently Mac OS X.

was designed for software implementations on 32-bit little endian machines and is regarded as the fastest of the commonly used hash functions. SHA-1 like MD5 allows for possible SIMD architecture packet-level optimizations, but shows no promise for other parallelization techniques.

4.1.3 RIPEMD-160

RIPEMD-160 is a 160-bit hash function operating on 32-bit words. RIPEMD-160 was designed by the RIPE consortium as a more secure replacement for RIPEMD (a 128-bit hash function with similarities to MD4). The number of chaining variables (32 bits each), is increased from three to five from RIPEMD to RIPEMD-160 and the number of rounds for each block from three to five [10]. Like nearly all hash functions RIPEMD-160 has block-to-block recursive dependance (provided by the chaining variables) preventing any attempt of computing blocks in parallel. RIPEMD and RIPEMD-160 both however have some intrinsic parallelism, computing two halves of the each block in parallel. These two sets of 32-bit operations could theoretically be done in parallel on a 64-bit processor viewing the 64-bit processor as a 2×32 -bit SIMD processor. Processors and VPUs larger than 32 or 64 bits could also use a packet-level parallelism as suggested for other hash algorithms, applying the same algorithm to two or more blocks at once on a 128-bit or larger processor or VPU. I found no mention of the number of gates required to implement RIPEMD in hardware, however it should also be possible to implement an efficient BitSlice version of this depending on the number of gates required for RIPEMD-160. RIPEMD, like SHA-1 and MD5 shows potential for packet-level SIMD optimizations, but also due to its minimal intra-round parallelism should be more efficiently implemented as-is on 64 bit RISC architectures.

4.1.4 Tiger

Tiger is a 192-bit hash function operating on 64-bit words. Tiger was designed by Ross Anderson and Eli Biham to work efficiently on 16, 32 and 64-bit processors. Tiger performs eight parallel 8 bit lookups for each round using 64-bit S-Boxes¹⁴ which take 8-bit lookup values, thus effecting every bit of the final word with each lookup. The results of these lookups are combined in a series of boolean expressions and carried as two of three 64-bit chaining variables. Tiger, like all other hash functions, is recursive dependent due to its chaining variables and does not allow block-level parallelism (i.e. distribution across multiple processors). Unlike other algorithms however Tiger works well on 64-bit architectures, achieving speeds 2.5 times that of SHA-1 on 64-bit machines according to Anderson [5]. Tiger unfortunately does not appear at first glance to work as well on 64-bit or larger VPU's however since it is dependent on 8-bit lookups, which are scalar operations and would require moving out of the vector registers. Tiger achieves it's efficiency on 64-bit processors due to their RISC design. Tiger does not appear to be widely used at present. Tiger, unlike the others mentioned is the only hash function designed for 64-bit processors and appears to be the fastest hash function on 64-bit architectures. Tiger appears to take good advantage of RISC design and should run well on 64-bit hardware and perform efficiently with (SIMB based) block-level parallelism on any $n \cdot 32$ bit architecture.

¹⁴Substitution-Boxes: a non-linear function which maps from $[0, 2^n] \rightarrow [0, 2^m]$, commonly implemented in a table fashion. S-Boxes refer to the table form. This same function is sometimes also referred to as f but f can correspond to other parts of the round function depending on the algorithm.

4.2 Block Cyphers (Secret Key Cryptography)

Block cyphers are the most common type of cryptography, and are used for many general purpose tasks, including encrypting large blocks of data, generating large random numbers, and generating another class of cyphers called “stream cyphers” – used for very long, continuous sets of data, such as multimedia streams. Block cyphers are truly the work-horse of cryptography.

Block Cyphers have several different modes of operation, and generally the ability to answer the problem of parallelization depends on the mode in which one uses the cypher.

Those modes are:

1. ECB (Electric Code Book) - Each block of the cypher is computed separately. This mode is the most commonly used in real world applications, but is less secure than any of the other modes described and is vulnerable to attacks under which a single plaintext, or partial plaintext is known [6]. For modern algorithms with large block sizes, the number of known plaintexts which required for an attack is extremely large. It a good idea regardless when using block cyphers in this mode to change keys often (at least every $n/2$ blocks where n is the smallest number of required plaintexts for a known attack). This mode lends well to parallelization as we will see below.
2. CBC (Cypher Block Chaining) - Each block is computed after first XORing the plaintext of this block with the cypher text from the previous block. This introduces bit dependancy from block to block and assures that two cypher blocks which appear identical have no relation in the plain text. This is generally one of the most difficult to parallelize since every block depends directly on the previous block. Single packet,

block-level parallelism is impossible when using this mode.

3. CFB (Cypher FeedBack) - Each block of cyphertext is computed by first encrypting the previous block's cyphertext (again) and then XORing that result with the plaintext of this block. Parallelization of block cyphers in this mode shows similar difficulties to those experienced in CBC mode.

Other factors, however, including block-size effect our ability to parallelize cypher block algorithms. Such difficulties are explained in detail below.

4.2.1 DES - Data Encryption Standard

DES is a 32-bit block cypher which belongs to a class of cyphers called Feistel networks. In Feistel networks, blocks are divided into equal sized high and low components, one component has a non-linear function applied to it and the result of that application is eXclusive OR'd (XORed) with the other. DES suffers on modern processors not only from a 32-bit dependancy, but also from inefficiencies in its standard 32-bit implementation which cause often only four or six bits of each 32-bit register to be used. DES is however the most popular block cypher, designed back in 1974 [2]. DES was originally intended only for a few years of use, but has survived over 28 years, and is still regarded as cryptographically sound, even if it is limited by a short key length and small block size [1].

As DES is by far the most commonly used block cypher, there have been several attempts at parallelization including a most ingenious suggestion by Eli Biham, commonly referred to as the BitSlice implementation, and employed in various modern algorithms, including Serpent by Ross Anderson and Eli Biham. BitSlice implementations solve the problem of

DES not using registers efficiently during computation. BitSlice implementations regard each n bit processor as an $n \times 1$ SIMD processor and perform the hardware implementations of DES on this SIMD array [11]. This turns out to be significantly faster, due to the extreme inefficiency of DES on complex instruction set processors with registers larger than 8 bits. Biham also states that his BitSlice implementation has been quicker on RISC processors as well. A BitSlice implementation can efficiently compute up to x blocks in parallel on a x bit processor. [6].

The problem of parallelization of DES is addressed differently when DES is used in the different modes mentioned above. Of the modes listed, DES when used in ECB mode is the most easy to parallelize. A DES implementation in ECB mode can take multiple blocks at a time processing them simultaneously, then append one to another to form the final message. CBC or CFB in DES are not readily parallelized, since each block depends on the computed cyphertext of the block prior. The BitSlice implementation given above also does not function with DES in CBC mode.

An alternative packet-level implementation could be possible. In such a packet-level parallel implementation, a server would write all data needing encryption to a buffer. The server implementation would then use an x -bit BitSlice implementation to process up to x different packets in parallel one block from each at a time. This packet-level parallelism should work regardless of the mode of the cypher used provided that the mode is the same for all packets computed in parallel. To the best of my knowledge no such system has been attempted to date, but from my understanding of Biham's work it should be possible and would offer busy servers such as Amazon.com or ATM controllers a boost in their crypto processing power. This packet-level parallelism is also even more efficient when packet lengths

for all n packets are the same.

Also worth noting, is that the limitations of CBC and CFB modes on parallelization are *not* present during the the decryption stage of DES. This is due to the fact that all encrypted blocks are already known, thus all blocks of the message can be decrypted simultaneously, and then XORed with the appropriate cyphertext. This allows full block level parallelization when running decryption of DES cyphertext under any mode.

DES was never designed to be run on parallel architectures, and does not even run on current 32-bit architectures with particular efficiency. But due to it's long history as a cryptographic standard DES has substantial research surrounding it. This research has provided efficient implementations of DES for modern and future processors , including the ECB block level parallelization, the BitSlice implementation, and the CBC/CFB decryption optimization.

4.2.2 3DES - “Tripple-Des”

3DES is merely the application of the DES cypher three times over a single message, using three different keys. This is accomplished by chaining encryption-decryption-encryption together, in any of the modes mentioned above. This effectively triples the number of rounds of the DES algorithm, and triples the secret key length. Just like DES, when 3DES is used in ECB mode it is easily parallelized by distributing packets among various processors, or over a vector and using a VPU. The block-level parallelization possible with ECB mode can be exploited best when performed in combination with Biham's BitSlice implementation.

Due to the inverse block scheduling seen during the decryption stage, 3DES is used in CBC mode is unfortunately more difficult to parallelize than I had originally thought.

3DES encryption runs from block $0 \rightarrow n$, but decryption runs from $n \rightarrow 0$, making direct block-level pipelining impossible. One can however still get a speed boost in 3DES CBC and CFB modes by decrypting all blocks in parallel using the decryption trick mentioned for DES. I know unfortunately of no library which exploits this decryption trick when using 3DES. 3DES shows similar parallel implementations to those seen for DES, and although never designed for future hardware, can be implemented efficiently there.

4.2.3 Serpent

Serpent is a modern 128-bit block cypher using 256-bit keys for computation and accepting a variable key size. Serpent was designed by Ross Anderson, Eli Biham, and Lars Knudsen and was one of five finalists for the AES competition [15]. Serpent was designed with principles similar to those of the BitSlice variant of DES, and offers encryption better than 3DES at speeds comparable to DES[6]. Due to Serpent's design, Serpent works well with any size processor 1-bit or greater, can easily be parallelized out to Symmetric Multi-Processing (SMP) architectures, VPUs or multi-cored chips. Serpent seems to receive comparatively little usage at current even though it may offer slightly greater cryptographic strength than AES. One potentially interesting future project would be to implement a VPU or SMP optimized version of Serpent and compare the speeds with efficient AES implementation on the same platform. Serpent, unlike DES and 3DES was designed specifically for efficient computation on any sized processor and additionally works well on VPUs and multiple processors.

4.2.4 Twofish

Twofish is a 128-bit block cypher which accepts key lengths up to 256 bits. Twofish belongs to the class of algorithms referred to as Feistel Networks. Twofish was designed by Bruce Schneier and was also one of the 5 AES finalists. Because of the larger block size it is also safer to use Twofish in ECB mode than it would be for DES or 3DES. This larger block size might allow a parallel ECB mode Twofish to be used for many more years than DES or 3DES[17]. As a Feistel network Twofish should also be parallelizable during decryption in any mode. Twofish should also have a corresponding BitSlice implementation which may or may not be faster depending on the number of gates required. Twofish, like DES and 3DES does not seem to have been designed with parallelization in mind, but because of its similarities to DES should be parallelizable in the same ways.

4.2.5 Rijndael - the American Encryption Standard

Rijndael is a variable block length and variable key length cypher accepting blocks and keys of sizes 128, 192, 256. Rijndael was designed by Joan Daemen and Vincent Rijmen, and was the winner of the 1998 AES solicitation[18]. The Rijndael algorithm consists of four steps per round: byte substitution, word rotation, word swapping, and round key addition [9]. The designers claim an efficient parallelization of Rijndael on 64-bit RISC processors, commenting that each of these four operations operate on byte-words and should be computable in parallel [8]. They note specifically that on RISC architectures, the bit-swap operation should automatically be done in parallel. I agree with their comments that a 64-bit RISC version should be efficient, but it is not however as readily apparent to me

that an efficient implementation of Rijndael exists on VPU enabled systems, or systems with larger 128-bit or non-RISC processors. Further analysis would be required to determine these limitations, and due to the relatively newness of the cypher such analysis was not already available, and lies beyond the scope of this paper. Rijndael was designed for the AES solicitation, which included efficient implementation on future processors as a goal, does not appear to have been stressed as much as efficient memory requirements and efficient implementation on smart-cards and other hardware devices. This said, Rijndael still shows promise for efficient implementation on 64-bit risk architectures, and it is my prediction that due to it's official status as AES, efficient implementations will be quickly found for Vector Processing Units, larger SIMD architectures, and multiple processor systems.

4.2.6 RC6

The RC6 block cypher is a variable block length and variable key length cypher accepting block and keys, of any size. RC6 was one of five finalists in for the National Institute of Standards and Technology's (NIST) AES solicitation. RC6 is an evolutionary improvement on RC5, designed by Rivest, Robshaw, Sidney and Yin[16]. RC6 is most simply viewed as computing parallel copies of RC5 over a double sized block and swapping half of the parrallel chaining variables after each round. (There is some initial and final round computations which make the algorithm a bit more complex, but the details of such are not necessary here.) This leaves us with two parallel 32-bit dependent block cyphers which expand well to a 64-bit RISC processor capable of efficient 32-bit multiplies. RC6 should have an efficient implementation on a 64-bit RISC processor, but does not appear as if it would lend itself well to a VPU implementation or implementation in parallel beyond two 32-bit processors[16]. Eli

Biham's BitSlice idea could also be applied here for processors and VPUs with register sizes beyond 64 bits, but further parallelization does not seem obvious. RC6, like all the other AES candidates has not been around long enough for extensive review of its parallelization capabilities. It does not however seem to be designed with as much architectural flexibility in mind as say Serpent or possibly Rijndael but still offers at least a good implementation on 64 bit processors, leaving some work to be done towards VPU and other multi-processor implementations.

4.3 Public-Key Cryptography

Public Key Cryptography is a cryptography variant whereby there exists not only a secret key, but also a public key. With the public key, any individual can encrypt data which is only decryptable by the original secret key. It is also essential in a PKC system that encrypting the data with the public key lends no information about the secret key. There are several different PKC systems, but I will only write much about the most popular: RSA. One factor limiting my discussion are the absence of any published standards for PKC¹⁵ thus implementations and designs vary from version to version, making precise discussion difficult. Another factor is that I feel discussing the types of PKC algorithms can be as effective as discussing the individual algorithms themselves.

PKC infrastructures, differ greatly from Block-Cyphers or Secret Key cryptography. PKC works from the assumption that there exist functions for which the application of the function is comparatively simple, yet the inverse of such is computationally hard, specifically

¹⁵The The Institute of Electrical and Electronics Engineers, Inc. (IEEE) P1363 working group is addressing this issue, but they have as of yet no final specifications for PKC.

at least NP-hard. This allows for a separate encrypting key and decrypting key. The encrypting key, performs the non-invertible function, and the secret key provides the necessary clues to quickly find the original plaintext on which the function was applied without having to compute the function's inverse.

There are a few different functions which are currently believed to have this non-invertible property and are used in PKC. The first and most common problem used is the problem of factoring large composite numbers into numbers composed of two large primes. It is comparatively simple to generate large prime numbers, and equally simple to multiply large primes. But it is computationally difficult to factor large numbers, as exemplified by such real world problems of factoring for large primes such as GIMPS (Great Internet Mersene Prime Search). It is based off of this property that we have the RSA and factoring-based Public Key Crypto systems [1].

A second is a class of problems called discrete logarithms, and deals with finding n given only some y such that $y = g^n$. The problem of discrete logarithms is simple for integers but much more difficult when dealing with a large finite field using modular arithmetic. The problem of discrete logarithms used in several different PKC systems and is considered as hard as factoring and is believed to be NP-hard [1].

A third class of problems are those which compose the elliptic curve cyptosystems. These are mostly the same problem as discrete logarithms, but implemented over a field which consists of all points along an ellipse. Elliptic curve cyptosystems are attractive because math along the curve can be computed very efficiently on both hardware and software, and compete with RSA (factoring) for speed [1] [3].

There are several other problem classes including the knapsack problem and finding the

shortest vector in a lattice which have been used in PKC systems. None of these systems are however particularly popular at current and thus will not be mentioned further here [1].

4.3.1 RSA - Prime Factorization

The RSA - PKC system is the most common PKC system in use, and RSA keys are the heart of popular PKC infrastructures such as Pretty Good Privacy (PGP) or the compatible GNU Privacy Guard (GPG). RSA was developed by Rivest, Shamir and Adleman. RSA depends on our inability to factor large composite numbers into large primes. RSA's dependence during encryption on large amounts of BigNum¹⁶ math lends it well to data-level parallelization.

Due to RSA's dependence on BigNum math, the efficiency of the corresponding BigNum libraries directly affects these algorithms ability to work well on parallel architectures. Various VPUs have accompanying vector libraries which handle BigNum math at high optimization with sizes of 128, 256, 512, 1024-bit or larger integers [3]. These BigNum functions, combined with traditional BigNum implementations based on smaller word sizes can provide optimizations on architectures containing VPUs¹⁷. Likewise since the math involved in these computations uses integers larger than several registers, BigNum libraries can utilize even larger CPUs achieving a near linear speed vs. word size gain. Many basic mathematical functions also have optimized parallel implementations, which can allow efficient BigNum optimizations on platforms with SMP architectures [12]. PKC based on prime factorization

¹⁶Big Number, or Large Number math. Mathematical operations which require the use of more than a single register to be completed. In this case exponentiation of integers at least 1024 bits

¹⁷This would be another interesting project, having found no BigNum libraries which in turn use vector libraries to do more efficient BigNum computation.

can be computed on parallel architectures with improvement over traditional architectures, and that improvement is based on efficient BigNum computation on that platform.

5 Final Thoughts

The analysis presented here is unfortunately relatively brief and superficial, serving as an overview of some of that which I researched this past term. Further analysis is certainly required on the subject of parallelization of existing cryptographic algorithms in order to assure proper utilization of future architectures by both our legacy cryptographic algorithms, and our more modern algorithms. From the cursory analysis performed here, one can see already many positive movements towards the efficient implementation of cryptography on future architectures. It is my belief that if some of the implementations which I proposed here, including packet-level parallelization via BitSlice implementations, decryption optimizations in 3DES and other Feistel networks, and BigNum library implementations using vector libraries were to be implemented, that many of the legacy cryptographic algorithms could be implemented with much greater efficiency on the computer hardware of the future.

Potentials for implementations of individual algorithms varies widely. The hash functions in general, have no chance of gains via intra-round parallelization, however do show some promise for a packet-level parallelization in situations where many hashes are computed at the same time. MD5 and SHA-1 both appear very trapped in their 32-bit worlds, neither shows serious promise of efficient implementations on newer hardware, with the exception of using SIMD processing to compute more than one simultaneous hash. The RISC processor design reduces some of inefficiency acquired when moving these 32-bit dependent algorithms

to 64 or more bit processors, but RISC does not offer help for VPU or other parallel implementations. An efficient SIMD implementation of RIPEMD-160 may be possible on 64 bit or higher architectures, but of the current hash functions only Tiger shows real promise towards efficient operation on 64-bit architectures. None of the hash functions seem ready for VPUs or processor word sizes beyond 64 bits¹⁸

The block cyphers are a mixed, but generally positive, group. DES and other Feistel networks allow BitSlice implementations, parallel computation in ECB mode, and parallel decryption. The AES finalists are all very new algorithms and therefore do not have nearly as much analysis to draw from as DES. There are likely several parallelization techniques for the AES algorithms yet to be discovered for the AES submissions, but all of the AES finalist algorithms show some promise of efficient implementations on tomorrow's architectures:

- Rijndael, the winner of the AES solicitation, shows promise for intra-round parallelization, however may not yet take advantage of all types of parallelization, including VPUs.
- Serpent (regarded as the “runner up”) seems particularly well designed to the task of parallelization, and shows promise of efficient implementations on architectures with much larger words than those which we use today.
- RC6 shows promising performance on upcoming architectures for at least the short term, but may not be easily efficiently implemented on processors with word sizes greater than 64 bits.

¹⁸I would suggest that future hashing algorithms be based on word sizes larger than current processors, and be computed in a way not requiring the word size of the algorithm and the word size of the processor to be identical. If this were achievable, it would allow hashing functions to be more useful for longer.

- TwoFish showed no more readiness for parallelization than DES or any other Feistel networks, but as a Feistel network, much of the previous analysis applies. Thus TwoFish may have a BitSlice implementation, and be able to use parallel processing during decryption just like DES.
- MARS, the fifth finalist, was omitted from my review here, but have read the specification and would qualify its potentials for parallelization similar to those of TwoFish, it being yet another Feistel network [7].

My research suggests that the block cypher algorithms we have now, although not all designed directly for future architectures, do all have at least mostly efficient implementations on future hardware.

Finally Public Key Cryptography, as developed of a discipline as it is, is still not widely accepted by the consumer. I see this quickly changing in the near future with armed forces' issuing of the Common Access Card (a java based smart card) which will contain unique Public and Private keys for each recipient. The future for Public Key Cryptography in terms of parallelization also looks bright, especially due to it's complete dependance on mathematical functions for which many parallel systems were initially designed. This allows the same PKC systems to run efficiently on any platform dependent only on a lower level, but well studied, problem of how to do large number math fast.

Despite my initial warning words, my analysis has shown that many of our present day algorithms, although not specifically designed for the hardware of tomorrow, have been found to work efficiently on current and future platforms with a few intelligent modifications. I would caution however that many of these implementations although theoretically possible

remain only on paper and have not yet been implemented in software on many architectures. There is still much work for the implementors of cryptography: to move cryptographic libraries from 32-bit dependent versions running on newer hardware, to truly optimized versions for that hardware. I also caution the cryptographers, that although we saw some awareness of architectures with the AES submissions, 4 of the five algorithms were not designed specifically for the computers of the future, but rather only had "possible efficient implementations" on those architectures, focusing instead (and appropriately so), on the computers of today, and smaller processors such as smartcards. There is still much work to be done towards the parallelization of crypto, but we a promising start towards "modern" cryptography.

References

- [1] <http://www.ssh.com/support/cryptography/algorithms/>.
- [2] Federal information processing standards publication. Technical Report 46-3, National Institute of Standards and Technology, 1999.
- [3] vdsp library. Technical report, Apple Computer, 2001.
- [4] Selim G. Akl. Parallel real-time computation: Sometimes quantity means quality. In *ISPAN: Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*. IEEE Computer Society Press, 2000.
- [5] Ross Anderson and Eli Biham. Tiger: A fast new hash function, 1996.
- [6] Eli Biham. A fast new DES implementation in software. *Lecture Notes in Computer Science*, 1267, 1997.
- [7] C. Burwick, D. Coppersmith, E. D’Avignon, R. Gennaro, S. Halevi, C. Jutla, S. Matyas, L. O’Connor, M. Peyravian, D. Safford, and N. Zunic. Mars — a candidate cipher for aes.
- [8] J. Daemen and V. Rijmen. Aes proposal: Rijndael.
- [9] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The block cipher SQUARE. *Lecture Notes in Computer Science*, 1267, 1997.
- [10] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996.

- [11] Hans Eberle. A high-speed DES implementation for network applications. *Lecture Notes in Computer Science*, 740:521–539, 1993.
- [12] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays Trees Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992.
- [13] E. Nahum, S. O’Malley, H. Orman, and R. Schroepel. Towards high-performance cryptographic software, 1995.
- [14] E. Nahum, D. Yates, O. Orman, and H. Schroepel. Parallelized network security protocols, 1996.
- [15] Anderson R., E. Biham, and L. Knudsen. Serpent: a flexible block cipher with maximum assurance, 1998.
- [16] Ronald L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 block cipher.
- [17] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. On the twofish key schedule. In *Selected Areas in Cryptography*, pages 27–42, 1998.
- [18] Michael Welschenbach. *Cryptography in C and C++*. Springer-Verlag New York, 2001.