

An Adaptive Cryptographic Engine for IPSec Architectures*

Andreas Dandalis and Viktor K. Prasanna
Department of Electrical Engineering-Systems
University of Southern California, Los Angeles, USA
{dandalis, prasanna}@halcyon.usc.edu
<http://maarcII.usc.edu/>

Jose D. P. Rolim
Centre Universitaire d'Informatique
Université de Genève
24 Rue General Dufour, 1211 Genève 4, Switzerland
Jose.Rolim@cui.unige.ch

Abstract

Architectures that implement the Internet Protocol Security (IPSec) standard have to meet the enormous computing demands of cryptographic algorithms. In addition, IPSec architectures have to be flexible enough to adapt to diverse security parameters. This paper proposes an FPGA-based Adaptive Cryptographic Engine (ACE) for IPSec architectures. By taking advantage of FPGA technology, ACE can adapt to diverse security parameters on the fly while providing superior performance compared with software-based approaches. For example, for the final candidate algorithms of the Advanced Encryption Standard (AES), our techniques lead to throughput speed-up of 4 – 20 while the key-setup latency time is reduced by a factor of 20 – 700 compared with software-based approaches. We also develop a compression technique that reduces the memory requirements of ACE without the need for dedicated hardware. Though data compression has been extensively studied before, we are not aware of any prior work that addresses the compression problem of FPGA-based embedded systems with respect to the implementation cost. Using our technique, we demonstrate up to 40 % savings in memory for various configuration bit-streams.

1 Introduction

The enormous success of Internet has made the Internet Protocol (IP) one of the primary communication protocols in our days. However, communicating over the Internet involves significant security risks since the Internet is an unprotected network. Therefore, the need for securing the Internet has become a fundamental issue, especially for transmitting confidential data (e.g., electronic commerce, electronic banking, Virtual Private Networks).

For securing the Internet traffic, Internet Protocol Security (IPSec) standard was developed by the Internet Engineering Task Force. The IPSec standard extends the IP protocol by securing the IP traffic at the IP level using cryptographic methods. IPSec has been adopted by all leading vendors and will be the future standard for secure communications on the Internet [16]. It is also rapidly becoming the industry standard for Virtual Private Networks (VPNs) [7].

Cryptography is the fundamental component of IPSec. Therefore, architectures that implement IPSec have to meet the enormous computing demands of cryptographic algorithms. Moreover, since the security parameters are dynamically negotiated by the communicating entities, IPSec architectures have to be also flexible enough to adapt to diverse security parameters (e.g., cryptographic algorithms, cryptographic operation mode, key).

In this paper we focus on the encryption/decryption aspects of the IPSec cryptographic component. Our goal is to design and implement an Adaptive Cryptographic Engine (ACE) using FPGAs. Such an engine consists of a hardware component (FPGA) and a memory component. FPGA configurations of cryptographic algorithms are stored in the memory in the form of a cryptographic library. The FPGA

*This research was performed as part of the MAARCII project. This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-99-1-0004 monitored by Fort Huachuca.

is configured on-demand based on the cryptographic library and then performs the required encryption/decryption tasks. Thus, ACE can provide hardware-like time performance with software-like flexibility.

The cryptographic library of ACE is designed using a domain-specific mapping approach. The domain is defined by the algorithm and the target FPGA architecture. Each variant of the algorithm corresponds to a sub-domain. Based on a specific domain, algorithm-specific configurations are derived. Our key idea is to synthesize bit-streams at runtime by merging a *skeleton* with a sub-domain configuration. A *skeleton* is the intersection of all the configurations in a domain, that is, a “parameterized” bit-stream that corresponds to the elementary functions of an algorithm.

We consider the final candidate algorithms of the Advanced Encryption Standard (AES) [1] to form the cryptographic library. We have chosen these algorithms because of the projected key role of AES in protecting electronic data flow. Each algorithm was implemented using the Virtex FPGA series devices [17]. For each implementation, we provide precise time performance results. In [6] and [15], only the cryptographic cores of three final candidate algorithms (Serpent, RC6, Twofish) were implemented. However, in both papers [6, 15], no results were provided regarding the key-setup of the algorithms. In our implementations, besides throughput results, we provide key-setup latency results. The latency metric is the key measure for IPsec where a small amount of data is processed per key and key-context switching occurs repeatedly. The key-setup latency time is the time required for adapting the algorithm to the input key. In software implementations, the cryptographic process cannot commence before the key-setup process for all the rounds is completed. As a result, the key-setup latency time equals to the key-setup time. On the contrary, in FPGAs, each cryptographic round can commence as early as possible since the key-setup process can run concurrently with the cryptographic process. Compared with software implementations, the achieved throughput speed-up was 4–20 while the key-setup latency time was reduced by a factor of 20–700.

We also address the equally important issue of compressing the configuration memory required to store the cryptographic library. The cost-effectiveness of the cryptographic library strongly depends on its memory requirements. Our goal is to derive a lossless compression technique with high decompression efficiency. High decompression efficiency includes high decompression rates and minimal hardware requirements. However, the speed and cost constraints of the ACE environment (and similar FPGA-based platforms) differentiate the resulting compression problem from conventional software-based applications. A new compression technique is required that meets the constraints and the requirements of the ACE environment. We are not aware of

any prior work that addresses the compression problem of FPGA-based platforms with respect to the implementation cost.

Our novel compression technique achieves high decompression rates without the need for any dedicated hardware. Moreover, it is applicable to various FPGA devices since it does not depend on individual features of the configuration mechanism. Using our technique, we demonstrated 7–40% savings in memory for various configuration bit-streams. Our technique is based on the well-known LZ compression algorithm [12]. However, a major deviation from LZ-based algorithms is the calculation of the compression ratio and the construction of the dictionary. In conventional LZ-based algorithms, only the compressed data is considered in the calculation of the compression ratio. The compressed data is stored in a secondary storage media or transmitted over a network. The dictionary is reconstructed on-line and the extra memory required is provided by the “host”. However, in embedded FPGA-based systems, no secondary storage media is available and the extra required memory has to be considered in the calculation of the compression ratio.

An overview of the IPsec standard and its cryptographic component is given in Section 2. In Section 3, an overview of the proposed Adaptive Cryptographic Engine is presented. Our key ideas for designing the cryptographic library are described in Section 4. Implementation results of the AES candidate algorithms are also given and comparisons with software implementations are made. The proposed compression technique and related results are demonstrated in Section 5. Finally, in Section 6, possible future work is described and concluding remarks are made.

2 Private-Key Cryptography for IPsec

The enormous advances in network technology have resulted in an amazing potential for changing the way we communicate and do business over the Internet. However, in the case of confidential data, the cost-effectiveness and globalism provided by the Internet are diminished by the main disadvantage of public networks: *security risks*. The significantly increasing growth in the confidential data traffic over the Internet makes the security issue a fundamental problem. Confidential data has to be protected against security threats including loss of privacy, loss of data integrity, identity spoofing, and denial-of-service among others [2]. Consequently, applications such as electronic banking, electronic commerce, and Virtual Private Networks (VPNs) require an efficient and cost-effective way to address the security threats over public networks.

The Internet Protocol Security (IPsec) standard is an ongoing effort of the Internet Engineering Task Force that has been adopted by all leading vendors. It will be the future standard for secure communications over the Internet [16].

Besides its wide range of applicability to Internet-based applications, IPSec is also rapidly becoming the standard of choice for VPNs [7]. VPNs utilize public networks in order to establish a secure private network. As a result, drastic reduction of cost and superior network efficiency can be achieved. The estimated savings can be from 20% to 80% by switching from leased lines, serving branch offices or serving remote access users [7]. The remarkable potential of the VPN market is an evidence of the significance of IPSec in the global electronic business. From US\$ 224 million in the year 1998, the VPN market is forecast to grow to US\$ 13,000 million in the year 2004 [7].

The IPSec standard is a framework of open standards for ensuring secure private communication over the Internet [2]. It extends the IP protocol by securing the IP traffic at the IP level using cryptographic methods. Since it provides security at the IP level, there is no need for changing the end systems, the applications, and the intermediate network infrastructure. The only required changes occur at the end points where the encrypted packets enter/exit the public network. Consequently, the implementation and management costs are reduced significantly. However, the major advantage of IPSec is its flexibility: it is not based on a particular cryptographic method. The cryptographic methods to be used are negotiated between the communicating entities. IPSec provides an open framework to implement any cryptographic algorithm. As a result, the IPSec standard makes it possible for security systems developed by different vendors to interoperate [10].

In order to protect IP traffic from security threats, IPSec provides the following services [2]:

- Confidentiality: data encryption.
- Authenticity: proof of sender.
- Integrity: data tampering detection.
- Replay Protection: preventing unauthorized data re-sending.
- Key Management: negotiating security associations and key exchanging.

Initially the two communicating parties negotiate and establish a security association (SA) for protecting the transferred data. An SA determines the cryptographic methods and the related keys to be utilized. The cryptographic methods include both private and public key cryptography, keyed hash algorithms, and digital certificates. Each SA is restricted by its lifetime after the expiration of which a new SA has to be established. The lifetime of an SA can be determined in terms of absolute time or amount of transmitted data. Usually, a small amount of data is processed per key and key-context switching occurs repeatedly.

In this paper, we focus on private-key cryptography (i.e., confidentiality) for IPSec architectures. The implementation of a cryptographic algorithm must achieve high processing rate to fully utilize the available network bandwidth. Otherwise, the cryptographic component of the IPSec architecture becomes a performance bottleneck. The implementation must also support fast key-context switching. Otherwise, additional latency is introduced that is critical in latency-bound applications (e.g., video on-demand, IP telephony). Finally, in order to keep the IPSec architecture independent of the implemented cryptographic algorithms, it must be possible to add new implementations or modify existing ones. Then, the IPSec architecture can be updated with state-of-the-art cryptographic algorithms and standards.

Known IPSec architectures [7] utilize software-driven or ASIC-based solutions for implementing private-key cryptographic algorithms. Software-driven approaches provide the flexibility required to keep the architecture to be flexible to adapt to various cryptographic algorithms. However, since cryptographic algorithms have enormous processor requirements, software-driven solutions are inadequate for data-transfer rates higher than those found in two T1 lines [13]. In this case, ASIC-based architectures can provide significant performance advantages over software-driven solutions. Major companies, such as Cisco, AT&T, and Lucent, endorse specialized encryption/decryption chips integrated in their routers. However, any update of these specialized chips becomes very costly. Even though the price of such chips is dropping dramatically, the question that still remains is what to do with the old “boxes” [7]. The ultimate solution for the problem would be a reconfigurable processor that can provide software-like flexibility with hardware-like performance.

2.1 FPGA-based Cryptography

FPGA devices are highly promising alternatives for implementing private-key cryptographic algorithms. Even though ASICs can achieve superior performance compared with FPGAs, their flexibility is restricted by the design parameters provided during fabrication. Thus, the replacement of such application-specific chips is very costly. However, if the required performance can not be achieved by FPGA implementations, ASICs solutions are superior. On the other hand, compared with software-based solutions, FPGA implementations can achieve superior performance and security.

The fine-granularity of FPGAs matches extremely well the operations required by private-key cryptographic algorithms (e.g., bit-permutations, bit-substitutions, look-up table reads, boolean functions). Moreover, the inherent parallelism of the algorithms can be efficiently exploited by FP-

GAs. As a result, superior processing rates can be achieved. For example, for the AES final candidate algorithms, the throughput speed-up is 4 – 20 compared with software implementations (see subsection 4.2).

Besides throughput, FPGA implementations can also achieve agile key-context switching. Key-context switching includes the generation of key-dependent data for each cryptographic round (e.g., subkeys, look-up tables). A cryptographic round can commence as soon as its key-dependent data is available. In software implementations, the cryptographic process cannot commence before the key-setup process for all the rounds is completed. As a result, excessive latency is introduced making key-context switching inefficient. On the contrary, in FPGAs, each cryptographic round can commence as early as possible since the key-setup process can run concurrently with the cryptographic process. As a result, minimal key-setup latency can be achieved. For example, for the AES final candidate algorithms, the key-setup latency time is reduced by a factor of 20 – 700 compared with software implementations (see subsection 4.2).

Finally, security issues make FPGA implementations more advantageous than software-based solutions. An encryption algorithm running on a generalized computer has no physical protection [14]. Hardware cryptographic devices can be securely encapsulated to prevent any modification of the implemented algorithm. In general, hardware-based solutions are the embodiment of choice for military and serious commercial applications (e.g., NSA authorizes encryption only in hardware) [14].

3 Adaptive Cryptographic Engine

The proposed Adaptive Cryptographic Engine (ACE) is an FPGA-based architecture (see Figure 1) that can provide the speed and flexibility required by IPsec. ACE can be dynamically adapted to cryptographic algorithms of different Security Associations (SAs) at runtime. As we will show shortly, agile adaptation to different key-contexts can also be achieved by FPGA implementations. In addition, the cryptographic library can be updated by new configurations by updating the memory contents. As a result, ACE can adapt to any cryptographic algorithm. The key problem in the implementation of ACE is the design of the cryptographic library. We have addressed both the problems of deriving as well as compressing the configurations of various cryptographic algorithms.

ACE consists of an FPGA device(s), a cryptographic library, and a configuration controller (see Figure 1). The FPGA is configured on the fly by the configuration controller. Subsequently, adaptation to the input key occurs and the data encryption/decryption commences. The configuration controller determines the configuration to be chosen based on the established SA (shown as request in Figure 1).

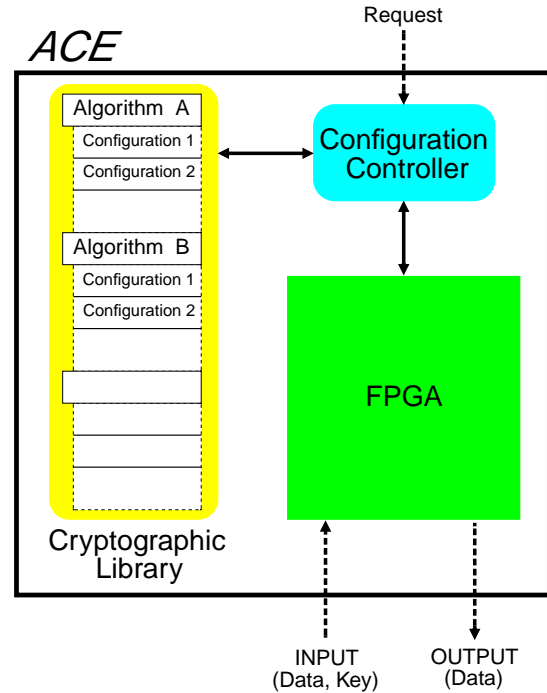


Figure 1. Architecture of ACE

The cryptographic library consists of FPGA configurations stored in the memory. Each configuration corresponds to a cryptographic algorithm or one of its variants.

The core of ACE is reconfigurable hardware. FPGAs are a highly promising technology for high-performance, adaptive architectures for IPsec. FPGA-based implementations can provide the high processing rates required by high-speed networks. Moreover, FPGAs can be reconfigured on the fly providing the high degree of flexibility required in dynamically changing environments.

The cryptographic library consists of FPGA configurations of private-key cryptographic algorithms. Each cryptographic algorithm can correspond to various configurations. Such algorithm variants differ from each other in terms of the cryptographic operation mode, the key length, the number of rounds, etc. The configurations are stored in the memory and can be updated to enhance the adaptability of ACE.

The configuration controller is indispensable for configuring the FPGA(s). Besides being the interface between the cryptographic library and the FPGA(s), it also resolves the external requests. The hardware overhead required for decompressing the stored configuration bit-streams (please see section 5) is minimal. Only memory reads are required similar to the memory reads required for configuring the FPGA(s).

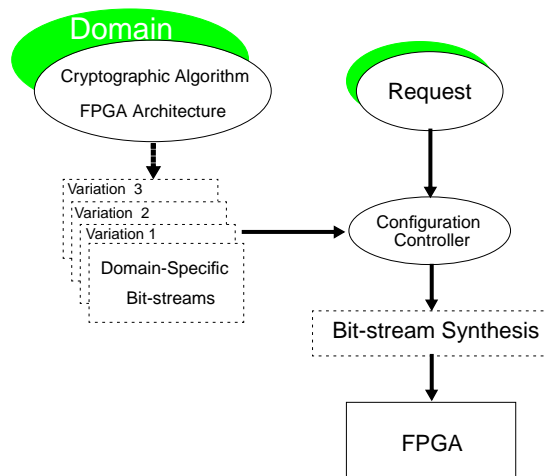


Figure 2. Domain-Specific Bit-stream Synthesis

4 Cryptographic Library Design

The cryptographic library consists of configuration bit-streams of various cryptographic algorithms. A domain-specific approach is utilized to design the library. The domain is defined by the algorithm and the target FPGA architecture. Each variant of the algorithm corresponds to a different sub-domain. Based on a specific domain, algorithm-specific configurations are derived. Our key idea is to synthesize bit-streams at runtime by merging a *skeleton* with a sub-domain configuration. A *skeleton* is the intersection of all the configurations in a domain, that is, a “parameterized” bit-stream that corresponds to the elementary functions of an algorithm.

By exploiting the low-level details of the FPGA architecture, hardware-optimized designs can be derived by matching the algorithm requirements with the low-level hardware details. As a result high performance can be achieved. Furthermore, the time performance and the area requirements are accurately determined beforehand. This is particularly important in run-time environments where the parameters of the problem are not known *a priori* but time and area constraints must be satisfied.

The bit-stream synthesis occurs at runtime by merging the *skeleton* with a sub-domain configuration (see Figure 2). Thus, a sub-domain corresponds to the modifications of the *skeleton* that lead to a configuration of an algorithm variant. As a result, the configurations of a domain can be efficiently represented avoiding data redundancy, that is, the *skeleton* configuration data. The bit-stream merging can be realized by the configuration controller by providing it with the sequence of the required memory reads. On the other hand, by utilizing partial reconfiguration, the *skeleton* can

be overwritten on-chip resulting in the requested configuration. In either case, simple memory read operations are utilized.

In [4], we have utilized a domain-specific and instance-dependent approach to solve graph problems using FPGAs. The main idea was to specialize the hardware based on the input data in order to achieve high performance. However, in the case of the cryptographic library, the run-time adaptation is algorithm-dependent. While data-dependent adaptation may result in higher performance, it also results in implementations that involve security risks. A rule-of-the-thumb for hardware cryptographic implementations is that their performance should be independent of the input data in order to avoid time-based attacks.

4.1 AES Performance Evaluation

The significance of the Advanced Encryption Standard (AES) [1] led us to choose the final AES candidates as the basis of the cryptographic library. Providing precise time performance and area requirements results, we can evaluate the practical effectiveness of ACE. The performance metrics are throughput and key-setup latency. The throughput metric indicates the amount of data encrypted/decrypted per time unit after the initialization of the algorithm. The key-setup latency denotes the time required to adapt to the input key. While throughput indicates the bulk-encryption capability of the implementation, key-setup latency indicates the capability of agile key-context switching. The latency metric is the key metric for IPSec where a small amount of data is processed per key and key-context switching occurs repeatedly. The key-setup latency time is the time required for adapting the algorithm to the input key. In software implementations, the cryptographic process cannot commence before the key-setup process for all the rounds is completed. As a result, the key-setup latency time equals to the key-setup time. On the contrary, in FPGAs, each cryptographic round can commence as early as possible since the key-setup process can run concurrently with the cryptographic process. Finally, the FPGA configuration time can be safely ignored since it can be amortized by the time required to complete an SA negotiation and exchange keys.

Development of the AES is an on-going effort of the National Institute of Standards and Technology (NIST) and the cryptographic community. AES will be a public algorithm designed to protect sensitive government information. It will also serve as an important security tool to support the dynamic growth of electronic commerce. The development effort started on 1997 and, eventually, fifteen candidate algorithms were submitted from all over the world. On August 9, 1999, NIST announced the final five candidates algorithms of the AES standard.

The AES algorithm(s) is a private-key cryptographic al-

gorithm that supports at the minimum block sizes of 128-bits and key sizes of 128-, 192-, and 256-bits. It will replace the aging Data Encryption Standard (DES), which was adopted by NIST in 1977 as a Federal Information Processing Standard used by federal agencies and the private sector to encrypt information [1]. Having chosen the final candidate algorithms, the AES development effort has entered its final stage where the final algorithm(s) will be chosen.

Regarding performance estimates, detailed analysis has been done in terms of software-based and smartcard-based implementations. Performance results of FPGA-based implementations were shown in [6, 15]. In [6], only the results of implementing the cryptographic core of one final candidate algorithm (Serpent) using FPGAs were shown. In [15], the results of implementing the cryptographic core of two other candidate algorithms (RC6, Twofish) were shown using their own non FPGA-based reconfigurable architecture (CMU PipeRench). However, no results regarding the key-setup of the algorithms were provided in both [6, 15].

4.2 Performance Results

We have chosen the Virtex FPGA series [17] as the hardware target for the implementations. For mapping onto Virtex devices, we have used the Foundation Series v2.1i software development tool. The operation parameters of the tool remained the same for all the implementations. All the results were based on placed-and-routed implementations (device speed -6). The implementations included both the key-setup component and the cryptographic core along with their control circuit.

Besides throughput results, we also provide key-setup latency results, that is, the time required for adapting the algorithm to the input key. Among the various time-space tradeoffs, we focused primarily on time performance. Our goal was to maximize throughput for the cryptographic core of each candidate algorithm. We have exploited the inherent parallelism of each cryptographic core and the low-level hardware features of FPGAs to enhance the performance. Moreover, the latency issue was of primary interest, that is, the cryptographic core has to commence its operation as early as possible. We have designed the key-setup component to sustain the processing rate of the cryptographic core and to achieve minimal latency.

For each algorithm, we have implemented a single round of the encryption block cipher for 128-bit data blocks using 128-bit keys. The implemented round is reused repeatedly until encryption has been completed. Similar performance analysis can be performed for certain operation modes that allow concurrent processing of multiple blocks of data. For example, in the ECB mode of operation, multiple blocks of data can be processed concurrently since each data block is encrypted independently. Consequently, if p rounds are im-

Table 1. Key-setup latency results and comparisons

AES Algorithm	Latency (μ s)		Latency block encryption time	
	Software	Our	Software	Our
MARS	38.11	1.96	7.91	3.12
RC6	25.07	0.17	5.93	0.15
Rijndael	33.93	0.07	8.39	0.20
Serpent	56.99	0.08	3.33	0.09
Twofish	63.99	0.18	13.15	0.25

Table 2. Throughput results and comparisons

AES Algorithm	Throughput (Mbits/sec)		Speed-up
	Software	Our	
MARS	26.56	101.88	3.83
RC6	30.29	112.87	3.72
Rijndael	31.64	353.00	11.15
Serpent	7.48	148.95	19.91
Twofish	26.31	173.06	6.58

plemented, a throughput speed-up of p can be achieved compared with a “single-round” implementation.

Our performance results are compared with the software-based results of the “NIST’s Efficiency Testing for Round 1 AES candidates” [1]. The reference platform was a Pentium Pro with 64 MB RAM running at 200 MHz. As noted in [1], NIST used the optimized code provided by the submitters of the candidate algorithms.

In Table 1, the key-setup latency results of our implementations and the software counterparts are shown. The results are represented both as absolute time and as the fraction over the corresponding encryption time of one 128-bit block of data. Clearly, the FPGA implementations achieve significant reduction of the key-setup latency time. On the contrary, the key set-up time of the software implementations is equivalent to the encryption time of 3-13 blocks of data. In FPGAs, each cryptographic round can commence as early as possible since the key-setup process can run concurrently with the cryptographic core. In software implementations, the cryptographic core can not commence before the key-setup process for all the rounds is completed and the key-setup latency time equals to the key-setup time. Thus, while FPGA implementations favor agile key-context switching, the software implementations require relatively excessive time for key-context switching.

In Table 2, encryption throughput results are shown and comparisons with the software implementations are made.

Compared with the software-based results, the throughput speed-up was 4 – 20. While software implementations can not achieve processing rates greater than 30 Mbits/sec, our FPGA implementations achieve processing rates greater than 100 Mbits/sec. For one reason, software implementations can not exploit the inherent parallelism of the algorithms. For another, the operations required by the cryptographic algorithms can be executed more efficiently in FPGAs than in a general-purpose computer. The fine-granularity of FPGAs matches extremely well operations such as bit-permutations, bit-substitutions, look-up table reads, and boolean functions among others.

By using a superior computer configuration than the reference platform of NIST Efficiency Test, higher throughput can be achieved for the software implementations. Even in this case, the speed-up of the FPGA implementations can still be remarkable (e.g., *Rijndael*, *Serpent* implementation). Moreover, by realizing "multiple-round" FPGA implementations, the throughput speed-up can be significantly improved for operation modes that allow concurrent processing of multiple blocks of data. Regarding area requirements, the high-capacity of Virtex makes it possible that each algorithm can fit in one device. Summarizing, FPGA implementations indeed achieve superior time performance compared with the software counterparts. Besides the throughput speed-up, the key advantage of the FPGA implementations is the significant reduction of the latency time.

5 Configuration Compression

The cost-effectiveness of the cryptographic library is strongly related to its memory requirements. Given the variety of the cryptographic algorithms and the corresponding configuration size, configuration compression can result in significant savings in memory. Usually, the configuration size is in the order of *Mbits*. The main idea is to store compressed configurations in the memory and perform decompression at runtime. Finally, the decompressed data is fed to the configuration mechanism of the FPGA. Our goal is to develop a lossless compression technique that leads to high decompression rates without the need for dedicated hardware resources.

5.1 Our Compression Technique

Our compression technique is based on the principles of dictionary-based compression. Even though statistical methods can achieve higher compression ratios, we preferred a dictionary-based approach because statistical methods require dedicated hardware resources for decompression. Dictionary-based compression is a lossless compression technique. The main idea is to encode variable-length

strings of symbols as single codewords [12]. The codewords form an *index* to a phrase dictionary. Compression occurs, if the codewords are smaller than the phrases that they replace. Decompression can be as simple as a look-up table operation. In general, finding a dictionary that results in optimal compression has exponential complexity.

In our scheme, the dictionary corresponds to configuration data while an index corresponds to the way that a configuration is being synthesized. Configuration synthesis corresponds to decompression. For each cryptographic algorithm, we derive a dictionary and its index off-line. The decompression occurs at runtime by parsing the dictionary with respect to its index. As a result, only memory read operations are required and thus high decompression rates can be achieved. Our compression technique consists of two phases. First, the dictionary and its index are derived simultaneously. Then, a heuristic is used to reduce the memory requirements of the dictionary by merging repeated patterns.

We derive each dictionary based on the principles of the LZW algorithm [12]. In LZW, the dictionary is preloaded with the symbols of the alphabet. Therefore, the number of the preloaded phrases is equal to the number of symbols in the alphabet. As compression proceeds, the algorithm finds all the phrases that are prefix strings of the preloaded phrases. The dictionary index is also derived simultaneously with the dictionary.

In conventional LZW algorithms, the dictionary is constructed on-line and it also includes phrases that are not referenced by any codeword. This happens because, as compression proceeds, the algorithm keeps all the generated prefix strings of the preloaded phrases. This is performed regardless of whether these strings are referenced by a codeword or not. However, this is not a problem in software-based applications. Only the index that is stored in a secondary storage media or transmitted over the network is considered in the calculation of the compression ratio. There is no need to store or transmit the dictionary since it can be reconstructed on-line based on its index.

However, in our case, the memory requirements of the dictionary must also be considered. The dictionary memory requirements derived by LZW are comparable to the index size. In contrast to conventional LZW algorithms, we derive the dictionary by adding only phrases that are referenced by a codeword. A phrase is represented as a singly-linked list of symbols. Furthermore, the dictionary is derived in such a way that allows to merge phrases that have common suffix strings. In this way, the memory requirements can be reduced significantly. For example, instead of storing in the memory the phrases "COMPUTING" and "COMPUTATION", we can store instead the strings "COMPUT", "ING", and "ATION", and link them together.

After having derived the dictionary and its index, we re-

Table 3. Memory requirements of the bit-streams

AES Algorithm	Bitstream Size (bits)	Redundancy Factor
MARS	3833655	0.6890
RC6	2705363	0.4534
Rijndael	3833655	0.7623
Serpent	2705363	0.6296
Twofish	6510917	0.7163

duce the memory requirements of the dictionary by selectively decomposing phrases in the dictionary. The main idea is to replace frequently-occurred strings among phrases by a new phrase. As a result, while memory savings can be achieved for the dictionary, additional codewords are also introduced leading to index expansion. In the following, a greedy heuristic is described that tries to find strings that can result in overall memory savings.

Given the derived dictionary, we first identify repeated strings that can result in potential memory savings. The memory savings depend on the string length l_i and the frequency of occurrence t_i . Besides this, the index expansion depends on the codeword splits s_i (i.e., number of new codewords introduced). By deducting a string from a phrase, new codewords are introduced for all the prefix strings of the phrase (including the phrase itself). In our heuristic, after finding a repeated string, we selectively delete it only at positions where the savings in memory are greater than the corresponding index expansion. Moreover, we concentrate only on strings that include nodes that are referenced only once. Otherwise, the string extends over larger number of prefix strings and lower possibility of overall memory savings. Further details of our compression technique can be found in [3].

5.2 Compression Results for AES

The bit-streams of the AES algorithms shown in Table 3 were compressed utilizing our compression technique. Both the index and the dictionary memory requirements are considered in the calculation of the compression ratio. The compression ratio metric indicates the fraction of the compressed bit-stream over the original one.

Our compression results are compared with the results derived by using the LZW algorithm [12] to generate the dictionary and its index. However, in software-based applications, only the index size is considered in the calculation of the compression ratio. In addition, statistical encoding schemes are utilized for compressing the derived index. As a result, superior compression ratios (i.e., 10-20 %) can be achieved by using LZ-based commercially available soft-

Table 4. Compression results and comparisons

AES Algorithm	Compression Algorithm	Index Size (bits)	Dictionary Size (bits)	Compression Ratio
MARS	LZW	2641500	3822130	168.60 %
	Our Scheme			
	Phase I	2348000	1125128	90.59 %
	Phase II	2850360	446269	85.99 %
RC6	LZW	1226839	1810550	110.43 %
	Our Scheme			
	Phase I	1082505	678983	65.11 %
	Phase II	1254610	371558	60.11 %
Rijndael	LZW	2922732	4228354	186.53 %
	Our Scheme			
	Phase I	2597984	1162128	98.08 %
	Phase II	3189570	395830	93.52 %
Serpent	LZW	1703451	2511450	153.25 %
	Our Scheme			
	Phase I	1603248	830304	89.95 %
	Phase II	1794240	336732	78.76 %
Twofish	LZW	4663980	6743490	175.20 %
	Our Scheme			
	Phase I	4404870	1938400	97.42 %
	Phase II	5049690	664033	87.75 %

ware programs (e.g., *compress*, *gzip*).

In Table 3, besides the size of the bit-streams, their redundancy factor is also shown. The redundancy factor corresponds to the data redundancy of each bit-stream. In our case, this factor is represented by the fraction of the index derived by LZW over the bit-stream size. Therefore, for a given bit-stream, the redundancy factor is a lower bound on the compression ratio.

The compression results achieved by our compression technique are shown in Table 4. The first phase (Phase I) of our technique corresponds to the generation of the dictionary and its index while the second one (Phase II) corresponds to our heuristic.

The sizes of the dictionaries that were derived by LZW were comparable with the sizes of their indices. Therefore, negative compression occurred, that is, the memory requirements of the dictionary and its index were greater than the memory requirements of the original bit-stream.

However, by using our technique to generate the dictionary (Phase I), the sizes of the dictionaries were reduced by a factor of 2.6 – 3.5. This occurred due to our method of constructing the dictionary and representing it in the memory. As a side-effect, since the dictionary entries were re-

duced significantly, the size of the index was also reduced by a small factor (1.05 – 1.15). Compared with the LZW compression results, superior compression ratios were achieved (65 – 98 %).

Finally, the overall savings in memory were improved further by utilizing our heuristic (Phase II). The size of the dictionary was reduced by a factor of 1.8 – 2.9 compared with the dictionary sizes derived in Phase I. On the contrary, the memory requirements of the indices were increased due to the introduction of new codewords. However, in every case, the index size expansion was amortized by the savings in the memory of its dictionary. As a result, the achieved compression ratios were improved further by a constant of 5 – 11 %.

5.3 Comparison with Related Work

Various lossless compression techniques have been extensively published in the literature. The majority of these techniques have been developed for software-based applications. However, in embedded systems, the implementation cost becomes very significant. In [5, 11], dictionary-based compression techniques were utilized for code minimization in embedded processors.

In [11] a fixed-size dictionary was used for compressing programs of size in the order of hundreds of bits. The dictionary was built based on a benchmark of programs of various instruction sets. No detailed information was provided regarding the algorithm used to build the dictionary. The authors mainly focused on tuning the dictionary parameters to achieve better compression results based on the specific set of programs. However, such an approach is unlikely to achieve the same results for FPGA configurations where the bit-stream is a data file and not an instruction-based program. In addition, Huffman encoding was used for compressing the codewords. As a result, dedicated hardware resources were utilized for decompressing the codewords.

In [5], the dictionary was built by solving a set-covering problem. The dictionary representation was based on the External Pointer Macro compression model. According to this model, a phrase is called by pointing to a memory address and reading as many consecutive memory addresses as the phrase length. A heuristic was developed to merge dictionary entries by subsuming a dictionary entry i by another entry j if it was a suffix of entry j . While this heuristic results in a minimal-size dictionary (i.e., minimal number of entries), it also results in entries of maximal-length. This happens because the goal was to substitute maximal number of entries by keeping the ones that “cover” maximal number of them, that is, the longest ones. Even though this approach is different from ours, it is interesting to note that checking as well as the frequency of appearance of each string (as in our heuristic), a minimal memory-size dictio-

nary can be derived instead of a minimal-size one. Moreover, the size of the considered programs was 0.5-10 Kbits and the achieved compression ratios (i.e. size of the compressed program as fraction of the original program) were approximately 85-95 %. Since the compression results of each approach depend on data sets of different characteristics, it is unfair to make any compression ratio comparisons.

Work related to FPGA configuration compression has been reported in [8, 9]. In [8], the proposed technique took advantage of the configuration mechanism characteristics of the XC6200 architecture. Therefore, the technique is applicable only to the XC6200 architecture. In [9], runlength compression techniques for FPGA configurations have been described. Addresses were compressed by using runlength encoding while data was compressed by using LZ compression (sliding-window method [12]). Dedicated on-chip hardware was required for both methods. The bit-streams (0.1-1 Kbits) of a benchmark were used to fine-tune the parameters of the proposed methods. A 16-bit size window was used in the LZ implementation. While this window size led to good results for the considered bit-streams, it is impractical for the size of the AES bit-streams. Moreover, larger size windows impose a fairly high hardware penalty with respect to the buffer size as well as the support hardware [9].

6 Conclusions

In this paper, an Adaptive Cryptographic Engine (ACE) was proposed to perform the encryption/decryption tasks required by IPSec. ACE is an FPGA-based architecture that can provide the speed and flexibility required for IPSec architectures. Our main goal was to design the cryptographic library of ACE. The proposed cryptographic library was based on “parameterized” configuration bit-streams. We considered the AES final candidate algorithms as the base of our cryptographic library. Compared with software-based implementations, throughput speed-up of 4 – 20 was achieved while the key-setup latency time was reduced by a factor of 20 – 700. We also addressed the equally important issue of compressing the configuration bit-streams of the library. Good compression ratios were achieved without the need for dedicated hardware resources. Though data compression has been extensively studied in the past, we are not aware of any prior work that addresses the compression problem of FPGA-based platforms with respect to the implementation cost. Using our technique, we demonstrated up to 40 % savings in memory for various configuration bit-streams.

Future work includes further evaluation of the efficiency of our compression technique by applying it on bit-streams corresponding to regular, iterative computation structures (e.g., pipelines). Moreover, we will explore the possible

compression improvement by deriving a unified dictionary for different domains.

The work reported here is part of the USC MAARCII project (<http://maarcII.usc.edu>). This project is developing novel mapping techniques to exploit dynamic reconfiguration and facilitate run-time mapping using configurable computing devices and architectures. A domain-specific mapping approach is being developed to support instance-dependent mapping. Moreover, computational models and algorithmic techniques are being developed to exploit self-reconfiguration using FPGAs. Finally, the idea of “active” libraries is exploited to develop a framework for automatic dynamic reconfiguration.

References

- [1] Advanced Encryption Standard, <http://www.nist.gov/aes/>
- [2] Cisco Systems, Inc., “IPSec”, White paper, http://www.cisco.com/warp/public/cc/cisco/mkt/security/encryp/tech/ipsec_wp.htm
- [3] A. Dandalis, “Dynamic Logic Synthesis for Reconfigurable Devices”, PhD Thesis, Dept. of Electrical Engineering - Systems, University of Southern California. Under Preparation.
- [4] A. Dandalis, A. Mei, and V. K. Prasanna, “Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices,” Reconfigurable Architectures Workshop, April 1999.
- [5] S. Devadas, S. Laio, and K. Keutzer, “On Code Size Minimization Using Data Compression Techniques”, Research Laboratory of Electronics Technical Memorandum 94/18, Massachusetts Institute of Technology, 1994.
- [6] A. J. Elbirt, C. Paar, “An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher”, Eighth ACM International Symposium on Field-Programmable Gate Arrays, February 2000.
- [7] D. Fowler, “Virtual Private Networks. Making the Right Connection”, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.
- [8] S. Hauck, Z. Li, and E. J. Schwabe, “Configuration Compression for the Xilinx XC6200 FPGA”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 8, pp. 1107-1113, August, 1999.
- [9] S. Hauck, W. D. Wilson, “Runlength Compression Techniques for FPGA Configurations”, IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.
- [10] IPSec Developers Forum, “What is IPSec”, <http://www-ip-sec.com/IPSec.info.html>
- [11] C. Lefurgy, P. Bird, I-C. Cheng, and T. Mudge, “Improving code density using compression techniques”, 29th Annual IEEE/ACM Symposium on Microarchitecture, December 1997.
- [12] M. Nelson, J-L. Gaily, “The Data Compression Book”, M&T Books, New York, 1996.
- [13] B. Robinson, “Plans for a Secure Future”, tele.com, Issue 418, September 20, 1999.
- [14] B. Schneier, “Applied Cryptography”, John Willey & Sons, Inc., 2nd edition, 1996.
- [15] R. R. Taylor, S. C. Goldstein, “A High-Performance Flexible Architecture for Cryptography”, Workshop on Cryptographic Hardware and Embedded Systems, August 1999.
- [16] SSH IPSEC Express, <http://www.ipsec.com/products/ipsec/>
- [17] Virtex Series FPGAs, <http://www.xilinx.com/products/virtex.htm>