IPSecco: A Lightweight and Reconfigurable IPSec Core

Benedikt Driessen, Tim Güneysu, Elif Bilge Kavun, Oliver Mischke, Christof Paar, Thomas Pöppelmann
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{benedikt.driessen, tim.gueneysu, elif.kavun, oliver.mischke, christof.paar, thomas.poeppelmann}@rub.de

***Abstract*—In this paper we propose a reconfigurable lightweight Internet Protocol Security (IPSec) hardware core. Our architecture supports the main IPSec protocols; namely Authentication Header (AH), Encapsulating Security Payload (ESP), and Internet Key Exchange (IKE). In this work, the cryptographic algorithms and their modes of operation, which are at the heart of the IPSec protocols, are implemented in hardware. Instead of re-implementing common IPSec configurations, which are deemed "too heavy" for pervasive devices, we evaluate efficient implementations of standardized and/or well-known lightweight and hardware-friendly algorithms. In particular, we examine different versions of Present, Grøstl, Photon, and a very compact ECC core. As a consequence, we present IPSecco, a core with adequate security and only moderate resource requirements, making it suitable for lightweight devices. We selected the Xilinx Spartan family of Field Programmable Gate Arrays (FPGA) as target platform due its low-power footprint and reduced costs compared to other FPGAs. Our results show that it is possible to realize a high performance IPSec core even on members of the Spartan-3 family.**

***Keywords*-Lightweight; IPSec; FPGA; Reconfigurability**

I. Introduction

With the technological development in today's world, more and more computing devices enter the market and many of them are connected over the Internet or local networks in order to communicate with each other. An increasing number of these devices are resource-constrained devices and used in pervasive computing applications. These lightweight devices also need to connect to the Internet for device-to-device communication and product updates – mostly over a wireless network. There must be a common language for devices across networks to share information with each other. The Internet Protocol (IP) [1]–[3] is the primary communication protocol for transferring data between parties across a network. It defines datagram structures that are encapsulating the data to be delivered. For resource-constrained environments like embedded systems, there is even a lightweight version of IP, namely lwIP [4], which reduces the resource utilization.

The rapid increase in the utilization of computing devices led to security problems especially in communication over networks. Nowadays, many devices require authentication and encryption of the data they receive and send. The same is valid for area-constrained devices, many of them provide critical data. For devices with no resource limitation, a security enhancement to IP, called IPSec [5]–[7], has already been proposed. IPSec defines a family of protocols to provide security services such as confidentiality – to prevent undesired access attempts to the data transmission, data integrity – to make sure that the transferred data is not changed, and authentication – to identify the information source. In IPSec, there are different protocols to provide mentioned services. For instance, the Authentication Header (AH) protocol provides data authentication. The Encapsulating Security Payload (ESP) protocol defines mechanisms for confidentiality and data integrity. Finally, the Internet Key Exchange (IKE) protocol is used for establishing secure connections. These protocols use different cryptographic primitives such as encryption, hashing and modular arithmetic in order to provide security services. A minimum set of algorithms, which must be supported in an IPSec implementation for AH, ESP, and IKE protocols, was defined in "Cryptographic Suites for IPSec" [6], [7] for standardization purposes. For example, in "Cryptographic Suite B" [7], the AES [8] cipher is used in Galois/Counter Mode (GCM) [9] to provide authenticated encryption. The Hashed Message Authentication Code (HMAC) [10] construction is used with the Secure Hash Algorithm (SHA) [11] for AH services. For exchanging keys between parties, IKE uses the Diffie-Hellman key exchange [12]. However, none of these recommendations are actually targeted at resource-constrained devices. To the best of our knowledge, there exists no standardized lightweight IPSec protocol in the literature. In this paper, in order to have a lightweight IPSec core, we exchange regular algorithms with lightweight ones – standardized and/or well-known lightweight algorithms. For instance, instead of AES we use Present [13], which is already standardized by ISO/IEC [14]. As hash functions we evaluate SHA-3 candidate Grøstl [15] and the lightweight proposal Photon [16], which uses the Present Sbox.

Due to already mentioned lightweight device constraints, a lightweight crypto core must be low-cost and low-power. Software solutions suffer from low performance (when compared to hardware) and some hardware implementations, such as ASIC (Application-Specific Integrated Circuit) implementations, lack the flexibility and programmability offered by software. Hence, using an FPGA platform for the designed hardware core seems to be the perfect solution to achieve our goals and reconfigurability. We implement both the encryption/hashing algorithms and modes of operation in hardware. Selecting Xilinx Spartan FPGAs as the target platform provides us with reconfigurability, we therefore can switch between different lightweight algorithms or implementations depending on our needs with less effort. For these platforms, we evaluate and propose the IPSecco-80 and IPSecco-128 cores, which are configured to achieve 80-bit and 128-bit symmetric security.

The paper is organized as follows: In Section II previous works are discussed. Section III describes the proposed IPSecco architecture, and Section IV discusses our results. Finally, we conclude the paper together with future directions in Section V.

II. Related work

Hardware-related implementations of IPSec specific functionality have been relatively scarce, although the last year has seen a spark of activity.

In [17] the authors propose an architecture for implementing IPSec on a Xilinx Virtex-4. Cryptographic primitives are realized as a reconfigurable co-processor which is attached to a MicroBlaze soft-core. The MicroBlaze is responsible for handling the protocol layer and reconfiguring the co-processor according to the type of primitive that is required. The supported primitives are programmed into the co-processor *on-demand*, i.e., the co-processor supports only one primitive at a time, which allows for a lower overall resource consumption. However, since partial reconfiguration comes with a time penalty for switching the crypto core, this approach does not allow for extremely high throughput in a typical setting.

The authors of [18] propose an architecture which targets IPSec and SSL. The architecture is optimized for throughput rather than area and is implemented on a Xilinx Spartan-3. A soft-core handles the protocol layer and distributes cryptographic operations to an array of crypto engines, which are instantiated in parallel. Equipped with adequate buses, this system allows for an extremely high throughput while maximizing device resource utilization.

The authors of [19] implement their design on a Xilinx Virtex-5. They propose to use four different types of cores, designed to handle the AH and ESP packets of IPSec. In contrast to the previously discussed papers, the system in this paper allows one to flexibly choose how many cores of which type are to be used in the IPSec processor. This feature is enabled by a flexible bus architecture connecting these cores. The key exchange phase of IPSec is not considered in this design, hence there are no cores using asymmetric cryptography.

In [20], the authors propose an IPSec implementation on a Xilinx Virtex-2 Pro FPGA. Again, the protocol layer is handled by a soft-core and reconfigurable hardware implements AES and HMAC. Asymmetric cryptography is also not considered in this paper.

III. Implementation

Our implementation of IPSecco supports a tight coupling with a soft-core, such as Xilinx' MicroBlaze or PicoBlaze, as depicted in Figure 1. The main idea here is to let the soft-core handle the protocol layer of IPSec, whereas our core executes all cryptographic operations, thus accelerating the supported cryptographic primitives. Contrary to mentioned related work, our focus is on supplying a small core able a handle all IPSec requirements while maintaining a very low footprint.

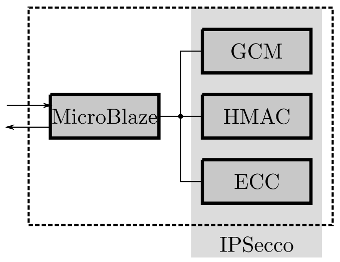

Figure 1. Integrating IPSecco

	AH (HMAC)	ESP (GCM)	IKE
IPSecco-80	Photon	Present-80 (B)	ECC (secp160r1)
IPSecco-128	Grøstl	Present-128 (S)	ECC (secp256r1)

Table I
IPSecco configurations

Our core supports Present-80 and Present-128, Grøstl, Photon and ECC. For both Present variants we have implemented a BRAM-based version – for extra low area usage and a serial version – optimized for higher speeds (compared to BRAM-based version). To complement the respective version with a hash function, we use a BRAM-based implementation of Photon and a serial implementation of Grøstl. Considering the key sizes and different versions, the overall IPSecco core is available in two different configurations, see Table I. IPSecco-80 provides a security level equivalent to 80-bit symmetric security, while the IPSecco-128 core achieves 128-bit security.

The design choices and implementation details of the three distinct blocks (enabling AH, ESP, IKE of IPSec) of our core are given in the following.

A. Message Authentication

In order to provide data integrity and authenticity in the AH mode, IPSec defines a symmetric message authentication code (MAC). The biggest advantage of a MAC is its relatively high speed compared to digital signatures. However, MACs are symmetric so that both parties have to agree on a joint key, and also do not ensure confidentiality of the transmitted information.

IPSec requires the usage of the HMAC construction. which provides a fixed size authentication tag for arbitrary messages and can be provably secure under some circumstances. The HMAC value of a message x is computed as

$$\mathrm{HMAC}_k(x) = H\left((k^+ \oplus opad)||H(k^+ \oplus ipad)||x\right)$$

where $opad$ and $ipad$ are padding constants. In order to make the construction secure, it is required that the key length used is equal or greater than the output length of the employed hash function H.

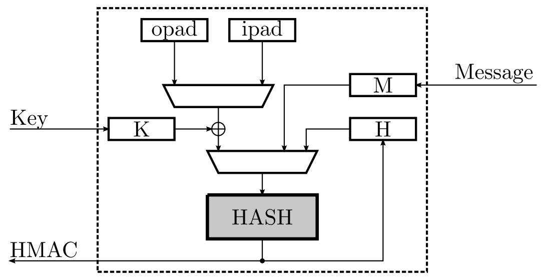

Figure 2. Architecture of the HMAC core

The implementation of an HMAC wrapper, given a hash function H, is rather straightforward, see Figure 2. We only require a shift register, implemented as SLR that stores the key (as it is needed twice), and also the hash output after the first call to the hash function has been processed. We have implemented two modern, lightweight hash functions, to be used in our wrapper.

1) PHOTON*:* One implementation option we provide for IPSECCO is the lightweight hash function PHOTON, which is designed for hardware efficiency and low area consumption. PHOTON is based on the well-established and analyzed sponge framework [21] and supports several parameter sets for different levels of security. In this work we have implemented PHOTON-160/36/36, which provides 80-bit collision resistance and has an input block length of 36 bits. The state is represented as a 7×7 nibble matrix, which is first initialized with some constants. For each message block, the message is XORed with the content of the state and afterwards a permutation is applied (12 times for every input block). This permutation consists of adding constants, appyling the PRESENT Sbox to every entry of the state, rotating some rows and finally mixing columns of the state. After all message blocks have been absorbed, the squeezing phase begins in which the hash output is generated by further applications of the permutation and extraction of parts of the state.

We have implemented PHOTON as an Application-Specific Processor (ASP) that stores its program code and also the state in a dual-port block RAM in order to save slice resources. Some operations are directly applied to the state (e.g., Sbox or XORing of constants) and do not involve any registers. Other instructions, like shifting and re-ordering of entries of the state, are performed with the help of two 4-bit registers. One register is also able to execute Galois field arithmetic, which is necessary in the MixColumns layer. Constants are generated by LFSRs, which are also used to trigger the execution of static branch instructions and hold the state (e.g., the number of applications of the permutation layer to a message block).

2) GRØSTL*:* GRØSTL is a collection of hash functions and one of the five SHA-3 finalists. GRØSTL can return message digests from 8 to 512 bits in 8-bit steps and the variant returning n bits is called GRØSTL-n. The input message M is padded before hashing and split into l-bit message blocks, then each message block is processed by the compression function f together with the l-bit chaining input (with an initial value of $h_0 = iv$). Note that, for GRØSTL variants with n up to 256 – as in our case, l is defined to be 512. After the last message block is processed; the output of the hash function passes through an output transformation, whose output size is n bits (where $n \leq 2l$).

Our serialized GRØSTL architecture is similar to the one of [22]. There is only a single block for both P and Q operations in order to save area, which also allows us to use the same block for both the f and the output transformation functions. The design basically implements a modified version of the serial AES-like data flow in [23]. While the message is processed in P mode, it is also stored inside a temporary register. In the Q mode, the result of P is stored inside the temporary register while the message is restored. Then, it is processed in Q mode and the result is combined with the P result, which is restored from the temporary register, and the previous hash value. For more details on this serial design, we refer to [22].

Instead of using our GRØSTL implementation as part of HMAC, another possibility is to use a different message authentication method. In their submission to NIST [24], the authors of GRØSTL state that an envelope construction such as

$$\mathrm{MAC}_k(x) = H(k^+||x^+||k)$$

offers security similar to HMAC, but requires only one iteration of the hash function and no wrapper.

B. Authenticated Encryption

The Galois/Counter Mode (GCM) provides authenticated encryption with low overhead and was originally designed [9] for 128-bit block ciphers, such as AES. However, with lightweight block ciphers such as PRESENT [13], a modification was proposed [25] to extend GCM to ciphers with a block size of only 64-bit.

Encryption with GCM allows for four different inputs: an initialization vector IV, a plaintext P, a key K, and additional data A, which is not encrypted but authenticated. In our notation, A is split into m blocks $A_1, A_2, ..., A_m$ of 64 bits and P (and thus also C) in n blocks $P_1, ..., P_n$. The output of the encryption step is a ciphertext $C_1, ..., C_n$ and an authentication tag T, which can be used to validate the integrity of C and A.

We have adapted and slightly modified the GCM proposal for 64-bit block sizes in order to correct what we believe[1] is a flaw in the document. Accordingly, GHASH64(H, A, C) is defined by the recursive relation

$$X_i = H \cdot \begin{cases} 0 & i = 0, \\ (X_{i-1} \oplus A_i) & \forall i = 1, ..., m-1, \\ (X_{i-1} \oplus \mathcal{P}(A_m^*)) & i = m, \\ (X_{i-1} \oplus C_i) & \forall i = m+1, ..., m+n-1, \\ (X_{i-1} \oplus \mathcal{P}(C_n^*)) & i = m+n, \end{cases}$$

[1]We have tried to contact the authors of the respective document but received no answer.

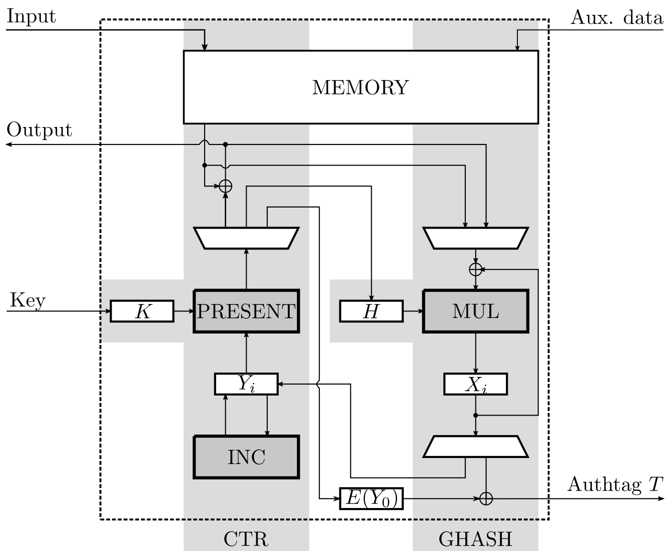

Figure 3. Architecture of the PRESENT/GCM core

and finally

$$\text{GHASH64}(H, A, C) = (((X_{m+n} \oplus \mathcal{L}(A)) \cdot H \oplus \mathcal{L}(C)) \cdot H.$$

Here, $\mathcal{P}(\cdot)$ returns a zero-padded string of 64 bits, $\mathcal{L}(\cdot)$ returns the zero-padded bit length of its input. Building on the construction of GHASH64, the authenticated encryption operations are defined by the following equations:

$$\begin{aligned} H &= \text{E}(K, 0^{64}) \\ Y_0 &= \begin{cases} IV||0^{31}||1 & \text{if len(IV) = 32} \\ \text{GHASH64}(H, \{\}, IV) & \text{otherwise} \end{cases} \\ Y_i &= \mathcal{I}(Y_{i-1}) \quad \forall i = 1, ..., n \\ C_i &= P_i \oplus \text{E}(K, Y_i) \quad \forall i = 1, ..., n-1 \\ C_n &= P_n \oplus \mathcal{M}_u(\text{E}(K, Y_n)) \\ T &= \mathcal{M}_t(\text{GHASH64}(H, A, C) \oplus \text{E}(K, Y_0)) \end{aligned}$$

Here, $\mathcal{I}(\cdot)$ denotes the 32-bit incrementation while $\mathcal{M}_x(\cdot)$ returns the x most significant bits. In the original proposal [25] we find

$$Y_0 = \begin{cases} 0^{31}||1||IV & \text{if len(IV) = 32} \\ \text{GHASH64}(H, \{\}, IV) & \text{otherwise} \end{cases}$$

which, in case the length of the IV is 32 bits, severely limits the possible counter values Y_i to 2^{32} instead of 2^{64} possibilities.

Figure 3 shows the architecture of the PRESENT/GCM core. Since GCM is basically a combination of the well-known CTR mode and multiplication/addition in GF(2^{64}), this is also reflected in our design. Our design can be used for encryption as well as for decryption – both modes include calculating the authentication tag T. In case of decryption, the computed T has to be verified by the user of our core. Note that Figure 3 depicts a simplified view on the implemented architecture.

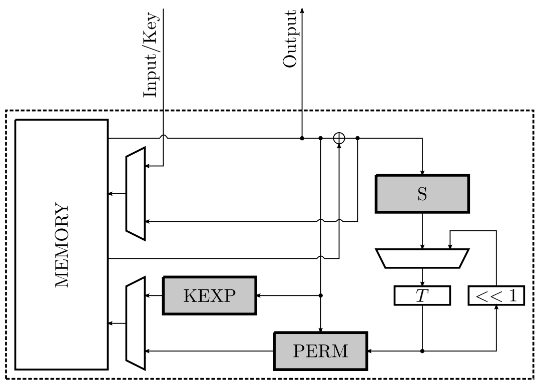

Figure 4. Architecture of the BRAM-based PRESENT implementation

1) PRESENT-BRAM: PRESENT *ultra-lightweight* cipher is a substitution-permutation network with 64-bit block size and 80/128-bit key size. It has 32 rounds, which consist of key addition, substitution, and permutation. Note that, the last round is only the key addition step. For IPSECCO, we implement PRESENT for both key options, PRESENT-80 and PRESENT-128.

For the BRAM-based implementation of PRESENT, we refer to [26] presented at ReConfig 2011. The design makes use of the existing BRAMs on the FPGA for the internal state storage, which has an impact on the reduction of the slice count. In [26], only the design for PRESENT-128 is presented. For our work, we also implement PRESENT-80 – the difference of this design is in the last phase of the operation flow (key schedule). The overall data flow of PRESENT-80 is similar to PRESENT-128; however, in the key scheduling part we have less clock cycles as a result of the shorter key length. The main modification here is in the addressing scheme. As mentioned before, the cycle count is less than PRESENT-128; however, the non-symmetric nature of PRESENT-80's addressing logic does not allow for a significant reduction of the slice count. In total, PRESENT-80 takes 969 clock cycles while PRESENT-128 takes 1062 clock cycles (note that we have implemented the on-slice Sbox version). As stated above, besides the key scheduling part the data flow for both PRESENT-80 and PRESENT-128 is more or less the same. The general block diagram for both versions is given by Figure 4.

2) PRESENT-SERIAL: In the serial implementations of PRESENT-80 and PRESENT-128, we use a 16-bit datapath approach instead of the regular 4-bit datapath as depicted in [27]. Using a 4-bit datapath would certainly decrease the slice count as the overall circuit uses less combinational circuitry. However, we also want to have a better performance compared to regular serial implementations. Therefore, we selected the 16-bit datapath for the serial implementations of PRESENT. The data is processed in 16-bit chunks, which means we only update 16 bits of the state in each clock cycle.

As can be seen in Figure 5, we have four 16-bit state registers and five 16-bit key registers in serial PRESENT-

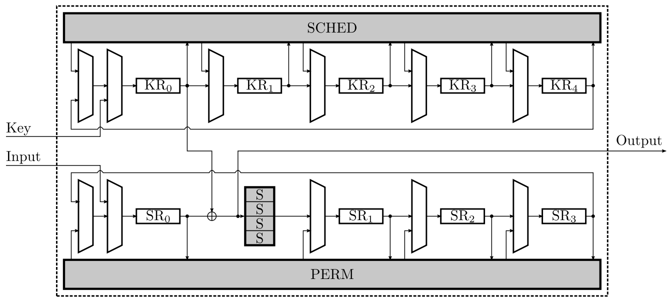

Figure 5. Architecture of the serial PRESENT-80 implementation

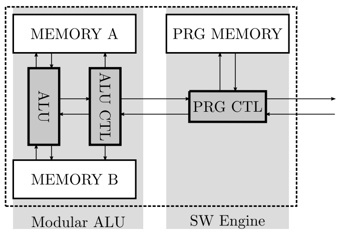

Figure 6. Architecture of the ECC core

80 implementation. These registers normally act like shift registers, except during the data load and permutation/key scheduling. We take the data and key input in the first round and we start processing at the same time. The key addition and substitution are performed at the same time in each cycle on the output of registers SR_0 and KR_0. Note that we have four 4-bit Sboxes for our 16-bit state in the substitution step. At the end of each round the permutation is applied on the full state as it is not possible to serialize the permutation step in a *cheap* way. The same is true for key scheduling, we therefore update the whole key registers at the end of each round, in parallel to permutation.

The serial implementations for PRESENT-80 and PRESENT-128 are similar. The main difference of PRESENT-128 is the additional three key registers for key storage. As a result of having a larger key size, the PRESENT-128 implementation takes more clock cycles than PRESENT-80. One round of PRESENT-80 is processed in 5 clock cycles while it takes 8 cycles for PRESENT-128. Furthermore, there is an additional multiplexer at the input of SR_1 to select between *key-added & Sboxed* and *unprocessed* SR_0 output, as we have to wait for a few cycles before the permutation process at the end of each round (which is done in parallel with the key scheduling to preserve the data flow).

C. Key Exchange

RFC 6379 [7] defines the recommended cryptographic primitives for IPSec for security levels of 128-bit and above. For key exchange, and a security level equivalent to AES-128, either RSA or use of prime field elliptic curve cryptography is required. We opted for ECC, the given curves are the same as recommended by NIST and earlier by SECG [28]. The naming conventions are slightly different, so the NIST curve ECC-p256 was formerly known as secp256r1. While NIST only standardized the secp curves from 224-bit upwards, there also exist recommended curves for 160-bit and 192-bit arithmetic, named secp160r1 and secp192r1.

At ReConfig 2011, a reconfigurable ECC-p core [29] using a microcode approach [30] was introduced. The design uses a very small arithmetic and logic unit (ALU) and a control unit which is able to read and decode certain instruction sequences from BRAM. It is able to perform modular arithmetics from point addition and doubling up to point multiplication. Because the program code (e.g., how to perform doubling, addition, a fast NIST reduction, etc.) and all necessary constants (base point, etc.) are stored in BRAM, it is possible to switch from one curve to another just by changing the stored program code and data in one BRAMs. This can happen on the fly after initialization of the FPGA.

The ability to change the prime curve for ECC operations just by swapping the memory content makes this design very suitable for less expensive, older FPGAs. This directly benefits the Xilinx Spartan-3 Series, which do not support partial reconfiguration. We have therefore chosen to implement a similar design as [29] but instead of implementing ECC-p224/secp224r1 we implemented secp160r1 alongside ECC-p256/secp256r1 to cover a wider range of security levels. As ECC-p256/secp256r1 provides an equivalent security level as AES-128 or PRESENT-128, secp160r1 is a very good match for lower security cryptographic primitives like the implemented PRESENT-80.

The architecture of our ECC module is depicted in Fig. 6 and is, as mentioned before, very similar to the one of [29]. A control unit (PRG CTL) reads and decodes instructions from one BRAM and passes that information to the modular ALU. That unit features a small arithmetic unit using a 16-bit data path which can perform basic operations like multiplication of two 16-bit values, addition, subtraction, and comparison. It also contains two dual-port BRAMs to be able to read both operands and store the result of a computation in a single clock cycle.

Compared to [29] our design is slightly smaller, most likely because we focused only on Xilinx Spartan FPGAs with dedicated Multipliers/DSPs. Thus, there was no need to optimize the data path to make the same design fast on different FPGA architectures.

IV. RESULTS AND DISCUSSION

Table II and Table III show our synthesis results for the two representatives of Xilinx' Spartan family. We have evaluated two major design strategies, BRAM-based implementations and serial implementations. The BRAM-based approach makes use of the FPGA's on-board BRAMs in order to reduce the slice count. It also comes with a performance penalty and is therefore more suitable for low throughput applications. Our serial implementations are also implemented with low area requirements in mind but still allow for faster processing

	#SLICEs	#LUTs	#FFs	#BRAMs	#MULs
PRESENT-80 (B)	84	135	38	1	0
PRESENT-80 (S)	131	222	153	0	0
PRESENT-80/GCM (B)	356	630	448	3	0
PRESENT-80/GCM (S)	402	714	561	2	0
PRESENT-128 (B)	84	130	38	1	0
PRESENT-128 (S)	161	295	201	0	0
PRESENT-128/GCM (B)	356	632	450	3	0
PRESENT-128/GCM (S)	443	771	609	2	0
PHOTON (B)	91	160	50	1	0
PHOTON/HMAC (B)	133	235	80	1	0
GRØSTL (S)	349	647	376	0	0
GRØSTL/HMAC (S)	601	1119	693	0	0
ECC-160	569	1028	507	3	1
ECC-256	569	1028	507	3	1
IPSECCO-80	**1107**	**1986**	**1123**	**7**	**1**
IPSECCO-128	**1613**	**2918**	**1809**	**5**	**1**

Table II
CHARACTERISTICS OF OUR IMPLEMENTATION FOR XILINX SPARTAN-3

	#SLICEs	#LUTs	#FFs	#BRAMs	#MULs
PRESENT-80 (B)	26	83	38	1	0
PRESENT-80 (S)	48	167	153	0	0
PRESENT-80/GCM (B)	138	455	451	3	0
PRESENT-80/GCM (S)	164	515	572	2	0
PRESENT-128 (B)	31	82	38	1	0
PRESENT-128 (S)	62	231	201	0	0
PRESENT-128/GCM (B)	141	454	459	3	0
PRESENT-128/GCM (S)	174	512	624	2	0
PHOTON (B)	30	89	50	1	0
PHOTON/HMAC (B)	60	152	80	1	0
GRØSTL (S)	136	411	385	0	0
GRØSTL/HMAC (S)	275	808	760	0	0
ECC-160	221	630	482	3	1
ECC-256	221	630	482	3	1
IPSECCO-80	**424**	**1301**	**1101**	**7**	**1**
IPSECCO-128	**670**	**1950**	**1866**	**5**	**1**

Table III
CHARACTERISTICS OF OUR IMPLEMENTATION FOR XILINX SPARTAN-6

of data (when compared to the BRAM versions). They are suitable for medium throughput applications.

As expected, the resource consumption of our design is significantly lower than that of previously published work on IPSec FPGA implementations. As example, in [17], AES, SHA-256, and a modular exponentiation core for Elliptic curve Diffie-Hellmann was implemented on a Xilinx Virtex-4 FPGA. While the Virtex-4 series is significantly more powerful then our Spartan-3 target, it employs a similar 4-input LUT architecture. Therefore, the slice count as well as LUT and flip-flop usage can be compared to each other. Their AES core uses 1862 slices while our similarly secure PRESENT-128 implementation in the smallest option (BRAM) only requires 84 slices. The area difference is still large when looking at the hash functions; their SHA-256 implementation requires 924 slices while a similarly secure, serial GRØSTL only requires 347 slices.

Reducing the security requirements to an equivalent symmetric security of 80 bits would further reduce the necessary slice count, especially since we then could use a low area, BRAM-based implementation of PHOTON instead of GRØSTL. Note that these numbers alone of course do not allow a fair comparison, since the cores of [17] most likely allow a higher throughput while we primarily focused on minimizing the slice count. For the modular exponentiation core, the slice count is quite comparable; namely 499 slices and three DSP48s in [17] versus 569 slices and one 18-bit block multiplier in our design.

V. CONCLUSION AND FUTURE WORK

In this work, we have presented how lightweight building blocks for the execution of the popular IPsec protocol can be efficiently realized on reconfigurable hardware. Our research shows that even a complex protocol suite, which requires several cipher standards and modes of operation, can be implemented even on very low-cost and energy-efficient Spartan-3 FPGAs. By selecting algorithms which are generally designed for efficient realization in hardware, and by adapting them to the specification of our target device, we were able to achieve a significant reduction in terms of area usage compared to related work. With our results we show that standardized security can be achieved on reconfigurable hardware with only a modest investment into slice resources. This leaves more space for other functionality and target hardware.

As a lightweight IPSec core that will probably be used mostly on low-power devices deployed in the field, an attacker could employ side-channel analysis to obtain secret keys or introduce faults into the crypto cores or management modules. As protection mechanisms against such attacks usually require large amounts of device resources and make analysis hard, we did not consider them in this work. However, for future work we plan to integrate generic protection layers [31] as well as countermeasures that are specifically designed for the usage within lightweight applications and tailored to the used ciphers.

Another interesting topic would be the investigation of different combinations of algorithms, e.g., CLEFIA, KECCAK, or HIGHT and potential options for resource sharing. Note that a generic IPSec core requires hash functions, block ciphers, as well as an asymmetric construction. These primitives are occupying valuable resources on the device but are not always executed simultaneously (e.g., HMAC is not needed when only ESP mode with authenticated encryption enabled is used). One common solution for efficient resource sharing is partial reconfiguration which is only available on more powerful but also more expensive and larger FPGAs. Therefore, we plan to put further effort into the integration and sharing of components between different ciphers (e.g., SBox look-up tables, block RAMs or registers for keys) in order to further reduce the size of our design.

ACKNOWLEDGMENT

In this project Oliver Mischke has been part-financed by the European Union, Investing in your future, European Regional Development Fund. The work described in this paper has also been supported by the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II.

REFERENCES

[1] "RFC-791," http://www.ietf.org/rfc/rfc791.txt.

[2] "RFC-1349," http://www.ietf.org/rfc/rfc1349.txt.

[3] "RFC-2474," http://www.ietf.org/rfc/rfc2474.txt.

[4] "lwIP," http://savannah.nongnu.org/projects/lwip/.

[5] "RFC-4301," http://www.ietf.org/rfc/rfc4301.txt.

[6] "RFC-4308," http://www.ietf.org/rfc/rfc4308.txt.

[7] "RFC-6379," http://www.ietf.org/rfc/rfc6379.txt.

[8] NIST, "Federal Information Processing Standards Publication 197."

[9] M. Dworkin, "NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," NIST, Tech. Rep., November 2007.

[10] "RFC-4868," http://www.ietf.org/rfc/rfc4868.txt.

[11] NIST, "Federal Information Processing Standards Publication 180-2."

[12] "RFC-5903," http://www.ietf.org/rfc/rfc5903.txt.

[13] A. Bogdanov, G. Leander, L. R. Knudsen, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT - An Ultra-Lightweight Block Cipher," in *Proceedings of CHES 2007*, ser. LNCS, no. 4727. Springer-Verlag, 2007, pp. 450 – 466.

[14] I. 29192-2:2012, "Information technology – Security techniques – Lightweight cryptography – Part 2: Block ciphers," http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=56552.

[15] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen, "Grøstl – a SHA-3 candidate," Submission to NIST (Round 3), 2011. [Online]. Available: http://www.groestl.info/Groestl.pdf

[16] J. Guo, T. Peyrin, and A. Poschmann, "The PHOTON Family of Lightweight Hash Functions," in *CRYPTO*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed., vol. 6841. Springer, 2011, pp. 222–239.

[17] A. Salman, M. Rogawski, and J.-P. Kaps, "Efficient Hardware Accelerator for IPSec Based on Partial Reconfiguration on Xilinx FPGAs," in *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs*, ser. RECONFIG '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 242–248.

[18] H. Wang, G. Bai, and H. Chen, "A Gbps IPSec SSL Security Processor Design and Implementation in an FPGA Prototyping Platform," *Journal of Signal Processing Systems*, vol. 58, no. 3, pp. 311–324, Mar. 2010.

[19] Y. Niu, L. Wu, L. Wang, X. Zhang, and J. Xu, "A Configurable IPSec Processor for High Performance In-Line Security Network Processor," in *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, ser. CIS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 674–678.

[20] J. Lu and J. Lockwood, "IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '05. Washington, DC, USA: IEEE Computer Society, 2005.

[21] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Cryptographic sponge functions," Submission to NIST (Round 3), 2011. [Online]. Available: http://sponge.noekeon.org/CSF-0.1.pdf

[22] E. B. Kavun and T. Yalcin, "On the Suitability of SHA-3 Finalists for Lightweight Applications," ser. The Third SHA-3 Candidate Conference, 2012.

[23] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen, “Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core,” in *Proceedings of the 9th EUROMICRO Conference on Digital System Design*, ser. DSD ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 577–583. [Online]. Available: http://dx.doi.org/10.1109/DSD.2006.40

[24] P. Gauravaram, L. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. Thomsen, “Grøstl–a SHA-3 candidate,” *Submission to NIST*, 2008.

[25] D. McGrew and J. Viega, “The Galois/Counter Mode of Operation (GCM),” Tech. Rep., May, 2005.

[26] E. B. Kavun and T. Yalcin, “RAM-Based Ultra-Lightweight FPGA Implementation of PRESENT,” *Reconfigurable Computing and FPGAs, International Conference on*, vol. 0, pp. 280–285, 2011.

[27] C. Rolfes, A. Poschmann, G. Le, and C. Paar, “Ultra-lightweight implementations for smart devices - security for 1000 gate equivalents,” in *in Proceedings of the 8th Smart Card Research and Advanced Application IFIP Conference Ű CARDIS 2008, ser. LNCS*. Springer-Verlag, 2008, pp. 89–103.

[28] S. for Efficient Cryptography Group, “SEC 2: Recommended Elliptic Curve Domain Parameters,” Available at http://www.secg.org/collateral/sec2_final.pdf, September 2000.

[29] M. Varchola, T. Güneysu, and O. Mischke, “MicroECC: A Lightweight Reconfigurable Elliptic Curve Crypto-processor,” in *ReConFig*. IEEE Computer Society, 2011, pp. 204–210.

[30] J. Vliegen, N. Mentens, J. Genoe, A. Braeken, S. Kubera, A. Touhafi, and I. Verbauwhede, “A compact FPGA-based architecture for elliptic curve cryptography over prime fields,” in *ASAP*. IEEE Computer Society, 2010, pp. 313–316.

[31] T. Güneysu and A. Moradi, “Generic Side-Channel Countermeasures for Reconfigurable Devices,” in *CHES*, ser. Lecture Notes in Computer Science, vol. 6917. Springer, 2011, pp. 33–48.