2012 IEEE 18th International Conference on Parallel and Distributed Systems

Dictionary Attack on TRUECRYPT with RIVYERA S3-5000

Ayman Abbas[1], Claas Anders Rathje, Lars Wienbrandt, Manfred Schimmler
Department of Computer Science
Christian-Albrechts-University of Kiel
24098 Kiel, Germany
Email: {aab,car,lwi,masch}@informatik.uni-kiel.de

***Abstract*—The popular free encryption software TRUECRYPT uses whole device or partition encryption as well as encrypted container files to protect sensible data from unauthorized access. Several combinations of encryption algorithms and hash functions used for the key derivation can be chosen by the user. This paper regards the combination with SERPENT as encryption algorithm and WHIRLPOOL as hash function for the key derivation. A dictionary attack has been implemented for this combination using the FPGA-based high-performance computer RIVYERA S3-5000. The achieved performance reaches more than 200,000 passwords per second. Compared to 820 passwords per second, achieved by a fully threaded Intel Core i7-970 system at 3.2GHz using the Crypto++ library, this leads to a speedup of more than 247 with energy savings of about 99%.**

***Keywords*-known-plaintext dictionary attack; SERPENT; WHIRLPOOL; PBKDF2; FPGA; reconfigurable high-performance computing;**

I. INTRODUCTION

With the increasing importance of cryptography in information systems cryptanalysis has gained predominant attention in research. One popular encryption system to protect personal and sensitive data from unauthorized access is TRUECRYPT [1], a free, on-the-fly and cross-platform open source software. The encryption using this kind of software is fast enough not to form any drawbacks for all-day use. However, for the forensic analysis it is hard to access the plain data if the password is not known. To recover a password, one possibility is to try to read the master keys from a memory dump [2], [3]. However, the container has to be already mounted at the time of memory acquisition, which is not the general case for a forensic analyzer who operates on shut down hardware. A different approach is a dictionary attack, where the header keys have to be derived for each individual password candidate from a huge list to decrypt the container header and finally recover the password. This process may be very time consuming, depending on the size of the dictionary, the kinds of implied password variations, and, of course, the speed of the underlying attack system. Our test system (Intel Core i7 970 @ 3.2GHz) with a cracking software using the Crypto++ libraries [4] reached a speed of about 820 passwords per second. Therefore, a trivial approach is to try different password lists on a single or several computers, such as a cluster system, with appropriate cracking software (e.g. *unprotect.info* [5] or the commercial *DNA* software tool [6]). However, the speed increases linear with the number of nodes in the cluster and with increasing requirements for power, space and cooling for each additional node. Hence the demand for cheaper and easier maintainable solutions arises. Solutions like TrueCrack [7] utilize GPU cores for containers encrypted with an AES RIPEMD-160 combination, but do not reach a significant speedup compared to a single node as well. Anyway, to our knowledge, other combinations have not been implemented for GPUs yet.

Another approach is to take benefits from the high performance of reconfigurable Field Programmable Gate Arrays (FPGAs), which is described in this work. The high-performance supercomputer RIVYERA S3-5000 [8] is based on 128 Xilinx Spartan3-5000 FPGAs, providing ample resources to expect a high speedup against our test system.

In detail, we are addressing TRUECRYPT v7.1 containers encrypted with the SERPENT [9] cipher and the WHIRLPOOL [10] hash function using an arbitrary password list stored in a dictionary. Existing implementations of the SERPENT and the WHIRLPOOL algorithms on FPGAs [11]–[14] can not be easily used for decrypting TRUECRYPT containers, since they are not directly optimized for this attack, e.g. TRUECRYPT uses Serpent in XTS mode. Hence, we try to decrypt the part of the container header containing the encrypted known plaintext string "TRUE" with a combined implementation of the SERPENT decryption algorithm in XTS mode [15] and the *Password Based Key Derivation Function* (PBKDF2) [16] algorithm using HMAC-WHIRLPOOL [17] on all 128 FPGAs of a fully equipped RIVYERA S3-5000.

Our measured performance reaches more than 200,000 passwords per second, i.e. a standard dictionary (e.g. a summary over common words in different languages) with about 17.5 million passwords in less than 90 seconds.

II. TRUECRYPT

TRUECRYPT [1] is an encryption software system which is used for on-the-fly encrypting and decrypting data stored on a mass storage device. On-the-fly encryption (OTFE)

[1]corresponding author

1521-9097/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPADS.2012.23

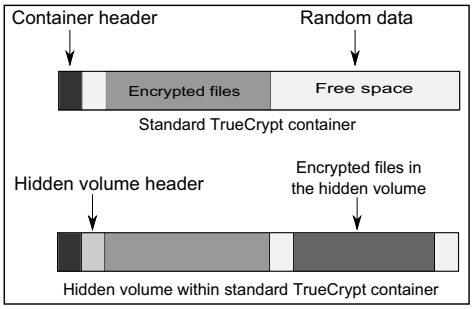

Figure 1. The header format of a TRUECRYPT container.

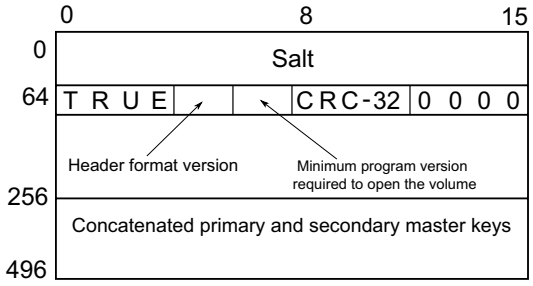

Figure 2. Header format of a TRUECRYPT container.

denotes that data is encrypted and decrypted automatically before being saved or loaded.

There are two types of TRUECRYPT volumes, file-hosted (TRUECRYPT container) and partion/device-hosted. The dictionary attack which has been implemented in this work targets the former type which is searching for the correct password to decrypt a TRUECRYPT container. The standard TRUECRYPT container is divided into three main spaces: header, encrypted data and free space containing random data. This is illustrated in Fig. 1.

The header is stored in the first 512 bytes of the container. The first 64 bytes of this header state the salt which is saved unencrypted. The remainder of the header (the next 448 bytes after the salt) is encrypted in XTS-mode [15] using two header keys. XTS-mode is commonly used to encrypt data on block-oriented storage devices. The header keys, required to encrypt and decrypt the container header, are derived using the PBKDF2 algorithm which takes the salt and the user-supplied password as inputs. As Fig. 2 shows, the correctly decrypted header contains in addition to the salt the ASCII string "TRUE", the CRC-32 checksum of the decrypted bytes 256-511 and the primary and secondary master keys (XTS-mode), required to encrypt and decrypt the actual user data in the container. On creation of a TRUECRYPT container, the salt, the primary and the secondary master key are generated using a random number generator.

TRUECRYPT supports eight encryption algorithms: AES [18], SERPENT [9], TWOFISH [19] and five different combinations of cascaded algorithms, i.e. AES-TWOFISH, AES-TWOFISH-SERPENT, SERPENT-AES, SERPENT-TWOFISH-AES and TWOFISH-SERPENT. In the current version of TRUECRYPT all these encryption algorithms are used in XTS-mode [15]. The supported hash algorithms for the PBKDF2 [16] to derive the header keys are RIPEMD-160 [20], SHA-512 [21] and WHIRLPOOL [10]. The user is able to select an encryption and hash algorithm during the process of creating a TRUECRYPT container. One of the main features of TRUECRYPT is the ability to create a hidden volume within the standard TRUECRYPT container, as shown in Fig. 1. The hidden volume contains a header and separate space for the encrypted data as well.

While mounting a TRUECRYPT container, the container header and the hidden volume header are read in to the RAM and the user is asked to enter the password. TRUECRYPT tries to decrypt the container header using all the possible combinations of the encryption and hash algorithms in the process of trial and error. The decryption is considered successful if the first 4 bytes of the decrypted header contain the ASCII string "TRUE", and if the CRC-32 checksum of the decrypted bytes 256-511 match its stored value. TRUECRYPT then takes the primary and secondary master keys from the decrypted header and uses them to decrypt any encrypted files within the container. However, if the decryption fails for all combinations, TRUECRYPT tries to decrypt the hidden volume header in the same way, even if there are no hidden volumes within the container. If this succeeds, the hidden volume is mounted. Otherwise, the mounting process is terminated due to one of the following errors: wrong password, corrupted container, or not a TRUECRYPT container.

The process of decrypting the header of a TRUECRYPT container consist of two main steps, which are described in the following. The first step is to derive the header keys using the PBKDF2 algorithm. The second step is to decrypt the encrypted area of the header using the derived keys.

A. Header Keys Derivation

TRUECRYPT derives the header keys (HK) using the PBKDF2 algorithm [16]. This is specified as follows:

$$HK = PBKDF2(P, S, c, hkLen)$$

whereby P is the password, S is the salt, c is the number of iterations and $hkLen$ is the header keys length. The salt S is a 512-bit random number generated in the creation process of the TRUECRYPT container. It makes it impossible for a potential attacker to generate a general list of keys derived from a password list to use against every TRUECRYPT container. The number of iterations c specifies how many times the underlying pseudorandom function has to be iterated. These iterations increase the time required to make a brute force or dictionary attack significantly. In this work HMAC-WHIRLPOOL is implemented as pseudorandom function whereby TRUECRYPT specifies 1000 iterations

for this function. The header key length $hkLen$ depends on the used encryption algorithm. SERPENT in XTS mode, which is used in this work, requires two keys, each of length 256bit, i.e. $hkLen = 512\text{bit}$. The length of the output of the pseudorandom function is 512bit as well, leading to the following equation:

$$HK = PBKDF2(P, S, c, 512) = U_1 \oplus U_2 \oplus .... \oplus U_c$$

whereby

$$\begin{aligned} U_1 &= HMAC\text{-WHIRLPOOL}(P, S \parallel INT(i)) \\ U_2 &= HMAC\text{-WHIRLPOOL}(P, U_1) \\ &\vdots \\ U_c &= HMAC\text{-WHIRLPOOL}(P, U_{c-1}) \end{aligned}$$

and $INT(i)$ is a four byte representation of the block index i, and $\parallel$ is the concatenation operator. In our case, only block is required since the length of the derived key is equal to the size of a WHIRLPOOL-hash resulting in i being constantly 1.

HMAC-WHIRLPOOL*:* The Keyed-Hash Message Authentication Code (HMAC) is a message authentication code (MAC) calculated using a secret key and a cryptographic hash function. HMAC-WHIRLPOOL (HW) is the algorithm, which used to calculate the MAC of a message M using a secret key K and WHIRLPOOL as cryptographic hash algorithm as defined below:

$$HW(K, M) = W((K_0 \oplus opad) \parallel W((K_0 \oplus ipad) \parallel M))$$

whereby W is the WHIRLPOOL hash function, $ipad$ is the byte $0x36$ repeated 64 times and $opad$ is the byte $0x5c$ repeated 64 times again. K_0 is obtained from the key K by zero-padding when it is less than 64 bytes, hashed to 64 bytes if longer or taken as it is in case it equal to 64 bytes. In this work the password P states the key K and the message M is the concatenation of the salt S and $INT(i)$ in the first iteration. Thereafter, the message M will be the result of the previous iteration U_{i-1}.

WHIRLPOOL*:* WHIRLPOOL is a one-way, collision resistant cryptographic hash function. It produces a 512bit hash value of a message of length less than 2^{256}. It was adopted by the International Organization for Standardization in (ISO/IEC)10118-3 [22] and approved by NESSIE [23]. WHIRLPOOL is constructed from a compression function, which is based on 512-bit block cipher W and follows the Miyaguchi-Preneel scheme [24]. According to the rules in [10] the message M has to be padded before being hashed. Afterwards, it is split into n 512-bit blocks $m_1, m_2, \ldots, m_n$. The block cipher W takes m_i as plaintext and H_{i-1} as cipher key and the result is used as followed to compute the hash value H_i:

$$H_i = W_{H_{i-1}}(m_i) \oplus H_{i-1} \oplus m_i$$

Where H_0 is the initialization vector consisting of an (8×8) matrix of zero bytes.

The cipher W treats the m_i and H_i blocks as (8×8) byte matrix, and each byte is interpreted as a polynomial in the Galois field $GF(2^8)$. The hash value of the message M is obtained after processing the last block m_n. The block cipher W processes ten rounds of the following transformations (intermediate results are called *state matrices*):

1) Non-linear layer: each byte in the state matrix is substituted using an S-box.
2) Cyclical permutation: each column j in the state matrix is rotated downwards by j positions.
3) Linear diffusion layer: modular multiplication of the state matrix and a constant matrix (generator matrix) in $GF(2^8)$.
4) Key addition: XOR-operation of the state and the round key matrices.

The non-linear layer substitutes each polynomial $b \in GF(2^8)$ in the state matrix with another polynomial $SBox(b) \in GF(2^8)$.

The key schedule, which generates the ten round keys from the cipher key H_{i-1}, consists of ten rounds as well performing the same transformations. The round keys for the key schedule are ten predefined constants inherited from the first 80 bytes of the S-box.

B. Container Header Decryption

For the dictionary attack described in this work, it is sufficient to decrypt only the first 128bit block after the salt of the header. As shown in Fig. 2, this block starts with the encrypted ASCII string "TRUE". We have implemented the SERPENT decryption algorithm in XTS-mode to decrypt this block using the derived header keys. If the first four bytes of the decryption of this block contains the ASCII string "TRUE" and the last four bytes equal zero the password from which the header key was derived is highly considered as the correct password.

SERPENT*:* SERPENT [9] is a symmetric-key block cipher. It consists of a 32-rounds substitution-permutation network operating on four 32bit words. The cipher accepts a user key of length less than or equal to 256bit to encrypt 128bit plaintext P to 128bit ciphertext C. The algorithm can be described according to [9] as follows:

$$\begin{aligned} B_0 &:= IP(P) \\ B_{i+1} &:= R_i(B_i) \qquad \text{i= 0,...,31} \\ C &:= FP(B_{32}) \end{aligned}$$

whereby IP denotes the initial permutation, FP the final permutation and

$$\begin{aligned} R_i(B_i) &= L(S_i(B_i \oplus K_i)) \qquad \text{i= 0,...,30} \\ R_i(B_i) &= S_i(B_i \oplus K_i) \oplus K32 \qquad \text{i=31} \end{aligned}$$

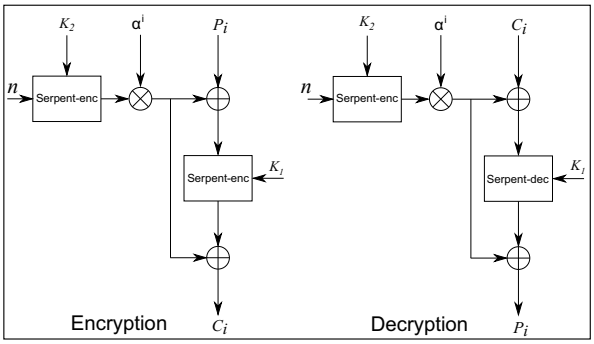

Figure 3. SERPENT encryption and decryption in XTS-mode. K1 and K2 define the header keys, $\otimes$ represents the multiplication of two polynomials over the binary field $GF(2)$ modulo $(x^{128} + x^7 + x^2 + x + 1)$, α is a primitive element of $GF(2^{128})$, n is the data unit index, and i is the cipher block index within a data unit.

The cipher uses eight different S-boxes $S_0, S_1, \ldots, S_7$. Each S-box is used in four rounds (round R_i uses S-box $S_{(i \bmod 8)}$). The S-box in SERPENT substitutes a 4-bit input by a 4-bit output. In the equation above K_i is the round key and L is the application of the linear transformation.

The decryption in SERPENT uses the inverse of S-boxes and linear transformation whereby the rounds obtain the round keys in inverse order.

Key Schedule: SERPENT requires 33 round keys to encrypt a 128bit plaintext. The user key is used in the key schedule to generate these 33 round keys. The length of each round key is 128bit as well. The user key is padded with zeros to 256bit if its length is less. Thereafter, this key is split into eight 32bit words $w_{-8}, w_{-7}, \ldots, w_{-1}$. The intermediate key $w_0, \ldots, w_{131}$ is calculated using the following affine recurrence:

$$w_i \quad := \quad (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) <<< 11$$

whereby $\phi = 0x9e3779b9$ is a 32bit constant, and the round keys are obtained after using S-boxes as stated in [9].

XTS-mode: XTS is a mode of operation designed to encrypt the data on block-oriented storage devices. XTS [25] is a tweakable mode, based on Rogaway's XEX (Xor-Encrypt-Xor) mode [26] and uses the technique of ciphertext stealing. This mode was approved by NIST [15] and by the IEEE standard for cryptographic protection of data on block-oriented storage devices [25]. Fig. 3 illustrates the process of encryption and decryption using SERPENT in XTS-mode.

In this work, the first 128bit block C_i, containing the ASCII string "TRUE", is decrypted using SERPENT in XTS-mode. This is the first block ($i = 0$) within the first data unit ($n = 0$), whereby the size of each data unit in TRUECRYPT is 512 bytes. The keys are obtained from the derived key (DK) such that: $DK = K_1 \parallel K_2$.

III. RIVYERA ARCHITECTURE

The massively parallel FPGA-based hardware platform RIVYERA was first introduced as COPACOBANA 5000 for applications in the fields of bioinformatics [27]–[29]. It is the direct successor of the DES-Cracker COPACOBANA [30], [31] which proved its cryptanalytical capabilities in attacking other ciphers, e.g. A5/1 [32], as well.

We have implemented the dictionary attack on the SERPENT/WHIRLPOOL encrypted TRUECRYPT container on the specific RIVYERA S3-5000 rev 1, developed and distributed by SciEngines GmbH [8]. The in-built multiple FPGA-based supercomputer and a standard server grade mainboard, running a Linux operating system on an Intel Core i7-970 processor with 8GB of RAM, state the two basic elements this architecture consists of. The FPGA computer is equipped with up to 128 user configurable Xilinx Spartan3-5000 FPGAs, distributed over 16 FPGA cards, each containing eight user FPGAs. Additionally, a DRAM module with a capacity of 32MB is attached to each user FPGA. However, the DRAM is not required for our application.

The FPGAs are connected by a systolic-like bus system, i.e. each FPGA on an FPGA card is connected with two neighbors forming a ring including a communication controller. The FPGA card slots are connected as a ring as well, providing the connection of the communication controllers on each FPGA card. At least one communication interface of the FPGA cards is connected via PCIe to the host mainboard forming the communication link to the host software. This way, the distribution of the passwords from the dictionary to all FPGAs can be performed very quickly such that no large buffer memory is required on FPGA side (for more details see Sect. IV).

For application development, SciEngines provides an API for software development and hardware design on the RIVYERA architecture, i.e. an API controlling the data transfer between the host software and the FPGAs including broadcast facilities, and an API for the user defined hardware configuration of the FPGAs controlling the data transfer to other FPGAs and the host. A picture of the RIVYERA S3-5000 and the overview of its hardware structure is shown in Fig. 4. The chosen design allows a small packaging. Thus, RIVYERA is packed in a standard rack mountable $3U$ housing and powered by two redundant 650W power supplies.

IV. FPGA IMPLEMENTATION OF SERPENT AND WHIRLPOOL DICTIONARY ATTACK

To implement our attack against TRUECRYPT a PBKDF2 component with WHIRLPOOL and a SERPENT decryption core in XTS-mode are required. Since SERPENT can be implemented reasonable fast while still demanding a comparable low amount of resources [13], the PBKDF2 demands a lot of iterations and RAM for data. Hence, regarding

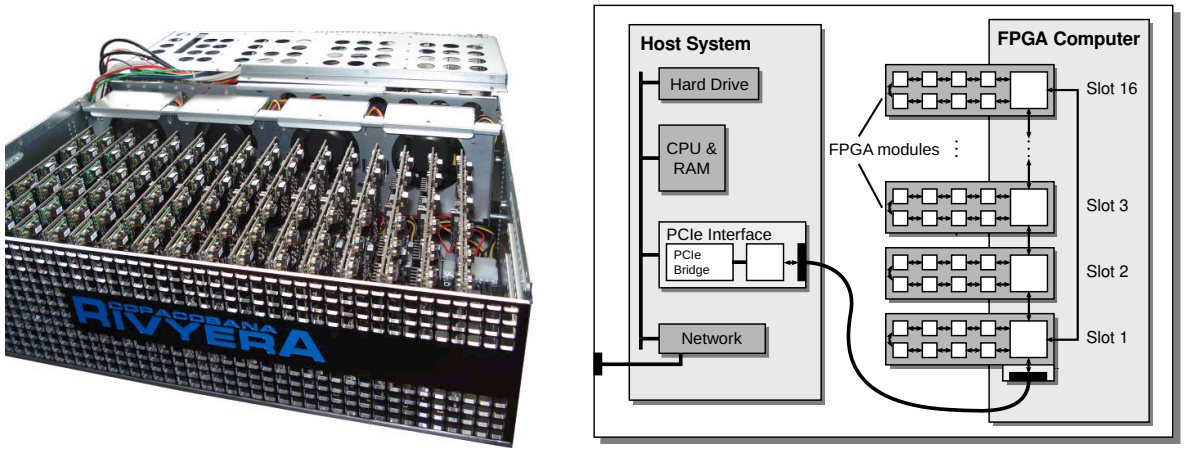

Figure 4. RIVYERA S3-5000 picture and hardware structure.

performance, the focus has been set on the key derivation. We implemented two independent key derivation cores along with one decryption core on each of the FPGAs as shown in Fig. 5. The host PC initially broadcasts the salt and the first cipher-block to all the FPGAs, afterwards 100 passwords are sent to the first FPGA before switching to the next one. The top entity equally distributes the incoming passwords in both FIFOs for the KDFs. The transmission speed of the RIVYERA-API is sufficient to keep the FIFOs filled as long as passwords are being sent from the dictionary. The passwords from the input FIFOs are being processed by the key derivation cores. The generated keys are put in their corresponding output FIFOs from where an alternating multiplexer utilizes the SERPENT decryption. The derived keys are identified by a counter to recover the correct password in the case of a successful decryption. Since the decryption needs less time than deriving eight keys (four from each pipelined key derivation core, s. Section IV-A), a single decryption core is sufficient to handle the incoming data. Afterwards, the decrypted data is compared to the known plain text parts within the first and the last four bytes of the decrypted ciphertext. In case of a match, the comparison core sends the password count back to the host. All 128 FPGAs are configured with the same configuration file.

A. Key derivation

A complete overview of the key derivation data flow on each FPGA is illustrated in Fig. 6. The salt is constant among all passwords and is kept in distributed RAM of the FPGA. As mentioned above the key derivation is the most time consuming part in the core since it has to perform 1000 iterations. The PBKDF2 can be implemented following the reference pseudo code in [16]. It is implemented as finite state machine utilizing a pipelined WHIRLPOOL core. The HMAC [33] steps have been merged into the same state

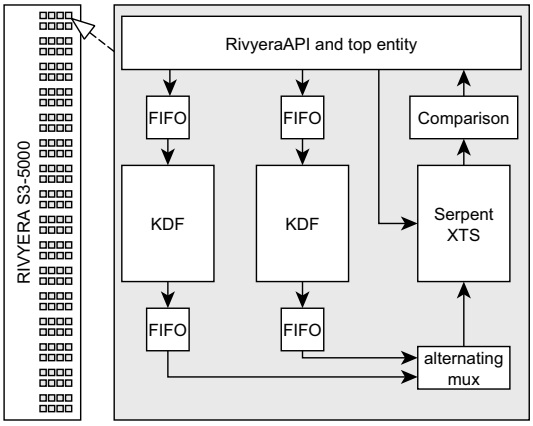

Figure 5. FPGA core design overview. Two key derivation (KDF) cores along with a decryption core (SERPENT XTS)

machine to utilize the same RAM resources for temporary values.

WHIRLPOOL: As already mentioned, we concentrated on the WHIRLPOOL core to maximize its performance. The WHIRLPOOL hash function allows a variety of implementation tradeoffs [10]. Regarding hardware implementations, efforts have been made towards compact designs [11] in contrast to fast designs [14]. We decided to take a full 512bit wide approach to avoid a big overhead in RAM usage. This may be caused if cores with a smaller data width would be implemented since large temporary values inside the PBKDF2 around each WHIRLPOOL core would have to be stored. The WHIRLPOOL core is implemented using a four stages pipeline, representing the four internal layers of the algorithm described before. This leads to four independent datasets being processed concurrently in consecutive stages. It internally feeds itself ten times before the hashed block becomes valid at the output as illustrated in Fig. 7.

The *non-linear layer* is the parallel application of a non-

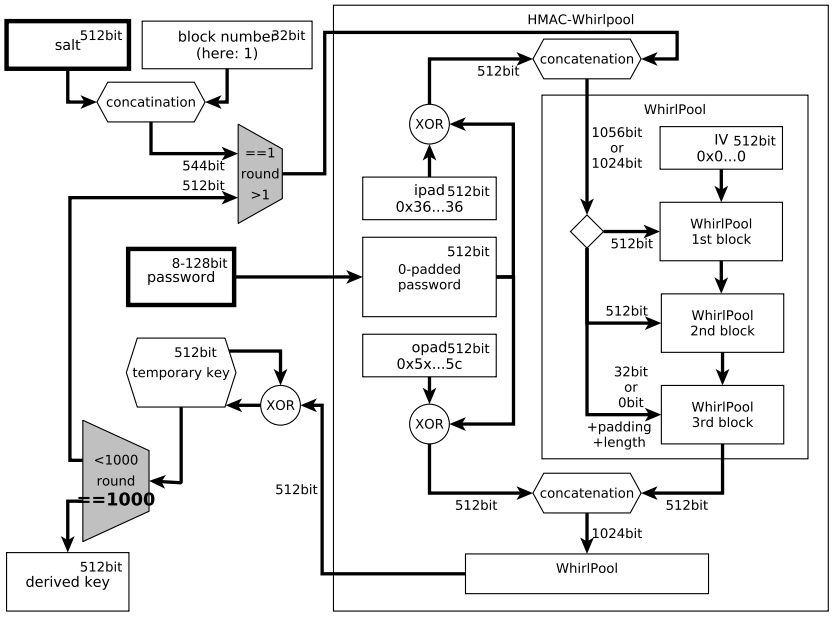

Figure 6. Key derivation data flow overview.

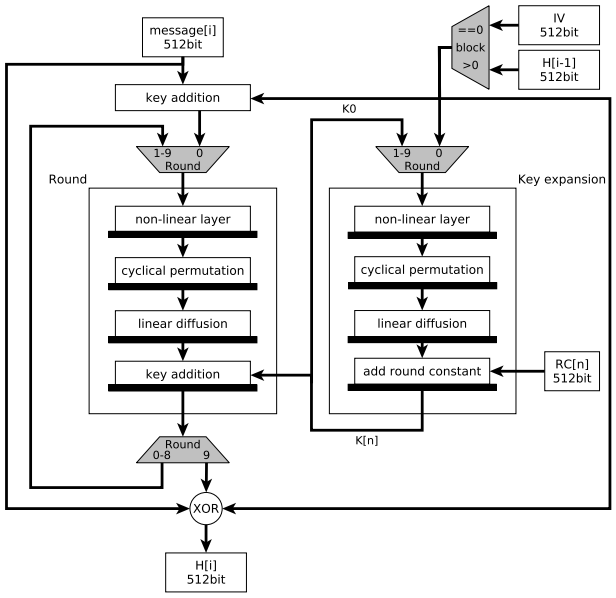

Figure 7. WHIRLPOOL-core data flow.

linear 8bit LUT (lookup table) with 256 entries to each 8bit block of the 512bit data representing each element of the (8×8) matrix. Alternatively, it can be implemented using three different 4bit LUTs as well [10] which is a common practice for hardware implementations where 4bit LUTs can be used as primitives. For this implementation targeting a Spartan3 FPGA, we chose this alternative version to reduce utilization of device resources. Considering a total of 256 substitutions per FPGA, tests showed that a full LUT implementation requires more than three times as many slices as the combined 4bit variant. The disadvantage of the combined 4bit LUT approach is a longer data path resulting in a higher latency (lower clock speed [14]). However, it turns out that the substitution is not the bottleneck. Hence, we do not expect drawbacks in runtime. The design still meets the timing requirements implied by the RIVYERA-API which is clocked with 50MHz.

The *cyclical permutation layer* is a shift operation on each of the columns in the (8×8) matrix, resulting in a simple rewiring in hardware. We decided against merging this layer into the next step to deepen the pipeline up to a multiple of two and make the routing easier by adding a register to the data-path.

The *linear diffusion layer* is a matrix multiplication with the generator matrix containing only multiplication factors 1, 2, 4, 5, 8 and 9. Due to the Galois Field used for this arithmetics, the addition of two polynomials can be done by eight XOR-gates. The multiplication by 2 can be achieved by shifting and the use of three XOR-gates, yielding in 25 XOR-gates for the computation of all required factors [11]. As the other layers described above, this layer has been implemented to perform all 256 calculations on this FPGA in only one clock cycle.

The *key addition layer* states the addition of a round key constant for the key inside the key schedule as well as the addition of the computed round key for the message. Since the round key constants have values unequal to zero only

in the first 64 bits, we used a 64bit LUT for the constants along with 64 XOR-gates and 512 XOR-gates to add the round key to the current intermediate block of the message.

As described previously, the input message block and previous hash block is required after the compression function of WHIRLPOOL. Those values are kept in a 512bit wide block RAM since the utilization of distributed RAM would require more device resources than available.

B. Decryption

As mentioned above, for the XTS mode, we require the SERPENT decryption as well as the encryption. However, encryption and decryption are very similar. Mainly, the inverse functions of the S-box and linear transformation are used for the decryption in comparison to the encryption. SERPENT has been implemented with very low device usage using bitslice operations (motivated by [12]). For the current application we implemented a SERPENT core that has fairly low resource usage but still performs the decryption process faster than the key derivation is able to deliver new keys.

Both, decryption and encryption, have been implemented with a data width of 32bit. We decided against unrolling of SERPENT rounds, since it would require a lot of device resources. The improvement of the speed would have no effect, since the slower PBKDF2 is not able to keep up anyway. Even less ressources were assigned to the XTS mode since in our case, the required polynomial multiplication α^i with $i = 0$ is the identity of the input and hence, we did not need to implement the multiplication.

V. PERFORMANCE EVALUATION AND CONCLUSION

Our design described above has been implemented using the Xilinx ISE 13.2 development software tool. The device utilization of the targeted Spartan3-5000 FPGA is about 90% slices and 52% block RAM including RIVYERA communication API regarding the available slices. The core frequency has been set to 50MHz.

We compared our design to a Crypto++ v5.6.1 [4] based software implementation of the dictionary attack, compiled with the GNU g++ v4.1.2 compiler for a Linux operating system including SSE2 optimizations. The software performance was measured on an Intel Core i7-970 at 3.2GHz, taking the advantages of all six CPU cores (12 threads with Hyper-Threading).

The measured speed and the energy consumption of the PC compared to a fully utilized RIVYERA S3-5000 and a single Spartan3-5000 FPGA of the RIVYERA is listed in Tab. I. With a performance of 203,000 passwords per second on the RIVYERA S3-5000, a speedup of 247 is reached, compared to the PC implementation. This shows, even a single FPGA of the RIVYERA S3-5000 is capable to outperform a current PC system, resulting in a fully equipped RIVYERA S3-5000 outperforming a PC cluster with 247 nodes.

Regarding energy consumption, RIVYERA requires only 590W while our test system consumes already 225W (measured with a customary power measurement device). This implies the power consumption of a PC cluster of 247 nodes with nearly the same performance as RIVYERA would be more than 55,575W, which is an increasement of 94 times, leading to energy savings with RIVYERA to about 99%.

Concluded, if it is known that a TRUECRYPT container has been encrypted using SERPENT and WHIRLPOOL hash, and the user has chosen a password based on a dictionary entry, this password can be easily retrieved with RIVYERA in reasonable time. In further research we are addressing the problem that the encryption method used for the attacked container is generally unknown. We plan to implement the remaining TRUECRYPT encryption and hashing methods as well (i.e. AES, TWOFISH, RIPEMD-160, and SHA-512), such that a dictionary attack can be performed on all provided encryption methods and combinations. It is also possible to extend an existing dictionary or create a new one from scratch by an intelligent password generator. This could be done on-the-fly on the host PC, which is responsible for the password distribution on the RIVYERA.

Furthermore, with a portation of this and future designs to the new Spartan6-based RIVYERA S6-LX150, we expect another speedup factor of at least four compared to our current implementation.

REFERENCES

[1] "TrueCrypt – Free Open-Source On-the-fly Encryption," http://www.truecrypt.org.

[2] S. Balogh and M. Pondelik, "Capturing encryption keys for digital analysis ," in *Intelligent Data Acquisition and Advanced Computing Systems(IDAACS*. IEEE, 2011, pp. 759–763.

[3] C. Hargreaves and H. Chivers, "Recovery of Encryption Keys from Memory Using a Linear Scan," in *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1369–1376. [Online]. Available: http://dl.acm.org/citation.cfm?id=1371602.1371819

[4] "Crypto++ ," http://www.cryptopp.com/.

[5] WinAbility Software Corp., "unprotect.info," http://unprotect.info.

[6] AccessData, "DNA – Distributed Network Attack," http://www.accessdata.com.

[7] "TrueCrack ," http://code.google.com/p/truecrack/.

[8] "SciEngines GmbH," http://www.sciengines.com.

[9] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A proposal for the advanced encryption standard."

Table I
COMPARISON OF PERFORMANCE AND ENERGY CONSUMPTION.

Target Architecture	Passwords p. second	Speedup (vs. PC)	Energy consumption	Energy per 10^8 passwds
RIVYERA S3-5000	203,000	247	590W	0.08kWh
Spartan3-5000	1,585	1.93	-	-
Core i7 970 @ 3.2GHz (6 cores / 12 threads)	820	1	225W	7.62kWh

[10] P. S. Barreto and V. Rijmen, "The WHIRLPOOL Hashing Function," in *Submitted to NESSIE, September 2000. Revised May 2003*, 2003.

[11] T. Alho, P. Hämäläinen, M. Hännikäinen, and T. D. Hämäläinen, "Compact hardware design of whirlpool hashing core," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '07. San Jose, CA, USA: EDA Consortium, 2007, pp. 1247–1252.

[12] R. J. Anderson, E. Biham, and L. R. Knudsen, "Serpent and Smartcards." in *CARDIS*, ser. Lecture Notes in Computer Science, J.-J. Quisquater and B. Schneier, Eds., vol. 1820. Springer, 1998, pp. 246–253.

[13] A. J. Elbirt and C. Paar, "An FPGA implementation and performance evaluation of the Serpent block cipher," in *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, ser. FPGA '00. New York, NY, USA: ACM, 2000, pp. 33–40.

[14] M. McLoone, C. McIvor, and A. Savage, "High-speed hardware architectures of the whirlpool hash function." in *FPT*, G. J. Brebner, S. Chakraborty, and W.-F. Wong, Eds. IEEE, 2005, pp. 147–162.

[15] M. Dworkin, *The XTS-AES Mode for Confidentiality on Storage Devices, NIST Special Publication 800-3E*, National Institute of Standards and Technology, Jan. 2010.

[16] RSA Data Security Inc., "RSA Laboratories, PKCS #5 v2.0: Password-Based Cryptography Standard," 1999, available at: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf.

[17] NIST, *The Keyed-Hash Message Authentication Code (HMAC) (FIPS PUB 198-1)*, National Institute of Standards and Technology, Jul. 2008.

[18] ——, *Advanced Encryption Standard (AES) (FIPS PUB 197)*, National Institute of Standards and Technology, Nov. 2001.

[19] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Twofish: A 128-Bit Block Cipher," in *in First Advanced Encryption Standard (AES) Conference*, 1998.

[20] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A Strengthened Version of RIPEMD," in *Proceedings of the Third International Workshop on Fast Software Encryption*. London, UK: Springer-Verlag, 1996, pp. 71–82. [Online]. Available: http://dl.acm.org/citation.cfm?id=647931.740583

[21] NIST, "Secure Hash Standard, FIPS 180-2," 2002.

[22] ISO, *ISO/IEC 10118-3:2004: Information technology – Security techniques – Hash-functions – Part 3: Dedicated hash-functions*. International Organization for Standardization, Feb. 2004.

[23] "NESSIE. New European Schemes for Signature, Integrity and Encryption. IST-1999-12324 ," https://www.cosic.esat.kuleuven.be/nessie/.

[24] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. CRC Press, Inc., 1997.

[25] IEEE, *Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices(IEEE Std 1619-2007)*, Institute of Electrical and Electronics Engineers, Inc., Dec. 2007.

[26] P. Rogaway, "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC," 2003.

[27] M. Schimmler, L. Wienbrandt, T. Güneysu, and J. Bissel, "COPACOBANA: A Massively Parallel FPGA-Based Computer Architecture," in *Bioinformatics – High Performance Parallel Computer Architectures*, B. Schmidt, Ed. CRC Press, Jul. 2010, pp. 223–262.

[28] L. Wienbrandt, S. Baumgart, J. Bissel, F. Schatz, and M. Schimmler, "Massively parallel FPGA-based implementation of BLASTp with the two-hit method," in *ICCS2011, Procedia Computer Science*, vol. 1, 2011, pp. 1967–1976.

[29] L. Wienbrandt, S. Baumgart, J. Bissel, C. M. Y. Yeo, and M. Schimmler, "Using the reconfigurable massively parallel architecture COPACOBANA 5000 for applications in bioinformatics," in *ICCS2010, Procedia Computer Science*, vol. 1, 2010, pp. 1027–1034.

[30] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp, "Cryptanalysis with COPACOBANA," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1498–1513, Nov. 2008.

[31] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, and M. Schimmler, "How to Break DES for €8,980," in *SHARCS2006, Cologne, Germany*, 2006.

[32] T. Gendrullis, M. Novotný, and A. Rupp, "A Real-World Attack Breaking A5/1 within Hours," *Lecture Notes In Computer Science*, vol. 5154, pp. 266–282, 2008.

[33] F. I. Processing, P. J. Bond, U. Secretary, A. L. Bement, and W. M. Director, "Fips pub 198," Tech. Rep., 1990.