Attacking Atmel's CryptoMemory EEPROM with Special-Purpose Hardware

Alexander Wild, Tim Güneysu, and Amir Moradi

Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{alexander.wild, tim.gueneysu, amir.moradi}@rub.de

Abstract. Atmel's CryptoMemory devices are non-volatile memories with cryptographically secured access control. Recently, the authentication mechanism of these devices have been shown to be severely vulnerable. More precisely, to recover the secret key the published attack requires only two to six days of computation on a cluster involving 200 CPU cores. In this work, we identified and applied theoretical improvements to this attack and mapped it to a reconfigurable computing cluster, known as RIVYERA. Our solution provides significantly higher performance exceeding the previous implementation by a factor of 7.27, revealing the secret key obtained from the internal state in 0.55 days on average using only 30 authentication frames.

1 Introduction

In 2002 Atmel introduced a secure memory device with authentication called CryptoMemory [2,13] which is basically an Electrically Erasable Programmable Read-Only Memory (EEPROM) augmented with a secure access control unit.

Due to the low cost and simplicity of deployment the device is employed in a wide range of commercial products, e.g., as key storage of the HDCP system in NVIDIA's graphic cards [16], Labgear's digital satellite receivers [15], Microsoft's Zune Player [7] and SanDisk's Sansa Connect [9] using the CryptoMemory as part of their DRM system implementation. Further examples of CryptoMemory deployment are printer and printer cartridge manufacturers like Dell, Ricoh, Xerox, and Samsung [14]. Furthermore, Atmel's CryptoMemory is placed in authentication tokens from Digitrade [6] and Datakey Electronics [1].

The specification of the Atmel cipher was kept secret till ACM CCS 2010 where Garcia et al. presented their findings obtained from reverse engineering [8]. They also showed significant weaknesses by analyzing the authentication protocol. One year later Biryukov et al. published a more efficient method which – with a probability of 50% – is capable of extracting the secret key from 30 authentication recordings [4]. This attack runs on a computing cluster with 200 Central Processing Unit (CPU) cores and needs two to six days to recover the secret. Another attack based on power side-channels has also been reported in [3]. This attack lasting a few minutes, however, needs physical access to the device and a special side-channel measurement setup to extract the secret key from about 100 power traces.

Our contribution: In this work, we improve and map the best known cryptanalytic attack on CryptoMemory devices published in [4] to special-purpose hardware, namely the RIVYERA S3-5000 reconfigurable computing cluster [17]. Our improvement of the cryptanalytic setup in addition to our hardware-based implementation leads to a speedup factor of 7.27 compared to the previously reported results. In short, our implementation is able to extract the internal state of the cipher from 30 authentication frames within 0.55 days on average which impressively demonstrates that none of the products mentioned above can be considered as secure. Given a cluster such as RIVYERA, our attack configuration is also a power-efficient solution. For a run with 30 frames, the hardware cluster consumes 8.6KWh instead of 245.76KWh the CPU cluster per attacked device.

Outline: In Section 2 we provide preliminary information on CryptoMemory, previously published attacks and RIVYERA. Improvements of the attack are presented in Section 3. Section 4 and 5 deals with our implementation architectures before we compare our results in Section 6 with those of a CPU-based implementation. Finally, our conclusions are given by Section 7.

2 Background

In this section we briefly restate the required background of our work. The section includes specification of the targeted cipher, the underlying protocol, the attack of [4], and our computing cluster.

2.1 CryptoMemory Stream Cipher

The cipher state consists of four shift registers - the left, middle, right and feedback register.

Definition 1. *The state* $S = (l, m, r, f)$ *is an element of* $\mathbb{F}_2^{117}$ *and consists of:*

$$\begin{aligned}
&\textit{Left Register:} && l = (l_0, l_1 \ldots, l_6) \in (\mathbb{F}_2^5)^7 \\
&\textit{Middle Register:} && m = (m_0, m_1 \ldots, m_6) \in (\mathbb{F}_2^7)^7 \\
&\textit{Right Register:} && r = (r_0, r_1 \ldots, r_4) \in (\mathbb{F}_2^5)^5 \\
&\textit{Feedback Register:} && f = (f_0, f_1) \in (\mathbb{F}_2^4)^2
\end{aligned}$$

For every cipher tick, the input $a \in \mathbb{F}_2^8$ is processed during the transition of S to the successor state S'. The state transition is executed in three steps. First, the input values a and f are merged and XORed to several bits in the l, m, and r registers. Second, the left, middle, and right registers are shifted to the right and their feedback value is calculated with the help of a bitwise left rotation and modular addition. Third, the f register is shifted to the left and the new

calculated cipher output nibble becomes the new first element of f. Figure 1 provides an overview of the cipher operation. The core operations of the cipher are given by the following definitions:

Definition 2. *The bitwise left rotation operator $L : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is defined by:*

$$L(x_0 x_1 \dots x_{n-1}) = (x_1 \dots x_{n-1} x_0)$$

.

Definition 3. *Let $\oplus$ be a bitwise XOR operator $\mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2^n$.*

Definition 4. *The modular addition operator $\boxplus : \mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2^n$ is defined as:*

$$x \boxplus y = \begin{cases} x + y(\mod 2^n - 1) & \text{if } x = y = 0 \text{ or } x + y \neq 0\ (\mod 2^n - 1) \\ 2^n - 1 & \text{otherwise} \end{cases}$$

Definition 5. *Let a and b be defined as: $a \in \mathbb{F}_2^8$ and $b = a \oplus f_0 f_1$. Further, the successor state $S' = (l', m', r', f')$ is defined as follows:*

$$\begin{array}{lll} l'_0 := l_3 \boxplus L(l_6), & l'_3 := l_2 \oplus b_3 b_4 b_5 b_6 b_7, & l'_{i+1} := l_i \quad i \in \{0,1,3,4,5\} \\ m'_0 := m_5 \boxplus L(m_6), & m'_5 := m_4 \oplus b_4 b_5 b_6 b_7 b_0 b_1 b_2, & m'_{j+1} := m_j \quad j \in \{0,1,2,3,5\} \\ r'_0 := r_2 \boxplus r_4, & r'_2 := r_1 \oplus b_0 b_1 b_2 b_3 b_4, & r'_{k+1} := r_k \quad k \in \{0,2,3\} \\ f'_0 := f_1, & f'_1 := output(S') & \end{array}$$

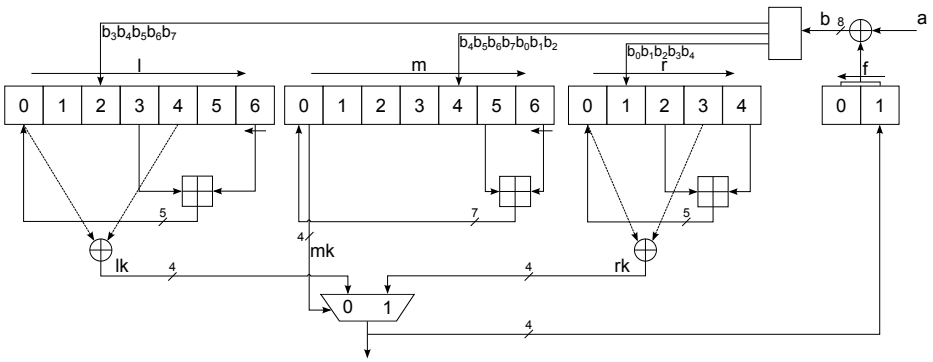

Fig. 1. The CryptoMemory keystream generator [4].

Definition 6. *The cipher output function is defined as follows while i is a bit-selector:*

$$output(S')_i = \begin{cases} lk_i = (l'_0 \oplus l'_4)_{i+1}, & \text{if } m'_{0,i+3} = 0 \\ rk_i = (r'_0 \oplus r'_3)_{i+1}, & \text{if } m'_{0,i+3} = 1 \end{cases} \quad i \in \{0, \dots, 3\}$$

Definition 7. *Let suc be the state transition function with input a, S and output $S' = suc(a, S)$. Further, $suc^n(a, S)$ is defined as multiple application of suc transforming S into its n-th successor state.*

$$suc^1(a, S) := suc(a, S)$$
$$suc^n(a, S) := suc^{n-1}(a, suc(a, S)) \; for \; n > 1$$

2.2 Mutual Authentication Protocol

Apart from the authentication between reader and memory, the authentication protocol is used to initialize the CryptoMemory device and requires the exchange of three messages. The first one contains a nonce $nt \in (\mathbb{F}_2^8)^8$ sent from the memory to the reader. The second message consists of another nonce $nr \in (\mathbb{F}_2^8)^8$ and a calculated authenticator $ar \in (\mathbb{F}_2^4)^{16}$ sent from reader to the memory device. As the last message, the memory device calculates its authenticator $at \in (\mathbb{F}_2^4)^{16}$ and sends it to the reader. Figure 2 depicts the protocol from which the resulting tuple (nr, nt, ar, and at) is defined as an authentication frame.

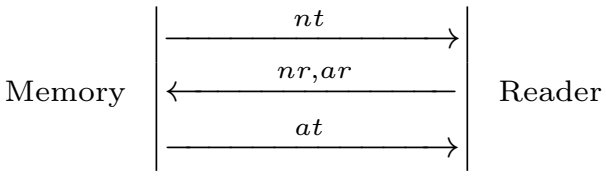

Fig. 2. The authentication protocol [4].

The authenticators ar and at are made by concatenating output nibbles of specific cipher states.

$$S_0 := 0$$
$$S_{i+1} := \mathrm{suc}(nr_i, \mathrm{suc}^3(nt_{2i+1}, \mathrm{suc}^3(nt_{2i}, S_i))) \qquad i \in \{0, \ldots, 3\}$$
$$S_{i+5} := \mathrm{suc}(nr_{i+4}, \mathrm{suc}^3(k_{2i+1}, \mathrm{suc}^3(k_{2i}, S_{i+4}))) \quad i \in \{0, \ldots, 3\}$$
$$S_9 := \mathrm{suc}^5(0, S_8), \quad S_{10} := \mathrm{suc}(0, S_9), \quad S_i := \mathrm{suc}^6(0, S_{i-1}) \quad i \in \{11, 13, \ldots, 23\}$$
$$S_i := \mathrm{suc}(0, S_{i-1}) \quad i \in \{12, 14, \ldots, 24\}, \quad S_i := \mathrm{suc}(0, S_{i-1}) \quad i \in \{25, 26, \ldots, 38\}$$
$$ar_i := \mathrm{output}(S_{i+9}) \quad i \in \{0, 1, \ldots, 15\}$$
$$at_0 := 15, \quad at_1 := 15, \quad at_i := \mathrm{output}(S_{i+23}) \quad i \in \{2, 3, \ldots, 15\}$$

2.3 A Probabilistic Attack on CryptoMemory

This section introduces the published attack by Biryukov et al. [4] which requires knowledge of some eavesdropped authentication frames to reconstruct the cipher state S_8 and S_4. Then a meet-in-the-middle attack is applied to extract the secret key k from the cipher states. The reconstruction of S_8 is the most computationally intensive part and is based on three phases. We start with the generation of state candidates for the right register r. Based on these given candidates we similarly obtain state hypotheses for the left register l. As the third step, we finally compute candidates matching the middle register m for each given left-right register tuple.

Right Register. The attack is based on an exhaustive search for all possible S_{24} states of the right register r and uses a correlation test to filter invalid guesses. This is performed by guessing r, calculating the register output for 16 consecutive ticks and counting the equivalent bits to ar_{14}, ar_{15}, at_i for $i = 2, \ldots 15$. If the sum of coincident bits is below a certain threshold T_r, the candidate is discarded. These steps are repeated until all candidates for r are checked.

Left Register. The next step is to recover the left register and is repeated for each remaining candidate of r. First, lk_i is defined as the intermediate output of l and $l_{S_0} = \{l_0, l_1, l_2, l_3, l_4, l_5, l_6\}$ as the starting point for the calculation. In fact, some bits of lk are known - in particular the cipher output bits that cannot be created by the right register.

Combining the state update function and the state output function, this results in the following equations for the first six output nibbles of the left register:

$$lk_1 = (l_3 \boxplus L(l_6)) \oplus l_3, \quad lk_2 = (l_2 \boxplus L(l_5)) \oplus l_2, \quad lk_3 = (l_1 \boxplus L(l_4)) \oplus l_1,$$
$$lk_4 = (l_0 \boxplus L(l_3)) \oplus l_0, \quad lk_5 = (l_7 \boxplus L(l_2)) \oplus l_7, \quad lk_6 = (l_8 \boxplus L(l_1)) \oplus l_8.$$

The equations above show that lk_i depends only on two variables. Based on these equations the following sets are defined which hold tuples of l_i and l_j.

$$H_0 = \{l_3, l_6\}, \quad H_1 = \{l_2, l_5\}, \quad H_2 = \{l_1, l_4\}, \quad H_3 = \{l_0, l_3\},$$
$$H_4 = \{l_7, l_2\}, \quad H_5 = \{l_8, l_1\}, \quad H_6 = \{l_9, l_0\}.$$

With the known parts of lk_i, H_i can be reduced to only those that match the aforementioned equations. The cardinality of each set strongly depends on the number of known bits in lk_i. Let $N_H(lk_i)$ be the number of known bits in lk_i. Note that register cells are used in more than one set. For example, l_3 is part of H_0 and H_3.

$$H_{0,3} = H_0 \cap H_3 = \{l_3\}, \quad H_{1,4} = H_1 \cap H_4 = \{l_2\},$$
$$H_{2,5} = H_2 \cap H_5 = \{l_1\}, \quad H_{3,6} = H_3 \cap H_6 = \{l_0\}$$

H_i and H_{i+3} can be further reduced by keeping only tuples that consist in l_{3-i} and create the intersection set $H_{i,i+3}$. Additionally, H_0, H_3, and H_6 are combined to $H_{0,3,6} = \{l_0, l_3, l_6\}$ to do a similar reduction by keeping only possible intersection values.

A yet unresolved problem is to choose a good starting point S_0 to maximize the reduction effect. A solution to this problem $\Psi(i)$ can be obtained from

$$\Psi(i) = \sum_{j \in \{1,3,4,8\}} N_H(lk_{i+j}) \quad \text{for } 1 \leq i \leq 7.$$

This function considers the reduction effect on $A = \{l_0, l_1, l_3, l_4, l_6\}$ of a chosen starting state S_0. Let $J = \arg\max_{1 \leq i \leq 7} \Psi(i)$; then, the optimal starting point is S_{24+J} to have the maximum reduction effect on A.

Theorem 1. *If A is defined as $A = \{l_0, l_1, l_3, l_4, l_6\}$ then $\{lk_0, lk_1, lk_3, lk_4, lk_7, lk_8, lk_{11}, lk_{15}\}$ depend only on A, $\{lk_{-1}\}$ on $\{l_5\}$ and A, and $\{lk_5, lk_{12}\}$ on A and l_2 for any chosen starting point l_{S_0}.*

Theorem 1 points out that some register cells exist with more impact on the output stream than others. Hence, the best starting point is the one with the most known bits in A. The proof of Theorem 1 can be found in [5]. Note that $\Psi(i)$ is defined over $1 \leq i \leq 7$ and if $J = 7$, only lk_i up to $i = 8$ can be used for the reduction. Due to the fact that J can be at minimum 1, the (lk_{-1}, lk_5) tuple can be combined with H_1 similar to the intersection set $H_{0,3,6}$ for further reduction.

After the reduction steps, the remaining set H_1 and A are combined to reconstruct all possible internal states S_{24+J} of the left register. In order to cover all lk_i the created candidates are clocked forward and finally backward from l_{S_0} to the original state S_{24}. Keep in mind for this step that l_0 and l_1 has to be XORed with their corresponding feedback byte to get the original values. For further reduction all restored candidates are filtered with the same correlation test as that of the right register but using T_l as the chosen threshold.

Middle Register. The most time-consuming part is the recovery of the middle register that we mapped to hardware as explained in Section 5. Assume that possible candidate pairs for the left and the right register have been generated according to the two steps expressed before. These candidates represent the state S_{24}. The following steps are then performed for each candidate pair.

Let mk_i be the output bits of the middle register. Some bits of mk_i can be restored with the help of rk_i and lk_i. Note that mk_i are the four right most bits of m_j. Due to that, information about m_0, m_7, m_8, ..., m_{21} is extracted. In order to use all gathered information about the middle register cells the attack starts from the state S_{30}.

Depending on the output and update function the following equations are extracted:

$$mk_7 = m_7 \boxplus L(m_0), \qquad mk_i = m_i \boxplus L(m_{i-1}) \quad i \in \{8, \ldots, 15\}.$$

Similar to the reconstruction of the left register, all possible tuples for the middle register cells of state S_{30} are grouped together to form the sets Q_i. For example, Q_0 holds all tuples of m_7 and m_0 that are able to create mk_7. In comparison to the reconstruction of the left register, there are some known bits of m_i that do not depend on other register cells. The cardinality of Q_i also correlates with the number of known bits in mk_{i+7}, m_{i+7}, and m_{i+6}.

$$Q_0 = \{m_7, m_0\}, \quad Q_1 = \{m_8, m_7\}, \quad Q_2 = \{m_9, m_8\}, \quad Q_3 = \{m_{10}, m_9\},$$
$$Q_4 = \{m_{11}, m_{10}\}, \quad Q_5 = \{m_{12}, m_{11}\}, \quad Q_6 = \{m_{13}, m_{12}\}, \quad Q_7 = \{m_{14}, m_{13}\},$$
$$Q_8 = \{m_{15}, m_{14}\}$$

Figure 3 shows which information is used by each Q_i. It is shown that all gained information is used in the sets and which relations they have. In the following it is explained how these relations are used to minimize the number of possible S_{30} candidates.

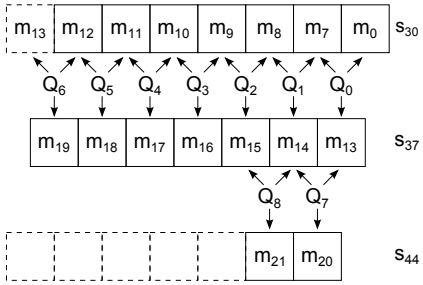

Fig. 3. This diagram shows in which data set which information is processed, structured by the candidate state of the middle register.

Q_i and Q_{i+1} can be shrunk by keeping only tuples that exist in the intersection set $Q_{i,i+1}$ with the same pattern value m_j. To maximize the reduction effect a good starting point needs to be chosen again. Let $I = \arg\min_{0 \le i \le 8} |Q_i|$ where $|Q_i|$ is the cardinality of Q_i. Then, the reduction process is started from $Q_I = \{m_j, m_k\}$ with $k = 0$ or $k = j - 1$. In other words, each Q_i is compared with Q_{i+1} for $I \le i \le 7$. Q_i is also compared with Q_{i-1} for $1 \le i \le I$.

$$Q_{0,1} = Q_0 \cap Q_1 = \{m_7\}, \quad Q_{1,2} = Q_1 \cap Q_2 = \{m_8\}, \quad Q_{2,3} = Q_2 \cap Q_3 = \{m_9\},$$
$$Q_{3,4} = Q_3 \cap Q_4 = \{m_{10}\}, \quad Q_{4,5} = Q_4 \cap Q_5 = \{m_{11}\}, \quad Q_{5,6} = Q_5 \cap Q_6 = \{m_{12}\},$$
$$Q_{6,7} = Q_6 \cap Q_7 = \{m_{13}\}, \quad Q_{7,8} = Q_7 \cap Q_8 = \{m_{14}\}$$

Now, the reduced Q_i sets are combined to fill the middle register cells six downto three of state S_{30}. This partially filled register is checked immediately by

$$mk_{14} = (m_{14} \oplus b_{S_{36}}) \boxplus L(m_{13} \oplus b_{S_{35}}) = (m_8 \boxplus L(m_7)) \boxplus L(m_7 \boxplus L(m_0)),$$
$$mk_{15} = (m_{15} \oplus b_{S_{37}}) \boxplus L(m_{14} \oplus b_{S_{36}}) = (m_9 \boxplus L(m_8)) \boxplus L(m_8 \boxplus L(m_7)).$$

This verification is performed by calculating m_{13}, m_{14} and m_{15} from m_0, m_7, m_8, and m_9. Then the new calculated values are XORed with their feedback byte and it is checked if the tuple (m_{14}, m_{13}) is included in Q_7 and if the tuple (m_{15}, m_{14}) is a part of Q_8. If this is not the case, the partial candidate is discarded and register cells 3 to 6 are filled with the next combination. Otherwise, cell 2 to 0 are filled from Q_3, Q_4, and Q_5 in the same way like cell 3 and 4 to complete the register candidate of state S_{30}. In order to get full cipher candidates of the same state the middle register is clocked backwards to state S_{24}.

As the final step of the state recovering process, the complete internal state $S_{24} = (l, m, r, f)$ is clocked backwards to state S_8 and the corresponding output is compared with ar_{13} to ar_0. This final step usually filters all invalid candidates. A correct state S_8 of a frame only persists if it was previously not discarded by the correlation tests performed on the right and left register candidates.

2.4 RIVYERA Special-Purpose Hardware Cluster

In this work we employ the reconfigurable RIVYERA computing cluster system which is specially designed to process cryptanalytic tasks. The Redesign of the Incredibly Versatile Yet Energy-efficient, Reconfigurable Architecture (RIVYERA) cluster is populated with 128 Spartan-3 XC3S5000 Field Programmable Gate Arrays (FPGAs) distributed over 16 card modules. The modules are plugged into a backplane that provides a systolic ring bus interconnect for high-performance communication. Additionally, a host PC is attached to the ring bus via PCI Express and both systems are installed in a 19" rackmount system [10, 11, 18, 19].

3 Advanced Candidate Filtering

The attack described previously creates candidates for each register sequentially. The candidates for the left register are chosen from the output stream of a right register candidate, and the middle register candidates are based on the output stream of a left and a right register candidate. The output function of the left and the right register is a simple XOR. The XOR operation of the binary complement $\overline{x}_0$ and $\overline{x}_1$ of an arbitrary x_0 and x_1 results in the same output y.

The update function of both register acts as following:

$$l_3 \boxplus L(l_6) = l_0 \qquad r_2 \boxplus r_4 = r_0 \tag{1}$$
$$\bar{l}_3 \boxplus L(\bar{l}_6) = \bar{l}_0 \text{ when } l_3 \neq L(\bar{l}_6) \qquad \overline{r}_2 \boxplus \overline{r}_4 = \overline{r}_0 \text{ when } r_2 \neq \overline{r}_4 \tag{2}$$

The probability that the condition in Equation (2) is not given for a register is $\frac{1}{32}$. To create the output stream the right register candidate is clocked 16 times.

So the probability that the condition is not met during this time is $(1 - \frac{1}{32})^{16} = 0.6017$. Summarizing the previous facts leads to a 60% chance that r as well as $\overline{r}$ produce the same output stream. In case r passes the correlation test, $\overline{r}$ passes the correlation test as well. This behavior also occurs for left register candidates. Due to the fact that the left register candidates are only based on the cipher output stream and the right register output stream, r and $\overline{r}$ produce the same left register candidate list. The middle register candidates are also based only on the output streams lk and rk which means that the tuples (l, r), $(\overline{l}, r)$, $(l, \overline{r})$ and $(\overline{l}, \overline{r})$ produces the same middle register candidate list, when the conditions in Equation (2) are satisfied during register output generation.

The attack performs inverted cipher ticks for a register candidate triple (l, m, r) and checks whether it matches to the known ar_i nibbles. For an inverted cipher tick a modular subtraction is necessary which is defined as follows:

Definition 8. *The modular subtraction operator* $\boxminus : \mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2^n$ *is defined as:*

$$x \boxminus y = \begin{cases} x - y(\mod 2^n - 1) & \text{if } x \neq y \\ 2^n - 1 \text{ or } 0 & \text{otherwise} \end{cases}$$

Note that the modular subtraction is non-injective. In case of $x = y$, the result of the modular subtraction can be 0 or $2^n - 1$. The attack, should consider both cases; in the later steps the wrong guess will be filtered out when not matching with ar_i.

For the modulo subtraction we observe a similar behavior as for the modulo addition. The condition in Equation (3) is due to the non-injectivity of the operator.

$$\begin{aligned} l_0 \boxminus l_4 &= L(l_6) & r_0 \boxminus r_3 &= r_4 \\ \overline{l}_0 \boxminus \overline{l}_4 &= L(\overline{l}_6) \text{ when } l_0 \neq l_4 & \overline{r}_0 \boxminus \overline{r}_3 &= \overline{r}_4 \text{ when } r_0 \neq r_3 \end{aligned} \tag{3}$$

Summarizing all these facts leads to the following conclusion. The attack performs inverse cipher ticks for the triple (r, m, l) to check its consistency with the known ar_i nibbles. If r, $\overline{r}$ and l, $\overline{l}$ exist in the list of candidates, we validate the triples $(l, m, \overline{r})$, $(\overline{l}, m, r)$, $(\overline{l}, m, \overline{r})$ at the same time. Also, if (l, m, r) is not the correct internal state, $(l, m, \overline{r})$, $(\overline{l}, m, r)$, and $(\overline{l}, m, \overline{r})$ will not be the correct one either. Therefore, we can remove $\overline{l}$ and $\overline{r}$ from the list of candidates which generate the same output stream as l and r. For a remaining candidate S_8 the complementary left and right register candidates have to be checked separately if they are feasible as well. Our experiments have shown that with this additional filtering the number of right and left register candidates are reduced to 68%. In total we only process – on average – 46.24% of the original left and right candidate list.

4 Mapping Components to Hardware

Most parts of the attack will be executed in software and only the most time-consuming parts are mapped to hardware. In this context, the interfaces between software and hardware are of major importance to allow a smooth transition of data in both directions. As a first step we implement the calculations of the middle register reconstruction process in hardware. The transition from software to hardware at this point requires only a very limited number of data transfers. Moreover, this is indeed the most time-consuming part (about 98.8 % of the attack time).

Attacking m begins with the reconstruction of Q_i and searches for the smallest set. The subsequent reduction on this step compares possible register cell candidates. In our hardware implementation we should merge these two steps so that the generation and reduction of the register cell candidates are performed at once. In order to check whether m_i is part of Q_j and Q_{j+1} it is necessary to check if (a) m_i contains the known bits from the fragmentary middle register output stream and (b) there must be at least one m_{i+1} and one m_{i-1} each of which contains the fragmentary known bits and is not removed. Each of them also must be able to create in conjunction with m_i an arbitrary m_{k+1} and m_k, respectively, that each contains the corresponding known bits. If both conditions are fulfilled, m_i is a valid register cell candidate.

Due to our merging technique we do not know which set is the smallest one. So we always start the generation with Q_8 and continue the calculation iteratively until we have created Q_0. With Q_0 we perform the generation and reduction steps again for all sets from Q_0 to Q_8. During the creation of the sets, either register cell candidates from previously generated sets are used or the candidates are generated with the help of the known mk_i bits as described in Section 5.1.

After generation the valid tuples need to be stored in memory. One problem is that we do not know in advance how many valid tuples we will receive but we have to allocate a fixed amount of memory in hardware. So we assume the worst case memory complexity for the Q_i sets: $2^7 \cdot 2^7 \cdot 14 \cdot 9$ bits. This translates to 126 Block Random-Access Memory (BRAM) blocks with 18kB each, but a Spartan-3 5000 only provides 108 BRAMs. Due to the sequential nature of the reduction, an on-the-fly calculation of the candidates will result in an enormous increase of time. An alternative method is to store the information of valid tuples in relation matrices. In a relation matrix the information of the register cell values is encoded in the position of a special flag which indicates if the register cell combination is valid or not. The usage of relation matrices reduces the memory complexity to: $2^7 \cdot 2^7 \cdot 9$ bits, which needs in total only 9 BRAM blocks to hold the necessary information. This storage method directly leads to the next challenge: the efficient reconstruction of register cell values. Obviously, due to the cell candidate dependencies a bitwise search for each candidate is ineffective.

Finally, the hardware instantiation of an Inverse Cipher Tick (ICT) is not trivial as well. Each modular subtraction for an ICT is non-injective so for some values the result is ambiguous and incorrect values need to be sorted out a few

ICTs later. This backtracking behavior complicates a straightforward hardware implementation using parallelism or pipelining techniques so that we decided to implement multiple iterative ICT modules instead for maximum performance.

5 Implementation

In this section we give an overview of the hardware implementation of the attack including advanced candidate filtering. Each of the 128 FPGAs is configured with the same configuration. The design contains two independent attack cores to which a controller forwards data depending on which module is waiting for a new dataset. Each attack component contains a module to generate Q_i tables that iteratively creates register cell candidates and stores them in the BRAMs. Then, a module reconstructs complete middle register candidates from the previously generated relation matrices (Buffered Pipeline) and distributes the candidates to a free ICT module. The ICT module performs inverse cipher ticks and validates the candidate by examining its compliance with ar_i. Figure 4 depicts the top-level design of our implementation.

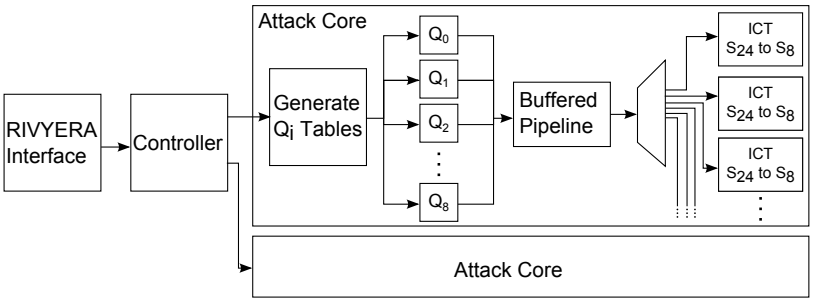

Fig. 4. The top-level design of our hardware implementation.

5.1 Generating Q_i Tables

The Q_i generating module iteratively fills Q_i with valid register cell tuples and starts with Q_8. First, the design uses just a counter and the partial reconstructed output stream to reduce the amount of possible candidates for m_{15}, m_{14} and m_{13}. The possible register cell candidates are then fed into a modular adder that calculates m_{21} and m_{20} which are directly verified by the known bits. Based on the results, a BRAM block for Q_8 is filled with a stream of bits that represents valid tuples for m_{15} and m_{14}. After the calculation of Q_8 is completed, the module continues with the calculation of Q_7. At this point possible previously defined candidates for m_{14} are present in memory and the module repeats these steps until Q_0 is generated. Next it performs the same procedure again in the

reverse direction, i.e., from Q_1 to Q_8 to achieve a maximum effect reducing the number of possible register cell candidates. Figure 5 shows an overview of the structure of the module.

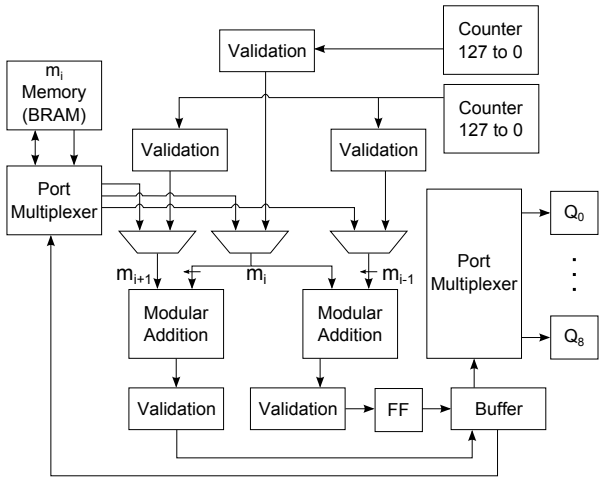

Fig. 5. Design of the Q_i generation module.

5.2 Buffered Pipeline

The goal of this unit is to efficiently extract complete middle register candidates from Q_i within the BRAM memory. In most cases the relation matrices in memory are rarely filled and a challenge is to find the bits set in Q_i and decode their corresponding position one after another.

In order to decode the position of a set bit in a block a priority decoder can be used. However for a large blocksize, e.g., 32-bit, the complexity of the priority decoder grows enormously requiring a lot of resources. To save the resources we filter one set bit out of the block and use a simple decoder to extract the position of this single set bit. The filtering is realized with the following approach:

Let α be a binary block. Instead of using a priority decoder one can calculate $\alpha \wedge (\alpha \oplus (\alpha - 1))$ which contains at most a single one bit and passes this to a binary decoder. This process can be iteratively repeated by replacing α by $\alpha \oplus (\alpha \wedge (\alpha \oplus (\alpha - 1)))$. For clarification an example is given in the following:

$$\begin{aligned}\alpha &= \ldots 101001000\\ \alpha - 1 &= \ldots 101000111\\ \alpha \oplus (\alpha - 1) &= \ldots 000001111\\ \alpha \wedge (\alpha \oplus (\alpha - 1)) &= \ldots 000001000 \text{ (decoder input)}\end{aligned}$$

This technique always filters the right most set bit from an arbitrary binary block. In our implementation this filtering process is repeated until each one bit in a block is appropriately decoded.

5.3 Inverse Cipher Tick

The ICT module is an iterative module which performs inverse cipher ticks until S_8 of a cipher state candidate is reached or a candidate does not generate the known output nibbles ar_i. The module starts with the receipt of an incoming candidate of S_{24} and forwards it to its First In, First Out (FIFO) unit. A candidate coming out of the FIFO is fed into the modular subtractor which calculates the right most cells of the three main state registers. Due to the non-injective property of modular subtraction, the output of the modular subtractor is selected by a special flag (0 by default). A decision unit calculates these flags and ensures that – in case of multiple ambiguous results – all possible combinations are considered. Two final modules update the feedback register and validate with ar_i. In case of a positive result, the newly generated state is fed into the FIFO for the next ICT– until either finally S_8 is reached or validation fails in a later step. Figure 6 depicts a block diagram of the ICT module.

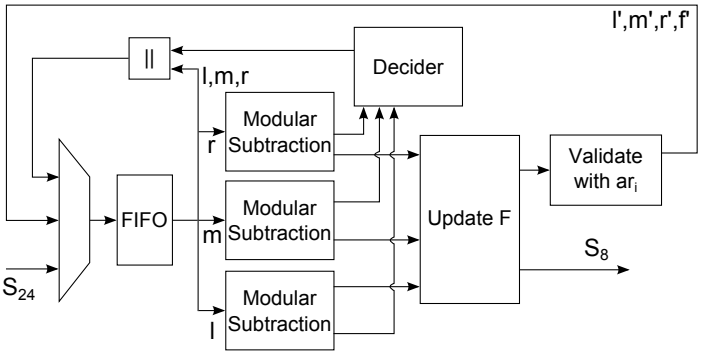

Fig. 6. The internal structure of an ICT module.

6 Results

In this section we present the results of our hardware implementation obtained using Xilinx ISE Foundation 14.3 for synthesis and place and route. The design with two attack cores of which each contains 17 ICT modules is synthesized and runs at the frequency of 100MHz. The utilized resources on each Spartan-3 5000 are shown in Table 1.

Essentially, the attack speed strongly depends on the frequency of operation and the number of attack components and ICT modules per core. The integrated FIFO component of each module has a data width of 125 bits which results in the utilization of 4 BRAM primitives on a Spartan-3. The complete hardware design is limited by BRAM blocks, i.e., a generic design configuration with two attack cores based on BRAM can only instantiate 7 ICT modules per core. For a better resource utilization a dedicated LUT-based version of the ICT modules was generated to instantiate the internal FIFO. This alternative implementation allows us to instantiate 10 additional ICT modules per core. Table 1 shows the resource consumptions of both ICT versions on a Spartan-3 5000.

Table 1. Resource consumption of a dedicated RAM-based ICT BRAM-based ICT and for the complete design (two attack components and 17 ICT) on a Spartan-3 5000 FPGA

Resource	DRAM ICT	BRAM ICT	Complete	Available
Slices	808	482	28.298	33.280
Lookup Tables (LUTs)	1058	469	45.600	66.560
Slice Registers	460	317	28.199	66.560
BRAM	0	4	103	104

Next we compare the throughput of the CPU-based cluster implementation in [4] with our hardware implementation on RIVYERA. Note that exact cycle counts are not available for the implementation given in [4]. Therefore we restrict our comparison to the data as shown in Table 2.

In order to determine the attack speed of our solution, we measure the validation time for a left and right register pair. 300 randomly generated frames are chosen to compute the average time needed for register pair validation. On average one attack unit is able to check one left/right register candidate tuple in 0.8 seconds. To have a 50% chance for a successful attack we need 30 frames similarly as stated in [4]. On average 23 right and $2^{19.527}$ left register candidates are generated out of 30 frames what leads to a total running time of the attack to reconstruct the internal state in about 13 hours (0.55 days).

Apart from performance, the cost for running an attack is of utmost importance. The CPU-based attack was run on a rented Amazon Elastic Compute Cloud (EC2) cluster but unfortunately, RIVYERA is not for rent. For a fair comparison, we therefore compare the running costs of the attack by estimating

Table 2. Comparison between the RIVYERA and CPU cluster implementation.

Aspect	**RIVYERA**	**CPU**
Parallelization	128 FPGAs with 2 attack cores and 17 ICT	200 CPU cores
Clock Cycles per ICT step	5	2^7
Clock Frequency [GHz]	0.1	2.26
Candidate Reduction (l/r)	0.4624	1
Total Time [days]	0.55	$\frac{2+6}{2} = 4$
Total improvement factor	$\frac{4}{0.55} = 7.27$	1
Performance equivalency	1 FPGA $\equiv$	11.36 CPUs
Power Consumption per Device [KW]	0.65	2.56
Power Consumption per Attack [KWh]	$0.65 \cdot 13.23 = 8.6$	$2.56 \cdot 96 = 245.76$
Cost Reduction	28.58	1

the power consumption for both attack implementations. The RIVYERA S-3 5000 takes on average $650W$ while two Intel Xeon L5640 CPUs including peripherals approximately demand $(60W \cdot 2) + 40W = 160W$ [12] for the complete system. To run the attack in the given time as stated in [4], at least 16 such computing systems are required. The power consumption in Table 2 shows again the advantage of special-purpose hardware over CPU-based attack clusters.

Despite the performance improvement with the Spartan-3 5000, we can achieve even higher performance with later FPGA devices. In particular, the RIVYERA S6-LX150 which can be equipped with 256 Spartan-6 LX150 offers by far more logic and performance but was not available in the course of this work. However, to provide at least estimates, we adapted our design for the Spartan-6 LX150 on which we can instantiate the double amount of attack cores with 17 ICT modules each. Additionally, we can run the design at double clock frequency due to the newer FPGA technology (200MHz) which results in an additional performance speed-up by factor of four.

7 Conclusion

The hardware implementation presented in this work improves the attack on CryptoMemory devices by Biryukov et al. [4] by introducing an additional candidate filtering step reducing the computation complexity to a half. By mapping the most time consuming parts to FPGA hardware, our solution runs in total 7.27 times faster than the previously reported results using 30 authentication frames. This enables the complete recovery of the secret internal state of the CryptoMemory cipher on average in less than 0.55 days. Finally, our hardware attack is 28.58 times cheaper considering power consumption compared to [4] using a CPU-based cluster.

Acknowledgements. The authors would like to thank Alex Biryukov, Ilya Kizhvatov and Bin Zhang for useful discussions and for their kindness providing parts of their attack script.

References

1. Atmel Corporation. CryptoMemory for Removable Storage Devices and Reprogrammable Keys. `http://www.cryptomemorykey.com/pdfs/AtmelCryptoMemoryFlier.pdf`, as of April 19, 2013.
2. Atmel Corporation. CryptoMemory specification. `http://www.atmel.com/Images/doc5211.pdf`, as of April 19, 2013, 2007.
3. J. Balasch, B. Gierlichs, R. Verdult, L. Batina, and I. Verbauwhede. Power Analysis of Atmel CryptoMemory - Recovering Keys from Secure EEPROMs. In *CT-RSA 2012*, volume 7178 of *LNCS*, pages 19–34. Springer, 2012.
4. A. Biryukov, I. Kizhvatov, and B. Zhang. Cryptanalysis of the Atmel Cipher in SecureMemory, CryptoMemory and CryptoRF. In *ACNS 2011*, volume 6715 of *LNCS*, pages 91–109. Springer, 2011.
5. A. Biryukov, I. Kizhvatov, and B. Zhang. Cryptanalysis of The Atmel Cipher in SecureMemory, CryptoMemory and CryptoRF. *IACR Cryptology ePrint Archive*, page 22, 2011.
6. Digitrade GmbH. `http://www.digittrade.de/shop/index.php/cat/c66_HS256S-High-Security.html`.
7. B. Dipert. The Zune HD: more than an iPod touch wanna-be? EDN, 2009.
8. F. Garcia, P. van Rossum, R. Verdult, and R. Wichers Schreur. Dismantling SecureMemory, CryptoMemory and CryptoRF. In *CCS 2010*, pages 250–259. ACM, 2010.
9. M. Giacomelli. SanDisk Sansa Connect. `http://www.rockbox.org/wiki/SansaConnect`.
10. T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008.
11. T. Güneysu, G. Pfeiffer, C. Paar, and M. Schimmler. Three Years of Evolution: Cryptanalysis with COPACOBANA. In *SHARCS 2009*, pages 9–10, 2009.
12. Intel. Intel Xeon Processor 5600 Series: Product Brief. Available via `http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-5600-brief.pdf`.
13. M. Jarboe. Introduction to CryptoMemory. *Atmel Applications Journal*, 3:28, 2004.
14. S. József. AT88SC0204 ChipResetter. `http://chipreset.atw.hu/6/index61.html`.
15. Labgear. Labgear HDSR300 High Definition Satellite Receiver. User Guide. `http://www.free-instruction-manuals.com/pdf/p4789564.pdf`, as of April 19, 2013.
16. NVIDIA Corporation. Checklist for Building a PC that Plays HD DVD or Blue-ray Movies. `ftp://download.nvidia.com/downloads/pvzone/Checklist_for_Building_a_HDPC.pdf`, as of April 19, 2013.
17. SciEngines GmbH. `http://www.sciengines.com`.
18. Xilinx. Spartan-3 FPGA Family: Complete Data Sheet. *Product Documentation*, November 2005.
19. Xilinx. Spartan-3 Generation FPGA User Guide. *Product Documentation*, June 2011.