The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/83892

Please be advised that this information was generated on 2017-11-10 and may be subject to change.
Dismantling SecureMemory, CryptoMemory and CryptoRF

Flavio D. Garcia  Peter van Rossum  Roel Verdult  Ronny Wichers Schreur

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands.
{flaviog,petervr,rverdult,ronny}@cs.ru.nl

ABSTRACT
The Atmel chip families SecureMemory, CryptoMemory, and CryptoRF use a proprietary stream cipher to guarantee authenticity, confidentiality, and integrity. This paper describes the cipher in detail and points out several weaknesses. One is the fact that the three components of the cipher operate largely independently; another is that the intermediate output generated by two of those components is strongly correlated with the generated keystream. For SecureMemory, a single eavesdropped trace is enough to recover the secret key with probability \(0.57\) in \(2^{39}\) cipher ticks. This is a factor of \(2^{30.5}\) faster than a brute force attack. On a 2 GHz laptop, this takes around 10 minutes. With more traces, the secret key can be recovered with virtual certainty without significant additional cost in time. For CryptoMemory and CryptoRF, if one has 2640 traces it is possible to recover the key in \(2^{52}\) cipher ticks, which is \(2^{19}\) times faster than brute force. On a 50 machine cluster of 2 GHz quad-core machines this would take less than 2 days.

Keywords: stream ciphers, practical cryptanalysis, smartcard security, RFID

1. INTRODUCTION
This paper addresses the (in)security of the cryptographic mechanisms used in the Atmel product families SecureMemory, CryptoMemory, and CryptoRF.

These products are integrated circuits, consisting essentially of a small piece of memory and have cryptographic capabilities to authenticate and encrypt. They have two main application areas: in smart cards (ID and access cards, health care cards, loyalty cards, internet kiosk, energy meters, and e-government); and embedded, to authenticate one piece of hardware to another or for the secure storing of cryptographic data (printers and print cartridges, removable storage devices, set top boxes, access control systems, subassembly authentication, counterfeit protection, networked systems) [BJ04, CGY08, ZJS+09]. A concrete example is the widely sold NVIDIA graphics cards, which uses CryptoMemory as secure storage to protect the HDCP license keys\(^1\), required to play high-definition video\(^2\). It is also used in media players such as Microsoft’s Zune Player [Dip09] and SanDisk’s Sansa Connect\(^3\), for instance to securely store the device id and device certificate.

The Atmel chips AT88SC153 and AT88SC1608, which we call here the SecureMemory family, were introduced in 1999. A newer version, the CryptoMemory family, which includes the AT88SCxxxxC chips, with more advanced cryptographic features, was introduced in early 2002. These two families only have contact interfaces; in late 2003, the CryptoRF family, which is a variant of the CryptoMemory family with the AT88SCxxxxCRF chips, was introduced and also has a RF interface [Jar04].

The technology used in the chips is covered by US Patent 7395435 B2 [BCM08]. Details on the communication protocol used between a card (or slave device) and a reader (or master device) can be found in the documentation [Atm07]; although this does include details on how the cryptographic algorithms are called and on the message flow between card and reader, it does not provide detailed information on the cipher itself. As experience has shown with other chips from the same era, the secrecy of such a proprietary cipher does not inspire confidence in its cryptographic strength. Most notably, KeeLoq, which is mainly used in wireless car keys and the Mifare Classic, which has widespread application in public transport ticketing systems and in access control cards, have been thoroughly broken in the last few years [Bog07, CBW08, IKD+08, GrVRS09, GdKGGM+08, dKGHG08, NESP08, Coul09]. The cipher used in CryptoMemory, however, boasts a security stronger than DES; a security evaluation [Tec05] claims that the observed bias in the output is “trivial enough to confirm that CryptoMemory is impermeable to shortcut attacks”. Admittedly, sophisticated attacks were not in the scope of this evaluation. We should note at this point that a back door has been found in the SecureMemory chips AT88SC153/1608 [Fly08]: using ultra-violet light, an EEPROM fuse can be restored to its original value and the contents of the memory can be read

\(^1\)http://www.expreview.com/review/2007-10-17/1192604816d5951_2.html
\(^2\)http://download.nvidia.com/downloads/pvzone/Checklist_for_Building_a_HDPC.pdf
\(^3\)http://www.rockbox.org/wiki/SansaConnect
Our contribution

As part of our research we have reverse engineered the security mechanisms of the SecureMemory and CryptoMemory families (the CryptoRF family uses exactly the same cipher as the CryptoMemory family, so we ignore that distinction from now on). This includes the ciphers, the authentication and encryption mechanisms, and the password mechanisms. This was accomplished by disassembling various executable applications from Atmel (e.g., ECEStudio), that implement the ciphers in software. We wrote a prototype implementation and then we compared its output with the packages we observed from the cards.

We have implemented the full functionality of these families in software and released this under GPL licence\(^4\). Our implementation can communicate with genuine Atmel products using commercial smartcards and readers.

In this paper, we focus exclusively on the ciphers and the authentication mechanism; that is all that is needed to recover the keys.

SecureMemory and CryptoMemory use a stream cipher. It has an internal state of 109 bits (SecureMemory) or 117 bits (CryptoMemory), organized in three shift registers with a non-linear feedback function. Every clock tick, two of the registers produce 4 bits of output; the output of the third is used to select bits from one of the two other registers, producing 4 bits of keystream. The difference between SecureMemory and CryptoMemory, as far as the cipher is concerned, is that in CryptoMemory the last 8 bits of the output are remembered and are fed back into the shift registers every tick. Details are described in Section 3.

During initialization, 64-bit nonces from reader and card and a 64-bit shared key are used to set the initial state of the cipher; reader and tag subsequently authenticate by exchanging part of the following keystream. See Section 3.1 for details.

Next, we describe passive attacks against SecureMemory and CryptoMemory. We assume that an attacker has managed to eavesdrop a number of authentication sessions. For SecureMemory, we use a combination of a correlation attack (see for example [Rue92]) and a meet-in-the-middle attack to recover a significant part of the internal state of the cipher. Meet-in-the-middle attacks were also proved successful against other stream ciphers like A51 [BBK08] and KeeLoq [IKD+08]. Once we have the internal state of the cipher, we use a meet-in-the-middle attack to recover the shared secret key. This attack costs \(2^{39}\) time, taking ticks of the cipher as unit. On a 2 GHz laptop, the attack takes approximately 10 minutes. Even though this attack is rather straightforward, it serves as an intermediate, didactic step to the more involved attack on CryptoMemory.

\(^4\)Sample code available at [http://www.libnfc.org/documentation/examples/nfc-cryptorf](http://www.libnfc.org/documentation/examples/nfc-cryptorf)

For CryptoMemory, the principal idea behind the attack is the same, but it is much more complicated because of the presence of the feedback of the output into the cipher state. We still use a correlation attack to recover a large part of the internal state (two of the three registers). Unlike with SecureMemory, we do not recover this with certainty, but obtain a few hundred to a few thousand candidates. Because of the feedback, it is also not directly possible to roll back these partial states, so we have to search for the correct remaining part of the internal state (the third register). After this, we can again unroll and use a meet-in-the-middle attack to recover the shared secret key. Using approximately 2640 traces, this attack has a time complexity of \(2^{32}\) cipher ticks; it would take less than 2 days on a 50 machine cluster of 2 GHz quad-core machines. Details on these attacks are described in Section 4 (for SecureMemory) and Section 5 (for CryptoMemory).

We have discussed these vulnerabilities with Atmel in October 2009. To give the manufacturer ample time to take appropriate measures, we have not disclosed them until March 2010.

2. BACKGROUND

SecureMemory and CryptoMemory are ISO/IEC 7816 smartcards that communicate through a contact interface. CryptoRF is a ISO/IEC 14443-B smartcard that uses a contactless interface. CryptoMemory is also available in the form of an embedded EEPROM IC with a serial interface. The available memory ranges from 128 bytes to 32KB. The memory is split in multiple user zones and one system zone which stores the configuration of the smartcard.

![Figure 1: Logical memory structure](http://www.libnfc.org/documentation/examples/nfc-cryptorf)

The system zone is divided into six sections: identification, access control, cryptography, secret and password. Blank cards operate with a user-defined identifier which is customized by the system integrator. To achieve key diversification, the documentation recommends that the authentication keys should be derived from this identifier. The access control section is used to restrict read and write operations per user zone and define the requirements of authentication and encryption. To verify the legitimacy and prevent eavesdropping of data and passwords a mutual authentication with encryption is available.

3. THE CIPHERS

This section describes the stream ciphers that are used in SecureMemory and CryptoMemory. As mentioned in the introduction, the internal state of the cipher consists of three
shift registers; for CryptoMemory, an additional register remembers the last 8 bits of output.

**Definition 3.1.** A cipher state of SecureMemory $s$ is an element of $\mathbb{F}_2^{109}$ consisting of the following components:

1. the left-hand segment $l = (l_0, \ldots, l_6) \in (\mathbb{F}_2^5)^7$;
2. the middle segment $m = (m_0, \ldots, m_6) \in (\mathbb{F}_2^7)^7$;
3. the right-hand segment $r = (r_0, \ldots, r_4) \in (\mathbb{F}_2^5)^5$.

The cipher state of CryptoMemory is an element of $\mathbb{F}_2^{117}$. In addition to the three segments above, it also has the following component:

4. the feedback segment $f = (f_0, f_1) \in (\mathbb{F}_2^4)^2$.

Every tick, a cipher state $s$ together with an input $a \in \mathbb{F}_2^8$ evolves in a successor state $s'$. First, the input $a$ (and in case of CryptoMemory also the feedback segment $f$) are XORed into $s$ at several places, giving an intermediate state $\hat{s}$. Second, the left, right and middle segments are shifted to the right and new 0th entries are computed using a bitwise rotate and a modular addition. Thirdly and finally, the output segment is shifted to the left and a new 1st entry is computed using a non-linear function of the other segments, giving the successor state $s'$. The following sequence of definitions gives the details of this construction; see also Figure 2.

**Definition 3.2.** Define the bitwise left rotate operator $L: \mathbb{F}_2^n \to \mathbb{F}_2^n$ by

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

**Definition 3.3.** Define the modified modular addition operator $\oplus: \mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2^n$ (identifying elements of $\mathbb{F}_2^n$ with elements of $\{0, 1, \ldots, 2^n - 1\}$) by

$$x \oplus y = \begin{cases} x + y \pmod{2^n - 1} & \text{if } x = y = 0 \text{ or } x + y \neq 0 \\ 2^n - 1 & \text{otherwise,} \end{cases}$$

where $+$ (mod $2^n - 1$) denotes integer addition modulo $2^n - 1$.

Note that this operation is not injective when fixing one of the arguments: $x \oplus 0$ and $x \oplus (2^n - 1)$ both equal $x$ (unless $x = 0$, because $0 \oplus 0 = 0$ and $0 \oplus (2^n - 1) = 2^n - 1$). Also observe that the result of $\oplus$ is never 0 unless both operands are 0.

**Definition 3.4.** Let $s = (l, m, r, f) \in \mathbb{F}_2^{117}$ be a cipher state (for CryptoMemory; for SecureMemory, ignore $f$) and let $a \in \mathbb{F}_2^8$ be an input. We define the intermediate state $\hat{s} = (\hat{l}, \hat{m}, \hat{r}, \hat{f})$ and the successor state $s' = (l', m', r', f')$ as follows. For CryptoMemory, define $b := a \oplus f_0 f_1$; for SecureMemory, $b := a$. The intermediate state $\hat{s}$ is given by

$$\hat{l}_0 := \hat{l}_2 \oplus b_3 b_0 b_2 b_6$$
$$\hat{m}_4 := m_4 \oplus b_4 b_2 b_7 b_0 b_1 b_2$$
$$\hat{r}_1 := r_1 \oplus b_0 b_3 b_1 b_4$$

and all other entries are copied from $l$, $m$, $r$, and $f$. The successor state $s'$ is given by

$$l'_{i+1} := \hat{l}_i, \quad i \in \{0, \ldots, 5\}$$
$$m'_{i+1} := \hat{m}_i, \quad i \in \{0, \ldots, 5\}$$
$$r'_{i+1} := \hat{r}_i, \quad i \in \{0, \ldots, 3\}$$
$$l'_{0} := \hat{l}_2 \oplus L(\hat{l}_6)$$
$$m'_{0} := \hat{m}_5 \oplus L(\hat{m}_6)$$
$$r'_{0} := \hat{r}_2 \oplus \hat{r}_4.$$  

We call the rightmost 4 bits of $l_0 \oplus l_4$ the output of the left-hand segment (i.e., $l'_{0,1} \oplus \ldots \oplus l'_{0,4}$, a bitwise XOR) and denote it by output($l'$). We call the rightmost 4 bits of $r_0 \oplus r_4$ the output of the right-hand segment $r'$ (i.e., $r'_{0,1} \oplus \ldots \oplus r'_{0,4}$) and denote it by output($r'$). The output of the cipher state $s'$, denoted by output($s'$) is defined by

$$\text{output}(s')_i = \begin{cases} \text{output}(l')_i, & \text{if } m'_{0,i+3} = 0; \\ \text{output}(r')_i, & \text{if } m'_{0,i+3} = 1 \quad i \in \{0, \ldots, 3\}. \end{cases}$$

Note how the rightmost 4 bits of the middle segment acts as a selector; it selects either a bit from the left-hand segment or a bit from the right-hand segment to be included in the output. For CryptoMemory, we define

$$f'_0 := \hat{f}_1 (= f_1)$$
$$f'_1 := \text{output}(s').$$

Figure 2: The cipher
Define the transition function succ that takes an input a and a state $s$ and outputs the successor state $s'$. We write succ$^n(a, s)$ denoting succ$(a, s)$ when $n = 1$ and succ$^{n-1}(a, \text{succ}(a, s))$ when $n > 1$.

Note that the whole feedback segment can be reconstructed from the other three segments: $f_1 = \text{output}(s)$, but also $f_0$ can be reconstructed: shift all segments in $s$ one to the left and take output. To be precise, we have

$$f_{0,i} = \begin{cases} l_{i,i+1} \oplus l_{i+1,i+1} & \text{if } m_{1,i+3} = 0 \\ r_{1,i+1} \oplus r_{4,i+1} & \text{if } m_{1,i+3} = 1. \end{cases}$$

(2)

(So the rightmost 4 bits of $m_3$ act as a selector between the rightmost 4 bits of $l_1 \oplus l_3$ and $r_1 \oplus r_4$.)

3.1 Initialization and authentication

The cipher is initialized during the authentication protocol. At the beginning of this protocol, tag and reader exchange nonces as depicted in Figure 3. The tag nonce is scrambled together with the first half of the reader nonce and fed into the cipher. Subsequently, the shared key scrambled together with the second half of the reader nonce is fed in. Then, the cipher produces keystream that will be used as authenticator for both reader and tag. The precise definitions follow.

Let $nt \in (F_2^8)^8$ be a tag nonce, $nr \in (F_2^8)^8$ a reader nonce, and $k \in (F_2^8)^8$ the shared key between the tag and the reader. In the terminology from Section 2 and the Atmel documentation, $nt$ is the cryptogram $C_t$, $k$ is the secret key $G$, and $at$ below is the cryptogram $C_{t+1}$. Section 2 does not mention $nr$ and $ar$ below, but in Atmel’s documentation they are $Q$ and $CH$ respectively. The initial cipher state

$$\begin{align*}
(s_0) & nt_0 \ nrt_0 \\
(s_1) & nt_2 \ nrt_2 \\
(s_2) & nt_4 \ nrt_4 \\
(s_3) & nt_6 \ nrt_6 \\
(s_4) & k_0 \ krt_4 \\
(s_5) & k_2 \ krt_5 \\
(s_6) & k_4 \ krt_6 \\
(s_7) & k_6 \ krt_7 \\
(s_8) & \text{nt} \ nrt
\end{align*}$$

Starting from the state called $s_0$, we feed in $nt_0$ (three times for CryptoMemory), then $nt_1$ (three times for CryptoMemory), then $nr_0$, and arrive at the state called $s_1$, etc. Note that these states are non-consecutive, e.g., $s_1$ is not the successor of $s_0$; we have only named those states that are needed for the description.

The authentication protocols for CryptoMemory and SecureMemory are similar, although the keystream bits used as authenticators are different in both cards. The precise definitions follow.

3.1.1 SecureMemory Authentication

After initialization, the cipher produces keystream that will be used for mutual authentication of tag and reader. In SecureMemory, every second output nibble is discarded. The keystream bits used as authenticators for the tag $at \in (F_2^{16})^4$ and for the reader $ar \in (F_2^{16})^5$ are interleaved. We define the following relevant states:

$$\begin{align*}
(0) & : succ^2(0, s_{i-1}) \\
& : \{9, \ldots, 40\}.
\end{align*}$$

The authenticator nibbles for the tag are

$$\begin{align*}
at_1 & := \text{output}(s_{2i+9}) \\
at_1+1 & := \text{output}(s_{2i+10}) \quad i \in \{0, 2, \ldots, 14\}
\end{align*}$$

and the authenticator nibbles for the reader are

$$\begin{align*}
ar_1 & := \text{output}(s_{2i+11}) \\
ar_1+1 & := \text{output}(s_{2i+12}) \quad i \in \{0, 2, \ldots, 14\}.
\end{align*}$$

The overview of the output during the authentication phase of SecureMemory is as follows.

$$\begin{align*}
(s_8) & - at_0 - at_1 - ar_0 - ar_1 \\
(s_{12}) & - at_2 - at_3 - ar_2 - ar_3 \\
(s_{16}) & - at_4 - at_5 - ar_4 - ar_5 \\
(s_{20}) & - at_6 - at_7 - ar_6 - ar_7 \\
(s_{24}) & - at_8 - at_9 - ar_8 - ar_9 \\
(s_{28}) & - at_{10} - at_{11} - ar_{10} - ar_{11} \\
(s_{32}) & - at_{12} - at_{13} - ar_{12} - ar_{13} \\
(s_{36}) & - at_{14} - at_{15} - ar_{14} - ar_{15} \\
(s_{40}) & - \text{nt} - \text{nrt}
\end{align*}$$

Staring from the state called $s_8$, one output nibble is discarded, the following is called $at_0$, the next one is discarded, etc., until $ar_1$. The state then reached is called $s_{12}$, etc.

3.1.2 CryptoMemory Authentication

After initialization, CryptoMemory generates the reader authenticator first and then the tag authenticator. While generating the reader authenticator $ar \in (F_2^{16})^5$, five keystream
nibbles are discarded and then two nibbles (one byte) are used, with the exception of the first byte where only 4 nibbles are discarded. The tag authenticator \( at \in (\mathbb{F}_2^4)^{16} \) consists of the bitstring \( \text{0xff} \) followed by the next 14 (consecutive) keystream nibbles produced. To be precise, define the following sequence of states. Again, note that these states are not consecutive.

\[
\begin{align*}
s_9 & := \text{suc}^5(0, s_8) \\
s_{10} & := \text{suc}(0, s_9) \\
s_i & := \text{suc}^6 (0, s_{i-1}) \quad i \in \{11, 13, \ldots, 23\} \\
s_i & := \text{suc}(0, s_{i-1}) \quad i \in \{12, 14, \ldots, 24\} \\
s_i & := \text{suc}(0, s_{i-1}) \quad i \in \{25, \ldots, 38\}
\end{align*}
\]

Then the reader authenticator is defined as

\[
\text{ar}_i := \text{output}(s_{i+9}) \quad i \in \{0, \ldots, 15\}
\]

and the tag authenticator is defined as

\[
\begin{align*}
\text{at}_0 & := \text{0xf} \\
\text{at}_1 & := \text{0xf} \\
\text{at}_i & := \text{output}(s_{i+23}) \quad i \in \{2, \ldots, 15\}.
\end{align*}
\]

The following shows a schematic overview of the output during the authentication phase of CryptoMemory.

\[
\begin{align*}
(s_8) & \quad 4x- \quad \text{ar}_0 \quad \text{ar}_1 \quad 5x- \quad \text{ar}_2 \quad \text{ar}_3 \\
(s_{12}) & \quad 5x- \quad \text{ar}_4 \quad \text{ar}_5 \quad 5x- \quad \text{ar}_6 \quad \text{ar}_7 \\
(s_{16}) & \quad 5x- \quad \text{ar}_8 \quad \text{ar}_9 \quad 5x- \quad \text{ar}_{10} \quad \text{ar}_{11} \\
(s_{20}) & \quad 5x- \quad \text{ar}_{12} \quad \text{ar}_{13} \quad 5x- \quad \text{ar}_{14} \quad \text{ar}_{15} \\
(s_{24}) & \quad \text{at}_2 \quad \text{at}_3 \quad \text{at}_4 \quad \text{at}_5 \quad \text{at}_6 \quad \text{at}_7 \quad \text{at}_8 \\
(s_{31}) & \quad \text{at}_9 \quad \text{at}_{10} \quad \text{at}_{11} \quad \text{at}_{12} \quad \text{at}_{13} \quad \text{at}_{14} \quad \text{at}_{15}
\end{align*}
\]

Figure 4:

Starting from the state called \( s_8 \), the first 4 output nibbles generated are discarded, the next two are called \( \text{ar}_0 \) and \( \text{ar}_1 \), respectively, the next 5 are discarded again, etc. For future reference, note that an attacker who has observed an authentication trace sees 16 consecutive keystream nibbles, viz., \( \text{ar}_{14}, \text{ar}_{15}, \text{at}_2, \text{at}_3, \ldots, \text{at}_{15} \).

**Example 3.5.** Figure 5 shows an authentication trace using the shared key \( k = 0x4f794a463ff81d81 \); the first two bytes on every line are a command and the last two are a CRC.

### 4. ATTACKING SECUREMEMORY

We now turn our attention to attacking the ciphers. We start with an attack against SecureMemory. We assume that an attacker has eavesdropped a single authentication session; the attack we describe recovers the shared secret key with probability at least 0.57. By using more authentication sessions we can arbitrarily increase this probability.

The attack we describe takes place in two phases. First, we use a correlation attack to recover (the left-hand and right-hand segments of) the internal state of the cipher just after feeding in the key and just before generating the authenticators (called state \( s_8 \) in Section 3.1.1). Since the attacker knows \( nr \) and \( nt \), he can also compute the state just before feeding in the shared secret key (called state \( s_4 \)). Then, we use a meet-in-the-middle attack to recover \( k \).

The running time of the whole attack is \( 2^{38} \) ticks of the SecureMemory cipher, which on a single-core 2GHz laptop takes about 10 minutes.

#### 4.1 Recovering the internal state

There are two weaknesses in the SecureMemory cipher that make it possible to recover the internal state of the cipher. The first one is that there is a high correlation between the output of the right-hand (or left-hand) segment and the keystream itself. Consider a state \( s \) and its output nibble \( \text{output}(s) \). For those bits for which the middle segment chooses the right (\( m_{0,1} = 1 \)), the output bit is equal to the corresponding output bit of the right-hand segment \( \text{output}(s)_i = \text{output}(r)_i \). Assuming a uniform probability, this happens with probability \( \frac{1}{2} \). Where the middle segment chooses the left (\( m_{0,1} = 0 \)), the output bit is equal to the corresponding output bit of the left-hand segment \( \text{output}(s)_i = \text{output}(l)_i \), but with probability \( \frac{1}{4} \) this equals the output bit of the right-hand segment \( \text{output}(r)_i \) anyway. So, with probability \( \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{3}{4} \), an output bit of the right-hand segment equals the corresponding keystream bit (and similarly for the left-hand segment). The second weakness is that the three segments operate independently. So, knowing (or guessing) the right-hand (or left-hand or middle) segment of a state \( s \), the attacker can compute the right-hand (or left-hand or middle) segment of the successor state \( s' \).

**Definition 4.1.** Consider a guess \( r \) for the right-hand segment of the internal state of the cipher at the start of the authentication phase (state \( s_8 \) in Section 3.1.1) and consider the 16 output nibbles generated by the right-hand segment that are used to compute at and ar. We define the score of \( r \) to be the number of bits that these 16 output nibbles have in common with the actual keystream (i.e., with at and ar). A similar definition applies for a guess \( l \) of the left-hand segment.

So, for a wrong guess of the right-hand segment, one would expect the score to be 64 (half of 128 bits correct); for the correct guess, one would expect the score to be 96 (\( \frac{128}{2} \) of the 128 bits correct). So, the attacker iterates over the \( 2^{25} \) possible right-hand segments and computes their score. The one with the highest score is most likely to be the correct one. As an approximation, we can assume that the score for the \( 2^{25} - 1 \) wrong guesses is binomially distributed with parameters \( p = \frac{3}{4} \) and \( n = 128 \) (128 bits, each of which has a probability of \( \frac{3}{4} \) of being correct) and the score for the correct guess is binomially distributed with parameters \( p = \frac{1}{4} \) and \( n = 128 \). Also assuming, again as an approximation, that all these scores are independent random variables, the following proposition applies.

**Proposition 4.2.** Let \( X_1, \ldots, X_{2^{25}-1} \sim \text{Binom}(128, \frac{1}{4}) \) and \( Y \sim \text{Binom}(128, \frac{3}{4}) \) be independent random variables. Then

\[
P[\forall i. X_i < Y] \approx 0.57
\]
and

\[ P[\exists i.\{Y < X_i \land \forall j \neq i. X_j < X_i\}] \approx 0.24. \]

**Proof.** Since the \(X_i\)'s and \(Y\) are assumed to be independent, we get

\[
P[\forall i. X_i < Y] = \sum_{k=0}^{128} P[\forall i. X_i < k \land Y = k]
\]

\[
= \sum_{k=0}^{128} P[X_1 < k] \cdot 2^{25-1} \cdot P[Y = k] \approx 0.57
\]

and

\[
P[\exists i.\{Y < X_i \land \forall j \neq i. X_j < X_i\}]
\]

\[
= \sum_{k=0}^{128} P[\exists i. X_i = k \land \forall j \neq i. X_j < k \land Y < k]
\]

\[
= \sum_{k=0}^{128} 2^{25-1} \cdot P[X_1 = k \land \forall j \neq i. X_j < k \land Y < k]
\]

\[
= \sum_{k=0}^{128} (2^{25} - 1) \cdot P[X_1 = k] \cdot P[X_2 < k]^{25-2} \cdot P[Y < k]
\]

\[ \approx 0.24. \]

So, as an approximation, with probability 0.57, the correct right-hand segment has a score that is higher than the score of all the wrong ones. With probability of only 0.24, a wrong right-hand segment has a score higher than all the others (including the correct one). Therefore we get with probability \(0.81) one right-hand segment that scores higher than all the others. So we need, on average \(1/0.81 \approx 1.23\) traces to obtain such a single candidate. The conditional probability that this candidate is indeed the correct right-hand segment is \(0.57/0.81 \approx 0.71\). To obtain a trace in which the highest scoring right-hand segment is indeed the correct one, we need on average \(1/0.57 \approx 1.75\) traces.

From now on, assume that we have a trace for which exactly one right-hand segment scores higher than all the others. For the attack we assume that it is the correct right-hand segment \(r\).

Then we try to find possible candidates for the left-hand segment. Note that for those bits of the known keystream where the output of the right-hand segment does not produce the correct bit, the corresponding bit from the middle segment (the selector) must choose the left-hand segment and the corresponding bit from the output of the left-hand segment must equal that keystream bit. For instance, when \(r\) has a score of 96, this happens \(128 - 96 = 32\) times. So, the attacker iterates over all \(2^{32}\) possibilities for the left-hand segment, keeping only those that satisfy the above constraint. Experiments show that this leaves only between, approximately, 10 and 200 candidates.

At this point, one could also try to recover (candidates for) the middle segment, but as we will show in the next two sections, that is not even necessary. Later, when we are attacking CryptoMemory in Section 5, we do recover the middle part as well.

### 4.2 Unrolling the cipher

Whenever the input \(b\) is known, given a state \(s'\) it is possible run the cipher backwards and recover the previous state \(s\). We start by defining an inverse to \(\Box\).

**Definition 4.3.** Define \(\Box : F_2^n \times F_2^n \rightarrow F_2^n\) by

\[ x \Box y = x - y \pmod{2^n - 1}. \]

To reconstruct \(s\), we first unshift the cipher, recovering the intermediate state \(s'\):

\[
\hat{l}_i := l_{i+1}', \quad i \in \{0, \ldots, 5\}
\]

\[
\hat{m}_i := m_{i+1}', \quad i \in \{0, \ldots, 5\}
\]

\[
\hat{r}_i := r_{i+1}', \quad i \in \{0, \ldots, 3\}
\]

\[
\hat{l}_6 := \text{L}^{-1}(l_6' \Box l_4')
\]

\[
\hat{m}_6 := \text{L}^{-1}(m_6' \Box m_6')
\]

\[
\hat{r}_4 := r_6' \Box r_4'.
\]

There is, however, an issue here that needs special care: the modified modular addition operator \(\Box\) is not injective when fixing one argument. When \(l_6' = l_4' \neq 0\) there are two possible previous left-hand segments, namely, \(L(l_6')\) could be equal to either 0 or to 31. Therefore, in those cases we have to consider both possible predecessors. Similarly, splitting might happen in the middle and right-hand segments. This issue does not significantly hamper the possibility of unrolling the cipher since it only splits with probability 1/31 for the left-hand and right-hand segments and with probability 1/127 for the middle segment. Also, because the result of \(\Box\) is never 0 unless both operands are 0, there are some states that do not have a predecessor. This happens when \(l_6 = 0\), but \(l_4' \neq 0\).
Next, we simply XOR back the input to recover the previous state:

\[ l_4 := \hat{l}_4 \oplus b_5b_6b_7b_0b_7 \]
\[ m_4 := \hat{m}_4 \oplus b_4b_5b_6b_7b_0b_2 \]
\[ r_1 := \hat{r}_1 \oplus b_3b_4b_5b_6b_4. \]

Also note that when unrolling the cipher, the three segments operate independently.

### 4.3 Recovering the key

In this section we use a meet-in-the-middle technique to recover the secret key.

We recall from Section 3.1 that the cipher state \( s_4 \), just before feeding in the key \( k \), is known to an attacker. Observe that the transition function for the right-hand segment only uses five bits \( b_0 \ldots b_4 \) of the input. Therefore, by guessing 20 bits of the key, it is possible to roll the cipher 6 times (feeding in \( nr_4 \) and \( nr_5 \) accordingly) and compute \( 2^{20} \) possibilities for the right-hand segment of \( s_6 \). Similarly starting at the end; we have just recovered the right-hand segment for the state at the start of the generation of the authenticators, i.e., just after feeding in the key. By guessing the other 20 bits of the key, it is possible to unroll the cipher 6 times (feeding in \( nr_7 \) and \( nr_8 \) accordingly) and obtain another set of \( 2^{20} \) candidates for the right-hand segment of \( s_6 \). (Actually, rolling back there are more candidates because of the splitting; in practice this is at most a ten fold increase.) Intersecting these sets of states, since the right-hand segment has 25 bits of entropy and we have guessed 40 bits of the input, we get approximately \( 2^{15} \) candidates. In practice, because of the additional splitting when rolling back, we get between 30000 and 66000 candidates for the right-hand segment of \( s_6 \). Note that we do not just get this set of candidates, but for each candidate we also have the leftmost five bits of every byte of the key. So, in fact, we get between 30000 and 66000 candidates for the leftmost five bits of every byte of the key.

Similarly for the left-hand segment, the transition function uses only bits \( b_3 \ldots b_7 \). Just as before, but for each candidate segment \( l \) from Section 4.1, we guess twice 20 bits and we meet-in-the-middle at (the left-hand segment of) \( s_6 \). In practice, we get between 1000 and 3000 candidates. Also here, we do not just obtain this set of candidates, but for each candidate we have the rightmost five bits of every byte of the key. So we get between 1000 and 3000 candidates for the right most five bits of every byte of the key.

We now combine the first set of left-hand candidates and the second set of right-hand candidates. Of course, a left-hand candidate (40 bits of the key) can only be combined with a right-hand candidate (also 40 bits of the key) if the bits of the key that they share are equal. These are bits \( b_3 \) and \( b_4 \) of every byte; \( 8 \cdot 2 = 16 \) bits in total. So, we get between \( 2^{16} \cdot (30000/2^{16}) \cdot (1000/2^{16}) \approx 450 \) and \( 2^{16} \cdot (66000/2^{16}) \cdot (3000/2^{16}) \approx 3000 \) candidate keys. Simulating the whole authentication session, which now also involves running the middle segment, with each of these candidate keys reveals what the actual key is.

### 4.4 Complexity and time

As a basic unit of time, we take one tick of the Secure-Memory cipher. As a reference note that an authentication attempt takes 88 ticks which means that a naive brute force attack takes \( 2^{64} \cdot 88 = 2^{70.5} \) cipher ticks.

The most time in this attack is spent in recovering the set of 10 to 200 candidates for the left-hand segment. Here we have to loop over \( 2^{25} \) candidates; for each of those candidates we have to compute a maximum of 64 ticks of the cipher, two for each of the nibbles of \( ar \) and \( at \) since Secure-Memory skips every second output during the generation of the authenticators. (Note, we only have to compute the left-hand segment, so we count this as \( \frac{1}{2} \).) So this gives a complexity of \( 2^{39} \) ticks, comparable to the time it takes to simulate \( 2^{32.5} \) authentications. Note that for the right-hand segment we only have to loop over \( 2^{25} \) candidates and for the meet-in-the-middle key-recovery we only have to build tables of size \( 2^{20} \), so we can ignore that. Also note that when our authentication session does not have the desired property that a single right-hand segment has the highest score, we only have wasted the \( 2^{25} \) loop over the possible right-hand segments. Since this happens only with probability \( 1 - 0.81 = 0.19 \) we can ignore this time. After on average 1/0.81 \( \approx 1.23 \) traces, we do have a single candidate for the right-hand segment and with probability 0.71 we recover the key in \( 2^{39} \) ticks. This yields an average complexity of \( 2^{39}/0.71 = 2^{39.5} \) ticks, i.e., the complexity of simulating \( 2^{34} \) authentications. On average, this needs 1/0.57 \( \approx 1.75 \) authentication traces. In practice, running this attack on a single-core 2 GHz laptop takes about 10 minutes.

### 5. Attacking CryptoMemory

We now turn our attention to CryptoMemory. We describe an attack in three phases: recovering the internal state by means of a correlation attack; unrolling the internal state; and recovering the key by means of a meet-in-the-middle attack. There is, however, a major complication. Because the output of the cipher is fed back into the internal state, it is no longer possible to run the three segments independently, at least not under all circumstances. This seriously complicates the recovery of the internal state and also makes unrolling the cipher slightly harder. Finally, we propose a trade-off between the number of authentication traces needed and off-line computation time, similar to the one proposed by Biryukov et al. in [BMS05].

### 5.1 Recovering the internal state

The starting point for this attack is the same as the one described in Section 4. Although the three segments cannot be run independently as for Secure-Memory, because of the feedback from the keystream, it is possible to do so when the keystream is fully known. Now most of the time, CryptoMemory discards several output nibbles when generating keystream. When it generates the tag authenticator \( ar \), however, CryptoMemory does produce 16 consecutive keystream nibbles (namely the last 2 of \( at \) and the 14 of \( ar \) that it actually generates). Note, by the way, that SecureMemory never produces consecutive keystream nibbles, but there that is not needed for the attack anyway. Let \( ks \in (\mathbb{F}_2^1)^{16} \) be those 64 bits of keystream. Knowing that the keystream bits are equal to the output of either the
So far we have consumed seven out of sixteen keystream no extra information gain as we know a number of bits of the components of the middle segment. After that there is repeat this procedure six times, for which  are, i.e., some bits of  . On average, we know two bits out of  . We now check if the newly computed  satisfies the condition

if  , then  , and otherwise discard it.

Now that all bits of  are set, we iterate only over all 128 values of  . Again, we compute the successor state and check the corresponding condition. At this point  is shifted one position to the right and a new  is computed from the sum  and  , i.e., we have set all bits of  ,  , and  . We repeat this procedure four more times, until all seven words of the middle segment are set. This gives us a number of candidate states that depend on the overlap between  and  . These candidate states can be unraveled as we describe in Section 5.2 and verified against the  nibbles.

5.2 Unrolling the cipher

Just as with SecureMemory, it is also possible to unroll the CryptoMemory cipher. If we are at a state  and know the keystream  output by the previous state  , the procedure is merely as described in Section 4.2, taking care of setting the input  to  accordingly. If we do not know the keystream  , we first have to reconstruct it. Of course, reconstructing  is no problem, as  . Now, first of all, we compute  as in Section 4.2 (if splitting happens, consider one possibility for  ). Write  . By Equations (2) and (1), we have

\[
f_{0,i} = \begin{cases} 
    \hat{l}_{i,1,i} \oplus \hat{b}_{i,1,i} & \text{if } m_{1,i,0} = 0 \\
    \hat{r}_{i,1,i} \oplus \hat{r}_{i,1,i} & \text{if } m_{1,i,3} = 1 \\
    \hat{l}_{i,1,i} \oplus \hat{b}_{i,1,i} & \text{if } m_{1,i,3} = 0 \\
    \hat{r}_{i,1,i} \oplus \hat{b}_{i,1,i} & \text{if } m_{1,i,3} = 1.
\end{cases}
\]

Now note that  , so we can use the above equation to compute  . Now  , we can use the above equation again to compute  . Doing this twice more, we can also compute  and  . Using Equation (1), we can now compute  .

5.3 Recovering the key

As before, we use a meet-in-the-middle technique to recover the secret key. This time the computational complexity of the attack is higher due to the keystream feedback loop, which makes it impossible to treat the left-hand and right-hand segments separately.

Recall that the cipher state  , just before feeding in the key  , is known to an adversary and assume the adversary also knows the state  , e.g., by running the attack described in Section 5.1.

Then, the adversary simply guesses the first 32 bits of  and runs the cipher forward from  , until half way the initialization protocol, i.e., to state  . This produces a set  of candidate states for  .

Similarly, it guesses the last 32 bits of  and runs the cipher backwards (unrolls) from  . This produces a set  of candidate states (around 2 · 232); the additional factor is
because of the states with multiple predecessors). Since we are guessing only two times 32 bits and states consist of 117 bits, there is only one element in $S_f \cap S_r$, namely $s_e$. The guessed bits that lead to it constitute the actual key $k$.

5.4 Complexity and time

As a baseline let us first establish the complexity of a naive brute force attack, again taking cipher ticks as a basic unit of time. For each of the $2^{34}$ possible keys, we need 125 cipher ticks to simulate an authentication. This adds to a complexity of $2^{73}$ cipher ticks.

For the first phase of the attack, recovering the internal state, we need to iterate over all $2^{35}$ left-hand segments and for each of them we need to compute the score, which takes 16/3 cipher ticks. This is of complexity $2^{38}$ ticks and takes about half an hour on a 2 GHz laptop. We also need to do the same for all $2^{35}$ right-hand segments, but this can be ignored.

The number of left-right candidates produced depends on the values of $N_l$ and $N_r$ (defined in Section 5.1) and there is a trade-off between the number of traces required for the attack and its computational complexity. As an example we consider two reference configurations: $C_0 := (N_l = 55, N_r = 51)$ and $C_1 := (N_l = 56, N_r = 52)$. Take for example $C_0$. Our attack from Section 5.1 will only be able to recover the internal state for those traces with left-hand segment score higher than 55 and right-hand segment score higher than 51. The score of the left-hand and right-hand segments of the correct state are binomially distributed as $\text{Binom}(64, \frac{1}{2})$. Hence the probability that the score of the correct left-hand segment is at least 55 is approximately 0.025 and the probability that the score of the correct right-hand segment is at least 51 is approximately 0.239. Assuming these scores are independent, the probability that both scores satisfy the requirement is $0.025 \cdot 0.239 \approx 0.0060$. Experiments show that the distributions of these scores is indeed very close to binomially distributed, but they are not totally independent. The experimentally observed probability of having both scores meet the requirement is approximately 0.00185. So the expected number of authentication session needed is $1/0.00185 \approx 544$. The score of a random left-hand or right-hand segment is also binomially distributed, but with distribution $\text{Binom}(64, \frac{1}{2})$. The probability of a random segment having a score of 55 or higher is $0.177 \cdot 10^{-8}$ and the probability of a random segment having a score of 51 or higher is $0.95 \cdot 10^{-6}$. So, we expect around $0.177 \cdot 10^{-8} \cdot 2^{35} \approx 60.9$ left-hand segments to have a score of 55 or higher and around $0.95 \cdot 10^{-6} \cdot 2^{35} \approx 31.6$ right-hand segments to have a score of 51 or higher. So this gives approximately $60.9 \cdot 31.6 \approx 1920$ left-right candidates per trace. We established the number of candidates for the middle segment per left-right candidate empirically; we get on average 1.43 \cdot 10^9 candidates for the middle segment. For each middle segment we need to unroll the cipher at most 64 times.

This gives us a total complexity of $2^{55}$ cipher ticks, comparable to simulating $2^{48}$ authentications. Assuming we dispose of a cluster of 50 2Ghz quad core computers, all this computation would take about two weeks.

Considering $C_1$, the expected number of traces needed to find one in which the correct segment has a score of at least 56 and the correct right-hand segment has a score of at least 52 is $1/0.000378 \approx 2640$ traces. Here we expect only $9.5 \cdot 7.7 \approx 73$ left-right candidates. Experiments show that the number of middle segments per left-right candidate is approximately $2.12 \cdot 10^{19}$. This gives us a total complexity of $2^{42}$ cipher ticks, i.e., the computational complexity of $2^{45}$ authentications. With the same computational power this would take less than 2 days of computation.

Once we have recovered the internal state, we need to do the meet-in-the-middle approach described in Section 5.3 to recover the key. For each of the $2^{32}$ guesses, we need to compute 14 cipher shifts to build the set $S_f$. Compared to the complexity above, this is negligible. It does require a storage space of around 16 Gb though. Using a 2 GHz computer with enough internal memory, it takes about half an hour to construct and sort this table. Similarly, from the other side we need another $2^{36}$ cipher ticks to unroll the cipher. Since we only need the intersection of $S_f$ and $S_r$, we do not actually store $S_r$, but only do a logarithmic search for each element of $S_r$ in the table for $S_f$. This whole computation takes another half hour.

6. CONCLUSIONS

In this paper we have described the ciphers used in the product families SecureMemory, CryptoMemory, and CryptoRF. We have shown weaknesses of these ciphers, most notably the fact that the three components of the cipher operate independently (knowing the keystream, in the case of CryptoMemory and CryptoRF) and that there is a strong correlation between the intermediate output of two of those components and the generated keystream. We have shown that an attacker can use these weaknesses and eavesdropped sessions to recover the secret key. For SecureMemory, the attack has a time complexity of $2^{39}$ cipher ticks; in practice it takes around 10 minutes on a 2 GHz laptop. For CryptoMemory and CryptoRF, the attack has a time complexity of $2^{52}$ cipher ticks; on a cluster of 50 2 GHz quad core machines, it takes about 5 days to execute the attack. We have implemented the full functionality of the chips in software. We have also implemented the full attacks and tested this on genuine traces.

With this paper we hope to reinforce that proprietary cryptography often leads to insecure products. Security obscurity does not compensate for the lack of public scrutiny since sooner or later the design of the cipher will become public.

We provide more evidence that this kind of LFSR-like ciphers often result in insecure constructions. In particular, that the design techniques of this particular cipher are unsatisfactory.
7. REFERENCES


