SAT-based cryptanalysis of ACORN

Frédéric Lafitte\(^1\) and Liran Lerman\(^2\) and Olivier Markowitch\(^2\) and Dirk Van Heule\(^1\)

\(^1\) Department of Mathematics, Royal Military Academy, Brussels, Belgium
frederic.lafitte@rma.ac.be

\(^2\) Quality and Security of Information Systems, Département d’informatique,
Université libre de Bruxelles, Belgium
llerman@ulb.ac.be

Abstract. The CAESAR competition aims to provide a portfolio of authenticated encryption algorithms. SAT solvers represent powerful tools to verify automatically and efficiently (among others) the confidentiality and the authenticity of information claimed by cryptographic primitives. In this work, we study the security of the CAESAR candidate \textit{Acorn} against a SAT-based cryptanalysis. We provide the first practical and efficient attacks on the first and the last versions of \textit{Acorn}. More precisely, we achieve state recovery, key recovery, state collision as well as forgery attacks. All our results demonstrate the usefulness of SAT solvers to cryptanalyse all the candidates of the CAESAR competition, thereby accelerating the "test of time".

1 Introduction

Authenticated Encryption (AE) provides (at the same time) message confidentiality and data authentication. Currently, the cryptographic literature provides two ways to obtain secure authenticated encryption primitives: generic composition and dedicated algorithm. The generic composition primitives require the application of two primitives (with two different keys): (1) an encryption primitive providing confidentiality, and (ii) a Message Authentication Code (MAC) primitive providing data authentication. In 2000, Bellare and Namprempre described three different generic composition primitives (namely Encrypt-and-MAC, MAC-then-Encrypt, and Encrypt-then-MAC) and prove that only one generic composition primitive fulfils all the considered security notions [2]. Compared to generic composition, dedicated primitives increase the efficiency (in time and memory) of the authenticated encryption process by executing one primitive (using one key) that provides at the same time confidentiality and data authentication.

The Authenticated Encryption with Associated Data (AEAD) primitives represent a generalisation of AE that (1) authenticate a part of the message called associated data (e.g., routing information in network packets), and (2) encrypt and authenticate another part of the message (e.g., the body data in network packets). The open cryptographic “\textit{Competition for Authenticated Encryption: Security, Applicability, and Robustness}” (CAESAR) [15] was launched by Bernstein in order to find a suitable portfolio of AEAD primitives with security and performance exceeding the current standards AES-GCM [13] and AES-CCM [10]. The call for submissions of CAESAR resulted in 57 first-round candidates. Each proposal relies on different design goals such as high-speed in software and/or hardware devices, low memory footprint as well as side-channel attacks resistances.

The proliferation of international open competitions (including AES, SHA-3 and eSTREAM) boosts the researches in cryptanalysis of cryptographic primitives. However, the impressive increase of submissions to cryptographic competitions reveals a major problem related to the cryptanalytic effort required to provide
a suitable analysis\footnote{The AES competition received 15 candidates while the eSTREAM competition obtained 34 candidates and the SHA-3 competition had 51 candidates.}: the human verification of the degree of resilience claimed by designers of cryptographic primitives is prone to flaws in particular under time constraints. As a result, it appears clearly that cryptanalysts require automatic tools to assist the analysis of cryptographic primitives in order to formally verify properties related, in the present case, to confidentiality and authenticity as well as to reduce the required (test of) time to analyse each primitive. It is worth to note that automatic tools are also useful for the design of new primitives. More precisely, automatic tools stress the effects of different parameters such as the number of rounds, the nonlinear part as well as the linear layers.

In July 2015, the CAESAR team announced that the 30 selected candidates for the second round include the ACORN proposal \cite{16}. Two months later, the designer of ACORN submitted (to the second round of the CAESAR competition) a tweaked version of the first version of ACORN (denoted ACORN v2). This paper demonstrates the usefulness of our automatic tool by highlighting critical flaws in the first and the last version of ACORN (i.e., ACORN v1 and ACORN v2).

Related Work.

In 2014, Liu \textit{et al.} analysed the existence of slid pairs attacks in the first version of ACORN that provide the same internal state of ACORN from two distinct pairs of key and initialisation vector (IV), up to a clock difference, with probability 1 \cite{7}. They also found slid keys that provide (with probability 1) an identical state up to a clock difference using one key and two distinct IVs. Furthermore, they explored state recovery attacks using guess-and-determine and differential algebraic techniques. The time complexities of the described state recovery attacks are about $2^{180}$ and $2^{130}$ CPU cycles (vs. $2^{128}$ for a brute-force attack).

In 2015, Chaigneau \textit{et al.} provided a key recovery attack under the nonce-reuse setting (i.e., the adversary uses several times a single nonce to encrypt several different plaintexts with the same key) and the decryption-misuse setting (i.e., ACORN releases the plaintext although the tag verification fails) although the designer of ACORN claims no security in these settings \cite{3}. The described attack (on the first version of ACORN) required to have the same associated data field in order to have the same internal state before encrypting the plaintexts. This context enables a state recovery attack followed by a key recovery attack from the extracted state.

Few months later, Salam \textit{et al.} showed the existence of state collision attacks in ACORN v1 that can be exploited in a forgery attack (representing the ability to produce a valid tag for the authentication of the data which has never been queried by the user), i.e. a pair of inputs (plaintext or associated data) that leads to the same state. They assumed that (1) the adversary knows the internal state before the encryption step if the attacker uses the plaintext to generate collisions or (2) the adversary knows the key and the IV if the attacker uses the associated data to generate collisions \cite{12}. According to the authors, the presented attacks can be extended to the second version of ACORN.

In 2015, Josh \textit{et al.} provided some observations on the first version of ACORN. They found that the combination (with the exclusive OR operation) of the first keystream bits (i.e., the stream of bits combined with the plaintext in order to provide the ciphertext) for a fixed key and IV but different associated data becomes the scalar 0 \cite{4}.

Contribution.

We explore and demonstrate the capability of our automated search tool to the cryptanalysis of the ACORN primitive. More precisely, our tool is able (1) to recover an arbitrary number of (internal) states (of the
stream-based authenticated encryption primitive Acorn) that leads to a given tag, (2) to extract the secret key of Acorn from a recovered state, and (3) to find an arbitrary number of plaintexts (and associated data) leading to the same tag. For the later attack, we discuss fix points and multiple states collisions (which generalise the results of Salam et al. [12]).

We exemplify the ability of an adversary to control the content of the messages that lead to the same tag (knowing the key and the IV). Afterwards, we discuss the exploitation of this property in a realistic attack scenario, suggesting that Acorn is not suitable for a wide range of applications. Since our attack does not contradict the claims of Acorn’s author, this attack casts some doubt on the necessary security requirements of authenticated encryption schemes.

We also show that all our attacks can be quickly reproduced (1) with an ordinary desktop computer, and (2) without human analysis of the algebraic property of Acorn. All our results are based on a freely available tool called Cryptosat (written in R and freely available on our website\(^2\)) using the SAT-solver Cryptominisat (version 2.9.5) [14].

Outline.

The rest of the paper is organized as follows. Section 2 details the Acorn authenticated encryption primitive. Section 3 presents the SAT-based analysis of cryptographic primitives with Cryptosat. Section 4 shows the results of attacks against Acorn using Cryptosat. Section 5 concludes this paper with several perspectives of future works.

2 ACORN

The CAESAR candidates rely (among others) on a secret key \( K \in \mathcal{K} = \{0,1\}^k \), a nonce \( N \in \mathcal{N} = \{0,1\}^n \), a message payload \( P \in \mathcal{P} = \{0,1\}^* \) (requiring both confidentiality and data authentication services) and an associated data \( A \in \mathcal{A} = \{0,1\}^* \) (requiring data authentication service). An AEAD primitive maps the message \( P \) and the associated data \( A \) to a single binary string \( C \in \mathcal{C} = \{0,1\}^* \) that contains the encrypted form of \( P \) as well as additional message authenticated code called the tag \( T \in \mathcal{T} = \{0,1\}^t \) (where \( t \) represents the tag length) in order to authenticate \( P \) and \( A \), i.e.:

\[
\text{AEAD}: \mathcal{K} \times \mathcal{N} \times \mathcal{P} \times \mathcal{A} \rightarrow \mathcal{C}.
\]  

(1)

The inverse mapping of \( \text{AEAD} \) (denoted \( \text{AEAD}^{-1} \)) returns the original message payload \( P \) if the user supplies the correct values for \( K \), \( N \), \( C \) and \( A \), i.e.:

\[
\text{AEAD}^{-1}: \mathcal{K} \times \mathcal{N} \times \mathcal{C} \times \mathcal{A} \rightarrow \mathcal{P} \cup \bot,
\]  

(2)

where \( \bot \) is an error message returned if the tag computed by the receiver does not match the received tag \( T \) (in which case \( \text{AEAD}^{-1} \) releases no plaintext).

Acorn represents a lightweight authenticated encryption primitive based on a stream cipher using a 128-bit key and a 128-bit nonce (also known as initialization vector, or IV in short) with the requirement that the associated data length and the plaintext length are at most \( 2^{64} \) bits each. Acorn manipulates a single internal state (denoted \( S \)) of 293 bits for encryption and authentication. The state represents the concatenation of 6 Linear Feedback Shift Registers (LFSRs) (of lengths 61, 46, 47, 39, 37 and 59) and one register (of length 4). Acorn executes three different Boolean functions:

\(^2\)https://qualsec.ulb.ac.be/people/frederic-lafitte/cryptosat/
the function $\text{KSG}_{128}(S_i)$ that generates the keystream bit (i.e., a stream of bits added, bit-wise modulo 2, to the plaintext to form the ciphertext and vice versa) from the $i$-th generated state (denoted $S_i \in \{0,1\}^{293}$).

- the function $\text{FBK}_{128}(S_i, ca_i, cb_i)$ that computes the overall feedback bit using two constants $ca_i \in \{0,1\}$ and $cb_i \in \{0,1\}$ (where $ca_i$ and $cb_i$ represent the $i$-th bit of vectors $ca$ and $cb$), and

- the function $\text{StateUpdate}_{128}(S_i, m_i, ca_i, cb_i)$ that updates the state according to the $i$-th bit of the input message $m$ (denoted $m_i$), the vector $ca$ and the vector $cb$.

In more details, ACORN generates the $i$-th keystream bit $ks_i$ by computing:

$$ks_i = \text{KSG}_{128}(S_i)$$

$$= S_{i,12} \oplus S_{i,154} \oplus \text{maj}(S_{i,235}, S_{i,61}, S_{i,193}),$$

where $\oplus$ is the addition operation modulo 2, $S_{i,j} \in \{0,1\}$ represents the $j$-th bit of $S_i$ and

$$\text{maj}(x, y, z) = (x \land y) \oplus (x \land z) \oplus (y \land z).$$

ACORN generates the feedback bit (denoted $f_i$) by computing:

$$f_i = \text{FBK}_{128}(S_i, ca_i, cb_i, ks_i)$$

$$= S_{i,0} \oplus \neg S_{i,107} \oplus \text{maj}(S_{i,244}, S_{i,23}, S_{i,160})$$

$$\oplus \text{ch}(S_{i,230}, S_{i,111}, S_{i,66}) \oplus (ca_i \land S_{i,196})$$

$$\oplus (cb_i \land ks_i),$$

where:

$$\text{ch}(x, y, z) = (x \land y) \oplus (\neg x \land z).$$

ACORN updates the state by computing the following $\text{StateUpdate}_{128}$ function:

$$S_{i,289} = S_{i,289} \oplus S_{i,235} \oplus S_{i,230},$$

$$S_{i,230} = S_{i,230} \oplus S_{i,196} \oplus S_{i,193},$$

$$S_{i,193} = S_{i,193} \oplus S_{i,160} \oplus S_{i,154},$$

$$S_{i,154} = S_{i,154} \oplus S_{i,111} \oplus S_{i,107},$$

$$S_{i,107} = S_{i,107} \oplus S_{i,66} \oplus S_{i,61},$$

$$S_{i,61} = S_{i,61} \oplus S_{i,23} \oplus S_{i,0},$$

$$f_i = \text{FBK}_{128}(S_i, ca_i, cb_i, ks_i)$$

$$S_{i+1,j} = S_{i,j+1} \forall j \in \{0,1,\ldots,291\},$$

$$S_{i+1,292} = f_i \oplus m_i,$$

as clarified in Figure 1. In the following, for the sake of brevity, we detail only ACORN v2.

Based on the three functions $\text{KSG}_{128}$, $\text{FBK}_{128}$ and $\text{StateUpdate}_{128}$, ACORN executes four steps: (1) the initialization, (2) the processing of the associated data, (3) the processing of the plaintext, and (4) the tag generation as illustrated in Figure 2.

More precisely, during the initialization step, ACORN first initializes the state to the vector 0 before executing 1793 times the function $\text{StateUpdate}_{128}$ with the input parameters (related to the secret key and
Fig. 1: Concatenation of six LFSRs and a register of four bits. The function $f_i$ represents the overall feedback bit for the $i$-th step while $m_i$ represents the $i$-th bit of the input message.

the initialization vector) defined in Table 1. Note that the function $\text{StateUpdate}_{128}$ manipulates one bit at a time from the input parameters.

During the processing of the associated data, ACORN executes $\text{adlen} + 256$ times (where $\text{adlen}$ is the bit length of the associated data) the function $\text{StateUpdate}_{128}$ with the input parameters (related to the associated data) defined in Table 2.

During the processing of the plaintext and for each bit of the plaintext, ACORN combines one bit of the plaintext with the keystream (provided by the function $\text{KSG}_{128}$) and updates the state $S$ with the function $\text{StateUpdate}_{128}$ that takes as input the parameters defined in Table 3. ACORN executes this step $\text{pclen} + 256$ times (where $\text{pclen}$ represents the bit length of the plaintext).

Finally, during the tag generation step, the function $\text{StateUpdate}_{128}$ takes one bit at a time from three vectors $m$, $ca$ and $cb$ defined in Table 4, and the tag is given by the compression function that outputs the last (128) generated keystream bits.

Note that at each of the above steps, ACORN updates the internal state according to the value of the key, the IV, the associated data or the plaintext leading the keystream bits (and eventually the tag) to be related to these values.

<table>
<thead>
<tr>
<th>Vector</th>
<th>Index 0 to 127</th>
<th>128 to 255</th>
<th>256</th>
<th>257 to 1791</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>$K$</td>
<td>$IV$</td>
<td>$K_0 \oplus 1$</td>
<td>$K_1, \ldots, K_{127}, K_0, \ldots, K_{127}, K_0, \ldots, K_{127}, \ldots$</td>
</tr>
<tr>
<td>$ca$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$cb$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Values of input parameters to the function $\text{StateUpdate}_{128}$ during the initialization step where $K_i$ represents the $i$-th bit of the key $K$ and $IV$ represents the initialization vector.

3 SAT-based Analysis

Boolean satisfiability, abbreviated SAT, is the problem of deciding whether a given propositional formula $\phi(x_1, \ldots, x_n)$ can be satisfied, i.e. whether there exists a map $v : \{x_1, \ldots, x_n\} \rightarrow \{\text{true, false}\}$ such that $\phi(v(x_1), \ldots, v(x_n))$ is true.

In general, modern SAT solvers are algorithms that aim at answering instances of SAT by attempting to construct a satisfying valuation $v$. Any algorithm that attempts to solve (random) SAT instances will have
Fig. 2: Acorn executes four phases: (1) the initialization, (2) the processing of the associated data A, (3) the processing of the plaintext P and (4) the tag T generation.
a worst case exponential complexity. However, modern SAT solvers perform surprisingly well on very large (structured) instances, to the extent that it is often a good strategy to translate a difficult problem into SAT in order to harness the power of modern solvers.

In the following, the formula $\phi$ is represented in Conjunctive Normal Form (CNF): the formula $\phi$ contains a logical conjunction ($\land$) of clauses, where a clause represents a logical disjunction ($\lor$) of literals, and where a literal denotes a proposition $x_i$ or its negation (denoted $\neg x_i$).

Since 1999, cryptanalysts mention SAT-solvers as useful tools for cryptanalysis (see for example the work of Massacci [8], the work of Massacci and Marraro [9], and the work of Mironov and Zhang [11]). Our SAT-based cryptanalysis tool (called Cryptosat) is freely available on our website\(^3\) and allows to easily verify properties of any symmetric key algorithm of the ARX family (+S-boxes and bit-wise Boolean functions). For example, the tool was successfully applied to the analysis of stream ciphers (the ZUC primitive [6] as well as the compression functions of the MD4 and MD5 hash functions or the key schedule of WIDEA and MESH block ciphers [5]). In our experiments, Cryptosat uses the SAT-solver Cryptominisat (version 2.9.5) [14], leaving the investigation of other SAT-solvers as an interesting scope for further research.

Cryptosat transforms C++ code into a CNF formula thanks to operator overloading. For example, the exclusive-or operation (denoted $\oplus$) between two variables $x_1 \in \{0, 1\}$ and $x_2 \in \{0, 1\}$ can be represented with the following CNF formula:

$$\phi(x_1, x_2) = (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2).$$

(18)

The resulting formula is written in a text file formatted in DIMACS \(^4\) that contains three parts. The first part contains a set of comments beginning with the character c. The second part of the file contains the line p cnf nbvar nbclauses where nbvar indicates the number of variables and nbclauses represents the number of clauses in the formula. Finally, the last part of the file contains one line per clause (ended with the character 0) where a proposition $x_i$ (respectively $\neg x_i$) is represented by the integer $i$ (respectively $-i$). For example,

\(^3\) https://qualsec.ulb.ac.be/people/frederic-lafitte/cryptosat/

the CNF formula representing the exclusive-or operation between two variables $x_1 \in \{0, 1\}$ and $x_2 \in \{0, 1\}$ can be encoded in DIMACS as follows:

```plaintext
c XOR formulated in CNF and encoded in DIMACS
p cnf 2 2
1 2 0
-1 -2 0
```

*Cryptosat* is used with a timeout: its output is either SAT or UNSAT, depending on the output of *Cryptominisat*, or TIMEOUT when the SAT-solver exceeds the given timeout. In the SAT case, a map $v$ that satisfies the formula $\phi$ is returned. More precisely, as soon as we encoded in C++ the function $\text{AEAD}(K, N, P, A)$, *Cryptosat* converts the C++ code into a CNF formula (representing the C++ code) in which the propositions correspond to the manipulated bits of $\text{AEAD}(K, N, P, A)$. Afterward, the user of *Cryptosat* substitutes into the formula $\phi$ known values for $N$, $P$, and $A$ in order to obtain a simpler formula $\phi'(K)$ fed to the (*Cryptominisat*) SAT solver in order to extract the secret key $K$. It is worth to note that *Cryptosat* provides an interface that minimises user effort. Briefly, *Cryptosat* verifies at the same time the degree of resilience of the C++ code and the primitive’s design.

### 4 Experiments

This section demonstrates that Acorn v1 and v2 contain critical weaknesses leading to a state recovery attack (in Section 4.2), a key recovery attack (in Section 4.3), a state collision attack (in Section 4.4) as well as a forgery attack (in Section 4.5). The state recovery attack provides the probability that *Cryptosat* retrieves an internal state from a given fixed tag. The key recovery attack shows the probability that *Cryptosat* extracts the secret key from a known internal state. The state collision attack provides several internal states (manipulated before the tag generation) leading to a same given tag while the forgery attack provides several plaintexts and associated data (in which the values of a subset of bits can be selected by the adversary) leading to a same given tag. We estimate the probabilities and the time complexity of each attack by using 3,000 (randomly) generated tags or states, depending on the context.

#### 4.1 Experimental setup

The randomness used in all the experiments comes from the default pseudorandom number generator of R (i.e., Mersenne-Twister). In order to apply our attack, we installed *Cryptominisat*, a freely available tool in GitHub\(^5\). Then, we installed R, a free software environment available in the R Project\(^6\) before importing

\(^5\) [https://github.com/msoos/cryptominisat](https://github.com/msoos/cryptominisat)

\(^6\) [https://www.r-project.org](https://www.r-project.org)
our R package Cryptosat available in our website\footnote{https://qualsec.ulb.ac.be/people/frederic-lafitte/cryptosat/}. Eventually, we executed a set of R codes provided in the Appendices and referred to in the subsequent sections describing the attacks. Note however that we provide the codes for ACORN v2 but the same attacks can be applied on ACORN v1 by putting the parameters associated to the first version of ACORN. All the presented results can be reproduced with an AMD Opteron 6134 2.3 GHz using 1 core and 4 GB DDR3 ECC RAM 1.333 Ghz.

### 4.2 State recovery attack

The first experiment provides an intuition on the effectiveness of a SAT-based analysis on a reduced version of ACORN. Figure 3 shows the probability to extract an internal state (from which the tag is produced) of ACORN leading to the tag value as a function of (1) the number of iterations in the generation of the random given tag, and (2) the 	imeout\ (representing the limit of time before the adversary stops the execution of Cryptominisat). The results show that the probability to extract the target value (with a fixed timeout value) depends linearly (with a very small slope) on the number of iterations. Furthermore, the figure demonstrates that reduced versions of ACORN can be attacked in practice without any human analysis of the algebraic properties of the target primitive.

![Fig. 3: Estimated probability (with 3,000 randomly generated tags) to recover an internal state leading to the target tag value provided by ACORN as a function of (1) the number of iterations during the generation of the tag and (2) the timeout value (in seconds).](image-url)
The next experiment estimates the probability of the previously presented state recovery attack on the full version of ACORN v1 and ACORN v2. Recall that ACORN v2 executes 768 iterations during the tag generation while ACORN v1 executes 512 iterations. Table 5 provides the (estimated) probability to extract the (secret) internal state from the tag as a function of the value of the timeout. For example, an adversary extracts an internal state of ACORN v1 leading to a given tag value with probability 0.6240 when the timeout equals to 800 seconds. Surprisingly, our results also point out that an adversary requires the same execution time to extract the secret internal state (leading to the tag value) when targeting the first and the last version of ACORN. Indeed, an adversary recovers an internal state of ACORN v2 leading to the tag value with probability 0.6093 when the timeout equals 800 seconds. The R Code 1.1 (in the Appendix) allows to reproduce the state recovery attack for ACORN v2 from a tag (denoted TARGETMAC in the code) with a timeout (denoted TIMEOUT in the code) thanks to Cryptominisat (located in the path denoted SOLVER in the code).

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Timeout 200</th>
<th>Timeout 400</th>
<th>Timeout 600</th>
<th>Timeout 800</th>
<th>Timeout 1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACORN v1</td>
<td>0.1903</td>
<td>0.3683</td>
<td>0.5000</td>
<td>0.6240</td>
<td>0.6847</td>
</tr>
<tr>
<td>ACORN v2</td>
<td>0.1890</td>
<td>0.3610</td>
<td>0.4907</td>
<td>0.6093</td>
<td>0.7177</td>
</tr>
</tbody>
</table>

Table 5: Estimated probability (with 3,000 randomly generated tag) to extract the state stored before the tag generation from the tag as a function of the target primitive as well as the timeout value (in seconds).

Note however that the retrieved internal state (with the previously described attack) may differ from the internal state computed by the user of ACORN knowing the key, the IV, the associated data and the plaintext due to the surjectivity propriety of the mapping from the internal state (of 293 bits) to the tag value (of 128 bits), i.e. several internal states lead to the same tag. To fix this issue, we assume in the next experiment that the adversary knows (e.g. from a side-channel) a subset of bits of the target internal state representing the state manipulated when ACORN uses the key, the IV, the associated data and the plaintext. Table 6 provides the probability to extract the (right) internal state as a function of the number of known bits for ACORN v1 and ACORN v2 with a timeout of 1,000 seconds. The probability to extract the right internal state falls to zero when considering less than 263 known bits of the state with a timeout of 1,000 seconds. We could increase this probability by increasing the timeout, which constitutes an interesting future work.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Number of known bits</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACORN v1</td>
<td>283</td>
<td>0.1023</td>
</tr>
<tr>
<td></td>
<td>273</td>
<td>0.0137</td>
</tr>
<tr>
<td></td>
<td>263</td>
<td>0.0000</td>
</tr>
<tr>
<td>ACORN v2</td>
<td>283</td>
<td>0.0757</td>
</tr>
<tr>
<td></td>
<td>273</td>
<td>0.0517</td>
</tr>
<tr>
<td></td>
<td>263</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

Table 6: Estimated probability (with 3,000 randomly generated tag) to extract the right state stored before the tag generation from the tag as a function of the target primitive and the number of known bits of the right state with a timeout of 1,000 seconds.
4.3 Key recovery attack

The next experiment on ACORN v1 and ACORN v2 provides the (average) execution time required by Cryptosat to extract the secret key as well as the IV when the adversary knows the internal state, the plaintext and (optionally) the ciphertext. Since ACORN is based on a stream cipher, we expect ACORN to exhibit a similar resistance to this kind of attack (i.e., key-recovery from state is an important criterion for stream ciphers). Figure 4 shows the average execution time (in seconds) to extract the key and the IV with probability 1 as a function of the length of the plaintext.

The experiments show that the secret key and the IV can be (deterministically) derived as soon as the (previously presented) state recovery attack recovers the right internal state, i.e. the state just after processing the associated data and plaintext using that specific key-IV pair; indeed, other states can lead to the same tag. Furthermore, Cryptosat requires a (slightly) lower execution time to recover the secret key and the IV from ACORN v2 than from ACORN v1. The reason is that ACORN v1 executes 2560 iterations (before the tag generation step) independently of the size of the message while ACORN v2 executes (i.e., 2304) a lower number of iterations. Note however that, compared to ACORN v2, the cost of the higher number of iterations for ACORN v1 (independently of the size of the message) should vanish when the size of the message increases.

Code 1.2 (in the Appendix) allows to reproduce the key recovery attack on ACORN v2 from a known internal state (denoted STATE in the code), using a plaintext (denoted PLAINTEXT in the code) as well as the ciphertext (denoted CIPHERTEXT in the code).

Fig. 4: Average execution time (in seconds and estimated with 3,000 randomly generated tags) to extract the key and the IV from a known state (representing the input state to the tag generation step) for ACORN v1 and ACORN v2 as a function of the size of the plaintext (in bytes) knowing only the plaintext in 4(a), and knowing the plaintext and the ciphertext in 4(b).
4.4 State collision attack

The next experiment generalises the results of Salam et al. [12] by providing a multi-state collision attack (i.e., several internal states leading to the same tag value) instead of a single collision attack provided by Salam et al. In a technical point of view, in order to obtain several internal states leading to the same tag value, we fix the value of one different bit of the state (to the value 1) for each generated state, leaving Cryptosat to select the value of the other bits of the state. Table 7 provides five different states of ACORN v1 as well as five different states of ACORN v2 leading to the same tag. We extract each state of ACORN v1 with an average of 1641.98 seconds and each state of ACORN v2 with an average (over just 5 samples) of 570.736 seconds.

Code 1.3 (in the Appendix) provides the R instructions in order to accomplish in practice a multi-collision attack extracting a number of different states (denoted NUMBEROFSTATES in the code) knowing one tag value.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Primitive</th>
<th>State</th>
<th>Solving time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3150ada093af0e0f098a</td>
<td>ACORN v1</td>
<td>13a412610222da5ec865640fd85ba026f650491a7054a3d60bedb4e9ab0848ace3d9</td>
<td>956.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1a3da43b0973ed7b22d73116376e57a15cc40767a2ce9112f111b6272c3717afe47</td>
<td>766.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1529bbed69a296669c424546b51d546b0be4725e49646e3b39563f0a00c717a953a3464d</td>
<td>4649.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>02d32cc0648530605f541b0ba94c8d88b68eecc8e320660dd2d1c26d4be857c16bc415541</td>
<td>85.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>050db2e7775976ef9d5aed44f9c5e6551ec89b2b84caebf56796a6ce3463d7a169d3c</td>
<td>1751.41</td>
</tr>
<tr>
<td>3150ada093af0e0f098a</td>
<td>ACORN v2</td>
<td>11612197d0c34dfba3644474d410604571ca3852b2f7974537f9831c9af513d0763661273</td>
<td>391.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0d174afadb4e04560993d776ec82397b5792f511ec36d130941454e35347ac4613ac720718f</td>
<td>875.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1ce35f745bdf1781228a649b76d3430c745d4a90fd09d1159cae29781a7d0fffc5322fa6</td>
<td>74.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0f26c96fd301a6a4949559b0528267c83f01a522747fa3490a48e3f146207e2f5e5e28</td>
<td>491.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1357259c9a50ec39b86b5603ca8e487b6456ed772491b07953e27c14525b093a50f6d4d1</td>
<td>1021.27</td>
</tr>
</tbody>
</table>

Table 7: Execution time (in seconds) to extract the state (showed in hexadecimal) stored before the tag generation of ACORN v1 and ACORN v2 as a function of the tag value (displayed in hexadecimal). Note that the presented states contain 74 symbols representing 296 bits. However, only the (293) least significant bits of the state lead to the same tag.

4.5 Forgery attack

Similarly to the previous experiments, Table 8 provides the tag value when the plaintext is empty as well as four different plaintexts for ACORN v1 and ACORN v2 (extracted by Cryptosat) leading to the same tag.
knowing the key and the IV. Note that the adversary is not required to known the key and the IV if he knows
the internal state. Cryptosat requires a solving time to extract the plaintext of 517.35 seconds on average for
ACORN v1 and of 939.74 seconds on average for ACORN v2. Note that the same attack can be applied on the
associated data part. For example, Table 9 provides the same tag generated in the previous experiment but
with an empty associated data as well as with four different associated data for ACORN v1 and ACORN v2.

Code 1.4 (in the Appendix) shows the R code allowing to extract several plaintexts (including the plaintext
of length zero) leading to the same tag value. More precisely, the variable NUMBEROFPLAINTEXTS denotes the
number of different plaintexts to extract (including the plaintext of length zero).

Note however that, in the previous experiment, Cryptosat produces several plaintexts leading to the same
tag value but with possibly different internal states manipulated before the tag generation and after the
processing of the plaintext. Table 10 provides several plaintexts leading to the same internal state stored
before the tag generation (that leads eventually to the same tag value). There is a significant performance
gain by targeting the internal state without the knowledge of the tag value as done in the previous experiment
(although all the plaintexts lead to the same tag in this new experiment) since the adversary has access to
useful information. Furthermore, this experiment shows that an adversary can extract very quickly a fixed
point, i.e. several encrypted plaintexts leading to the same given internal state (in the present case the state
obtained with an empty plaintext). An attacker can reiterate this experiment by executing the Code 1.5
available in the Appendix.

We exemplify the forgery attack with a realistic scenario by selecting carefully the plaintext values. More
precisely, Table 11 shows 3 different plaintexts (of 64 bytes) representing three informative texts (of 15 or 17
characters depending on the context) encoded in ASCII leading to the same tag for ACORN v1 and ACORN
v2. The three (informative) messages are: “Bob gave 5Euro.”, “Bob gave 500Euro.” and “Eve gave 500Euro.”.
The ASCII encoding represents each character with 1 byte, leading the length of each text to be 15 or 17
bytes depending on the context. The unused bytes (i.e., 49 or 47 bytes) are filled by Cryptosat in order to
get the same tag. Cryptosat requires less than 640 seconds to find each plaintext for ACORN v1 and less than
130 seconds to find each plaintext for ACORN v2.

The presented forgery attacks imply that (1) a user (knowing the key and the IV) can easily repudiate
the authenticity of the plaintext and of the associated data (i.e., convince the receiver that the sender sent
another message), and (2) an (unreliable) communication channel can lose or modify the ciphertext and the
associated data without any detection by the receiver (i.e., a data integrity issue).

The forgery attack can also be applied in the context of Cryptographic Key Management Systems
(CKMS) [1]. For example, a malicious maintenance personal of the CKMS can produce a secret key for
each honest user (in which the metadata indicates the name of the user) that can also be used by a (set of)
malicious person(s) on a cryptographic module that verifies the access to the key (of an honest user) from the
tag. More precisely, for each (honest) user, the CKMS generates a secret key authenticated with a metadata
(indicating the name of the honest user) that leads to the same tag when modifying the metadata with the
name(s) of (a set of) malicious person(s). As a result, the cryptographic module cannot reliably distinguish
the use of the key between an honest user and a malicious person.

5 Conclusion and perspectives

There have been several efforts for gauging the security provided by ACORN v1 [3, 4, 7, 12]. We presented
practical (state recovery, key recovery, state collision and forgery) attacks against the full version of ACORN
v1 and ACORN v2. These attacks do no contradict the author’s security claims but casts some doubts on the
wide applicability of ACORN given the attack from Section 4.5.
The main advantage of our strategy based on Cryptosat, which can be applied to any other target primitive, lies in automation: Cryptosat requires no human analysis concerning the (algebraic) properties of the primitive. More generally, all our results highlight the usefulness of our tool in order to verify (quickly and easily) security properties of cryptographic primitives such as the presence of fixed points, time complexity estimations as well as probabilities estimation. All the presented attacks can be reproduced using the scripts given in the Appendix.

We plan three future works. First, we aim to extend our experiments by increasing the number of known tag values by the adversary (instead of the single tag value used during our experiments). Second, we plan to verify each candidate of the CAESAR competition in order to detect (with Cryptosat) flawed primitives. Third, in the long term, we aim to justify what makes cryptographic instances resilient to SAT-based attacks.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Tag</th>
<th>Plaintext</th>
<th>Solving time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACORN v1</td>
<td>7d945f7104c1d29f52a22b5e362c7f8</td>
<td>$\emptyset$</td>
<td>610.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5676f6d6ac385210838bee6901945034c273e539995bb76ff73829a3c017c4e98a9f5a10f293562d4c6582f0d7c856839e7a43fd89</td>
<td>1050.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cac11cf235041c798c35ca35fb297c19558b9f055a782b1659937a419a8f1c226764760586465daac8f87f53f9995bb78</td>
<td>359.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c6cf6c897f512fe739be01b9bde2bb7e6db46a5682f00953e9b4dd209663b21271f2dbd3d152155b613889e12da4ef923060d4002506efc84ed53</td>
<td>49.02</td>
</tr>
<tr>
<td>ACORN v2</td>
<td>607aceb1acfc94bfa7f19a575bf0365304</td>
<td>$\emptyset$</td>
<td>3170.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8f38f07a659eccbe2014e63a2a33c211266fa06632f939128fa7e6eb04b1673bd382ed083091c787923def84d102d15d840665105cb3d</td>
<td>227.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b87390ff2e389e3140f26ec5f24f164aa0f08625bbac76cfe3e573000583e3450f6c446262a4f4244b431cd74967c245b2e3dd8354acf</td>
<td>258.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2c8e62845ff7850d63b4a48c73d17a9b45ab6c248266542735e90d06edad795254bebd1e07261c98585c8a2aadbe5dd3a93401d8baf50</td>
<td>101.6</td>
</tr>
</tbody>
</table>

Table 8: Execution time (in seconds) to extract the plaintext (shown in hexadecimal) leading to the tag (displayed in hexadecimal) produced by ACORN v1 and ACORN v2 knowing the key (equal to $32fd8dc435218dffcde3f439018df16$ in hexadecimal) and the IV (equal to $a25a90cd57ff082ec9902c524085$ in hexadecimal). The symbol $\emptyset$ represents the empty plaintext.
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Tag</th>
<th>Associated data</th>
<th>Solving time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACORN v1</td>
<td>7d945f7104c1d2f52a22b5e3fd2e7f8</td>
<td>9106ff01d40dd3002c08598a6c446bc171f1ab77171ef94bc434f9f3f2697114484fcb134362aabed1683ba895d927d7117db1f1e5a2ea</td>
<td>1000.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f606cfe1db155b6148a7f6524456a45450a2b0fa587606a90d20820d709468861e9801ce8ac2e9a9faced37fd7b26d1e798eb69868868828a4</td>
<td>214.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e74a99b5fffd9244d0b2bb7c5854fba8f5ab777bed5f666f66f66f4f03f9f</td>
<td>871.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9b81302db0ff07dce3b5b6f6f527119845fa6aa22167e782a8177cc2277a4db17692b7b285869361e9395cc1b444ca8d29602d27ée7c0d</td>
<td>1402.92</td>
</tr>
<tr>
<td>ACORN v2</td>
<td>607aceb1aaf94bfa7f19a57b8365304</td>
<td>20ecf30a5833c4efcf3605ca62ca0a9e0a9c1e90c8aa7953b52620a55b32e8575ee8ab5f5cd103926466687e8c4490d86df47069e840f1</td>
<td>1695.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fee2b3ff4d0528e03f60846f7c8b4ddae56f6e2487fa21edeb8927ecd9ae1d7665407eb8bdf8fde39d7ed79c6e695485c59c01ca1a9e6315c</td>
<td>260.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ae563488f51e5aa258296337473ed5e7aaf82e65cf8f53b104530266</td>
<td>170.51</td>
</tr>
<tr>
<td></td>
<td></td>
<td>401b05ed51ffa73d46a8d8f681fd00dce07b0e3cca96227621d266036</td>
<td>67.29</td>
</tr>
</tbody>
</table>

Table 9: Execution time (in seconds) to extract the associated data (showed in hexadecimal) leading to the tag (displayed in hexadecimal) produced by ACORN v1 and ACORN v2 knowing the key (equal to 32fd8ec435218ddffc03f3f39018e6f16 in hexadecimal) and the IV (equal to a25a90cf57f0c082cca99602c524085 in hexadecimal). The symbol ∅ represents the empty associated data.
Table 10: Execution time (in seconds) to extract the plaintext (showed in hexadecimal) leading to the same tag (displayed in hexadecimal) as well as the same internal state manipulated before the tag generation produced by ACORN v1 and ACORN v2 knowing the key (equal to 32fd8dc435218dfced3ff439018cfd16 in hexadecimal) and the IV (equal to a25a90e5f57f0c0822a99602e524085 in hexadecimal). The symbol $\emptyset$ represents the empty plaintext.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Tag</th>
<th>Plaintext</th>
<th>Solving time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACORN v1</td>
<td>7d495f7104c1d29652a22b5e362e7f8</td>
<td>$\emptyset$</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td>819179aefff0966d81400db8a3e89bd148c58a658f5977468e8c862ca b267a7f851ed46cccc9cfd145c309acb3188d7012409c5b732e0</td>
<td>$\emptyset$</td>
<td>1.54</td>
</tr>
<tr>
<td>ACORN v2</td>
<td>607aceb1af9bb4b7f19a57b0365034</td>
<td>$\emptyset$</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td>6de7976ff97852770516f495163915de73ad00acfd69550005019d44d3 b6e922aaa8e7f7810823e10ec1114d6d515f0166aaabdc9990928</td>
<td>$\emptyset$</td>
<td>1.55</td>
</tr>
<tr>
<td></td>
<td>7212e6ff941a95aea007ec09df6c3522a405fed67c7d5d86db3a54a45 bd8553b3c6e9711b9c7559f92a3f8fa9d3d9c3f11e441a1e1a32</td>
<td>$\emptyset$</td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td>cc49d7bff0990903d7f129d9db9dd900601a99eb5f9e4ca359a0 db7eaa6ec7b6da70788d126f3a3ecfa9d34d9c9f11e441a1e1a32</td>
<td>$\emptyset$</td>
<td>1.32</td>
</tr>
<tr>
<td>Primitive</td>
<td>Tag</td>
<td>Informative text</td>
<td>Plaintext</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>-----------------</td>
<td>-----------</td>
</tr>
<tr>
<td><strong>ACORN v1</strong></td>
<td>1d60abed41a00f1900a3be8c5e3e4831</td>
<td>Bob gave 5Euro.</td>
<td>426f622067617665203575726f2e9f73bf75e1b3f9b65d6bd1e1e5df7783bcfae41d08451db91f0f4fd9e204e77c7de5793242357d9c0dabf4f2a86fe20bd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bob gave 500Euro.</td>
<td>426f622067617665203530304575726f2e58e23af47c3a4f4e4c7792a8b19d342b8a7e25d957cecf3881f6d2c7a255888830e49b8a29744dec2e54783b8f7f9e</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Eve gave 500Euro.</td>
<td>45766520676176652035304575726f2e4d62ed7d8d72d2a9af789fba2b276eac1d9d90072161642d0b8c80099929c0ff8a5157fc42d79d171d92584e8a1c160</td>
</tr>
<tr>
<td><strong>ACORN v2</strong></td>
<td>f4acf3a1470154a999addb1668b8cbe</td>
<td>Bob gave 5Euro.</td>
<td>426f622067617665203575726f2e46252a2fd2934bb6e12146a7a45dadb1e1c29353e321c0c9ca00ae2ba3644f253cc1595bc9430ad6d50f270747a13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bob gave 500Euro.</td>
<td>426f622067617665203530304575726f2e509e21b0e509823c52f1a1de305745ecc07edbb2a2474b3a3c89add1352e52b63afaad7f038fb8223fac132ce2dce</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Eve gave 500Euro.</td>
<td>4576652067617665203530304575726f2efa807669df93c1d1c7e85556357ed2a869a15b8d53545e9180e967e2f05b2ec43b47e82f19f2a487c4ae10d05a</td>
</tr>
</tbody>
</table>

Table 11: Execution time (in seconds) to extract the plaintext of 64 bytes (shown in hexadecimal) representing an informative text (of length 15 or 17 characters depending on the context) encoded in ASCII leading to the tag (displayed in hexadecimal) produced by ACORN v1 and ACORN v2 knowing the key (equal to 32fd8de435218dfffde3f439018cdf16 in hexadecimal) and the IV (equal to a25a90cdf57ff082eca99602c524085 in hexadecimal). The plaintext related to the informative text is showed in italic.
References


Appendix A

Code 1.1: R code to execute in order to recover the secret state manipulated by ACORN v2

```r
TARGETMAC <- rep("FF", 16) # example of target MAC
SOLVER <- "/cryptominisat" # the path to Cryptominisat
TIMEOUT <- 1000 # the timeout before to stop the execution of Cryptominisat

library("cryptosat")
target <- ACORNv2()
target$setParameter("keyIVsetup",FALSE)
target$setParameter("printPstate",TRUE)
target$setParameter("Aiterations",0)
target$setParameter("Piterations",0)
instance <- target$generateInstance()
for( i in 0:15 ) {
    varname <- paste("mac[",i,"]",sep="")
    instance$setEqual(varname, TARGETMAC[i+1])
}
for( i in 0:292 ) { # each bit of the STATE is encoded with 1 byte
    varname <- paste("pstate[",i,"]",sep="")
    instance$setEqual(varname, xbits=1:7, y="0000000", yformat="bin")
}
solution <- instance$solveWith(SOLVER,timeout=TIMEOUT)
if(!solution$isSAT()){
    cat("State not recovered\n")
} else{
    cat("State:")
    for( i in c(0:292)){
        cat(solution$getValueOf(paste("pstate[",i,"]",sep=""))," ")
    }
}
```
Code 1.2: R code to execute in order to recover the key and the IV knowing the state of Acorn v2 as well as the plaintext and the ciphertext

```r
SOLVER <- "/cmsat-2.9.5/build/cryptominisat" # the path to cryptominisat
TIMEOUT <- 200 # the timeout before to stop the execution of cryptominisat
STATE <- sample(c("00","01"),293,replace=T) # example with a random state
PLAINTEXT <- sample(c("00","01"),8,replace=T) # example with random plaintext
CIPHERTEXT <- sample(c("00","01"),8,replace=T) # example with random ciphertext

mlen <- length(PLAINTEXT)
library("cryptosat")
target <- ACORNv2()
target$setParameter("mlen",mlen)
target$setParameter("printPstate",TRUE)
target$setParameter("processTag",FALSE)
instance <- target$generateInstance()
if( mlen != 0){
  for ( i in 0:c(mlen -1) ) {
    instance$setEqual(paste("m[",i,"]",sep=""), PLAINTEXT[i+1])
    instance$setEqual(paste("c[",i,"]",sep=""), CIPHERTEXT[i+1])
  }
}
for( i in 0:292 ) {
  instance$setEqual(paste("pstate[",i,"]",sep=""), STATE[i+1])
}
solution <- instance$solveWith(SOLVER,timeout=TIMEOUT)
if(solution$isSAT()){  
cat("Key : ",paste(solution$getValueOf("key"),collapse=""),"
")
cat("IV : ",paste(solution$getValueOf("iv"),collapse=""),"
")
}
```

TARGETMAC <- rep("FF", 16)  # example of target MAC
SOLVER <- "/cmsat-2.9.5/build/cryptominisat"  # the path to Cryptominisat
TIMEOUT <- 1000  # the timeout before to stop the execution of Cryptominisat
NUMBEROFSTATES <- 4  # number of different states to recover (max: 294)

library("cryptosat")
target <- ACORNv2()
target$setParameter("keyIVsetup",FALSE)
target$setParameter("printPstate",TRUE)
target$setParameter("Aiterations",0)
target$setParameter("Piterations",0)
instance <- target$generateInstance()
for(i in 0:15) {
  varname <- paste("mac[",i,"],",sep="")
  instance$setEqual(varname, TARGETMAC[i+1])
}
for(sample in c(0:(NUMBEROFSTATES-1))) {
  for(i in 0:292) {
    varname <- paste("pstate[",i,"],",sep="")
    if(i== sample-1){
      instance$setEqual(varname, xbits=1:8, y="00000001", yformat="bin")
    } else{
      instance$setEqual(varname, xbits=1:7, y="0000000", yformat="bin")
    }
  }
  solution <- instance$solveWith(SOLVER, timeout=TIMEOUT)
  if(!solution$isSAT()){cat("State not recovered\n")
    }
  else{
    cat("State:")
    for(i in c(0:292)){
      cat(solution$getValueOf(paste("pstate[",i,"],",sep=""))," ")
    }
    cat("\n")
  }
}
Code 1.4: R code to execute in order to recover several plaintexts leading to the same tag provided by Acorn v2 when there is no plaintext (without exploiting a fixed point in the state transition function)

```r
SOLVER <- "/cmsat-2.9.5/build/cryptominisat" # the path to Cryptominisat
NUMBEROFPLAINTEXTS <- 4 # number of different plaintexts to recover (max: 294)
KEY <- rep("FF", 16) # example of key
IV <- rep("FF", 16) # example of IV
MLEN <- 56 # example of size of the recovered plaintexts

library("cryptosat")
target <- ACORNv2()
target$setParameter("mlen",0)
instance <- target$generateInstance()
for( i in 0:15 ) {
  instance$setEqual(paste("key[",i,"],",sep=""), KEY[i+1])
  instance$setEqual(paste("iv[",i,"],",sep=""), IV[i+1])
}
solution <- instance$solveWith(SOLVER)
mac <- solution$getValueOf("mac")
cat("MAC: ", paste(mac, collapse=""),"\n")
for(sample in c(2:NUMBEROFPLAINTEXTS)) {
  target <- ACORNv2()
  target$setParameter("mlen",MLEN)
  instance <- target$generateInstance()
  for( i in 0:15 ) {
    instance$setEqual(paste("key[",i,"],",sep=""), KEY[i+1])
    instance$setEqual(paste("iv[",i,"],",sep=""), IV[i+1])
    instance$setEqual(paste("mac[",i,"],",sep=""), mac[i+1])
  }
  instance$setEqual(paste("m[",sample-2,"],",sep=""), "ff")
solution <- instance$solveWith(SOLVER, timeout=1000)
  if(solution$isSAT()){
    m <- paste(solution$getValueOf("m"), collapse="")
    cat("Plaintext: ",m,"\n")
    cat("MAC: " , paste(solution$getValueOf("mac"), collapse=""),"\n")
  }
}
```
Code 1.5: R code to execute in order to recover several plaintexts leading to the same tag provided by Acorn v2 when there is no plaintext (exploiting a fixed point in the state transition function)

```r
SOLVER <- "/cmsat-2.9.5/build/cryptominisat" # the path to Cryptominisat
NUMBEROFPLAINTEXTS <- 4 # number of different plaintexts to recover (max: 294)
KEY <- rep("FF", 16) # example of key
IV <- rep("FF", 16) # example of IV
MLEN <- 56 # example of size of the recovered plaintexts

library("cryptosat")
target <- ACORNv2()
target$setParameter("mlen",0)
target$setParameter("printPstate",TRUE)
instance <- target$generateInstance()
for( i in 0:15 ) {
    instance$setEqual(paste("key[",i,"],",sep=""), KEY[i+1])
    instance$setEqual(paste("iv[",i,"],",sep=""), IV[i+1])
}
solution <- instance$solveWith(SOLVER)
mac <- solution$getValueOf("mac")
pstate <- solution$getValueOf("pstate")
cat("MAC: ", paste(mac,collapse=""),"\n")
for(sample in c(2:NUMBEROFPLAINTEXTS)){
    target <- ACORNv2()
    target$setParameter("mlen",MLEN)
target$setParameter("printPstate",TRUE)
    instance <- target$generateInstance()
    for( i in 0:15 ) {
        instance$setEqual(paste("key[",i,"],",sep=""), KEY[i+1])
        instance$setEqual(paste("iv[",i,"],",sep=""), IV[i+1])
    }
    for( i in 0:292 ) {
        instance$setEqual(paste("pstate[",i,"],",sep=""), pstate[i+1])
    }
    instance$setEqual(paste("m[",sample-2,"],",sep=""), "ff")
solution <- instance$solveWith(SOLVER,timeout=1000)

    if(solution$isSAT()){ 
        m <- paste(solution$getValueOf("m"),collapse="")
cat("Plaintext:" ,m," \n")
cat("MAC: ",paste(solution$getValueOf("mac"),collapse=""),"\n")
    }
}
```