Attacks on the DECT authentication mechanisms

Stefan Lucks[1], Andreas Schuler[2], Erik Tews[3], Ralf-Philipp Weinmann[4], and Matthias Wenzel[5]
<Stefan.Lucks@uni-weimar.de>, <krater@ccc-trier.de>,
<e_tews@cdc.informatik.tu-darmstadt.de>,
<ralf-philipp-weinmann@uni.lu>, <mazzoo@mazzoo.de>

[1] FB Informatik, TU Darmstadt
[2] FSTC, University of Luxembourg
[3] Chaos Computer Club Trier, Germany
[4] Bauhaus-University Weimar, Germany
[5] Chaos Computer Club München, Germany

Abstract. Digital Enhanced Cordless Telecommunications (DECT) is a standard for connecting cordless telephones to a fixed telecommunications network over a short range. The cryptographic algorithms used in DECT are not publicly available. In this paper we reveal one of the two algorithms used by DECT, the DECT Standard Authentication Algorithm (DSAA). We give a very detailed security analysis of the DSAA including some very effective attacks on the building blocks used for DSAA as well as a common implementation error that can practically lead to a total break of DECT security. We also present a low cost attack on the DECT protocol, which allows an attacker to impersonate a base station and therefore listen to and reroute all phone calls made by a handset.

1 Introduction

Digital Enhanced Cordless Telecommunications (DECT) is a standard for connecting cordless telephones to a fixed telecommunications network. It was standardized in 1992 by the CEPT, a predecessor of the ETSI (European Telecommunications Standards Institute) and is the de-facto standard for cordless telephony in Europe today. For authentication and privacy, two proprietary algorithms are used: The DECT Standard Authentication Algorithm (DSAA) and the DECT Standard Cipher (DSC). These algorithms have thus far only been available under a Non-Disclosure Agreement from the ETSI and have not been subject to academic scrutiny. Recently, the DSAA algorithm was reverse engineered by the authors of this paper to develop an open-source driver for a PCMCIA DECT card.

Our contribution: This paper gives the first public description of the DSAA as well as cryptanalytic results on its components. Furthermore we show two types of flaws that result in practical attacks against DECT implementations. One is a protocol flaw in the authentication mechanism, the other is a combination of a common implementation error combined with a brittle protocol deisgn.

The paper is structured as follows: Section 2 describes the authentication methods used in DECT. Section 3 details how easily a DECT base station can be impersonated in practice and outlines the consequences. Section 4 describes the DECT Standard Authentication Algorithm and explains how a weak PRNG can lead to a total break of DECT security. Section 5 presents the first public analysis of the DSAA. We conclude the paper in Section 6.

1.1 Notation and Conventions

We use **bold font** for variable names in algorithm descriptions as well as for input and output parameters. Hexadecimal constants are denoted with their least significant byte first in a `typewriter font`. For example, if all bits of the variable **b** are 0 except for a single bit, we write `0100` if $\mathbf{b}[0] = 1$, `0200` if $\mathbf{b}[1] = 1$, `0001` if $\mathbf{b}[8] = 1$ and `0080` if $\mathbf{b}[15] = 1$. Function names are typeset with a sans-serif font.

Function names written in capital letters like A11 are functions that can be found in the public DECT standard [5]. Conversely function names written in lowercase letters like step1 have been introduced by the authors of this paper. Functions always have a return value and never modify their arguments.

To access a bit of an array, the $[\cdot]$ notation is used. For example $\mathsf{foo}[0]$ denotes the first bit of the array foo. If more than a single bit, for example a byte should be extracted, the $[\cdot\ \ldots\ \cdot]$ notation is used. For example $\mathsf{foo}[0 \ldots 7]$ extracts the first 8 bits in foo, which is the least significant byte in foo.

To assign a value in pseudocode, the $\leftarrow$ operator is used. Whenever the operators $+$ and $*$ are used in pseudocode, they denote addition and multiplication modulo 256. For example $\mathsf{foo}[0 \ldots 7] \leftarrow \mathsf{bar}[0 \ldots 7] * \mathsf{barn}[0 \ldots 7]$ multiplies the first byte in bar with the first byte in barn, reduces this value modulo 256 and stores the result in the first byte of foo.

If a bit or byte pattern is repeated, the $(\cdot)^{\cdot}$ notation can be used. For example instead of writing `aabbaabbaabb`, we can write $(\mathtt{aabb})^3$. For concatenating two values, the $||$ operator is used. For example $\mathsf{aa}||\mathsf{bb}$ results in `aabb`.

1.2 Additional terminology

In the following we will use the terminology of the DECT standards [5,4]. To make this paper self-contained, we briefly explain the most important terms: A FT is fixed terminal, also called base station. A PT is a portable terminal, e.g. a telephone or handset. The Radio Fixed Party Identity (RFPI) is a 40-bit identifier of an FT. A PT is identified by a International Portable User Identity (IPUI), a value similar to the RFPI of variable length. In challenge/response authentications, responses are named RES1 or RES2. The value received during the authentication is called SRES1 or SRES2, and the value calcluated by the station (expected as a response) is called XRES1 or XRES2.

2 Authentication in DECT

The public standard describing the security features of DECT specifies four different authentication processes A11, A12, A21 and A22. These four processes are used for both authentication and key derivation and make use of an authentication algorithm A. DECT equipment conforming to the GAP standard [4] must support the DSAA to achieve vendor interoperability.

The algorithms A11 and A12 are used during the authentication of a PT. They are also used to derive a key for the DSC and to generate keying material during the initial pairing of a PT with a FT. The algorithms A21 and A22 are only used during the authentication of a FT. Furthermore the processes A11, A12, A21 and A22 are used to pair a PT with a FT.

2.1 Keys used in DECT

In most cases of residential DECT usage, the user buys a DECT FT, and one or more DECT PTs. The first step then is to pair the PTs with the FT, unless they have been bought as a bundle and the pairing was already completed by the manufacturer. This procedure is called key allocation in the DECT standards and described in more detail in Section 2.5. After this process, every DECT PT shares a 128 bit secret key with the FT, called the UAK.

In all scenarios we have seen so far, the UAK was the only key used to derive any other keys, but the DECT standard allows two alternative options:

- The UAK is used together with a UPI, a short 16-32 bit secret, manually entered by the user of the PT. The UAK und UPI are then used to derive a key.
- No UAK is used. Instead a short 16-32 bit secret called AC is entered by the user, which forms the key. The DECT standard suggests that the AC should only be used, if a short term coupling between PT and FT is required.

For the rest of this paper we will mainly focus on the first case, where the only the UAK is used.

2.2 Authentication of a PT and derivation of the DSC key

When a PT needs to authenticate itself against a FT, the procedures A11 and A12 are used. The FT generates two random 64 bit values RS and RAND_F and sends them to the PT.

The PT uses the A11 algorithm, which takes a 128 bit key UAK and a 64 bit value RS as input, and generates a 128 bit output KS, which is used as an intermediate key. The PT then uses the A12 algorithm, which takes KS and

RAND_F as input and produces two outputs: a 32 bit response called SRES1 and a 64 bit key DCK, which can be used for the DSC. SRES1 then is sent to the FT.

The same computation is done on the FT too, except here the first output of A12 is called XRES1 instead of SRES1. The FT receives the value SRES1 from the PT and compares it with his own value XRES1. If both are equal, the PT is authenticated.

2.3 Authentication of a FT

When a FT needs to authenticate itself against a PT, a similar procedure is used.

First the PT generates a 64 bit value RAND_P and sends it to a FT. Then the FT generates a 64 bit value RS and uses A21 to compute a 128 bit intermediate key KS from UAK and RS. Now A22 is used to compute a 32 bit response SRES2 from KS and RAND_P. A22 only generates SRES2 and no key for the DSC. The FT sends SRES2 and KS to the PT.

After having received KS, the PT can do the same computations. Here the output of A22 is called XRES2. The PT now compares XRES2 with the received value SRES2. If both are equal, the FT is authenticated.

This protocol might seem odd at the first look. As far as we know, the design goal was to build a protocol where a PT can be used in a roaming scenario, similar to GSM. Here, the home network provider could hand over a couple of (RS, KS) pairs to the partner network, which can then allow a PT to operate without having to know the UAK.

2.4 Mutual Authentication

The standard specifies different methods of mutual authentication:

- A direct method, which simply consists of executing A11 and A12 to authenticate the PT first followed by A21 and A22 to authenticate the FT.
- Indirect methods which involve a one-sided authentication of the PT together with a cipher key derivation that is used for data confidentiality.

However, even though the indirect methods are recommended for all applications except for local loop installations (see the reference configurations in Appendix F.2 of [5]), they are inherently flawed as they do not provide a mutual authentication at all. This indirect method is reminiscent of the case of GSM [1]. A derived cipher key does not necessarily have to be used, a FT may simply send a message indicating that it does not support encryption – it is an optional feature in the GAP standard. Moreover, even if encryption is enabled, being able to transmit encrypted messages under a derived key does not proof possession of this key: The FT may just replay authentication challenges, subsequently replaying encrypted messages that were previously recorded.

2.5 Key allocation

Most DECT systems allow an automatic pairing process. To initiate pairing the user switches both a PT and a FT to a dedicated pairing mode and enters the same PIN number on both devices[6]. This step needs to be repeated with all DECT PT devices. Each PT performs a handshake with the FT and a mutual authentication using the DSAA algorithm and the PIN as a shared secret is performed. During this handshake, three 64-bit random numbers are generated. However only a single 64 bit random number RS sent by the FT is used together with the PIN to generate the 128 bit UAK. For a 4 digit PIN number, there are only $2^{77.288}$ possible values for the UAK. If a flawed random number generator is used on the FT for which an attacker can predict the subset of random numbers generated during key allocation, the number of possibe UAKs shrinks accordingly. This can be exploited by sniffing challenge-response pairs ((RAND_F, RS), SRES1) at any time after key allocation that can be used as 32-bit filters. In practice we did indeed find weak PRNGs implemented in the firmware of several base stations – across a variety of vendors – in one specific case only providing 24 bits of entropy for the 64 bit value RS. This leads to a very practical and devastating attack against DECT PTs using vulnerable DECT stacks.

3 Impersonating a Base Station

As described in the previous section, in most cases authentication of the FT is optional. This makes DECT telephones vulnerable to a very simple, yet effective and practical attack: An attacker impersonates a DECT FT by spoofing its RFPI and faking the authentication of the PT. This is done by sending out random values RAND_F and RS for which any response SRES1 is accepted. Subsequently the impersonating FT simply rejects any attempts to do cipher mode switching. This technique is significantly simpler to implement than a protocol reflection attack and has been verified to work in practice by us.

We implemented this attack in practice by modifying the driver of a PCMCIA DECT card. The drivers and firmware for this card do not support the DECT Standard Cipher. Furthermore, the frames are completely generated in software which allows us to easily spoof the RFPI of another base station. Upon initialization of the card, the RFPI was read from the card and written to a structure in memory. We patched the driver such that the RFPI field in this structure was overwritten with an assumed RFPI value of our choosing directly after the original value was written there. Then we modified the routine comparing the RES values returned by the PT with the computed XRES values. We verified that we were indeed broadcasting a fake RFPI with a USRP [3] and a DECT sniffer that was written for the GNURadio framework by the authors.

For our lab setup, we used an ordinary consumer DECT handset paired to a consumer base station. We set the modified driver of our PCMCIA card

[6] Some DECT FTs are shipped with a fixed default PIN number – usually specified in the manual – which user has to enter as given on the DECT PT.

to broadcast the RFPI of this base station and added the IPUI of the phone to the database of registered handsets of the card. The device key was set to an arbitrary value. After jamming the DECT over-the-air communication for a short time, the handset switched to our faked base station with a probability of about 50%. From this point on, every call made by the phone was handled by our PCMCIA hard, and we where completely able to trace all communications and reroute all calls. No warning or error message was displayed on the telephone. Both the handset and the base station where purchased in 2008, which shows that even current DECT phones do not authenticate base stations and also do not force encrypted communication.

This attack shows that it is possible to intercept, record and reroute DECT phonecalls with equipment as expensive as a wireless LAN card, making attacks on DECT as cheap as on wireless LANs. Subsequently we also succeeded in converting this card to a passive sniffing device with a custom-written Linux and firmware[7].

4 The DECT Standard Authentication Algorithm

The algorithms A11, A12, A21, and A22 can be seen as wrappers around an algorithm, we call DSAA. The algorithm DSAA accepts an 128 bit key and a 64 bit random as input and produces a 128 bit output. This output is now modified as follows:

- A11 just returns the whole output of DSAA, without any further modification.
- A21 behaves similar to A11, but here, every second bit of the output is inverted, starting with the first bit of the output. For example if the first byte of output of DSAA is ca, then the first byte of output of A21 is 60.
- A22 just returns the last 4 bytes of output of DSAA as RES.
- A12 is similar to A22, except here, the middle 8 bytes of DSAA are returned too, as DCK.

5 Security analysis of the DECT Authentication

DSAA is surprisingly insecure. The middle 64 bits of the output of DSAA only depend on the middle 64 bits of the key. This allows trivial attacks against DSAA, which allow the recovery of all 128 secret key bits with an effort in the magnitude of about 2^{64} evaluations of DSAA. Even if attacks against the DSAA cannot improved past this bound, keep in mind the entropy problems of the random number generators that we found and described in Section 2.5.

Besides that, the security of DSAA mainly relies on the security of the cassable block cipher. Our analysis of cassable showed that cassable is surprisingly weak too.

[7] this software is available at `http://www.dedected.org`

Fig. 1. The four DSAA algorithms

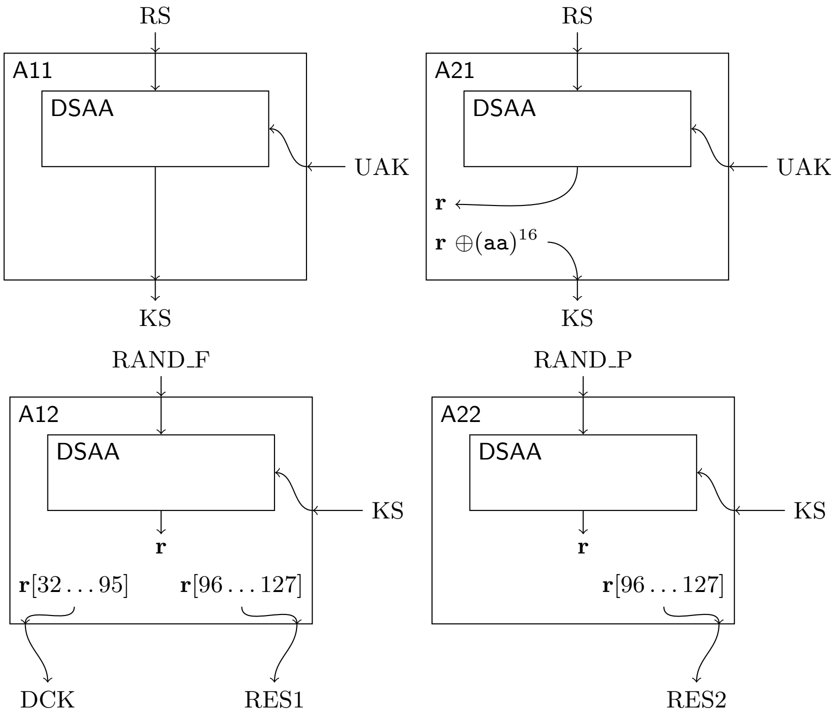

RS
A11
DSAA
UAK
KS
RS
A21
DSAA
UAK
$\mathbf{r}$
$\mathbf{r} \oplus (\mathsf{aa})^{16}$
KS
RAND_F
A12
DSAA
KS
$\mathbf{r}$
$\mathbf{r}[32 \ldots 95]$
$\mathbf{r}[96 \ldots 127]$
DCK
RES1
RAND_P
A22
DSAA
KS
$\mathbf{r}$
$\mathbf{r}[96 \ldots 127]$
RES2

Fig. 2. DSAA overview

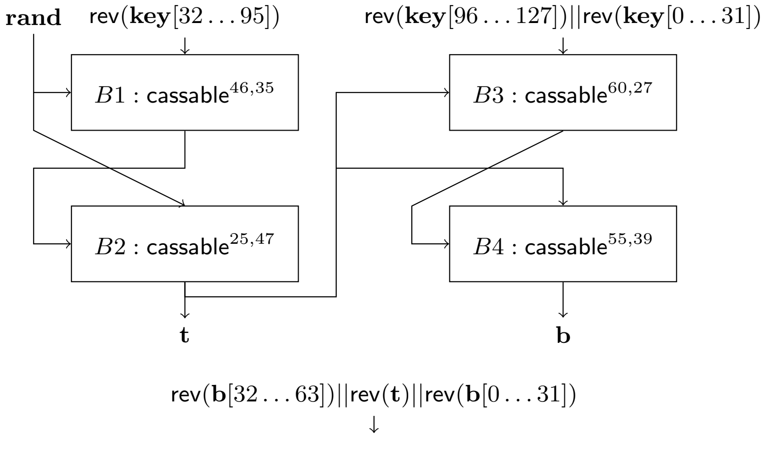

$\mathbf{rand}$
$\mathsf{rev}(\mathbf{key}[32 \ldots 95])$
$\mathsf{rev}(\mathbf{key}[96 \ldots 127])||\mathsf{rev}(\mathbf{key}[0 \ldots 31])$
$B1 : \mathsf{cassable}^{46,35}$
$B3 : \mathsf{cassable}^{60,27}$
$B2 : \mathsf{cassable}^{25,47}$
$B4 : \mathsf{cassable}^{55,39}$
$\mathbf{t}$
$\mathbf{b}$
$\mathsf{rev}(\mathbf{b}[32 \ldots 63])||\mathsf{rev}(\mathbf{t})||\mathsf{rev}(\mathbf{b}[0 \ldots 31])$
↓

Table 1. The DSAA S-Box (**sbox**)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	b0	68	6f	f6	7d	e8	16	85	39	7c	7f	de	43	f0	59	a9
10	fb	80	32	ae	5f	25	8c	f5	94	6b	d8	ea	88	98	c2	29
20	cf	3a	50	96	1c	08	95	f4	82	37	0a	56	2c	ff	4f	c4
30	60	a5	83	21	30	f8	f3	28	fa	93	49	34	42	78	bf	fc
40	61	c6	f1	a7	1a	53	03	4d	86	d3	04	87	7e	8f	a0	b7
50	31	b3	e7	0e	2f	cc	69	c3	c0	d9	c8	13	dc	8b	01	52
60	c1	48	ef	af	73	dd	5c	2e	19	91	df	22	d5	3d	0d	a3
70	58	81	3e	fd	62	44	24	2d	b6	8d	5a	05	17	be	27	54
80	5d	9d	d6	ad	6c	ed	64	ce	f2	72	3f	d4	46	a4	10	a2
90	3b	89	97	4c	6e	74	99	e4	e3	bb	ee	70	00	bd	65	20
a0	0f	7a	e9	9e	9b	c7	b5	63	e6	aa	e1	8a	c5	07	06	1e
b0	5e	1d	35	38	77	14	11	e2	b9	84	18	9f	2a	cb	da	f7
c0	a6	b2	66	7b	b1	9c	6d	6a	f9	fe	ca	c9	a8	41	bc	79
d0	db	b8	67	ba	ac	36	ab	92	4b	d7	e5	9a	76	cd	15	1f
e0	4e	4a	57	71	1b	55	09	51	33	0c	b4	8e	2b	e0	d0	5b
f0	47	75	45	40	02	d1	3c	ec	23	eb	0b	d2	a1	90	26	12

5.1 The cassable block cipher

The DSAA can be interpreted as a cascade of four very similarly constructed block ciphers. We shall call this family of block ciphers cassable. A member of this family is a substitution-linear network parametrized by two parameters that only slightly change the key scheduling. The block cipher uses 6 rounds, each of the rounds performing of a key addition, applying a bricklayer of S-Boxes and a mixing step in sequence. The last round is not followed by a final key addition, so that the last round is completely invertible, besides the key addition. This reduces the effective number of rounds to 5.

In the following we will describe how the cassable block ciphers are constructed. The functions $\sigma_i : GF(2)^{64} \rightarrow GF(2)^{64}$ with $1 \leq i \leq 4$ denoting bit permutations that are used to derive the round keys from the cipher key. The function $\lambda_i : (\mathbb{Z}/256\mathbb{Z})^8 \rightarrow (\mathbb{Z}/256\mathbb{Z})^8$ denotes the mixing functions used in the block ciphers, the function $\gamma : GF(2)^{64} \rightarrow GF(2)^{64}$ is a bricklayer transform that is defined as:

$$\gamma(A||B||\cdots||A) = \rho(A)||\rho(B)||\cdots||\rho(H)$$

with $A, B, \ldots, H \in GF(2)^8$ and $\rho : GF(2)^8 \rightarrow GF(2)^8$ denoting the application of the invertible S-Box that is given in Table 1. The linear transforms perform butterfly-style mixing:

$$\lambda_1 : (A, \ldots, H) \mapsto (2A+E, 2B+F, 2C+G, 2D+H, 3E+A, 3F+B, 3G+C, 3H+D)$$
$$\lambda_2 : (A, \ldots, H) \mapsto (2A+C, 2B+D, 3C+A, 3D+B, 2E+G, 2F+H, 3G+E, 3H+F)$$
$$\lambda_3 : (A, \ldots, H) \mapsto (2A+B, 3B+A, 2C+D, 3D+C, 2E+F, 3F+E, 2G+H, 3H+G)$$

The round keys $K_i \in GF(2)^{64}$ with $1 \leq i \leq 6$ are iteratively derived from the cipher key $K_0 \in GF(2)^{64}$ using the following parametrized function $\sigma_{(m,l)}$:

$$\sigma_{(m,l)} : (k_0, \ldots, k_{63}) \mapsto (k_m, k_{(m+l) \bmod 64}, k_{(m+2l) \bmod 64}, \ldots, k_{(m+63l) \bmod 64})$$

by simply applying a $\sigma_{(m,l)}$ to the cipher key i times:

$$K_i = \sigma^i_{(l,m)}(K)$$

The individual bytes of the key K_i can be accessed by $K_{i,A}$ to $K_{i,H}$.

To be able to compose the round function, we identify the elements of the vector space $GF(2)^8$ with the elements of the ring $\mathbb{Z}/256\mathbb{Z}$ using the canonical embedding. Given a fixed σ, the round function f_r for round r with $1 \leq r \leq 6$ that transforms a cipher key K and a state X into the state of the next round then looks as follows:

$$f_r : (X, K) \mapsto \lambda_{(((r-1) \bmod 3)+1)}(X \oplus \sigma^r(K))$$

The mixing layer To diffuse local changes in the state bits widely, the functions λ_i with $1 \leq i \leq 3$ (lambda1, lambda2, and lambda3 in the pseudo-code) are used. These form a butterfly network. At first look, it seems that full diffusion is achieved after the third round, because every byte of the state depends on every other byte of the input at this point. However, we made an interesting observation: The λ_i functions only multiply the inputs with either the constant 2 or 3. This means that for the components of the output vector formed as

$$c = (a * 2 + b) \bmod 256$$

the lowestmost bit of c will be equal to the lowestmost bit of b and not depend on a at all. This observation will be used in Section 5.2.

The S-Box The DSAA S-Box has a tendency towards flipping the lowest bit. If a random input is chosen, the lowest bit of the output will equal to the lowest bit of the input with a probability of $\frac{120}{256}$. For up to three rounds we were able to find exploitable linear approximations depending on the lowest bits of the input bytes, the lowest bits of the state and various bits of the key. Although this sounds promising, the linear and differential properties of the S-Box are optimal. Interpolating the S-Box over $GF(2^8)$ yields dense polynomials of degree 254, interpolation over $GF(2)$ results in equations of maximum degree.

The key scheduling The key bit permutation used in the key scheduling is not optimal for the cassable ciphers used in DSAA. Although the bit permutation could have a maximum order of 64, a lower order was observed for the cassable ciphers instantiated, namely 8 and 16.

5.2 A practical attack on cassable

The individual block ciphers used within the DSAA can be fully broken using differential cryptanalysis [2] with only a very small number of chosen plaintexts. Assume that we have an input $m = m_A||m_B||m_C||m_D||m_E||m_F||m_G||m_H$ with $m_i \in \{0,1\}^8$ and a second input $m' = m'_A||m'_B||m'_C||m'_D||m'_E||m'_F||m'_G||m'_H$ where every second byte is the same, i. e. $m_B = m'_B, m_D = m'_D, m_F = m'_F$, and $m_H = m'_H$. Now both inputs are encrypted. Let $s = s_{i,A}||\dots||s_{i,H}$ and $s' = s'_{i,A}||\dots||s'_{i,H}$ be the states after i rounds of cassable. After the first round, $s_{1,B} = s'_{1,B}, s_{1,D} = s'_{1,D}, s_{1,F} = s'_{1,F}$, and $s_{1,H} = s'_{1,H}$ holds. This equality still holds after the second round. After the third round, the equality is destroyed, but $s_{3,A} \equiv s'_{3,A} \bmod 2, s_{3,C} \equiv s'_{3,C} \bmod 2, s_{3,E} \equiv s'_{3,E}$, and $s_{3,G} \equiv s'_{3,G} \bmod 2$ holds. The key addition in round four preserves this property, with only the fourth application of the S-Box $\rho_{4,j}$ destroying it.

An attacker can use this to recover the secret key of the cipher. Assume the attacker is able to encrypt two such messages m and m' with the same secret key and see the output. He can invert the lambda3 and gamma steps of the last round, because they are not key-dependent. To recover the value of $s_{3,A} \oplus K_{4,A}$ and $s_{3,E} \oplus K_{4,E}$, he only needs 32 round key bits of round key 6 which are added to $s_{5,A}, s_{5,C}, s_{5,E}$, and $s_{5,G}$, and 16 round key bits of round key 5, which are added to $s_{4,A}$ and $s_{4,E}$. Due to overlaps in the round key bits these are only 38 different bits for cassable^{46,35}, 36 different bits for cassable^{25,47}, 42 different bits for cassable^{60,27}, and 40 different bits for cassable^{55,39}. After the attacker has recovered $s_{3,A} \oplus K_{4,A}, s_{3,E} \oplus K_{4,E}, s'_{3,A} \oplus K_{4,A}$ and $s'_{3,E} \oplus K_{4,A}$, he checks whether $s_{3,A} \oplus K_{4,A} = s'_{3,A} \oplus K_{4,A} \bmod 2$ and $s_{3,E} \oplus K_{4,E} = s'_{3,E} \oplus K_{4,E} \bmod 2$ holds. If at least one of the conditions is not satisfied, he can be sure that his guess for the round key bits was wrong. Checking all possible values for these round key bits will eliminate about $\frac{3}{4}$ of the key space with computational costs of about 2^k invocations of cassable, if there are k different key bits for the required round key parts of round key 5 and 6.

After having eliminated 75% of the key space, an attacker can repeat this with another pair on the remaining key space and eliminate 75% of the remaining key space again. Iterating this procedure with a total of 15 pairs, only about 2^{34} possible keys are expected to remain. These can then be checked using exhaustive search. The total workload amounts to $2^k + \frac{1}{4}2^k + \frac{1}{16}2^k + \frac{1}{64}2^k + \ldots + 2^{34}$ block cipher invocations which is bounded by $1.5834 \cdot 2^k$ for $k \geq 36$. For cassable^{25,47}, this would be about $2^{36.7}$.

An efficient implementation needs only negligible memory when every possible value of the k round key bits is enumerated and every combination is checked against all available message pairs. Only the combinations which pass their tests against all available pairs are saved, which should be about 2^{k-30}.

If the attacker can choose the input for cassable, he can choose 16 different inputs, where every second byte is set to an arbitrary constant. If the attacker can only observe random inputs, he can expect to find a pair in whcih every second byte is the same after 2^{16} random inputs. After $4 \cdot 2^{16}$ inputs, the expected number of pairs is about $4 \cdot 4 = 16$, which is sufficient for the attack. If not enough pairs are available to the attacker, the attack is still possible, however with increased computational effort and memory usage.

5.3 A Known-Plaintext Attack on Three Rounds Using a Single Plaintext/Ciphertext Pair

Three rounds of the cassable block cipher can be attacked using a single plaintext/ciphertext pair. This is of relevance as attacking B_4 or B_2 allows us to invert the preceding ciphers B_1 and B_3.

Assume a plaintext $m = m_A||m_B||m_C||m_D||m_E||m_F||m_G||m_H$ encrypted over three rounds. The output after the third round then is $S_3 = s_{3,A}, \ldots, s_{3,H}$. As in the previous attack, we can invert the diffusion layer λ_3 and the S-Box layer ρ without knowing any key bits, obtaining $(z_0, \ldots, z_7) := S_2 \oplus K_3$ with $z_i \in GF(2)^8$ for $0 \leq i < 8$. At this point the diffusion is not yet complete. For instance, the following relation holds for z_0:

$$\begin{aligned} z_0 = \rho((2 \cdot \rho(m_0 \oplus K_{1,A}) + \rho(m_4 \oplus K_{1,E})) \oplus K_{2,A}) + \\ \rho((2 \cdot \rho(m_2 \oplus K_{1,C}) + \rho(m_6 \oplus K_{1,G})) \oplus K_{2,C}) \oplus K_{3,A} \end{aligned}$$

Due to overlaps in the key bits, for the block cipher B_1 the value z_0 then only depends on 41 key bits, for B_2 on 36 key bits, for B_3 on 44 key bits and for B_4 on 46 key bits.

We can use the equations for the z_i as a filter which discard $\frac{255}{256}$ of the searched key bit subspace.

In the following, we give an example of how the attacks works for B2: Starting with z_0, we expect 2^{28} key bit combinations after the filtering step. Interestingly, the key bits involved in z_0 for B2 are the same as for z_2, so we can use this byte to filter down to about 2^{20} combinations. Another filtering step using both z_4 and z_6 will just cost us an additional 4 key bits, meaning we can filter about 2^{24} combinations down to about 2^8. All of these filtering steps can be chained without storing intermediate results in memory, making the memory complexity negligible.

For the remaining combinations we can exhaustively search through the remaining 24 key bits, giving a 2^{32} work factor. The overall cost of the attack is dominated by the first filtering step however, which means the attack costs about 2^{36} cassable invocations.

For B4, the key bit permutations work against our favor: After filtering with z_0 we expect 2^{38} key bit combinations to remain. Subsequently we filter with z_2, which causes another 6 key bits to be involved (z_4 and z_6 would involve 10 more key bits). This yields 2^{36} key bit combinations. Subsequently filtering with z_4 involves 8 more key bits, causing the number of combinations to stay at 2^{36}.

Finally we can filter with z_6, which adds 4 more key bits, bringing the number of combinations down to 2^{32}. As there are no more unused key bits left, we can test all of the 2^{32} key candidates. The total cost for this attack is again dominated by the first filtering step which requires 2^{46} cassable invocations. Again the attack can be completed using negligible memory by chaining the filtering conditions.

The attacks on B2 and B4 can be used to attack a reduced version of the DSAA where B1 and B3 are 6 round versions of cassable and B2 and B4 are reduced to three rounds. An attack on this reduced version costs approximately 2^{44} invocations of the reduced DSAA since approximately three 6 round cassable invocations are used per DSAA operation.

6 Conclusion

We have shown the first public description of the DSAA algorithm, which clearly shows that the algorithm only provides at most 64 bit of symmetric security. An analysis using only the official documents published by ETSI would not have revealed these information.

We could also show that the building blocks used for the DSAA have some serious design flaws, which might allow attacks with a complexity below 2^{64}. Especially the block cipher used in DSAA seems to be weak and can be completely broken using differential cryptanalysis.

Although 64 bit of symmetric security might be sufficient to hold off unmotivated attackers, most of the currently deployed DECT systems might be much easier attackable, because encryption and an authentication of the base station is not always required. This allows an attacker spending about 30$ for a PCMCIA card to intercept most DECT phone calls and totally breach the security architecture of DECT.

Currently, we see two possible countermeasures. First, all DECT installations should be upgraded to require mutual authentication and encryption of all phone calls. This should only be seen as a temporary fix until a better solution is available.

A possible long term solution would be an upgrade of the DECT security architecture to use *public* well analyzed methods and algorithms for key exchange and traffic encryption and integrity protection. A possible alternative could be IEEE 802.11 based Voice over IP phone systems, where networks can be encrypted using WPA2. These systems are currently more costly than DECT installations and still more difficult to configure than DECT phones for a novice user, but encrypt and protect all calls and signaling informations using AES-CCMP and allow a variety of different protocols for the key exchange. However it is open to debate whether these systems can provide a viable alternative to DECT systems because of their different properties in term of power consumption, radio spectrum and quality of service provided.

We would like to thank all the people who supported and helped us with this paper, especially those, whose names are not mentioned in this document.

References

1. Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 600–616. Springer, 2003.
2. Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO 1990*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1991.
3. Matt Ettus. *USRP user's and developer's guide.* Ettus Research LLC, February 2005.
4. European Telecommunications Standards Institute. ETSI EN 300 444 V1.4.2 (2003-02): Digital Enhanced Cordless Telecommunications (DECT); Generic Access Profile, February 2003.
5. European Telecommunications Standards Institute. ETSI EN 300 175-7 V2.1.1: Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 7: Security Features, August 2007.

A Pseudocode for the DSAA

The DSAA (see Algorithm 1) uses four different 64 bit block cipher like functions as building blocks. DSAA takes a random value **rand** $\in \{0,1\}^{64}$ and a key **key** $\in \{0,1\}^{128}$ as input and splits the 128 bit key into two parts of 64 bit. The first part of the key are the 64 middle bits of the key. DSAA calls the step1 function with the random value and the first part of the key to produce the first 64 bits of output, which only depend on the middle 64 bits of the key. Then the output of step1 is used to produce the second 64 bits of output using the step2 function and the second half of the key. Please note that the second half of the output only depends on the first half of the output and the second part of the key.

Algorithm 1 DSAA ($\mathbf{rand} \in \{0,1\}^{64}$, $\mathbf{key} \in \{0,1\}^{128}$)

```
$\mathbf{t} \leftarrow \mathsf{step1}(\mathsf{rev}(\mathbf{rand}), \mathsf{rev}(\mathbf{key}[32\dots95]))$
$\mathbf{b} \leftarrow \mathsf{step2}(\mathbf{t}, \mathsf{rev}(\mathbf{key}[96\dots127])||\mathsf{rev}(\mathbf{key}[0\dots31]))$
return $\mathsf{rev}(\mathbf{b}[32\dots63])||\mathsf{rev}(\mathbf{t})||\mathsf{rev}(\mathbf{b}[0\dots31]))$
```

We will now have a closer look at the functions step1 and step2. Both are very similar and each one uses two block cipher like functions as building blocks.

Algorithm 2 step1($\mathbf{rand} \in \{0,1\}^{64}, \mathbf{key} \in \{0,1\}^{64}$)

```
$\mathbf{k} = \mathsf{cassable}^{46,35}_{\mathbf{rand}}(\mathbf{key})$
return $\mathsf{cassable}^{25,47}_{\mathbf{k}}(\mathbf{rand})$
```

step1 takes a 64 bit key **key** and a 64 bit random value **rand** as input and uses two block ciphers to produce its output. The key is used as a key for the first cipher and the random value as a plaintext. The value **rand** then is used as an input to the second block cipher and is encrypted with the output of the first block cipher as the key.

Algorithm 3 $\mathsf{cassable}^{\textbf{start},\textbf{step}}_{\textbf{key}}(\mathbf{m} \in \{0,1\}^{64})$

```
$\mathbf{t} \leftarrow \mathbf{key}$
$\mathbf{s} \leftarrow \mathbf{m}$
for $\mathbf{i} = 0$ to $1$ do
    $\mathbf{t} \leftarrow \mathsf{sigma}(\mathbf{start}, \mathbf{step}, \mathbf{t})$
    $\mathsf{s} \leftarrow \mathsf{lambda1}(\mathsf{gamma}(\mathsf{s} \oplus \mathbf{t}))$
    $\mathbf{t} \leftarrow \mathsf{sigma}(\mathbf{start}, \mathbf{step}, \mathbf{t})$
    $\mathsf{s} \leftarrow \mathsf{lambda2}(\mathsf{gamma}(\mathsf{s} \oplus \mathbf{t}))$
    $\mathbf{t} \leftarrow \mathsf{sigma}(\mathbf{start}, \mathbf{step}, \mathbf{t})$
    $\mathsf{s} \leftarrow \mathsf{lambda3}(\mathsf{gamma}(\mathsf{s} \oplus \mathbf{t}))$
end for
return $\mathbf{s}$
```

To describe the block ciphers, we introduce a family of block ciphers we call cassable. These block ciphers differ only in their key schedule, where round keys are always bit permutations of the input key. All bit permutations used by cassable can be described by two numbers **start** and **step**.

The block cipher cassable itself is a substitution linear network. To mix the round key into the state, a simple XOR is used. Additionally, $\mathbb{Z}_{256}$-linear mixing is used for diffusion and an 8×8 S-Box for non-linearity of the round function.

Algorithm 4 $\mathsf{step2}(\mathbf{rand} \in \{0,1\}^{64}, \mathbf{key} \in \{0,1\}^{64})$

```
$\mathbf{k} = \mathsf{cassable}^{60,27}_{\mathbf{rand}}(\mathbf{key})$
return $\mathsf{cassable}^{55,39}_{\mathbf{k}}(\mathbf{rand})$
```

step2 is similar to step1, just two other bit permutations are used. The function rev simply reverses the order of the bytes of its input.

Algorithm 5 $\mathsf{rev}(\mathbf{in} \in \{0,1\}^{i*8})$

```
Ensure: Byte-reverses the input $\mathbf{in}$
  for $\mathbf{j} = 0$ to $i - 1$ do
      $\mathbf{k} \leftarrow i - j - 1$
      $\mathbf{out}[j * 8 \ldots j * 8 + 7] \leftarrow \mathbf{in}[k * 8 \ldots k * 8 + 7]$
  end for
  return $\mathbf{out}$
```

Algorithm 6 lambda1($\mathbf{in} \in \{0,1\}^{64}$)

```
$\mathbf{out}[0 \dots 7] \leftarrow \mathbf{in}[32 \dots 39] + 2 * \mathbf{in}[0 \dots 7]$
$\mathbf{out}[32 \dots 39] \leftarrow \mathbf{in}[0 \dots 7] + 3 * \mathbf{in}[32 \dots 39]$
$\mathbf{out}[8 \dots 15] \leftarrow \mathbf{in}[40 \dots 47] + 2 * \mathbf{in}[8 \dots 15]$
$\mathbf{out}[40 \dots 47] \leftarrow \mathbf{in}[8 \dots 15] + 3 * \mathbf{in}[40 \dots 47]$
$\mathbf{out}[16 \dots 23] \leftarrow \mathbf{in}[48 \dots 55] + 2 * \mathbf{in}[16 \dots 23]$
$\mathbf{out}[48 \dots 55] \leftarrow \mathbf{in}[16 \dots 23] + 3 * \mathbf{in}[48 \dots 55]$
$\mathbf{out}[24 \dots 31] \leftarrow \mathbf{in}[56 \dots 63] + 2 * \mathbf{in}[24 \dots 31]$
$\mathbf{out}[56 \dots 63] \leftarrow \mathbf{in}[24 \dots 31] + 3 * \mathbf{in}[56 \dots 63]$
return out
```

Algorithm 7 lambda2($\mathbf{in} \in \{0,1\}^{64}$)

```
$\mathbf{out}[0 \dots 7] \leftarrow \mathbf{in}[16 \dots 23] + 2 * \mathbf{in}[0 \dots 7]$
$\mathbf{out}[16 \dots 23] \leftarrow \mathbf{in}[0 \dots 7] + 3 * \mathbf{in}[16 \dots 23]$
$\mathbf{out}[8 \dots 15] \leftarrow \mathbf{in}[24 \dots 31] + 2 * \mathbf{in}[8 \dots 15]$
$\mathbf{out}[24 \dots 31] \leftarrow \mathbf{in}[8 \dots 15] + 3 * \mathbf{in}[24 \dots 31]$
$\mathbf{out}[32 \dots 39] \leftarrow \mathbf{in}[48 \dots 55] + 2 * \mathbf{in}[32 \dots 39]$
$\mathbf{out}[48 \dots 55] \leftarrow \mathbf{in}[32 \dots 39] + 3 * \mathbf{in}[48 \dots 55]$
$\mathbf{out}[40 \dots 47] \leftarrow \mathbf{in}[56 \dots 63] + 2 * \mathbf{in}[40 \dots 47]$
$\mathbf{out}[56 \dots 63] \leftarrow \mathbf{in}[40 \dots 47] + 3 * \mathbf{in}[56 \dots 63]$
return out
```

Algorithm 8 lambda3($\mathbf{in} \in \{0,1\}^{64}$)

```
$\mathbf{out}[0 \dots 7] \leftarrow \mathbf{in}[8 \dots 15] + 2 * \mathbf{in}[0 \dots 7]$
$\mathbf{out}[8 \dots 15] \leftarrow \mathbf{in}[0 \dots 7] + 3 * \mathbf{in}[8 \dots 15]$
$\mathbf{out}[16 \dots 23] \leftarrow \mathbf{in}[24 \dots 31] + 2 * \mathbf{in}[16 \dots 23]$
$\mathbf{out}[24 \dots 31] \leftarrow \mathbf{in}[16 \dots 23] + 3 * \mathbf{in}[24 \dots 31]$
$\mathbf{out}[32 \dots 39] \leftarrow \mathbf{in}[40 \dots 47] + 2 * \mathbf{in}[32 \dots 39]$
$\mathbf{out}[40 \dots 47] \leftarrow \mathbf{in}[32 \dots 39] + 3 * \mathbf{in}[40 \dots 47]$
$\mathbf{out}[48 \dots 55] \leftarrow \mathbf{in}[56 \dots 63] + 2 * \mathbf{in}[48 \dots 55]$
$\mathbf{out}[56 \dots 63] \leftarrow \mathbf{in}[48 \dots 55] + 3 * \mathbf{in}[56 \dots 63]$
return out
```

Algorithm 9 sigma($\mathbf{start}, \mathbf{step}, \mathbf{in} \in \{0,1\}^{64}$)

```
$\mathbf{out} \leftarrow (\mathsf{00})^8$
for $\mathbf{i} = 0$ to $63$ do
    $\mathbf{out}[\mathbf{start}] \leftarrow \mathbf{in}[\mathbf{i}]$
    $\mathbf{start} \leftarrow (\mathbf{start} + \mathbf{step}) \bmod 64$
end for
return out
```

Algorithm 10 gamma($\mathbf{in} \in \{0,1\}^{64}$)

```
for $\mathbf{i} = 0$ to $7$ do
    $\mathbf{out}[\mathbf{i} * 8 \dots \mathbf{i} * 8 + 7] \leftarrow \mathbf{sbox}[\mathbf{in}[\mathbf{i} * 8 \dots \mathbf{i} * 8 + 7]]$
end for
return out
```

B Test vectors for DSAA

To make implementation of these algorithms easier, we decided to provide some test vectors. Let us assume that A11 is called with the key K=ffff9124ffff9124ffff9124ffff9124 and the RS=0000000000000000 as in [5] Annex K. These values will be passed directly to the DSAA algorithm. Now, step1(0000000000000000, 2491ffff2491ffff) will be called. While processing the input, the internal variables will be updated according to Table 2. The final result after step2(ca41f5f250ea57d0, 2491ffff2491ffff) has been calculated is 93638b457afd40fa585feb6030d572a2, which is the UAK. The internal states of step2 can be found in Table 3.

Table 2. Trace of step1(0000000000000000, 2491ffff2491ffff)

algorithm	after line	**i**	**t**	**s**
cassable^{46,35}	5	0	0000000000000000	549b363670244848
cassable^{46,35}	7	0	0000000000000000	51d3084936beeaae
cassable^{46,35}	9	0	0000000000000000	20e145b2c0816ec6
cassable^{46,35}	5	1	0000000000000000	4431b3d7c1217a7c
cassable^{46,35}	7	1	0000000000000000	6cdcc25bbe8bc07f
cassable^{46,35}	9	1	0000000000000000	2037df9f8856a0a2
cassable^{25,47}	5	0	77fe578089a40531	cce76e5f83f77b4c
cassable^{25,47}	7	0	f5b720768a8a8817	c69973d6388f3cf7
cassable^{25,47}	9	0	552023ae0791ddf4	1cd81853ba428a2c
cassable^{25,47}	5	1	8856a0a22037df9f	ca643e2238dc1d1d
cassable^{25,47}	7	1	89a4053177fe5780	82fa43b0725dc387
cassable^{25,47}	9	1	8a8a8817f5b72076	ca41f5f250ea57d0

Table 3. Trace of step2(ca41f5f250ea57d0, 2491ffff2491ffff)

algorithm	after line	**i**	**t**	**s**
cassable^{60,27}	5	0	66f9d1c1c6524b4b	39ad15f5f68ab424
cassable^{60,27}	7	0	5bd0d66bf152e4c0	59e160ed3bb1189c
cassable^{60,27}	9	0	d5ebead34f434050	0bc33d7c093128b8
cassable^{60,27}	5	1	d2c057d860e3dd72	3f538f008a2b52f9
cassable^{60,27}	7	1	c6d2e3614c5953cb	ab826a7542ffa5c7
cassable^{60,27}	9	1	f158c640d3f27cc3	757782ad02592b4e
cassable^{55,39}	5	0	b0ec588246ea9577	40be7413fe173981
cassable^{55,39}	7	0	df212e1b790245e6	087978cbb37813af
cassable^{55,39}	9	0	e671b9d44296ee08	d97b8d2dbae583b9
cassable^{55,39}	5	1	2a0f207383ec575d	1340ba1df9d60b52
cassable^{55,39}	7	1	d022e4e81dd712ee	f7af7e62a1fa5ce6
cassable^{55,39}	9	1	04b3db206f4e7d03	08d87f9aef21c939

C Example of a weak PRNG used in DECT stacks

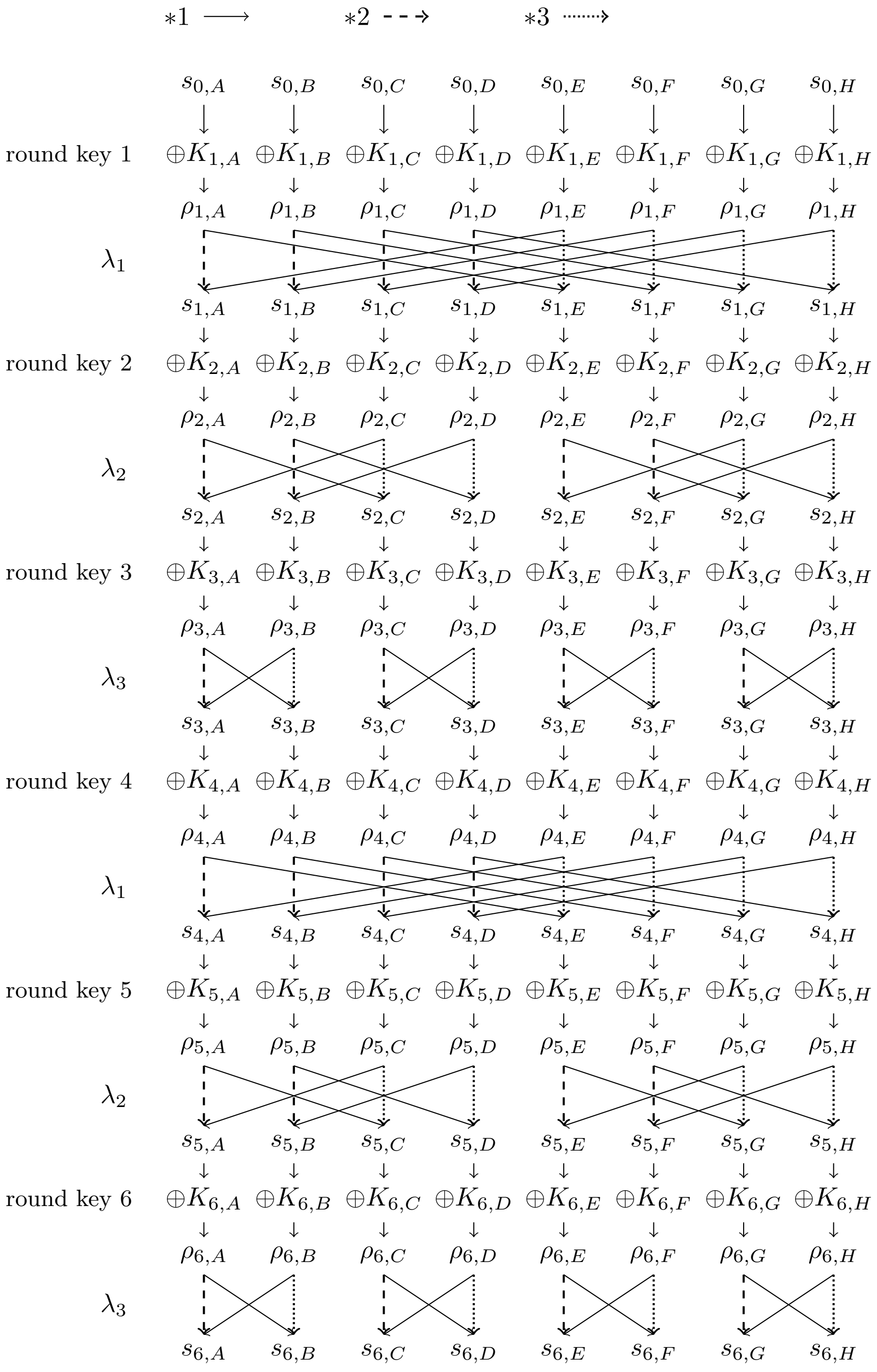
Algorithm 11 is a typical example for the quality of pseudo random-number generators (PRNGs) used in DECT stacks. Although it is supposed to provide 64-bit of randomness per nonce output, it only manages to use 24 bits of entropy. Moreover, the total number of distinct 64-bit **rand** values of this PRNG is only 2^{22} since outputs collide.

Algorithm 11 vendor_A_PRNG($\mathbf{xorval} \in \{0,1\}^8, \mathbf{counter} \in \{0,1\}^{16}$)

```
for $\mathbf{i} = 0$ to 7 do
    $\mathbf{out}[(\mathbf{i} * 8)) \ldots (\mathbf{i} * 8 + 7)] \leftarrow \lfloor \mathbf{counter}/2^i \rfloor \oplus \mathbf{xorval}$
end for
return out
```

The values produced by this particular PRNG can be easily stored in ASCII representation in a file just 68 Megabytes big. This means that to identify vulnerable implementations, an attacker or evaluator simply has to search for an intercepted **rand** in this text file.

D Structure of the **cassable** block cipher


∗1
∗2
∗3
s0,A s0,B s0,C s0,D s0,E s0,F s0,G s0,H
round key 1
⊕K1,A ⊕K1,B ⊕K1,C ⊕K1,D ⊕K1,E ⊕K1,F ⊕K1,G ⊕K1,H
ρ1,A ρ1,B ρ1,C ρ1,D ρ1,E ρ1,F ρ1,G ρ1,H
λ1
s1,A s1,B s1,C s1,D s1,E s1,F s1,G s1,H
round key 2
⊕K2,A ⊕K2,B ⊕K2,C ⊕K2,D ⊕K2,E ⊕K2,F ⊕K2,G ⊕K2,H
ρ2,A ρ2,B ρ2,C ρ2,D ρ2,E ρ2,F ρ2,G ρ2,H
λ2
s2,A s2,B s2,C s2,D s2,E s2,F s2,G s2,H
round key 3
⊕K3,A ⊕K3,B ⊕K3,C ⊕K3,D ⊕K3,E ⊕K3,F ⊕K3,G ⊕K3,H
ρ3,A ρ3,B ρ3,C ρ3,D ρ3,E ρ3,F ρ3,G ρ3,H
λ3
s3,A s3,B s3,C s3,D s3,E s3,F s3,G s3,H
round key 4
⊕K4,A ⊕K4,B ⊕K4,C ⊕K4,D ⊕K4,E ⊕K4,F ⊕K4,G ⊕K4,H
ρ4,A ρ4,B ρ4,C ρ4,D ρ4,E ρ4,F ρ4,G ρ4,H
λ1
s4,A s4,B s4,C s4,D s4,E s4,F s4,G s4,H
round key 5
⊕K5,A ⊕K5,B ⊕K5,C ⊕K5,D ⊕K5,E ⊕K5,F ⊕K5,G ⊕K5,H
ρ5,A ρ5,B ρ5,C ρ5,D ρ5,E ρ5,F ρ5,G ρ5,H
λ2
s5,A s5,B s5,C s5,D s5,E s5,F s5,G s5,H
round key 6
⊕K6,A ⊕K6,B ⊕K6,C ⊕K6,D ⊕K6,E ⊕K6,F ⊕K6,G ⊕K6,H
ρ6,A ρ6,B ρ6,C ρ6,D ρ6,E ρ6,F ρ6,G ρ6,H
λ3
s6,A s6,B s6,C s6,D s6,E s6,F s6,G s6,H