DECT Security Analysis

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Diplom Informatiker Erik Tews aus Lauterbach (Hessen)
Mai 2012 — Darmstadt — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Theoretische Informatik,
Kryptographie und Computeralgebra

DECT Security Analysis

Genehmigte Dissertation von Diplom Informatiker Erik Tews aus Lauterbach (Hessen)

1. Gutachten: Prof. Johannes Buchmann
2. Gutachten: Prof. Stefan Lucks

Tag der Einreichung: 5. September 2011
Tag der Prüfung: 19. September 2011

Darmstadt — D 17

Erik Tews <erik@datenzone.de>
Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-29328
URL: http://tuprints.ulb.tu-darmstadt.de/2932

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Contents

Abstract

DECT is a standard for cordless phones. The intent of this thesis is to evaluate DECT security in a comprehensive way. To secure conversations over the air, DECT uses two proprietary algorithms, namely the *DECT Standard Authentication Algorithm* (DSAA) for authentication and key derivation, and the *DECT Standard Cipher* (DSC) for encryption. Both algorithms have been kept secret and were only available to DECT device manufacturers under a *None Disclosure Agreement* (NDA). The reader is first introduced into the DECT standard. The two algorithms DSAA and DSC have been reverse engineered and are then described in full detail. At first, attacks against DECT devices are presented, that are based on faults made by the manufacturers while implementing the DECT standard. In the next Chapters, attacks against the DSAA and the DSC algorithm are described, that recover the secret keys used by these algorithms faster than by brute force. Thereafter, a attack against the DECT radio protocol is described, that decrypts encrypted DECT voice calls. Finally, an outlook over the next release of the DECT standard is presented, that is expected to counter all attacks against DECT, that are described in this thesis.

DECT ist ein Standard für schnurlose Telefone. Um die Funkübertragung zwischen DECT Geräten zu sichern, verwendet DECT zwei proprietäre Algorithmen, den *DECT Standard Authentication Algorithm* (DSAA) für die Authentifikation und Schlüsselableitung, sowie den *DECT Standard Cipher* (DSC) für die Verschlüsselung. Beide Algorithmen wurden geheim gehalten und waren nur DECT Geräteherstellern unter einem *None Disclosure Agreement* (NDA) zugänglich. Das Ziel dieser Arbeit ist eine umfassende Untersuchung der Sicherheit von DECT. Der Leser wird zuerst in den DECT Standard eingeführt. Die beiden ehemals geheimen Algorithmen DSAA und DSC wurden reverse engineered und sind hier mit allen Details beschrieben. Zuerst werden Angriffe auf DECT Geräte selbst vorgestellt, die weitestgehend auf Fehlern basieren, die von den Herstellern bei der Implementierung des DECT Standards gemacht wurden. In den nächsten Kapiteln werden Angriffe auf die Algorithmen DSAA und DSC selber vorgestellt, die es möglich machen die geheimen Schlüssel der Algorithmen schneller als durch eine erschöpfende Suche zu finden. Danach wird ein Angriff auf das DECT Protokoll selber vorgestellt, der es möglich macht, verschlüsselte Telefongespräche zu entschlüsseln. Zuletzt wird ein Ausblick auf die zukünftige Version des DECT Standards geboten, der voraussichtlich alle Angriffe, die hier beschrieben wurden, beheben wird.

1 Introduction

DECT is a standard for cordless phones, which is probably used by most Germans on a daily basis. Besides for cordless phones, DECT can also be used for remote speakers, baby phones, wireless payment systems, traffic control systems and a lot of other applications.

DECT standardization was finalized in 1992 and low cost consumer phones are available since 1994. It is safe to say that about 34 million DECT systems have been in use in Germany in 2009, and more than 800 millions have been sold worldwide.

The DECT standard is publicly available free of charge, except for one part: In order to protect sensitive information transmitted over DECT and to ensure authenticity of the devices, DECT uses two algorithms: The DECT Standard Cipher (DSC) and the DECT Standard Authentication Algorithm (DSAA). Both algorithms are not available to the public but only to DECT device manufactors, who in turn have to sign a none disclosure agreement.

At the end of 2008, no academic publication about DECT security had been published yet and no academic research about DECT security known to the author of this thesis had been made. In contrast, the GSM[1] system was at the time completely known to the public, including all formally secret algorithms and a lot of academic papers about GSM security have been published.

In contrast to public research, it had been assumed that intelligence agencies had a very good knowledge about DECT and had been using DECT as an exercise for signal intelligence (SIGINT) training[7]. The German *Bundesamt für Sicherheit in der Informationstechnik* (BSI) issued a warning that DECT might be not as secure as it could be and that thus sensitive information should not be transmitted using DECT[6].

1.1 Challenges in DECT Security

These information, the lack of knowledge and that I was using DECT too on a daily basis made me highly interested in DECT security and I started discussing DECT and DECT security with other people, a short time before the annual *CCC congress*, an event organized by the *Chaos Computer Club*, in 2007. There I met many other people who were interested in DECT too and started working on DECT security. We identified multiple challenges in DECT security:

- DSAA and DSC are two secret algorithms. In order to evaluate DECT security comprehensively, these two algorithms must be known to the public. This can be achieved by reverse engineering of those algorithms.
- DSAA is an authentication mechanism for DECT that is also used for key derivation. A weakness in DSAA could help an attacker to recover long-term or session keys for DECT. Therefore DSAA should be analyzed.

- DSC is a stream cipher that is used for encryption DECT. A weakness in DSC could help an attacker to recover information from a DECT conversation that is encrypted with DSC. To evaluate the security of DECT, DSC must be analyzed too.
- The mere use of an authentication scheme and a stream cipher is not sufficient to secure a mobile telecommunication network. To fully evaluate DECT security, the DECT protocol itself and how it uses DSC and DSAA need to be checked.
- Because DECT can be implemented in many different ways, implementations are of high interest as well. For example, a bad random number generator can compromise the security of a good cryptosystem.
- To accomplish all these tasks and to allow other researchers to work with DECT, there should be Open Source tools available for analyzing DECT systems.

By now in 2011, all parts of the DECT standard are known, various academic publications about DECT security have been written[23, 27, 29, 31, 25] and multiple Open Source programs are available that interact with DECT networks. All attacks discovered have been acknowledged by the European Telecommunications Standards Institute (ETSI) and the device manufactures, and countermeasures will be implemented. To counter all attacks, updates to the standard have been released and further updates are prepared by ETSI.

1.2 My Contribution

I as the author of this thesis contributed to this by helping in the reverse engineering of both DSAA and DSC. Both algorithms are now known to the public. I analyzed DSAA and contributed to the discovery of weaknesses in the algorithms and the building blocks of the algorithm. These results have been published at CT-RSA 2009[23] and ICWMC 2009[27]. I also analyzed DSC and designed a key recovery attack against the cipher that has been published at FSE2010[29] and ICISC 2010[31]. I also showed that the DECT protocol itself is not secure, even when used together with secure algorithms. This has been published at WISEC 2011[25]. I also helped in finding attacks against DECT phones and base stations from different vendors. Furthermore, I contributed to the development of tools that help in analyzing DECT devices. To achieve this, methods from symmetric cryptanalysis, reverse engineering and protocol design have been used. I am one of the authors of every academic paper[23, 27, 29, 31, 25], that has been published about DECT security so far.

1.3 Organization of this Thesis

This thesis is organized as follows: In Chapter 2, the DECT standard is summarized. The standard itself spans multiple documents and several hundreds of pages. Since this thesis just covers DECT security, only a small part of this standard needs to be known. A reader who is unfamiliar with DECT should read this Chapter first.

In Chapters 3 and 4, the two formerly secret algorithms DSAA and DSC are described in full detail. These algorithms are not part of the public DECT standard, but have been reverse engineered. These Chapters also introduce a notation for DSAA and a different notation for DSC that will be used later on to describe attacks on these algorithms. The reverse engineering of these algorithms is also summarized in these Chapters.

Chapter 5 describes attacks on DECT phones that rely mostly on mistakes made by the manufacturers, and are not weaknesses of the DECT standard. For example, some phones do not use encryption for voice calls, even though the DECT standard supports encryption. Other phones do not require authentication from the base station, meaning that an attacker can impersonate a base station and thus intercept calls. Another common mistake is using weak Pseudo Random Number Generators (PRNGs) which generate keys that can be easily guessed. This attack is also suitable for execution on a FPGA. To understand this Chapter, previous reading of Chapter 2 is recommended.

Chapter 6 describes attacks on the DECT Standard Authentication Algorithm (DSAA) itself. In a nutshell, the algorithm uses 128 bit keys; however, those keys can be recovered with an effort equal to about 2^{64} executions of DSAA. The main building block of DSAA is a custom block cipher that will also be heavily analyzed in this Chapter. To understand this Chapter, previous reading of Chapters 2 and 3 is recommended.

Chapter 7 describes a key recovery attack against the DECT Standard Cipher (DSC). Assuming that enough keystreams are available to an attacker, the cipher key can be recovered on a standard PC in minutes to hours. The attack presented here can also be accelerated by using a FPGA. To understand this Chapter, previous reading of Chapters 2 and 4 is recommended.

Chapter 8 describes an attack on the DECT protocol itself that can be used to decrypt a phone call without even having to attack DSAA or DSC itself. Even if both algorithms were to be replaced with secure variants, a call could still be decrypted using this attack. To understand this Chapter, merely the reading of Chapter 2 is required.

To counter these attacks, ETSI is preparing a new release of the DECT standard including two new algorithms DSAA2 and DSC2. Chapter 9 describes what can be expected in the upcoming releases of the DECT standard and how this will improve the security of DECT phones. To understand the importance of all these changes, a reading of all previous Chapters is strongly recommended.

Finally, I conclude in Chapter 10. Everybody who contributed to make this thesis possible is also mentioned in this Chapter.

2 DECT

In this Chapter, an overview of the DECT standard is given. The DECT standard[11] itself spans several hundred of pages and additional extensions of the standard exist. Because this thesis just covers DECT security, only a very small subset of this standard must be known to the reader to comprehend this thesis. Most details that are not required to comprehend the attacks presented in this thesis are not outlined in this Chapter.

The algorithms DSAA and DSC are later described in Chapters 3 and 4 in full detail and the methods used to reverse engineer these algorithms are outlined. Chapter 5 describes weaknesses in various DECT implementations. Attacks on the actual DECT standard are then described in Chapters 6, 7, and 8. However a reader who is unfamiliar with DECT should read this Chapter first, to understand the notation and terminology of these Chapters.

This Chapter only summarizes the public standard and contains no new findings. However, the text in this Chapter is based on my previous publications[23, 27, 29, 31, 25] about DECT.

2.1 DECT at a Glance

With more than 800 million devices sold worldwide[1], Digital Enhanced Cordless Telecommunications (DECT)[11] is one of the most common standards for short range cordless telephones. Besides for phones, DECT is used for many other applications like wireless payment systems, traffic control, access control and room monitoring. DECT networks usually consist of a single or multiple base stations named DECT Fixed Part (FP) in the standard and phones named DECT Portable Part (PP) linked with these base stations. For most residential use cases, only a single base station is operated with a small number of phones. A single base station can cover a single house or up to a few hundred meters in the open field. European systems operate at 1880 to 1900 MHz and have a maximum transmit power of 250 mW, while the North American version operates at 1920 to 1930 MHz and just uses a maximum transmit power of 100 mW. DECT systems can scale to many base stations and phones, and also support roaming as GSM[1] does.

To protect sensitive data transmitted over DECT, the standard provides authenticity (DECT Standard Authentication Algorithm, DSAA) as well as confidentiality (DECT Standard Cipher, DSC). Both algorithms were specially designed for DECT and are only available to DECT device manufactures who sign a non-disclosure agreement. DSAA is responsible for the initial pairing of a new phone with its base station and the generation of the long term key User Authentication Key (UAK). It is also used for authentication of phones and base stations and for key derivation to derive a session key (Derived Cipher Key (DCK)) for the DSC from the UAK.

[1] http://www.etsi.org/WebSite/NewsandEvents/201004_CATIQ.aspx

DSC is used for encryption. It is a stream cipher, which takes an Initialization Vector (IV) and a session key (Cipher Key, CK) to generate a keystream (cipher stream, CS) from it. If this key is derived using DSAA, we also refer to this key as Derived Cipher Key (DCK). Besides the actual payload (voice data), parts of the control traffic (C-channel traffic) are also encrypted, which for example contain the dialed number.

2.2 Radio Protocol

The DECT protocol can be divided into 5 layers: Physical Layer[12], MAC Layer[13], Data Link Control Layer[14], Network Layer[15] and the actual speech and audio coding[18].

Physical Layer[12] The Physical Layer is responsible for transmitting the individual frames. It defines the modulations, frequencies and frame formats used for the transmission and the time multiplexing, so that multiple devices can operate on the same frequency. It also monitors the radio environment to handle conflicts with other devices operating on the same frequency.

MAC Layer[13] The MAC layer establishes physical connections between devices. It also multiplexes multiple logical channels into a single physical channel.

Data Link Control Layer[14] The Data Link Control Layer provides reliable connection oriented or connection less services between two entities. Connection can either be made on the C-plane (control messages) or on the U-plane (payload transport).

Network Layer[15] The Network Layer provides call control and call management services, as well as mobility management so that resources in the central network or in the mobile part can be allocated and used.

Speech and Audio coding[18] Speech and Audio coding defines how the audio is encoded and decoded before it is transmitted.

The Physical Layer is of importance for the encryption used by DECT, because it defines which bits inside a single DECT frame are encrypted and how this is done. To understand the Physical Layer, we first have a look at the Time Division Multiple Access (TDMA) structure of a DECT network.

To allow multiple devices to transmit on the same frequency, DECT uses TDMA (Time Division Multiple Access), a period of 10 ms, known as **frame**, is divided into 24 **time slots**. In every frame, one of the first 12 time slots is used for transmissions from the base station to the phone (FP → PP), and the slot 12 time slots later is used for transmissions from the phone to the base station (PP → FP). In every time slot, 480 bits could be transmitted. A single DECT **full slot packet (P32)** has a total length of 32+64+320+4+4 = 424 bits and is divided into an S-, A-, B-, X- and an optional Z-Field. The remaining 56 or 60 bits are a guard period between the time slots.

The S-field (32 bits) is only a static preamble of a packet, which is used by the receiver to synchronize on the signal.

The A-field (64 bits) contains the packet header and can transport control traffic. Control traffic is separated into different logical channels, namely the C, M, N, P, and Q channels. A tag in the A-field header determines which channel is embedded in the A-field.

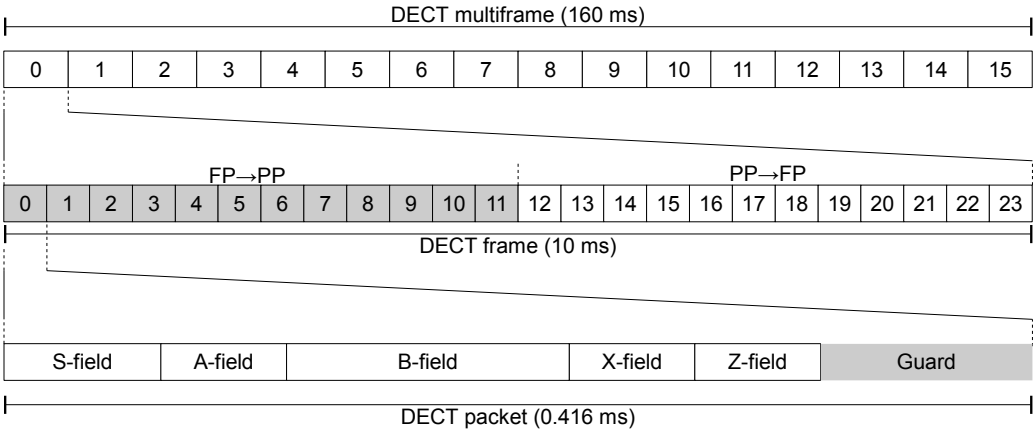
The B-field (320 bits) contains the actual payload, for example the voice data, when a phone call is made over DECT.

The X and Z fields (4 and 4 bits) are checksums to detect transmission problems.

DECT optionally supports sending shorter or longer packets (the standard also specifies the **P00**, **P80j** and other formats), if no payload is present. It is also possible to use different modulations so that more bits can be transmitted on a single frequency during a single time slot. Alternatively, two consecutive time slots can be used to send a single packet or a small packet can also be transmitted in just half a time slot. This is an optional feature, that can only be used, if it is supported by the phone and the base station.

An overview of the particular fields is given in Figure 1 and Figure 6. In addition to that, a period of 16 frames is called a **multiframe**, and a period of 25 multiframes is called a **hyperframe**. Base stations usually broadcast a beacon packet once per frame. Phones synchronize their timer on this beacon.

Figure 1.: DECT TDMA Structure



Most of the time, a phone only passively listens to the broadcasts of a base station. When there is traffic, for example a call is active or the base station needs to update the display of the phone, the phones establish a connection with the base station. A base station can request a new connection from the phone by broadcasting an `LCE-PAGE-REQUEST` message[15].

The DECT standard makes heavy use of timers, to specify how long certain procedures may take. Only one timer, namely the LCE.01 timer is of importance for this thesis. When a connection is not needed anymore by any upper protocol layers, the LCE.01 timer is started, which runs for 5 seconds. If there is no more activity on the connection within these 5 seconds, the connection is terminated.

An existing connection between a phone and a base station does not necessarily mean that a call is active. Instead, a base station might, for example, establish a connection just to update a phone display state to indicate that a new voicemail has arrived or a text has been received by the base station. All phones we

examined so far send packets with all bits set to 1 (`ff` in hex) in the B-field when there is no audio data present in the connection.

2.3 Identities and Addressing

DECT supports a complex scheme of addressing[16]. DECT handsets can carry multiple identities and DECT base stations can be grouped into different location areas and networks. However, these complex features are not used by any attack in this thesis. For residential use cases, we can summarize the identities and addressing scheme as follows:

Base Stations are identified by a Radio Fixed Part Identifier (RFPI), a 40 bit value, which should be unique for every base station in the world. Phones have at least a single International Portable User Identification (IPUI), which should be unique for every phone in the world too. They can also carry additional identities. For short term intervals a Temporary Portable User Identity (TPUI) can be assigned to a phone, which needs to be only unique for the current area, the phone is operating in.

2.4 Authentication and Key Derivation

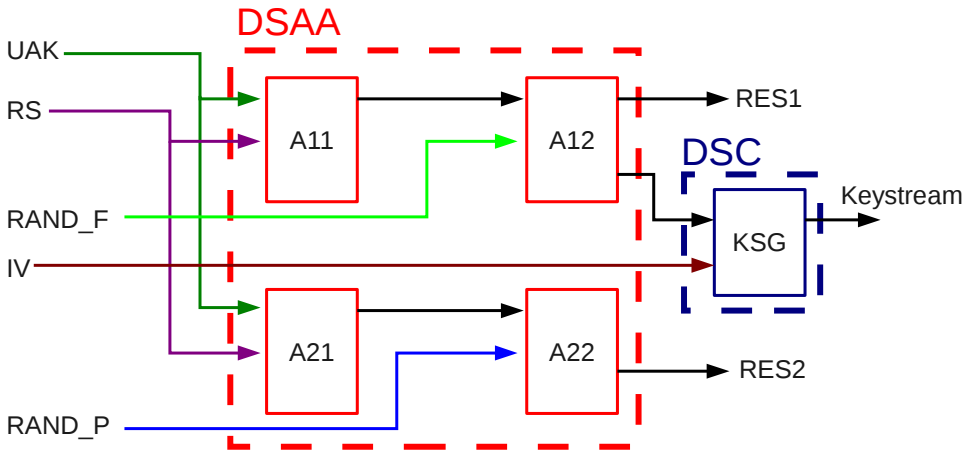
To ensure authentication of the communicating devices, DECT uses the DECT Standard Authentication Algorithm (DSAA). DECT provides procedures for authenticating the base station and for authenticating the phone. Mutual authentication can be achieved by executing both procedures sequentially. During authentication of the phone, a cipher key can be derived (Derived Cipher Key, DCK), that can later be used for encryption. This Section summarizes the authentication procedures, while the inner structure of the actual authentication algorithm (DSAA) is described in Chapter 3.

DSAA has been designed specially for DECT and was only available under an None Disclosure Agreement (NDA) to device manufacturers. During my research, DSAA has been reverse engineered and is later described in Chapter 3. The DECT standard also allows device manufacturers to replace DSAA with their custom algorithms, with the consequence that they would loose compatibility with all other DECT devices not using their authentication algorithm.

DSAA is a set of four algorithms, namely A11, A12, A21, and A22. The public interface to these algorithms is specified in the public part of the standard [17], but the algorithms themselves are not part of the public standard. To use these algorithms, a DECT base station must share a 128 bit secret key UAK with the base station (Section 2.4.3 describes how this key can be generated):

- A11 and A21 take a 128 bit input, usually a key (UAK), and a 64 bit random number RS, and generate a 128 bit intermediate key KS.
- A12 takes a 128 bit key KS, and a 64 bit random number RAND_F and generates a 32 bit value RES1 and a 64 bit cipher key DCK.
- A22 takes a 128 bit key KS, and a 64 bit random number RAND_P and generates a 32 bit value RES2.

Figure 2.: Security algorithms in DECT[17]



- If roaming would be used with DECT, the algorithms A11 and A21 would be executed in the home network and the key KS would be transferred to the roaming network, where A12 and A22 would be executed. If no roaming is used, all algorithms are executed anyway in the home network.

An overview of all these algorithms is given in Figure 2.

2.4.1 Authentication of a phone by base station

DECT supports two different authentication procedures using these algorithms: First, a base station can request authentication from a phone. To do so, the base station chooses two random numbers RS and RAND_F and sends them in an AUTHENTICATION-REQUEST[17] message to a phone. Now the phone computes a response to this challenge using the DSAA algorithms A11 and A12 with the UAK and these two random numbers as input. The result RES1 is transmitted in an AUTHENTICATION-RESPONSE[17] message to the base station, which performs the same computations and compares the received RES1 with the locally computed expected result XRES1. In addition to that, a 64 bit cipher key DCK is also generated by A12, which can be used for encryption later on. See Figure 3 for details.

2.4.2 Authentication of a base station by a phone

A phone can also request authentication from a base station. To do so, it picks just a single 64 bit random number RAND_P and sends it in an AUTHENTICATION-REQUEST message to the base station. The base station picks another 64 bit random number RS, and computes a response RES2 to the challenge sent by the phone using the DSAA algorithms A21 and A22 with UAK, RAND_P and RS as input. RES2 and RS are transmitted in an AUTHENTICATION-REPLY message to the phone, which compares it to the locally computed expected result XRES2 using the RS value send from the base station. No cipher key is

generated and the procedure does not affect the generated cipher key from the previous paragraph. An overview is given in Figure 4.

2.4.3 Initial pairing and key allocation

So far, we assumed that a phone and a base station share a 128 bit key UAK. For initial key allocation, when a phone connects to the base station for the first time, DECT GAP [19] defines a key allocation procedure, that requires both devices to share a common PIN number (usually 4 digits) and generates a 128 bit key (UAK) from it.

First, the PIN number is deterministically expanded into a 128 bit value AC. Then a base station picks two random numbers RS and RAND_F and sends them in an `KEY-ALLOCATE` message to the phone. The phone chooses a 64 bit random number RAND_P and computes:

Figure 3.: Authentication of a DECT PP

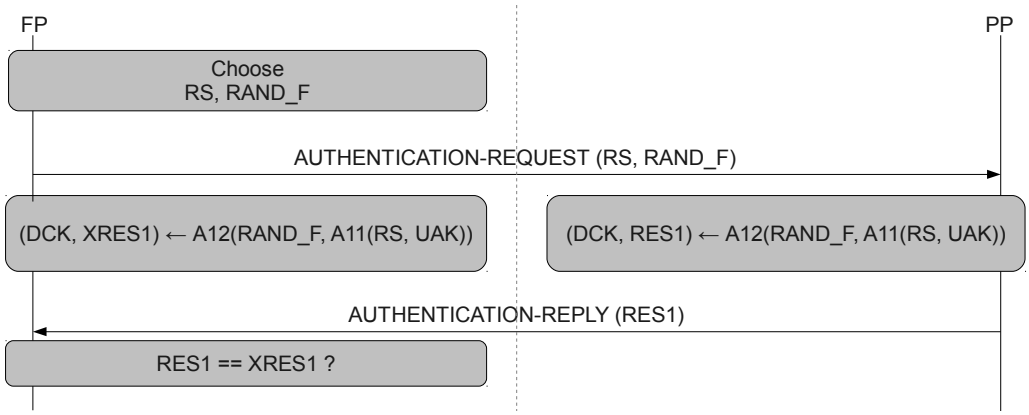


Figure 4.: Authentication of a DECT FP

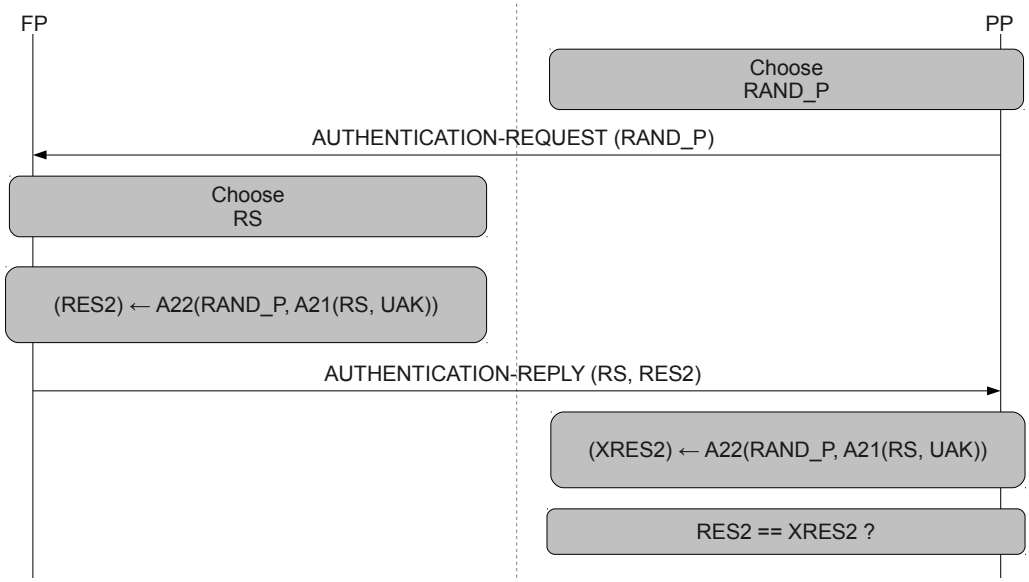
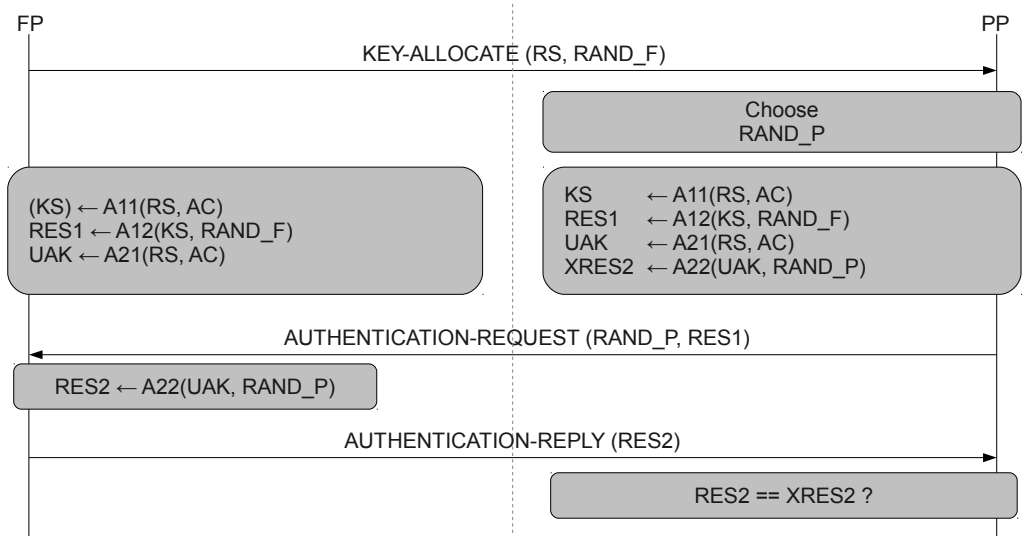


Figure 5.: DECT GAP key allocation



$$
\begin{aligned}
\mathsf{KS} &\leftarrow \mathsf{A11}(\mathsf{RS}, \mathsf{AC}) \\
\mathsf{RES1} &\leftarrow \mathsf{A12}(\mathsf{KS}, \mathsf{RAND\_F}) \\
\mathsf{UAK} &\leftarrow \mathsf{A21}(\mathsf{RS}, \mathsf{AC}) \\
\mathsf{RES2} &\leftarrow \mathsf{A22}(\mathsf{UAK}, \mathsf{RAND\_P})
\end{aligned}
$$

The phone responds with an `AUTHENTICATION-REQUEST` message containing RES1 and RAND_P. The base station compares the received RES1 with the locally computed expected result XRES1 and computes UAK and RES2 too, sends RES2 in an `AUTHENTICATION-REPLY` message back to the phone and accepts UAK as the new long term key. The phone compares the received RES2 with the locally computed expected result XRES2 and accepts UAK as the new long term key too. An overview of this procedure is given in Figure 5.

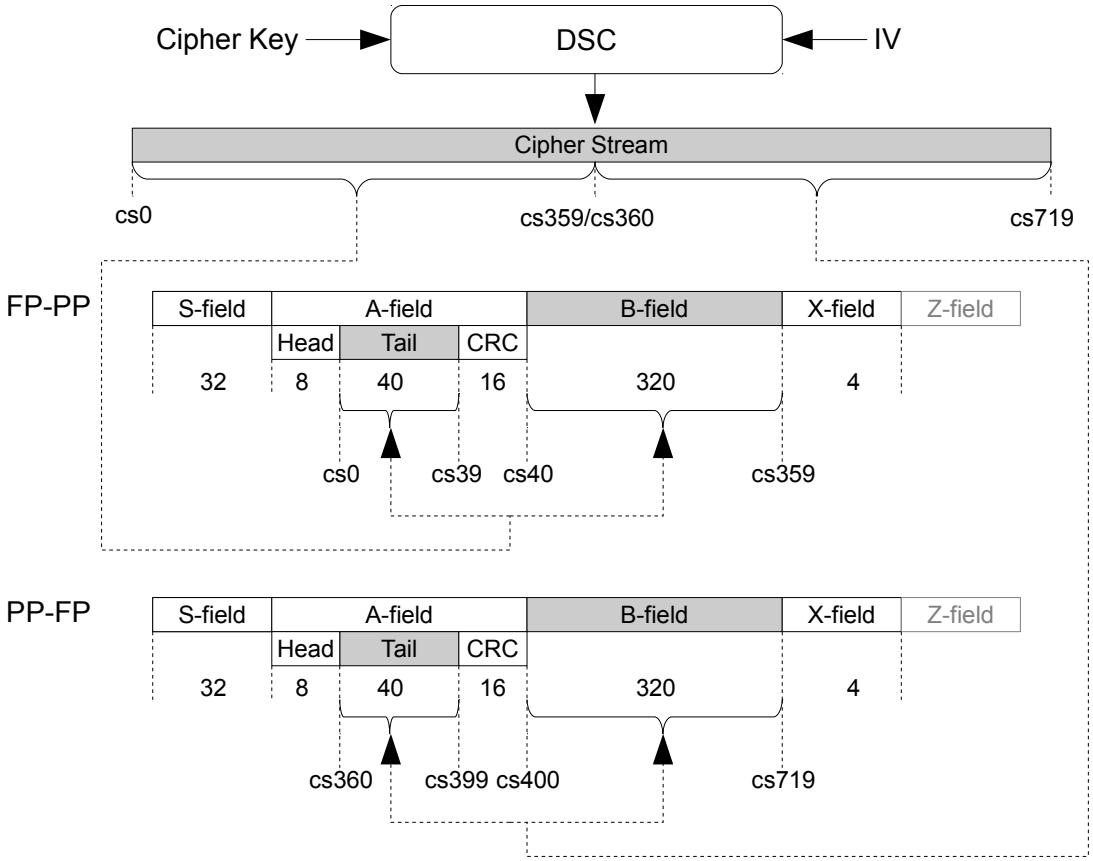
2.5 Encryption

For encryption, DECT defines a stream cipher, the DECT Standard Cipher (DSC). DSC takes a 64 bit Initialization Vector IV and a 64 bit Cipher Key (CK) as input and generates a keystream (cipher stream, cs) of arbitrary length from it. The cipher key CK is usually negotiated the beginning of a call or a connection using the DSAA algorithms A11 and A12 as shown in Figures 2 and 3. When the cipher key has been negotiated using the authentication algorithms, it is also named Derived Cipher Key (DCK). Even when there is no encryption in use, a base station continuously broadcasts a multiframe number embedded in a Q-channel message. The IV used is the frame number concatenated with the multiframe number, zero-padded to 64 bit length. Every packet in the same frame shares the same IV.

Usually, 720 bits of keystream (cs0 ... cs719) are generated for each IV: The first 360 bits (cs0 ... cs359) are used to encrypt the packet sent from the base station to the phone (FP → PP). The remaining 360

bits (cs360 ... cs719) are used to encrypt the packet sent from the phone to the base station (PP → FP) in the same frame. To encrypt a packet, the first 40 bits of keystream (cs0 ... cs39) are used to encrypt C-channel messages in the A-field, if the A-field contains C-channel traffic. If no C-channel traffic is present in the packet, the first 40 bits (cs0 ... cs39) are silently discarded. The remaining 320 bits (cs40 ... cs359) are XORed with the B-field to encrypt the payload present in the B-field. An overview of the process is given in Figure 6.

Figure 6.: Keystreams used in DECT (P32 full packet)



To enable encryption, the base station sends a `CIPHER-REQUEST`[17] message to the phone, indicating that it requests ciphering. The phone either confirms that using a MAC layer message, or sends a `CIPHER-REJECT`[17] message back to the base station, indicating that it is not capable of enabling ciphering. After ciphering has been confirmed by the phone, the base station sends a MAC layer message back to the phone as an acknowledgment. The next packet send or received will be encrypted.

3 Authentication – DSAA

The *DECT Standard Authentication Algorithm* (DSAA) is used for authentication and key derivation in DECT. DSAA has been kept secret and was only available to DECT device manufacturers under a None Disclosure Agreement (NDA). During my research on DECT security, DSAA was reverse engineered and analyzed. This Chapter gives a full description of DSAA in pseudocode, and various overview figures in Sections 3.2, 3.3, and 3.4. A reference implementation of DSAA written in C can be found in Appendix B.1. The reverse engineering of DSAA is described in Section 3.1. This Chapter also introduces the notation used to describe the internal parts of DSAA, which is later used for describing attacks on DSAA in Chapter 6.

Parts of this Chapter are joined work with Stefan Lucks, Andreas Schuler, Ralf-Philipp Weinmann, and Matthias Wenzel and have previously been published at CT-RSA 2009[23]. My main contributions to this Chapter were parts of the reverse engineering where a DECT kernel driver for Windows XP was analyzed on a live system using the integrated Windows XP kernel debugger. This made it possible to monitor inputs and outputs of function calls in the Windows XP kernel driver.

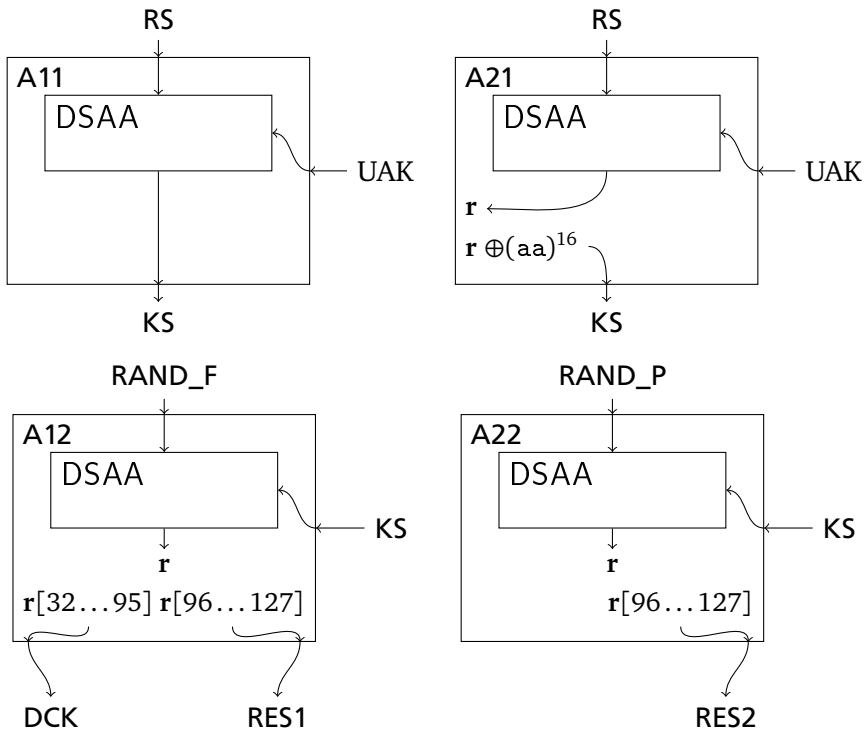
3.1 Reverse Engineering DSAA from Software

DSAA has been reverse engineered from various software implementations. When we compared multiple firmware images of DECT devices, we spotted something, that appeared to be a permutation table that permutes bytes. However, we were not able to find that table using a Google search and we could not spot any structure in that table, as one would expect for example for charset conversion or for signal processing. We assumed that this table is an S-Box from a not yet known cipher.

We used standard tools like IDA Pro and Binnavi to reverse engineer one of the binaries containing that table. We spotted a code fragment and found out that a value was XORed with another value, and then this value was substituted using that table.

We started reverse engineering the code that used this table and revealed something, that seemed to be a 64 bit block cipher. Further reverse engineering revealed the four DSAA algorithms A11, A12, A21, and A22. To verify our findings, we installed a DECT base station on a Windows XP system, that had DSAA implemented in the kernel driver. We used the Windows XP kernel debugger to intercept calls to the kernel driver when authentication was performed, and compared the inputs and outputs with our reverse engineered code. We also monitored the radio traffic between a phone and a base station, where we knew the UAK and compared the transmitted values with our code. In addition, we reran the communication trace contained in [17] Annex K. All of these attempts succeeded, so we could say with high confidence, that we successfully reverse engineered DSAA.

Figure 7.: The four DSAA algorithms



3.2 DSAA Internals at a Glance

DECT Authentication consists of four algorithms, namely A11, A12, A21, and A22, that were previously introduced in Section 2.4. They can be seen as wrappers around an algorithm, we call DSAA, that has been kept secret. The algorithm DSAA accepts an 128 bit value **key** and a 64 bit random number **rand** 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 (**r**[96...127])as RES1.
- A12 is similar to A22, except here, the middle 8 bytes of DSAA (**r**[32...95]) are returned too, as DCK.

An graphical overview is provided in Figure 7.

3.3 Notation and Conventions for DSAA

We introduce a special notation for DSAA. 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 [17]. Conversely function names written in lowercase letters like step1 have been introduced by us. 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 $(\texttt{aabb})^3$. For concatenating two values, the || operator is used. For example `aa||bb` results in `aabb`. The set of all n-bit strings is denoted by $\{0,1\}^n$. The set of the natural numbers from 0 to $n-1$ is denoted by $\mathbb{Z}_n$.

3.4 DSAA in Pseudocode

The DSAA (see Algorithm 1) uses four different 64 bit block cipher like functions as building blocks. DSAA takes a random value $\mathbf{rand} \in \{0,1\}^{64}$ and a second value $\mathbf{key} \in \{0,1\}^{128}$ 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 (bits 32 to 95) 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 output of step1 only depends on 64 bits of the **key** and **rand** and the output of step2 only directly depends on the other 64 bits of the **key** and the output of step2, and only depends transitively on **rand**.

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

1: $\mathbf{t} \leftarrow \mathsf{step1}(\mathsf{rev}(\mathbf{rand}), \mathsf{rev}(\mathbf{key}[32 \ldots 95]))$
2: $\mathbf{b} \leftarrow \mathsf{step2}(\mathbf{t}, \mathsf{rev}(\mathbf{key}[96 \ldots 127]) || \mathsf{rev}(\mathbf{key}[0 \ldots 31]))$
3: **return** $\mathsf{rev}(\mathbf{b}[32 \ldots 63]) || \mathsf{rev}(\mathbf{t}) || \mathsf{rev}(\mathbf{b}[0 \ldots 31]))$

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

1: $\mathbf{k} = \mathsf{cassable}^{46,35}_{\mathbf{rand}}(\mathbf{key})$
2: **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 $\mathsf{cassable}^{\mathbf{start},\mathbf{step}}_{\mathbf{key}}(\mathbf{m} \in \{0,1\}^{64})$

1: $\mathbf{t} \leftarrow \mathbf{key}$
2: $\mathbf{s} \leftarrow \mathbf{m}$
3: **for** $\mathbf{i} = 0$ to 1 **do**
4: $\quad \mathbf{t} \leftarrow \mathsf{bitperm}(\mathbf{start}, \mathbf{step}, \mathbf{t})$
5: $\quad \mathbf{s} \leftarrow \mathsf{mix1}(\mathsf{sub}(\mathbf{s} \oplus \mathbf{t}))$
6: $\quad \mathbf{t} \leftarrow \mathsf{bitperm}(\mathbf{start}, \mathbf{step}, \mathbf{t})$
7: $\quad \mathbf{s} \leftarrow \mathsf{mix2}(\mathsf{sub}(\mathbf{s} \oplus \mathbf{t}))$
8: $\quad \mathbf{t} \leftarrow \mathsf{bitperm}(\mathbf{start}, \mathbf{step}, \mathbf{t})$
9: $\quad \mathbf{s} \leftarrow \mathsf{mix3}(\mathsf{sub}(\mathbf{s} \oplus \mathbf{t}))$
10: **end for**
11: **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 permutation 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. Figure 11 shows an overview of the cassable structure.

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

1: $\mathbf{k} = \mathsf{cassable}^{60,27}_{\mathbf{rand}}(\mathbf{key})$
2: **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. The DSAA S-Box (**sbox**) can be found in Algorithm 11.

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

Ensure: Byte-reverses the input **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 out

Algorithm 6 mix1($\mathbf{in} \in \{0,1\}^{64}$)

1: $\mathbf{out}[0 \ldots 7] \leftarrow \mathbf{in}[32 \ldots 39] + 2 * \mathbf{in}[0 \ldots 7]$

2: $\mathbf{out}[32 \ldots 39] \leftarrow \mathbf{in}[0 \ldots 7] + 3 * \mathbf{in}[32 \ldots 39]$

3: $\mathbf{out}[8 \ldots 15] \leftarrow \mathbf{in}[40 \ldots 47] + 2 * \mathbf{in}[8 \ldots 15]$

4: $\mathbf{out}[40 \ldots 47] \leftarrow \mathbf{in}[8 \ldots 15] + 3 * \mathbf{in}[40 \ldots 47]$

5: $\mathbf{out}[16 \ldots 23] \leftarrow \mathbf{in}[48 \ldots 55] + 2 * \mathbf{in}[16 \ldots 23]$

6: $\mathbf{out}[48 \ldots 55] \leftarrow \mathbf{in}[16 \ldots 23] + 3 * \mathbf{in}[48 \ldots 55]$

7: $\mathbf{out}[24 \ldots 31] \leftarrow \mathbf{in}[56 \ldots 63] + 2 * \mathbf{in}[24 \ldots 31]$

8: $\mathbf{out}[56 \ldots 63] \leftarrow \mathbf{in}[24 \ldots 31] + 3 * \mathbf{in}[56 \ldots 63]$

9: **return out**

Algorithm 7 mix2($\mathbf{in} \in \{0,1\}^{64}$)

1: $\mathbf{out}[0 \ldots 7] \leftarrow \mathbf{in}[16 \ldots 23] + 2 * \mathbf{in}[0 \ldots 7]$

2: $\mathbf{out}[16 \ldots 23] \leftarrow \mathbf{in}[0 \ldots 7] + 3 * \mathbf{in}[16 \ldots 23]$

3: $\mathbf{out}[8 \ldots 15] \leftarrow \mathbf{in}[24 \ldots 31] + 2 * \mathbf{in}[8 \ldots 15]$

4: $\mathbf{out}[24 \ldots 31] \leftarrow \mathbf{in}[8 \ldots 15] + 3 * \mathbf{in}[24 \ldots 31]$

5: $\mathbf{out}[32 \ldots 39] \leftarrow \mathbf{in}[48 \ldots 55] + 2 * \mathbf{in}[32 \ldots 39]$

6: $\mathbf{out}[48 \ldots 55] \leftarrow \mathbf{in}[32 \ldots 39] + 3 * \mathbf{in}[48 \ldots 55]$

7: $\mathbf{out}[40 \ldots 47] \leftarrow \mathbf{in}[56 \ldots 63] + 2 * \mathbf{in}[40 \ldots 47]$

8: $\mathbf{out}[56 \ldots 63] \leftarrow \mathbf{in}[40 \ldots 47] + 3 * \mathbf{in}[56 \ldots 63]$

9: **return out**

Algorithm 8 mix3($\mathbf{in} \in \{0,1\}^{64}$)

1: $\mathbf{out}[0 \ldots 7] \leftarrow \mathbf{in}[8 \ldots 15] + 2 * \mathbf{in}[0 \ldots 7]$

2: $\mathbf{out}[8 \ldots 15] \leftarrow \mathbf{in}[0 \ldots 7] + 3 * \mathbf{in}[8 \ldots 15]$

3: $\mathbf{out}[16 \ldots 23] \leftarrow \mathbf{in}[24 \ldots 31] + 2 * \mathbf{in}[16 \ldots 23]$

4: $\mathbf{out}[24 \ldots 31] \leftarrow \mathbf{in}[16 \ldots 23] + 3 * \mathbf{in}[24 \ldots 31]$

5: $\mathbf{out}[32 \ldots 39] \leftarrow \mathbf{in}[40 \ldots 47] + 2 * \mathbf{in}[32 \ldots 39]$

6: $\mathbf{out}[40 \ldots 47] \leftarrow \mathbf{in}[32 \ldots 39] + 3 * \mathbf{in}[40 \ldots 47]$

7: $\mathbf{out}[48 \ldots 55] \leftarrow \mathbf{in}[56 \ldots 63] + 2 * \mathbf{in}[48 \ldots 55]$

8: $\mathbf{out}[56 \ldots 63] \leftarrow \mathbf{in}[48 \ldots 55] + 3 * \mathbf{in}[56 \ldots 63]$

9: **return out**

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

1: **out** $\leftarrow (00)^8$
2: **for i** $= 0$ to 63 **do**
3: **out**[**start**] $\leftarrow$ **in**[**i**]
4: **start** $\leftarrow$ (**start** $+$ **step**) mod 64
5: **end for**
6: **return out**

Algorithm 10 sub(**in** $\in \{0,1\}^{64}$)

1: **for i** $= 0$ to 7 **do**
2: **out**[**i** $*8 \ldots$ **i** $*8+7$] $\leftarrow$ **sbox**[**in**[**i** $*8 \ldots$ **i** $*8+7$]]
3: **end for**
4: **return out**

Algorithm 11 DSAA S-Box

	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

3.4.1 Test vectors for DSAA

To make implementations of these algorithms easier to verify, we decided to provide some test vectors. Let us assume that A11 is called with the key K=ffff9124ffff9124ffff9124ffff9124 and the random seed RS=0000000000000000 as in [17] 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 1. 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 2. As an additional help, a reference implementation of DSAA can be found in Appendix B.1.

Table 1.: 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 2.: 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

4 Encryption – DSC

The second secret algorithm used in DECT is the DECT Standard Cipher (DSC). DSC is a stream cipher used for encryption in DECT. During my research on DECT security, DSC was reverse engineered too. Compared to DSAA, no software implementations of DSC have been found and the reverse engineering was much more difficult compared to DSAA. This Chapter gives a description of DSC in Sections 4.1 and and a reference implementation of DSC written in C is presented in Appendix B.2. Because DSC is very different from DSAA, a different notation for DSC is introduced in Section 4.2. This notation is used to describe attacks on DSC too, which can be found in Chapter 7. Section 4.3 gives a full description of the internals of DSC. The reverse engineering of DSC is described in Section 4.4. Parts of this Chapter are joined work with Jan Krissler, Karsten Nohl, Andreas Schuler, and Ralf-Philipp Weinmann and have previously been published at FSE 2010[29].

My contribution to this Chapter is the reverse engineering of parts of the DECT Standard Cipher, especially the output combiner using a mixed software/firmware/hardware based approach, as well as the verification of the final cipher.

4.1 DSC at a Glance

The DECT Standard Cipher is an asynchronous stream cipher, that takes a 64 bit Cipher Key CK and a 35 bit Initialization Vector, IV, to generate keystream (cs). If the Cipher Key was generated using DSAA, we also refer to it as Derived Cipher Key (DCK). DSC is similar to GSM's A5/1 algorithm[8] and was reverse engineered from a DECT device using a combination of firmware probing and hardware reverse engineering.

The main components of DSC are:

- Three Linear Feedback Shift Registers (LFSRs) of length 17,19, and 21 bit in galois configuration[22], which are clocked asynchronously.
- An additional linear feedback shift register of length 23 bit in galois configuration, which is clocked synchronously, and is used for clocking control only.
- A clocking control unit, which takes input from all four registers, and controls the clocking of the three asynchronous registers.
- An output combiner, which takes input from the asynchronous registers and the last bit of output, and produces one bit of output.

In total, the internal state of DSC consists of 81 bits, 80 bits from the linear feedback shift registers, and 1 bit of memory from the output combiner.

Table 3.: Matrices describing linear operations on the internal state

Matrix	Dimension	Description
C_1	81×81	single clock register R1
C_2	81×81	single clock register R2
C_3	81×81	single clock register R3
L	81×128	load key and IV into state
S	6×81	extract the first two leading bits from R1, R2, and R3

4.2 Notation for DSC

The internal state of DSC is represented as a 81 bit vector, $s \in GF(2)^{81}$, composed of the state of four linear feedback shift registers and the memory bit of the output combiner.

Since state transitions are performed by linear operations, we will use matrices to describe them. The matrices in Table 4.2 represent the DSC operations. The matrices themselves are not presented in this thesis, but can be obtained from the source code presented in Appendix B.2 from the variables *R1_MASK, R2_MASK, R3_MASK* and *R4_MASK*. They are solely used for notation and only their effect is important to understand the rest of this Section and all attacks on DSC.

The output combiner $\mathcal{O}$ of DSC is a non-linear mapping, depending on the previous bit of output y and 6 bits of the state s. The DSC round function that translates a state into the next round's state is a non-linear mapping. The pre-ciphering phase which consists of loading the secret key and initialization vector (IV) into the DSC registers and then applying the round function i times is denoted $\mathcal{D}_i$. The i initialization rounds are referred to as pre-ciphering steps.

4.3 DSC Internals

The DECT Standard Cipher (DSC) is an irregularly clocked combiner with memory. Its internal state is built from 4 Galois LFSRs R1, R2, R3, R4 of length 17, 19, 21 and 23 respectively as well as a single bit of memory y for the output combiner. The bits of the state of the LFSR Ri shall be denoted by $x_{i,j}$ with the lowest-most bit being $x_{i,0}$. The taps of R1 are located at bit positions 5, 16; the taps of R2 are at bit positions 2, 3, 12, 18; the taps of R3 at bit positions 1, 20; the taps of R4 are at bit positions 8, 22. An overview of the circuit is given in Figure 4.3.2.

For each bit of output, register R4 is clocked three times whereas R1, R2, and R3 are clocked either two or three times. The clocking decision is determined individually for each of the irregularly clocked registers. The decisions depends linearly on one of the three lowest bits of R4 and the middle bits of the

other irregularly clocked registers. More specifically, the number of clocks c_i for each of the registers is calculated as follows:

$$\begin{aligned} c_1 &= 2 + (x_{4,0} \oplus x_{2,9} \oplus x_{3,10}) \\ c_2 &= 2 + (x_{4,1} \oplus x_{1,8} \oplus x_{3,10}) \\ c_3 &= 2 + (x_{4,2} \oplus x_{1,8} \oplus x_{2,9}) \end{aligned}$$

4.3.1 The output combiner

The output combiner is a cubic function that involves the lowest-most two bits of the registers R1, R2 and R3 as well as the memory bit y:

$$\begin{aligned} \mathcal{O}((x_{1,0}, x_{1,1}, x_{2,0}, x_{2,1}, x_{3,0}, x_{3,1}), y) = \; & x_{1,1}x_{1,0}y \oplus x_{2,0}x_{1,1}x_{1,0} \oplus x_{1,1}y \oplus \\ & x_{2,1}x_{1,0}y \oplus x_{2,1}x_{2,0}x_{1,0} \oplus x_{3,0}y \oplus \\ & x_{3,0}x_{1,0}y \oplus x_{3,0}x_{2,0}x_{1,0} \oplus x_{3,1}y \oplus \\ & x_{1,1}x_{1,0} \oplus x_{2,0}x_{1,1} \oplus x_{3,1}x_{1,0} \oplus \\ & x_{2,1} \oplus x_{3,1} \end{aligned}$$

The output of the combiner function generates a keystream bit which is also loaded into the memory bit for the next clock.

4.3.2 Key loading and initialization

Initially all registers and the memory bit are set to zero. The 35-bit IV is zero extended (most significant bits filled with zero) to 64 bits and concatenated with the 64 bit cipher key CK to form the session key K.

$$K = \text{zero\_extend}(\mathsf{IV})||\mathsf{CK}$$

The bits of K are clocked into the most significant bit of all four registers, bit by bit, starting with the least significant bit. During the key loading each LFSR is clocked once after each bit. After the session key has been loaded, 40 pre-cipher rounds are performed. In these pre-cipher rounds, the irregular clock control is used but the output is discarded. If one or more registers have all bits set to zero after executing 11 rounds, the most significant bit of the corresponding registers is set to 1 before starting the next round.

4.4 Reverse Engineering the DSC from Hardware

No software implementations of DSC have been found. Instead the starting point was a patent [2] describing the general structure of the DSC. From this document we learned that DSC is an LFSR-based design together with the lengths of the individual registers. Furthermore the patent discloses that the

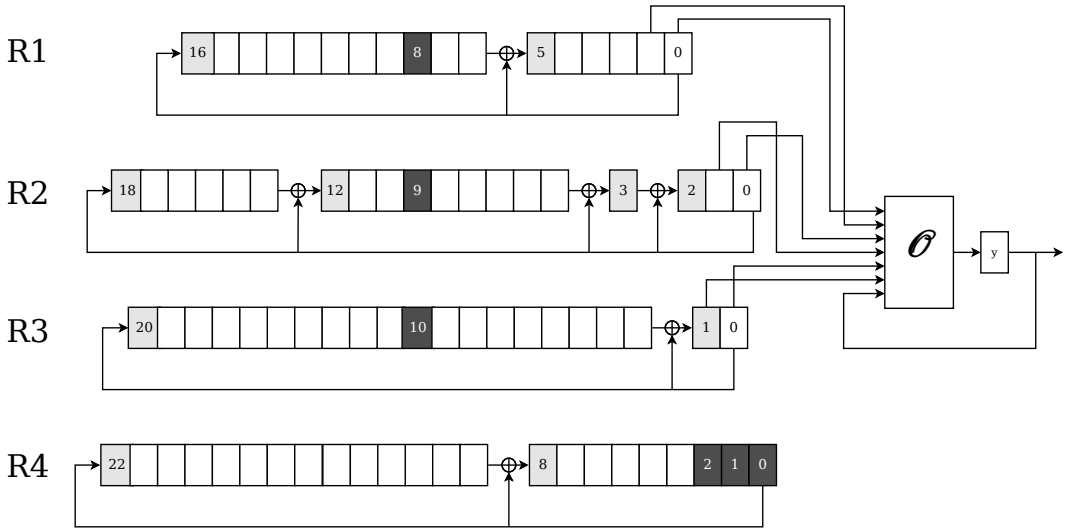


Figure 8.: The DSC keystream generator with LFSRs in Galois configuration. Bit positions that are inverted (white on black) are used in clocking decisions

cipher has an output combiner with a single bit of memory, irregular 2/3 clocking, and the number of initialization rounds. On the other hand the tap positions of the LFSRs, the clocking functions, the combiner function as well as the exact key loading routine are not described in this patent. The rule that after 11 initialization rounds a check had to be performed to make sure that no register is zero at that point is also stated in the patent.

Luckily, for the National Semiconductors SC14xxx DECT chipset that was used to implement the DECT sniffer described in Section 5.1, we found instructions that allow to load and store an arbitrary internal state of the stream cipher. Moreover, the stream cipher can be clocked in two modes: a regular clocking of the LFSRs for key loading and a second mode clocking irregularly as specified by the clocking functions, generating output. However we are not able to directly capture these output bits.

To reverse engineer the unspecified details of the cipher we proceed as follows: Using the first mode allows us to determine the tap positions of the LFSRs. After that, we are able to determine the clocking functions in the second mode by loading a random vector of low hamming weight into the internal state and observing how single-bit changes affect the clocking decisions.

The most elaborate part to reverse engineer is the output combiner function. To do this, we set up one machine with a modified firmware to send out frames containing zero-stuffed payloads. Another machine acting as the receiving side then *decrypts* these using a chosen internal state (no key setup), yielding keystream. Starting from random states, we sequentially flip single bit positions of the state and inspect the first bit to see whether the bit flip affected the output. If the output remains constant for a large number of random states, we assume that the flipped bit is not used in the output combiner.

Having identified the bits that indeed are fed into the combiner, we recover the combiner function by using multivariate interpolation for a number of keystreams. To do so, we iterated over all possible states for these bits, and observed the output of the combiner. We used the entire set of inputs and

outputs to represent the output combiner as a boolean polynomial using the *interpolation_polynomial* function of the *BooleanPolynomialRing* module of sage. Finally we determine the correct key loading by systematically trying different bit and byte-orders for both key and IV combined with both different orders of key and IV.

In parallel to the approach described above, we also reverse engineered the DSC cipher including its output combiner from silicon applying the techniques previously used to discover the Crypto-1 function [28].

5 Attacks on Implementations

This Chapter describes multiple attacks on DECT devices, which rely mostly on implementation flaws instead of problems in the standard. They can be fixed by using more features of the standard or by using a stronger random number generator. Attacks, that are even possible against devices that use alle security features of the standard and have no implementation flaws are later described in Chapters 6, 7, and 8. A reader, who is unfamiliar with DECT, should read Chapter 2 first, which gives a general overview of DECT.

Parts of this Chapter are joined work with Stefan Lucks, Alexandra Mengele, Hans Gregor Molter, Kei Ogata, Andreas Schuler, Ralf-Philipp Weinmann, and Matthias Wenzel and have previously been published at CT-RSA 2009[23] and ICWMC 2009 [27].

My main contributions to this Chapter are the supervision of a diploma thesis[30] together with Hans Gregor Molter written by *Kei Ogata* about the attack in Section 5.3. In addition to that, a second a second diploma thesis[26] written by *Alexandra Mengele* was supervised to investigate how common the problems described in Sections 5.1, 5.2, and 5.4 are. In addition to that, minor parts to the software, that is described in this Chapter, have been written by me.

5.1 Communication in Clear

As long as no encryption is used in DECT, the transmissions are not protected at all against an eavesdropper, and can be monitored. Even before I started my research on DECT, it was known by the German BSI [5], that some DECT devices don't support encryption and that DECT sniffers exist.

DECT does not use any features that make it difficult to monitor a radio transmission, like key dependent frequency hopping or similar techniques. However, as explained in Section 2.2, DECT is a TDMA system and the exact timing, when a signal was received, must be preserved during the post processing steps of the signal. This is difficult, but not impossible for general purpose hardware as the USRP[9]. The USRP has been successfully used to implement protocols like GSM[1] and TETRA[20].

To build a low-cost DECT sniffer, we used a Dosch-Amand DECT PCMCIA Card. The card was shipped with a Windows driver implementing a DECT base station[26]. To monitor DECT transmissions, a Linux driver was implemented. The driver implements two modes of operation:

fpscan The card tries to receive DECT base station beacons on every possible frequency (using frequency hopping) without synchronizing the timer to any of the beacons. This makes it possible to detect DECT base stations with their respective RFPI near the card.

ppscan The card scans for a specific RFPI and then locks on the channel the base station with that RFPI is broadcasting on. The cards timer is synchronized with the broadcasts of the base station. Packets

send by this base station and by phones communicating with this base station are then recorded in a file (using the common pcap format[2]).

The recorded packets can later be analyzed using tools like wireshark or any other tools, that are able to read pcap files. Custom tools to extract the C-channel data or the audio signal have been written. C-channel messages (which contain the dialed number) can be displayed and the audio can be converted to common wav-files. This compromises all DECT systems, where the adversary is in communication range of the phone and the base station. This toolset is available on `http://dedected.org/`. Today, a different toolset has been written by Patrick McHardy, which is available on `http://dect.osmoconbb.org/`, and can be used to monitor DECT transmissions too.

A list of base stations and phones, which are known not to encrypt their communication can be found in the diploma thesis[26] of Alexandra Mengele. She examined 36 devices, 14 of them did not encrypt their calls by default.

5.2 Late Encryption

Some DECT devices encrypt their communication by default, but enable the encryption only after the call has been established. The dialed phone number, which must be transmitted before the base station is able to establish the call, is transmitted in clear and can be read. The tools described in Section 5.1 can be used to retrieve the dialed number from those devices. A list of devices, which are known to enable their encryption after the call has been established, can be found in the diploma thesis[26] of Alexandra Mengele. She examined 36 devices, 6 of them encrypted their calls, but not the phone number, 16 of them encrypted the calls including the phone number by default.

After these results have been published, at least one manufacturer released an updated firmware for one of their base stations. The updated base station enables encryption at an earlier point of the call setup, so that the phone number is protected by the encryption too.

5.3 Weak PRNGs

Good PRNGs (pseudo random number generators) are vital to the security of DECT. As shown in Section 2.4.3, when two devices are initially paired and the UAK is generated, it is computed from the PIN number and a 64 bit random number RS chosen from the base station. If we assume, that the PIN number is never changed by a user from the default value (eg. 0000), then the only source of entropy for the UAK is this random number. If an adversary can guess this number, he can reconstruct the UAK and can reconstruct all secret keys which are derived from the UAK. Alternatively, the adversary could record these numbers during the initial pairing process. This completely compromises the security of DECT.

The DECT standard does not specify a PRNG algorithm, that must be used by all DECT devices. However, a random number generator is required to implement a DECT device, for a phone and for a base station

[2] `http://www.wireshark.org/`

too. If hardware sources for randomness are not available or are difficult to access, a PRNG is used. During my research on DECT, a PRNG algorithm (shown in Algorithm 12) was recovered from a DECT device. The algorithm has been found in devices from at least two different manufacturer, so that one can assume that it is widely deployed.[26]

Algorithm 12 vendor_A_PRNG($\mathbf{xorval} \in \{0,1\}^8, \mathbf{counter} \in \{0,1\}^{16}$)

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

The algorithm is a very good example for a weak PRNG in DECT. It takes only two inputs **xorval** and **counter** of length 8 and 16 bits (24 bits in total) and generates a 64 bit value **out**, which is used directly as RAND_F or RS by the examined devices. Regardless of the internal design, at most 2^{24} different outputs can be generated by such an algorithm and in this case, only 2^{22} different outputs are generated, because some inputs collide. As long as the standard PIN is used during pairing, only 2^{22} different UAKs can be generated. A table containing all 2^{22} possible outputs of the PRNG in ASCII representation has only a size of 68 MB.

After an attacker has recorded an authentication of a phone as described in Section 2.4.1 using the tools described in Section 5.1, he can lookup the recorded RAND_F or RS in a text file, to verify that this PRNG algorithm is actually used by the base station, and check every possible UAK against the recorded RES1 value in the transmission.

A single invocation of A21 is required to generate a UAK from a guessed random number and a PIN (see Section 2.4.3) and A11 and A21 using this UAK must be computed to verify it against an intercepted RES1 and RS (denoted as $\mathrm{RS}_{\mathrm{AUTH}}$ in Figure 9) value (see Section 2.4.1). In total, 3 invocations of DSAA or 12 invocations of the block cipher cassable are needed to compute and verify a single UAK. All possible PIN values can be checked in less than a minute on a *Core2Duo* CPU. Figure 9 shows an overview of this process.

5.3.1 Brute-Force attacks using an FPGA

If the PRNG can only generate 2^{22} different outputs, the UAK can be recovered in less than a minute on a laptop computer (assuming a known PIN number). To examine, if this attack is also possible, if the PRNG outputs many more than 2^{22} different values, or a subset of all possible outputs appears with a high probability, I decided to supervise a diploma thesis[30], implementing this attack on an FPGA (Field Programmable Gate Array). DSAA uses bit permutations, 8 bit additions, 8 bit multiplications which can be replaced by additions, and a 8 bit S-Box. These operations are well suited for an 8 bit microcontroller, but can be implemented efficiently on an FPGA. This was joint work with Kei Ogata, Gregor Molter and Ralf Weinmann. I was mainly responsible for the DECT specific parts, while Gregor Molter and Kai Ogata focused on the FPGA specific parts. The result has been published at ICWMC 2009[27].

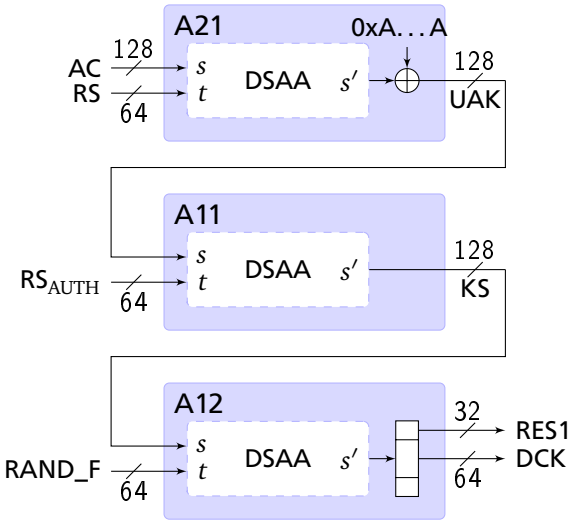


Figure 9.: Data flow of the Combined Authentication and Key Derivation

Implementation overview

We implemented our brute-force *Key Calculation Engine* (KCE) on a *XUPV5 Virtex-5 OpenSPARC Evaluation Platform* board. It is equipped with a *Xilinx Virtex-5 XC5VLX110T* Field Programmable Gate Array (FPGA).

For host PC communication one may utilize the board over the PCI bus, USB, or RS-232. We decided to communicate over the RS-232 to keep the communication-depended HW parts as small as possible. Using the PCI bus or the USB would lead to a higher resource utilization without any performance gain. RS-232 communication is sufficient for the small amount of data the host PC has to share with the FPGA.

The host PC application must be initialized with the eavesdropped values $\mathsf{RS}_{\mathrm{AUTH}}$, RAND_F, and RES1. It iterates over all possible PIN values, generating the ACs, and sends it to the board together with the eavesdropped ones. The hardware implementation takes those four parameters and calculates the result DCK, UAK, and RES1' for all possible weak random numbers RS. If RES1' matches RES1, the possible key candidates DCK and the UAK are send back to the host PC application.

Key Calculation Engine

The main computational effort of the Key Calculation Engine is done by three DSAA cipher blocks (see Figure 9). The upper DSAA cipher block embedded in the security process A21 represent the UAK generation of the On Air Key Allocation. The lower two DSAA cipher blocks embedded in the secret processes A11 and A12 represent the DSC Key Derivation. A single cipher block is shown in detail in Figure 10.

Pipeline stages:	3	6	12	72	144	72
DSAA shared:	no	no	no	no	no	yes
KCE components:	1	1	1	1	1	4
Delay [ns]:	125	$66.\overline{6}$	$33.\overline{3}$	6.25	5.263	$6.\overline{6}$
Throughput [key/s]:	$8*10^6$	$15*10^6$	$30*10^6$	$160*10^6$	$190*10^6$	$200*10^6$
Slice Flip-Flops:	2,225 (3%)	2,551 (3%)	3,521 (5%)	11,806 (17%)	20,168 (23%)	18,184 (26%)
Slice LUTs:	27,848 (40%)	28,233 (40%)	28,233 (40%)	30,017 (43%)	30,289 (43%)	42,840 (61%)
Occupied Slices:	8,019 (46%)	8,320 (48%)	8,317 (48%)	9,929 (57%)	10,060 (58%)	13,265 (76%)

Table 4.: Comparison of Pipeline Stages, Key Derivation Throughput and Utilized Resources

The DSAA takes two input values s, the key or subkey, and t, values from a random number generator. The output s' of it is a derived subkey which acts as the UAK , KS , or RES1 and DCK , respectively. The DSAA is split into two steps step1 and step2 . Each step builds half of the final DSAA output subkey s'.

The first step takes the inner 64 bits, i.e., bit 32 to 95, of the key s and the random value t. It derives a pseudo random number s' which builds the inner bits of the final subkey s' output of the DSAA. Also, this pseudo random number is feed into the next step's t. Additionally, the second step took the outer 64 bits, i.e., bit 0 to 31 and 96 to 127, of s. It calculates the remaining outer bits of the DSAA subkey s'.

Every step's part consists of six linearly concatenated microsteps, which are effectively rounds of cassable. They implement the actual subkeys derivation.

A microstep$_j^i$ takes two parameters s, the microsteps subkey, and t, its random value. The random values are shuffled by the i-th scheme of four possible butterfly networks, a bit-wise permutation, before they are passed to the next microstep. The shuffled random number is XOR-ed with the current subkey s and the result is byte-wise substituted by an sbox. Afterwords it is mixed by the j-th scheme of three possible mixing structures. A mixing structure connects two input bytes with an multiply-accumulate to each output byte. Chapter 3 gives a detailed mathematical description of bitpermi, sbox, and mixj.

Resource Sharing

The Key Calculation Engine is very resource intense. It utilizes 864 8-bit adders, 144 64-bit XORs, and 1152 8-bit ROMs for the sboxes. After having evaluated several designs, we decided to instanciate three DSAA blocks, to maximize the throughput. These three blocks don't share any resources (variant 1). In addition, we implemented a second design instanciating only a single DSAA block for the whole dataflow (variant 2).

Pipelining

The whole data flow from Figure 10 could be implemented as one big logic block which results in a high latency. Therefore, we implemented different pipelined versions of the variant 1 and 2. We placed pipeline registers at the different outputs of the logical blocks, i.e., at the outputs of DSAA, step1 and step2 , part1 and part2 , or microstep$_j^i$.

This leads to different pipeline stage length, i.e., 3, 6, 12, or 72 stages. All those pipeline registers have to be at least 64-bit wide, leading to a high amount of additionally utilized flip-flops. Please keep in

mind that both the DSAA and the steps partially forward input values to the next step or part . Pipeline registers also have to be added on those forwarded signals.

To see if we can break down the longest delay paths any further, we created a pipelining version where one additional pipelining register was placed into the critical path of each $\mathsf{microstep}_j^i$ resulting in a pipeline with 144 stages.

Results

The performance of our implementation, with different resource sharing schemes and pipeline stages, is shown in Table 4. The first five columns describe the implementation variant 1 with no resource sharing, but with different pipeline stages. The last column describes the implementation variant 2 which shares resources and utilize four brute-force components in parallel. In general, adding more pipeline stages results in lower signal delays and therefor in a higher throughput.

More throughput can be even achieved by further utilizing more brute-force components, but this performance gain is only small compared to the increased occupation of FPGA slices. The implementation variant 1 with 144 pipeline stages has a throughput to slices ratio of about 20,000 key derivations per slice, while the variant 2 with 72 pipeline stages has only a ratio of about 15,000 key derivations per slice.

Nevertheless, implementation variant 2 with 72 pipeline stages performs best in terms of throughput. To brute-force the used DCK of the described base station implementation with the weak random number generator, we have to search through an input space of 2^{22} for a single PIN or $2^{35.29}$ for all four-digit width PINs. Our implementation can brute-force 2^{22} ($2^{35.29}$) key derivations within 20.97 msec (210 sec).

Because RES1 is only 32 bit long, some false positives will be reported during such an attack. One can verify the reported UAK*s* by checking a second authentication with a different RES1, or one can use the recovered UAK to derive the DCK that was used to encrypt the phone call (if the call was encrypted) and decrypt the phone call. If the UAK is incorrect, it will not decrypt the phone call with a very high probability.

5.4 No Encryption Enforced

As described in Section 2.5, encryption is enabled by the base station by sending a `CIPHER-REQUEST` message to the phone. A phone can suggest to start ciphering by sending a `CIPHER-SUGGEST` message to the base station. Alexandra Mengele[26] didn't find a single phone, which requested encryption from the base station. She did not find a single phone requesting authentication from the base station too. As a result, as long as encryption is not enabled and authentication is not used, the UAK shared between the phone and the base station is not used at all. An attacker can impersonate a base station, not enable encryption and then monitor and reroute phone calls.

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 so that the RFPI field in this structure was overwritten with an RFPI value of our choice directly after the original value was written there. Then we modified the routine comparing the RES1 values returned by the phone with the computed XRES1 values. We verified that we were indeed broadcasting a fake RFPI.

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 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 disabling the original base station for a short time, the handset switched to our faked base station. 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 phone. 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 phone calls with equipment as expensive as a wireless LAN card, making attacks on DECT as cheap as on wireless LANs. Alexandra Mengele examined 36 different phones, none of them refused to communicate with our base station.

After these results have been published, at least one manufacturer released an updated firmware for one of their phones. With the new firmware, the phone drops calls, if encryption is not activated in the first seconds. This behavior can be disabled by the user of the phone.

5.5 Jamming Base Stations

In the last Section, we manually disabled the original base station to make a phone switch to our impersonated base station. This is acceptable for a lab setup, to demonstrate that a phone will not request authentication from a base station or display a warning, if encryption is not active during a call. However, during a real attack, the attacker might not be able to disable the original base station for a short time. This problem can be solved.

Patrick McHardy developed a tool named *hijack*, that is part of *libdect*, which broadcasts frames in the same channel and time slot as the original base station suggesting that the phone should change it's frequency and time slot. When the phone has received such a packet, it will switch to the frequency and time slot we suggested and switch to our base station.

Alternatively, an attacker could use radio jamming to make the phone loose communication with the base station, so that it starts scanning for a base station and might pick our impersonated base station by chance.

As a result, the attack described in Section 5.4 can also be applied, if the original base station can not be disabled.

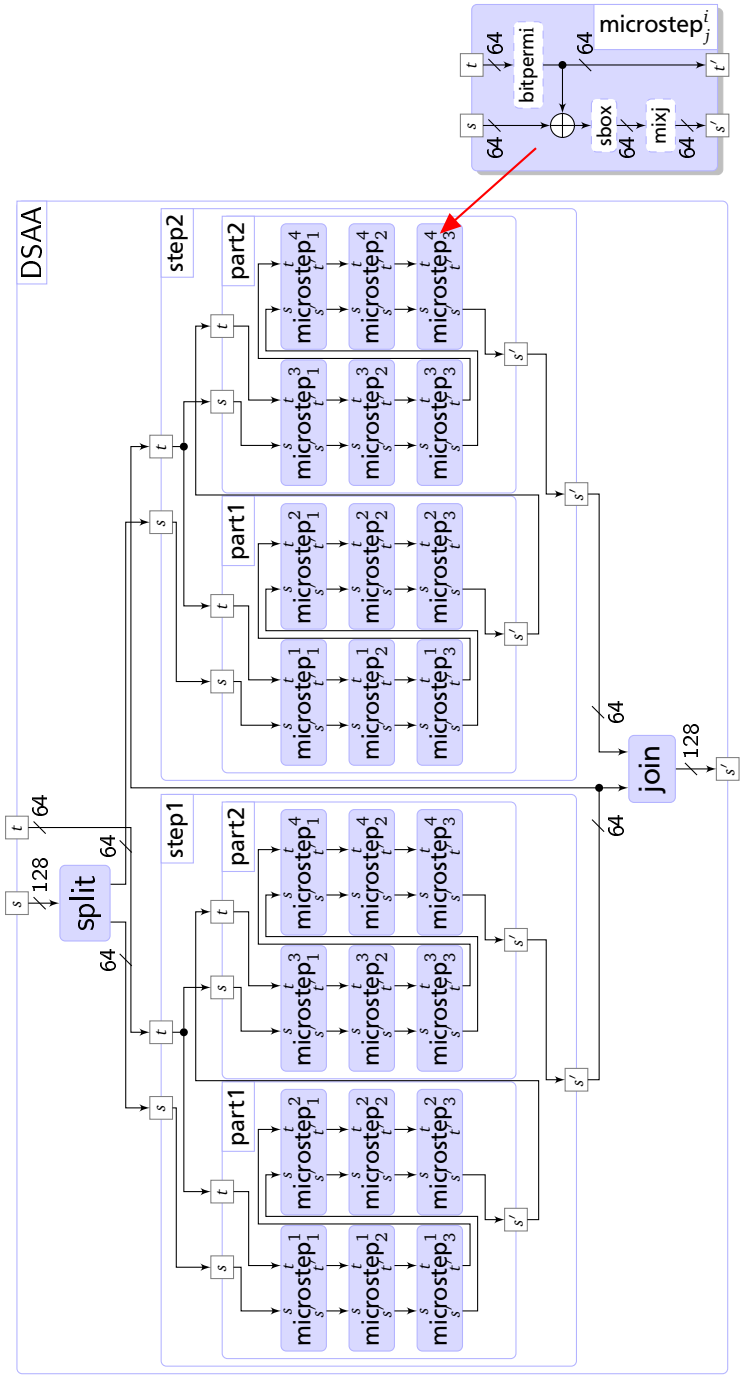


Figure 10.: Implementation of one DSAA cipher block

6 Attacks on DSAA

DECT uses DSAA for the generation of the long term key UAK as well as for the generation of the session key DCK. This makes DSAA extremely interesting for cryptanalysis since a compromise of the long term key would allow an attacker to decrypt every phone call and to impersonate a legitimate phone, no matter which security features of DECT are used. The compromise of a session key would allow an attacker to decrypt a single phone call. DSAA itself was formerly kept secret, but has been reverse engineered and is now described in Chapter 3.

Section 6.1 deals with a problem of the usage of DSAA in DECT GAP[10] that reduces the cryptographic strength of the long term key UAK to only 77.288 bits. A good long term key could have a cryptographic strength of 128 bits. In Section 6.2, another attack on DSAA is described that recovers the UAK with a efford of about 2^{64} calls to DSAA.

The main building block of DSAA is a family of block ciphers named cassable. In Section 6.3, this block cipher is analyzed. The attacks described in this Section show that cassable has structural weaknesses; however, those weaknesses have no direct effect on the security of DSAA itself. An overview of cassable is given in Figure 11.

Parts of this Chapter are joined work with Stefan Lucks, Andreas Schuler, Ralf-Philipp Weinmann, and Matthias Wenzel and have previously been published at CT-RSA 2009[23].

My main contributions to this Chapter are the attack in Section 6.2 and the linear analysis of cassable in Section 6.3.2.

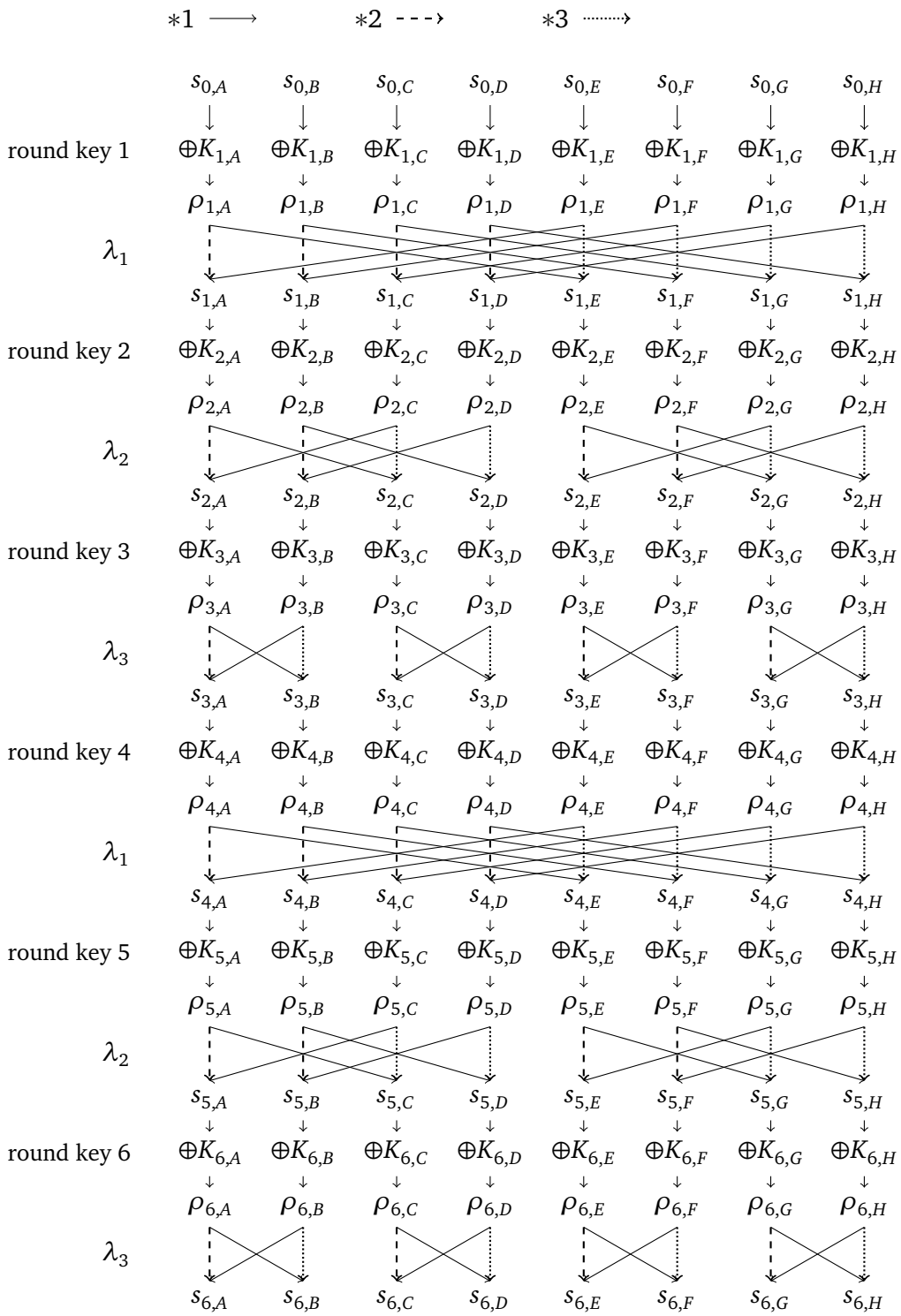
6.1 GAP Key Allocation Entropy Problem

This Section shows that the long term key UAK in DECT usually does not have the full entropy of 128 bits. Instead, only 77.288 bits of entropy can be expected, if the UAK is generated using the pairing process from DECT GAP[10].

Most DECT phones allow automatic pairing, as described in Section 2.4.3. The resulting long term key UAK is generated using DSAA from a 64 bit random number chosen by the base station and a PIN number the user has to enter. Since the PIN number is limited to 4 digits, at most $10^4 \approx 2^{13.288}$ different choices are possible. Assuming that the random number generator used by the base station can actually generate 2^{64} different random numbers, at most $2^{77.288}$ different UAKs can be generated. Section 5.3 describes, what happens, if the random number generator on the base station is weak and cannot generate 2^{64} different outputs.

Usually a DECT base station ships with a default PIN number which doesn't need to be changed by the user. If the PIN number was still the default PIN number during the pairing process, we can safely assume that the attacker knows this PIN number, which in turn means that only 2^{64} different UAKs can be generated using this specific PIN number.

Figure 11.: Structure of the cassable block cipher



As a result, the UAK only has at most 77.288 bits of entropy, and might even only have 64 bits of entropy if the default PIN number was used. This result was previously known[6] by the German *Bundesamt für Sicherheit in der Informationstechnik (BSI)*. No inside knowledge of the formerly secret DECT algorithms is required to note that.

However, if a DECT handset is sold pre paired with a DECT base station, the manufacturer can choose not to use DECT GAP pairing, to pair the devices. Directly writing the pairing and the UAK into the memory of the devices during production will most certainly be cheaper. These devices could be shipped with a UAK with really 128 bits of entropy.

6.2 Key Recovery in 2^{64} Operations

As described in Chapter 3, the middle 64 bits of the output of DSAA (bits 32 to 95) only depend on the middle 64 bits of the key (bits 32 to 95) and the 64 bit random number. This allows trivial attacks against DSAA, which allow the recovery of all 128 secret key bits with an effort in the magnitude of about $3 * 2^{64}$ evaluations of DSAA.

To recover the UAK, one proceeds as follows: An attacker needs to have access to a recorded phone call. He chooses a random UAK and iterates over all possible combinations of bits 32 to 95 and uses A11 and A12 with the inputs RAND_F and RS from the recorded phone call to generate all possible cipher keys. Each of the cipher keys is tested against the recorded phone call. One of the cipher keys will decrypt the phone call correctly, and bits 32 to 95 of the UAK, that generated this DCK can be assumed to be correct. If multiple inputs generate this cipher key, these bits need to be checked against a second phone call. Because only two of the 4 block cipher calls in DSAA are used to generate these bits of output, the workload for this procedure is only as high as 2^{64} calls to DSAA. In addition, 2^{64} trial decryptions with DSC need to be performed. If we assume that a trial decryption of DSC takes as long as a call to DSAA, the total workload is as high as $2 * 2^{64}$.

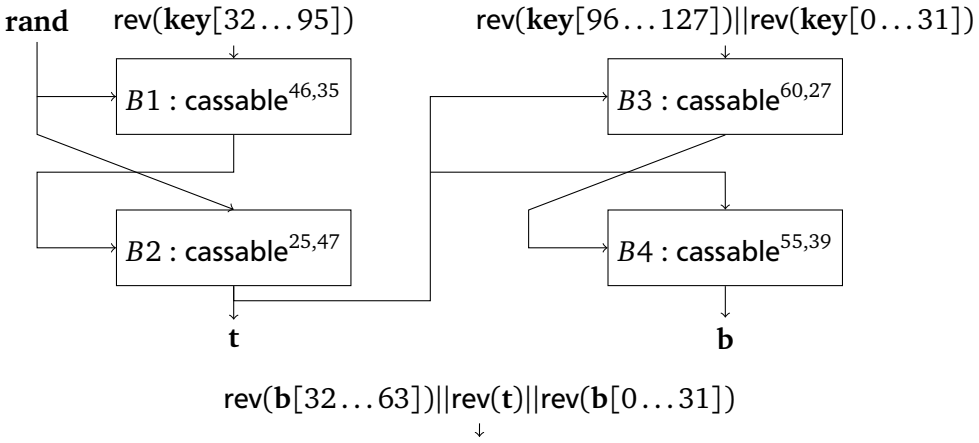
After this has been completed, bits 0 to 31 and 96 to 127 of the UAK needs to be determined. These bits generate the RES1 value in the phone call that was transmitted during the authentication at the beginning of the call. The attacker iterates over all possible values (using the correct bits for bits 32 to 95) and about 2^{32} inputs will generate the RES1 value. Theses candidates are then checked against a second or third phone call, until only a single input remains. Again, the workload for this part is not much more than 2^{64} calls to DSAA.

The total workload of the attack is roughly equivalent to $3 * 2^{64}$ calls to the block cipher. Even if attacks against the DSAA cannot improved past this bound, keep in mind the entropy problems of the random number generators described in Section 5.3.

6.3 Cryptanalysis of the cassable Block Cipher

Apart from the issues described above, 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. This Section shows

Figure 12.: DSAA overview



that cassable as a block cipher can be effectively broken using differential cryptanalysis, and that an effective attack on the full DSAA would be possible, if cassable would only use 3 instead of 6 rounds.

The DSAA can be interpreted as a cascade of four very similarly constructed block ciphers. We call this family of block ciphers cassable. A member of this family is a substitution-permutation 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 $\sigma_i : GF(2)^{64} \to GF(2)^{64}$ with $1 \leq i \leq 4$ denoting bit permutations that are used to derive the round keys from the cipher key. It is equivalent to bitperm in the pseudocode.
- $\rho : GF(2)^8 \to GF(2)^8$ denotes the application of the invertible S-Box that is given in Algorithm 11.
- The function $\lambda_i : (\mathbb{Z}/256\mathbb{Z})^8 \to (\mathbb{Z}/256\mathbb{Z})^8$ denotes the mixing functions used in the block ciphers. They are equivalent to mix1, mix2 and mix3 in the pseudocode. The function $\gamma : GF(2)^{64} \to GF(2)^{64}$ is a bricklayer transform that is defined as:

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

with $A, B, \ldots, H \in GF(2)^8$ and 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:

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

6.3.1 The mixing layer

In order to diffuse local changes in the state bits widely, the functions λ_i with $1 \leq i \leq 3$ (mix1, mix2, and mix3 in the pseudo-code) are used. These form a butterfly network visualized in Figure 11. At first glance 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 lowest most bit of c will be the lowest most bit of b and not depend on a at all. This observation will be used in Section 6.3.4.

6.3.2 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. Interpolating the S-Box over $GF(2^8)$ yields dense polynomials of degree 254, interpolation over $GF(2)$ results in equations of maximum degree.

An example for a 1 round approximation would be

$$s_{1,D} \equiv s_{0,H} \oplus K_{1,H} \bmod 2$$

that holds with probability:

$$p_1 = \frac{120}{256}$$

For a two round example, we can use the approximation:

$$s_{2,B} \equiv s_{1,D} \oplus K_{2,D} \bmod 2$$

that holds with a probability of:

$$p_2 = \frac{120}{256}^2 + \left(1 - \frac{120}{256}\right)^2 \approx \frac{128.5}{256}$$

Three rounds can be approximated with:

$$s_{3,A} \equiv s_{2,B} \oplus K_{3,B} \bmod 2$$

that holds with a probability of:

$$p_3 = (p_2 * p_1) + (1 - p_2) * (1 - p_1) \approx \frac{127.97}{256}$$

6.3.3 The key scheduling

The bit permutations used in cassable don't have maximum order. A permutation that permutes 64 bits could have order 64, but all permutations used by cassable only have order 8 or 16. This is just an observation and has not been exploited here.

6.3.4 A practical attack on cassable

The individual block ciphers used within the DSAA can be fully broken using differential cryptanalysis [4] 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.

$$m_B = m'_B, m_D = m'_D, m_F = m'_F, \text{ and } m_H = m'_H$$

Now both inputs are encrypted. Let $s = s_{i,A}||\ldots||s_{i,H}$ and $s' = s'_{i,A}||\ldots||s'_{i,H}$ be the states after i rounds of cassable. After the first round,

$$s_{1,B} = s'_{1,B}, s_{1,D} = s'_{1,D}, s_{1,F} = s'_{1,F}, \text{ and } s_{1,H} = s'_{1,H}$$

holds true. This equality still holds after the second round. After the third round, the equality is destroyed, but

$$s_{3,A} \equiv s'_{3,A} \bmod 2, s_{3,C} \equiv s'_{3,C} \bmod 2, s_{3,E} \equiv s'_{3,E}, \text{ and} s_{3,G} \equiv s'_{3,G} \bmod 2$$

still holds true. The key addition in round four preserves this property, but the fourth application of the S-Box $\rho_{4,j}$ destroys it.

An attacker can use this property to recover the secret key of the cipher. Assume the attacker is able to encrypt such two messages m and m' with the same secret key and see the output. He can invert the mix3 and sub 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

$$s_{3,A} \oplus K_{4,A}, s_{3,E} \oplus K_{4,E}, s'_{3,A} \oplus K_{4,A}, \text{ and } s'_{3,E} \oplus K_{4,A}$$

he checks whether

$$s_{3,A} \oplus K_{4,A} = s'_{3,A} \oplus K_{4,A} \mod 2, \text{ and } s_{3,E} \oplus K_{4,E} = s'_{3,E} \oplus K_{4,E} \mod 2$$

holds true. 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. When this procedure is iterated with 15 pairs in total, only about 2^{34} possible keys are expected to remain. These can then be checked by 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 at 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 which 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 a successful attack. Even if not enough pairs are available to the attacker an attack is still possible with increased computational effort and more memory usage.

6.3.5 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. However, this is just an academic attack, because cassable always uses 6 rounds as used in DSAA.

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 true for z_0:

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

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 that discards $\frac{255}{256}$ of the searched key bit subspace.

In the following paragraphs we give an example of how the attack 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 very same as for z_2, which means that 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 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 that the attack requires 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 remain 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.

7 Attacks on DSC

Besides DSAA, the second security algorithm used in DECT suffers from structural weaknesses too. DSC is vulnerable against a clock guessing attack that recovers the secret cipher key. This Chapter starts with a basic form of the attack in Section 7.2, that explains the basic idea of the attack. The attack is then extended to a much faster attack, that recovers the key in minutes to hours (assuming enough keystreams are available) in Section 7.3. Section 7.4 contains consideration, which must be taken into account, when trying to recover a key from an intercepted phone call. Section 7.5 shows, that the attack is still possible, if the first 40 bits of output of the cipher are missing. Section 7.6 contains further improvements to lower the data complexity of the attack. In Section 7.7 describes, how the main part of the attack can be implemented on a FPGA to make the attack faster.

One should read Chapters 2 and 4 first, which introduces the notation used in this Chapter to describe the cipher and the attack. Chapters 3, 5, and 6 are not required to understand this Chapter.

Parts of this Chapter are joined work with Jan Krissler, Karsten Nohl, Andreas Schuler, Michael Weiner, and Ralf-Philipp Weinmann, and have previously been published at FSE 2010[29] and ICISC 2010[31]. Parts of this Chapter have also been accepted to WISA 2010 as a short paper.

My main contributions to this Chapter are the design of the basic attack in Section 7.2 as well as the improvements in Sections 7.3, 7.5, 7.6. I also checked how applicable the attack is in Section 7.4. I also contributed minor parts to the high-speed implementations described in Section 7.7.

7.1 DSC at a Glance

The DECT Standard Cipher is a stream cipher described in Chapter 4. It generates a keystream of usually 720 bits from a 64 bit secret key CK and a 35 bit public initialization vector IV. Usually, many keystreams are generated using a fixed CK and different IVs. For this Chapter, we assume that an attacker can request as many keystreams for random IVs as he likes, and would like to recover the ciphers key CK. This scenario is usually named *Random-IV model* when analyzing stream ciphers. For a good stream cipher, an attacker should not be able to do this significantly faster, than by trying all possible keys. However, this is not the case for DSC.

The attack described in this Chapter is similar to the Ekdahl-Johansson attack[8] against A5/1 from GSM[1]. The attack, although it has a large data complexity, can be executed on a PC in hours and recovers the ciphers key with a high probability. However, we were not able to carry over later improvements of the Ekdahl-Johansson attack [24, 3] due to specific properties of A5/1 that are not present in the DSC.

DSC is stronger than A5/1 by indicators such as the non-linearity of the round and filter function, key size and state size. However, DSC as used in DECT is initialized in less than half the number of rounds when compared to A5/1 in GSM[1].

The attack on DSC presented here provides a trade-off between the number of available data samples, the time needed to calculate the secret key and the success probability.

7.2 Simple Clock Guessing

This Section shows that the keyspace for DSC can be reduced from 2^{64} possible keys to 2^{58} keys with a high probability. This is done only to explain the basic idea of the attack. The number of remaining keys is lowered to a much better bound in the next Sections, using this basic idea described here:

If an internal state for DSC is randomly chosen from a uniform distribution of all states, every irregularly clocked register clocks twice with 50% probability or three times with 50% probability. We assume for now that the probability that one register is clocked twice is independent of the clocking decision of the other irregularly clocked registers. The probability that one register is clocked k times during the pre-ciphering-phase is:

$$\binom{40}{k-80}2^{-40}$$

and the probability that a register has been clocked k times after i bits of output is:

$$\binom{40+i}{k-(80+2i)}2^{-(40+i)}$$

The total number of clocks per register after i bits of output is distributed according to a shifted binomial distribution with mode:

$$\left\lfloor \frac{i+1}{2} \right\rfloor + 2i + 100$$

In general, let

$$\mathsf{D}^{i,j,k} = \mathsf{S} \times \mathsf{C}_1^i \times \mathsf{C}_2^j \times \mathsf{C}_3^k \times \mathsf{L} \times (\mathsf{key}, \mathsf{iv})$$

be the state of the six bits of the registers which generate the output, after key and iv have been loaded, and register R1 has been clocked i, register R2 has been clocked j, and register R3 has been clocked k times.

The attack focuses on the internal DSC state from which the second bit of output is produced. An attacker who has observed the first bit of output knows the state z_0 of the memory bit of the output combiner. The second bit of output depends on 6 bits of the registers R1, R2, and R3. With a probability of:

$$p = \left(\binom{41}{21}2^{-41}\right)^3 \approx 2^{-9.09}$$

all of these irregularly clocked registers will be clocked exactly 103 times before the second bit of output z_1 is produced and we have

$$\mathsf{D}^{103,103,103}(\mathsf{key}, \mathsf{iv}) = \mathsf{S} \times \mathscr{D}_{41}(\mathsf{key}, \mathsf{iv})$$

with probability 1. If the number of clocks per register is different, this equation will hold by chance with a probability $\frac{1}{64} = 2^{-6.00}$. Therefore, we have:

$$\text{Prob}\left(\mathsf{D}^{103,103,103}(\mathsf{key},\mathsf{iv}) = \mathsf{S} \times \mathscr{D}_{41}(\mathsf{key},\mathsf{iv})\right) = p + \frac{1-p}{64} \approx 2^{-5.84}$$

Based on this guess, six affine-linear equations for an unknown key can be derived given that sufficiently many keystreams (about 2^{18}) are available. For every IV iv, the attacker computes

$$\mathsf{I}^{103,103,103} = \mathsf{D}^{103,103,103}(0,\mathsf{iv})$$

He then checks for every possible state s of the six output bits whether $\mathscr{O}(s,z_0) = z_1$ holds. If so, this is an indication that

$$\mathsf{D}^{103,103,103}(\mathsf{key},0) = s + \mathsf{I}^{103,103,103}$$

holds. After having processed all available keystreams, the attacker may assume the most frequent value for $\mathsf{D}^{103,103,103}(\mathsf{key},0)$ to be correct. Using these six affine-linear equations allows an attacker to recover the correct key by trying only 2^{58} instead of 2^{64} possible keys. This basic attack however is still too time-consuming to be practical on a single PC. We can refine the basic attack principle to give us a much faster attack that allows us to practically recover a DSC key on a PC.

Note that the attack scope can be extended to different assumptions for the number of clockings for the registers R1, R2 and R3. In the basic attack, we use the mode of the distribution of the number of clock of all registers. If the total number of clocking decisions is odd, another set of clockings with the same success rate always exists. For example, if the attacker assumes that R1 and R2 have been clocked 103 times as in the previous Section, but R3 has been clocked 102 times instead of 103 times, the previous attack works with the same computational effort and success rate. In total, there are 8 possible assumptions about the number of clocks with the same success rate as the previous attack.

However, these different assumptions share many equations for the key. An attacker will only obtain nine different affine-linear equations for the key using these eight assumptions (compared to six equations for a single assumption).

We can extending the attack scope further to increases the success rate of the attack but at an even smaller incremental gain per additional assumption, i.e. assuming that R1 has been clocked 101 times, and R2 and R3 have been clocked 102 times.

Another way to broaden the attack is to focus on different keystream bits. The basic attack only uses the first two bits of the output, z_0 and z_1. Instead of guessing how many times the registers have been clocked before producing z_1, one could guess how many times the registers have been clocked before z_2 is produced. For example, an attacker can try using z_1 and z_2 of the output and guess that R1, R2, and R3 have been clocked exactly 105 times. The resulting correlation will have the same success rate as the one from the basic attack, while other correlations based on further bits of the output will have a worse success probability. Using multiple output bits for a single clocking triplet is possible.

Combining these two improvements, we developed a more advanced key-recovery attack on the DSC that merely requires hours of computation on a PC given enough keystreams. We chose a clocking interval $C = [102, 137]$, and generated all $35^3 = 42875$ possible approximations with the number of clocks of R1 to R3 in C. We introduce new variables $x_{i,j}^{(t)}$ for the state of bit j of Register Ri after it has been clocked t times. Assuming that Ri has been clocked t times for an approximation gives us information about $x_{i,0}^{(t)}$ and $x_{i,1}^{(t)}$. In total, a clocking-interval of length l gives us information about $6l$ variables (3 registers, 2 variables per clocking amount). However, $x_{i,1}^{(t)} = x_{i,0}^{(t+1)}$ holds for all registers Ri, because $x_{i,1}^{(t)}$ is just shifted to $x_{i,0}^{(t+1)}$ with the next clock of Ri (see Figure 4.3.2 for details). Choosing a different feedback polynomial with a feedback position between the two bits contributing to the output combiner would destroy this structure. However for DSC, all feedback polynomials do not have a feedback position here. Effectively, this gives us information about $3(l+1)$ variables for a clocking interval of length l. We will always use $x_{i,0}^{(t+1)}$ instead of $x_{i,1}^{(t)}$ for the rest of this thesis.

There are also linear relations between these variables. For example $x_{1,5}^{(t+1)} = x_{1,6}^{(t)} \oplus x_{1,0}^{(t)}$ holds. In general, having determined a consecutive sequence of variables $x_{i,0}^{(t)}, x_{i,0}^{(t+1)}, \ldots$ for a register Ri, is equivalent to the output sequence of Ri. If more variables than the length of Ri have been determined, one might use these linear relations to check if a given assignment is feasible. However, we did not use this in our attack.

The success rate that register R1 is clocked i times, register R2 is clocked j times and register R3 is clocked k times after l bits of output have been produced is:

$$p_{i,j,k,l} = \binom{40+l}{i-(80+2l)}\binom{40+l}{j-(80+2l)}\binom{40+l}{k-(80+2l)}2^{-(40+l)3}$$

In theory, one could use all available bits of keystream for which the correlation has better than zero success rate, however after 19 bits of keystream, all of these correlations have negligible success probability. For example the probability that all registers have been clocked 137 times (the end of our clocking-interval) for the 19th bit of output is below 2^{-26}.

As in the basic attack, we evaluate all correlations separately and create a frequency table for every correlation. Following the ideas of Maximov et al.[24] we add the log-likelihood ratio $\ln \frac{p}{1-p}$ for $\mathsf{key} = s + \mathsf{iv}$ to every entry in the table, with

$$p = \sum_l p_{i,j,k,l} * [\mathcal{O}(s, z_{l-1}) = z_l] + \frac{1}{2}\left(1 - \sum_l p_{i,j,k,l}\right)$$

Here $[\mathcal{O}(s, z_{l-1}) = z_l] = 1$, if $\mathcal{O}(s, z_{l-1}) = z_l$, 0 otherwise.

Instead of writing the equations in all correlations as a linear combination of key bits, we now write all equations in the form:

$$x_{\{1,2,3\},0}^{(i)} = \{0, 1\}$$

Taking the entry with the highest probability from the frequency table of every approximations, we obtain $42875 * 6 = 257250$ equations with a given probability. Every approximation (42875) gives us information about the value of 6 state-variables. In total, these state-variables can have $2^6 = 64$ possible values. The value with the highest probability in the frequency table is most likely to be correct. We use the number of weighted votes for the top entry as an extend p_i how likely these equations are correct.

For every variable x, we take all the equations of the form $x = b_i, b_i \in \{0, 1\}$ with extend p_i and compute

$$s_x = \sum_i (2b_i - 1) * p_i$$

and assume that $x = 0$, if $s_x < 0$, or $x = 1$ otherwise.

Combining all equations to a single equation system gives 108 equations each one depending only on a single variable and a corresponding probability p_v for the correctness of this equation. We sort these equations according to $|p_v|$, rewrite all variables to key bits, and add them in order to a new linear equation system for the key bits. If adding a equation would make the resulting system unsolvable, we skip that equation. If enough linearly independent variables (for example 30) have been added to the system, we stop the process. We then iterate through all solutions to this system, and check every solution if it is the correct key, by comparing it to a few sample keystreams.

We created a proof-of-concept implementation of this attack written in Java. Processing all available keystreams and generating a linear equation system takes about 20 minutes using a SUN X4440 using 4 *Quad-Core AMD Opteron(tm) Processor 8356* running at 2.3 GHz. The main workload here is the generation of all the frequency tables for all approximations. The post processing and the generation of the final equation system is negligible. We think that this time can be reduced to a few minutes using parallel computation and a more efficient implementation. For the time for the final search of the correct key, see Section 7.7.

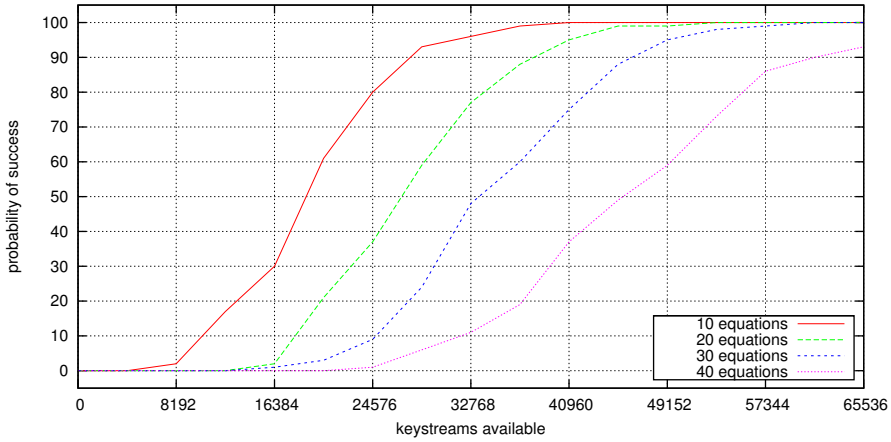
The success rate of this attack depends on the number of available keystreams and the number of equations in the final equation system for the keybits. Using more equations makes the final search for the correct key faster, but increases the probability of having at least one incorrect equation in the system which makes the attack fail. If i equations are used in the final system, one still needs to search through at most 2^{64-i} different keys to find the correct key (assuming the equation system is correct).

Using 30 equations in the final system (one still needs to check at most 2^{34} different keys), the attack was successful in 48 out of 100 simulations with 32768 different keystreams available. Using only 16384 keystreams, the success rate dropped down to 1%. With 49152 keystreams, the attack was successful in 95% of all simulations. If only 8 equations should be used, the attack had a success rate of 8% using just 8192 keystreams. However an adversary would need to conduct a final search for the key over 2^{56} different keys, which is roughly equivalent to a brute force attack against DES. See Figure 13 for details.

7.4 Keystream Recovery

To break the DSC, keystream needs to be recovered from the encrypted frames, which is only possible when the user data is known or can be guessed. Known user data is regularly sent over DECT's control

Figure 13.: Success rate of the attack



channel (C-channel). The C-channel messages (e.g., for a button press) share a common structure in which the majority of the first 40 bits stays constant. There are at most 50 C-channel packets sent per second which provides an upper bound on the number of known keystream segments from the C-channel. Especially in newer phones, the C-channel is extensively used for status updates including RSS feeds and other data communication which opens the possibility that a significant number of known keystream can be gathered.

Keystreams can also be collected from the voice channel (B-field), but assumptions have to be made about the voice being transmitted (i.e., segments of silence). Even when these assumptions do not hold in all cases, the data is still usable in the attacks outlined in the next Sections as they are error-resilient. A good overview how keystreams are used to encrypt DECT packets is given in Figure 6.

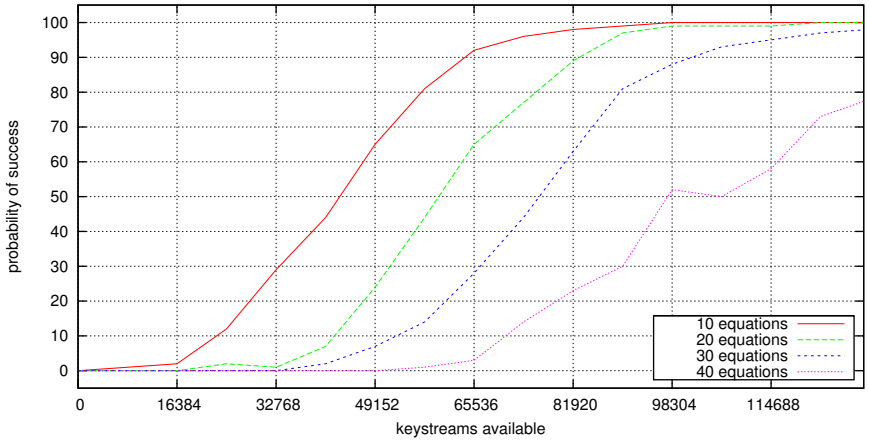
7.5 Extending the Attack to B-field Data

So far we assumed that the adversary has access to the first bits of output of DSC after the pre-ciphering phase. However, these bits are only used to encrypt the C-channel data in DECT. If C-channel data is not frequently used in a conversation, the adversary is unable to recover a sufficient number of keystreams using the techniques previously described. Henceforth, we adapt our attack to also work when the first 40 bits of keystream are not available.

To achieve this, we need to change the clocking interval from $[102, 137]$ to $[204, 239]$. We then use 21 bits of the keystream starting from bit 41. The best approximation which exists is to assume that every register has been clocked 202 times when the second bit for the B-field is produced. This happens with probability $2^{-10.527}$, instead of $2^{-9.0915}$ for the best approximation for the C-channel bits.

As expected, the number of keystreams required for the same success rate is increased by factor of 2-3. To make the attack work with a success rate of 50%, the attack requires 75,000 keystreams. Again we conducted 100 simulations to experimentally verify the success rate and to generate the plot in Figure 14.

Figure 14.: Success rate of the B-field attack



However, B-fields are sent 100 times per second from base station to the phone while a call is in progress. This allows to recover the corresponding keystreams in less than 13 minutes if a predictable plaintext pattern is used in the B-field.

7.6 Key Ranking

To improve the attack, we introduce a key ranking procedure. The attack generates equations of the form $\sum_i a_i k_i = \{0,1\}$ where k_i is a bit of the key and a_i is either 0 or 1. The left part of the equation only depends on the feedback polynomials of the registers. The right part of the equation is either 0 or 1, determined by a voting system. The difference between the number of votes for 0 and 1 is denoted by $|s_x|$ in Section 7.3. The equations are sorted by s_x and the topmost equations are assumed to be correct. Using many equations results only in a small remaining keyspace which needs to be searched, but increases the probability that at least one equation is incorrect and the key is not found in the set of solutions of the linear equation system.

For the standard attack, it is never necessary to compute the success probability of an equation explicitly. Instead one can just sort all equations by $|s_x|$, assuming that equations with a higher difference have a higher success probability. One can compute the explicit probability for an equation from $|s_x|$. First, one can simulate the attack against 100 random keys (using the same number of keystreams) and collect all generated equations with their voting difference and correctness. It is now possible to compute the success probability $P(|s_x|)$ of an equation using this data and a nearest neighbor smoother or similar methods (kernel smoother...).

Assuming that we have a set of equations e_i with respective individual success probabilities $P(|s_{x_i}|)$ and that the success probabilities are independent, we can run a best-first-search for the correct key (if we use 64 equations) or for the most promising sub key space (if less than 64 equations are used). We assume that the set of possible keys or sub key spaces is a directed graph $G = (V, E)$. A node v consists

of a vector c indicating which equations e_i are correct ($c_i = 0$) and which of them are incorrect ($c_i = 1$). The probability that this node represents the correct sub key space is:

$$\prod_i \left(|c_i - P(|s_{x_i}|)| \right)$$

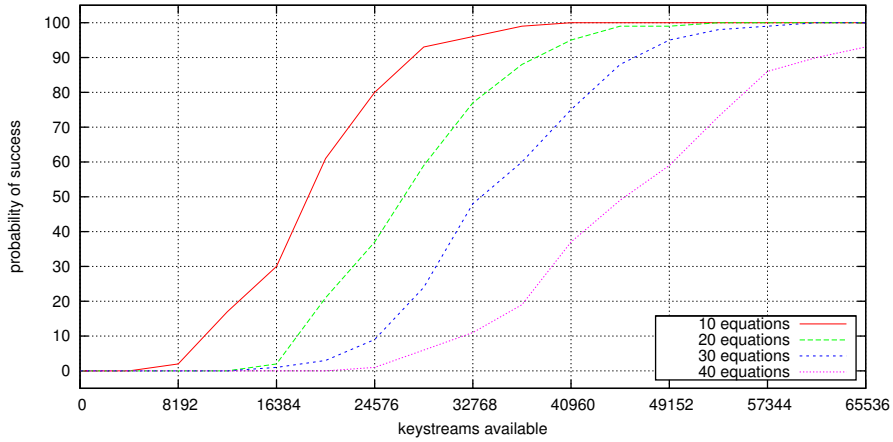
The node which the highest probability is the node with $c = (0, \ldots, 0)$ where all equations are assumed to be correct. An edge (v_1, v_2) exists if v_1 and v_2 differ only in a single equation, which is assumed to be correct in v_1 but assumed to be incorrect in v_2.

We can now run a best-first search for the correct sub key space on this graph starting at the node with the highest success probability. Using 64 equations would guarantee that all keys are visited in the exact order of probability, however we think that the number of equations should be limited for generating the keys to check and highly parallel hardware like CUDA graphic cards or FPGAs can be used in an efficient way. A high speed implementation of this last phase is described in [29].

7.6.1 Experimental results

Executing the attack from Section 7.3 against 100 randomly chosen keys only resulted in 48% success rate with 2^{15} keystreams available and 2^{34} keys checked. Using our new key ranking method allowed us to recover the key in 76% of all tests, with also 2^{34} keys checked in total. We used 42 instead of 32 equations, but checked the 1024 most likely sub key spaces. Figure 15 includes more details.

Figure 15.: Success rate of the improved attack



7.7 FPGA Implementation

FPGAs are very well-suited for an implementation of the final search phase of a key sub-space. Linear Feedback Shift Registers form the main part of the DSC algorithm, and they can be implemented much more efficiently on an FPGA than on a CPU or GPU platform. We decided to implement the time consuming final search for the correct key on an FPGA.

Basic Implementation Idea

An FPGA implementation of the improved DSC attack requires the knowledge of a valid reference *(IV, Keystream)* pair. It must iterate over all potentially valid cipher keys (according to an equation system), compute the keystream and compare it to the known keystream. Therefore, a cipher key generator, a DSC keystream generator and a compare unit comparing the keystream output to the reference keystream is necessary for the FPGA implementation.

When an attack against a particular key is run, the FPGA is given a reference tuple *(IV, Keystream)* as well as equation systems of the form $Ak = b$ that constrains the key space by only allowing keys k that satisfy all equations. The device shall report cipher keys that produce a reference keystream. Cipher Keys that generate the reference keystream must be sent to a PC which controls the FPGA.

The most convincing way to implement the key generator is using a counter or full-cycle LFSR that generates the *independent* bits and a combinatorial function generating the *dependent* bits that uses the *independent* bits as an input. The equation systems must be transformed beforehand for this purpose, such that the dependent bits are described as a function of the independent bits. The DSC keystream generator can be implemented straight-forward as described in Section 4.3.

Optimizations

Optimizations are possible on several levels compared to a straight-forward implementation:

Basic Improvements This includes sharing the key generator among several DSC units, removing unnecessary control signals, or keeping logic delays short by inserting registers on critical paths.

DSC Speedup The fundamental DSC implementation as described in Section 4.3 requires three clock cycles per bit of keystream output. This can be reduced to one clock cycle by multiplexing and re-arranging the feedback taps. The new tap positions were determined by using a matrix representation of the LFSRs. [2]

Key Loading Section 4.3 proposes to load the session key in 128 clock cycles. The same can be done in one cycle by loading the initial state in parallel instead of clocking it in. The combinatorial function transforming the cipher key into the initial state can be obtained by using the matrix representation of the LFSRs.

As a second step of improvement, the calculation of the full cipher key can be skipped: As described before, the *dependent* part of the cipher key is a combinatorial function of the *independent* cipher key bits. Therefore, the whole initial state can be expressed as a function of the independent cipher key bits.

Hard-Coding Where the plain attack from Section 7.3 proposes *one* equation system $Ak = b$, the key ranking allows us to reuse the matrix A and just invert one or more equations, i.e. modify b, if no key has been found for a particular sub-space.

Therefor, only the b vector needs to be loaded into the FPGA at run time, while the A matrix can be hard-coded into the design by a VHDL preprocessor. This saves hardware resources on the FPGA, reduces the complexity and eliminates potentially critical paths. potentially critical path is eliminated, as it is no longer necessary to combine *all independent* counter bits. The reference keystream can be hard-coded as well.

Early Abort A cipher key can be considered invalid as soon as one bit comparison to the reference keystream fails. Therefore, the next key can be loaded upon failure of a comparison. For a wrong key, the bit comparison already fails at the second keystream bit on average, such that $n - 2$ cycles are saved on average when the reference keystream has a length of n. Early Abort introduces non-determinism, however, which increases the control overhead.

Pre-Ciphering Pipeline With the Early Abort optimization, several DSC units are competing to be loaded with a new initial state. As the arbitration logic complexity rises with the number of competing units, this number is to be kept low. A good way to do this is outsourcing the pre-ciphering phase into a strictly sequential, deterministic pipeline. With this optimization, the state *after pre-ciphering* is directly loaded into the competing DSC units.

Input Buffering Idle time of the FPGA has a negative impact on the effective performance. Therefore, an input FIFO is inserted such that the PC can queue multiple tasks and the FPGA can directly load the next task as soon as the previous one is finished.

Implementation

For our implementation, a Xilinx Spartan-3E 1200 (XC3S1200E) FPGA on a Digilent Nexys 2 board was used. The PC communication was implemented via the on-board RS-232 interface.

Our final implementation includes all optimizations as described in the last sections. The runtime of the design is not entirely deterministic, as – for a specific keystream – the position of the first failing comparison with the reference keystream is not known in advance. Therefore, the key generator was given the ability to be paused, which is necessary when all available DSC units are busy.

One pipelined key generator was chosen to serve four DSC units. The complete design consumed only about 30% of the FPGA resources in total, such that three instances could be created on our device.

Performance Evaluation

This section compares the performance achieved by our FPGA implementation with the CUDA performance published in [29].

We used five different, randomly generated equation systems for evaluating the maximum frequency by synthesizing the design for each of the equation systems. Table 5 shows a summary of our results.

Table 5.: Performance Evaluation

	Max Frequency	Performance $[\frac{keys}{s}]$	Cost $[US\$]$	Cost-Performance
FPGA	140 MHz	$408.8 \cdot 10^6$	169	$2.42 \cdot 10^6 \frac{keys}{US\$ \cdot s}$
[29] CUDA / GTX 260	unknown	$148 \cdot 10^6$	190	$0.78 \cdot 10^6 \frac{keys}{US\$ \cdot s}$

8 Attacks on the Radio Protocol

So far, we have seen that several vulnerabilities in DECT implementations have been found, and the cryptographic algorithms DSAA and DSC have several severe problems. This Chapter shows that the DECT radio protocol can be attacked too, even if the standard has been accurately implemented and DSAA and DSC would be replaced with secure algorithms having the same interface. Even with these assumptions, an attacker can still decrypt the audio signal send from a phone to the base station during an encrypted call. The key idea behind this attack is to launch a replay attack against the phone to recover all keystream, which were used to encrypt the audio signal. Readers who are unfamiliar with DECT should read Chapter 2 first. Chapters 3, 4, 5, 6, and 7 are not required to understand this attack.

My contribution to this Chapter is the design as well as the implementation of this attack, based on a DECT kernel stack written by Patrick McHardy. The kernel stack as well as the userland utilities are available at `http://dect.osmocom.org/`. Parts of this Chapter are joined work with Andreas Schuler and Patrick McHardy, and have previously been published at WISEC 2011[25].

8.1 Outline of the Attack

The attack is executed in two phases: In the first phase, an encrypted call is passively recorded as described in Section 5.1. If the call would be unencrypted, the attack would be finished after this step. For encrypted calls, a second phase is executed, which decrypts the call.

The key idea in the second phase of the attack is, to use a replay attack against the phone to recover all keystreams, which were used to encrypt the original call. This step can be executed with some delay, so that an attacker can record some call first, and decrypt them later. However, he needs to be again in communication range of the phone. We first set up our base station impersonating the original base station of the victim. One can use the tool described in Section 5.5 to make the phone switch to our base station. Checking whether the phone has locked on our base station can be done by periodically broadcasting a `LCE-PAGE-REQUEST`[15] message from our base station. When the phone answers to the `LCE-PAGE-REQUEST`, it has locked on our base station.

8.2 Recovering the Keystreams

In the next step, we need to set the correct cipher key on the phone, which was used in the call we would like to decrypt. If no further call has been made and the phone was not switched off in the meantime, we could skip this step. Executing the step will only cost less than a second of additional attack time. Of course we do not know the key, otherwise, we could easily decrypt the call from our capture ourselves. However, the key was derived from the UAK and two random numbers RAND_F and RS using the A11 and A12 algorithms at the begin of the recorded call. We simply send again the `AUTHENTICATION-REQUEST` message, which was exchanged at the beginning of the recorded call. The phone will respond with an

AUTHENTICATION-REPLY message containing the response of the challenge we send. This procedure is described in Section 2.4.1.

If the response, which is included, is not equal to the response in our recording, we might have picked the wrong phone, recorded incorrect data or the UAK on the phone has been altered. Therefore, the following steps would be unlikely to succeed:

We now repeat the following procedure: Let n be the multiframe number of the first packet in our capture that we have not decrypted so far. We wait just before our base station broadcasts the next update of our multiframe number using a Q-channel message and set the multiframe number of our base station to $n - (t_d/16)$. Here, t_d is the time it takes from sending a CIPHER-REQUEST message to the phone until ciphering is enabled in the MAC layer and the first encrypted packet from the phone is sent in 10^{-2} seconds.

Then we broadcast our multiframe number update and send a CIPHER-REQUEST[15] message to the phone. The phone will receive our multiframe number update and update its multiframe number accordingly. After having received our CIPHER-REQUEST message, the phone will respond with an encryption control MAC layer message stating that it is ready to enable encryption. We now confirm that using a MAC layer message and the phone will signal that it will enable encryption now. The next packet sent by the phone will be encrypted using the same key as the original call and the same initialization vectors (frame numbers) will be used. From now on, we do not send any payload to the phone and wait until the LCE.01 timer on the phone has expired and the link is released. Because we have not established a phone call on the link or did run any other application on it, the B-fields of all frames sent by the phone just contained ff as plaintext. XORing all B-fields of the received frames with ff reveals the keystreams used to encrypt the frames in the original call. XORing these keystreams with the B-fields in the original call decrypts these B-fields, revealing the audio data sent from the phone to the base station in the original call. We now repeat that procedure until all original call frames have been decrypted. A good overview how keystreams are used to encrypt DECT packets is given in Figure 6.

After the call has been decrypted, we shut down our base station and the phone starts scanning for the original base station and locks on it again.

Choosing t_d too small when implementing the attack results in some keystreams at the beginning of the call not being recovered. Choosing t_d too large results only in a small performance decrease of the attack, so one might choose a slightly larger value for t_d in this step.

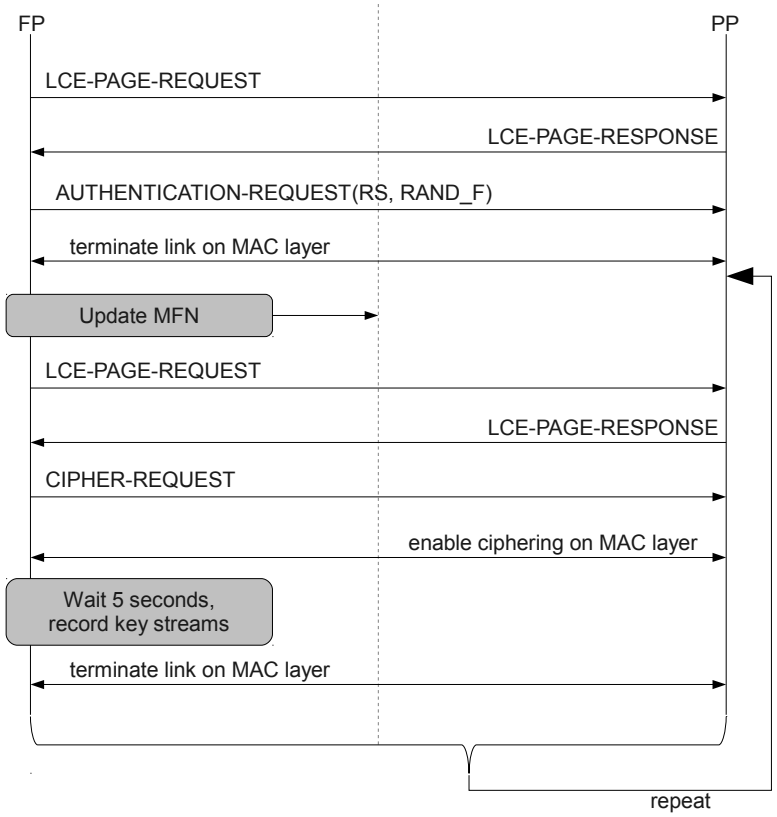
8.3 Implementation

We implemented a proof-of-concept of our attack. As basis, we used a DECT stack for the Linux kernel, which is available on http://dect.osmocom.org/. We needed to tweak three parts of the stack:

kernel We made a small change to the kernel code, that makes it possible to update the multiframe number in the kernel from the userland.

libnl We added another function, that passes a new multiframe number from the userland to the kernel.

Figure 16.: Attack overview



libdect This library implements all higher level functionality of the kernel stack. We added a small change to libdect so that a program using libdect can pass a new multi frame number to the kernel, using libnl. We also wrote a new program that uses libdect to implement the attack tool. Because libdect already implemented much of the functionality we needed, we were able to implement the tool in 222 lines of C-code.

To enable encryption on the link, we used a cipher key CK with all bytes set to zero (`00 00 00 00 00 00 00 00 00 00`), so that the kernel decrypted all the encrypted traffic with this key. To capture the encrypted frames, we used libpcap, which is also part of the DECT stack, which we did not need to modify. To reveal the keystream, we decrypted the received B-field with the cipher key `00 00 00 00 00 00 00 00 00 00` in software again, which revealed the original B-field received and then XORed the B-field with `ff` to reveal the keystreams.

We used a Siemens Gigaset AS150 phone to test our code. During implementation and execution, we observed the following effects:

- The phone accepted changes of the multiframe number without any problems, when the phone was in idle mode. However, it sometimes took the phone a few seconds until it was responsive again, after having changed the multiframe number.
- Changing the multiframe number after having established a connection, but before sending the `CIPHER-REQUEST` message caused the phone to drop the connection.

- Sometimes, the phone lost the connection with our base station and became unresponsive. Stopping our base station and starting it again a few seconds later solved the problem. Restarting the phone or any other kind of interaction with the phone was never required, so this would only cause a delay in the execution of the attack, but not render it impossible. We are not entirely sure yet, whether this is a problem related to the phone DECT stack or on our side.
- Sometimes, we received a few bits in the B-field incorrectly, resulting in some bits of the keystream recovered incorrectly. However, this only creates minor variations in the recovered audio stream, which usually will not be noticed.

  If this method would be used to recover a data-call (for example wireless internet access over DECT), this might be a problem. One could execute the attack multiple times to spot and correct variations in the keystreams recovered.

We also tested the attack successfully against a *Siemens Gigaset 4000 Classic* and a *T-Sinus 501* phone. Another test against an *AVM FritzFon* failed, due to general problems with our DECT stack, not related to the attack.

8.4 Experimental Results

We were able to establish a connection with the phone about once every 14 seconds. A single connection lasted for 5 seconds, until the phones LCE.01 timer expired. (In fact our fake base station implements a 5 seconds LCE.01 timer, so our base station closes the connection just before the phone would have closed it. Disabling this timer on our base station makes the phone close the connection a few moments later, because it also implements the timer.) As a result, we were able to recover about 500 keystreams in 14 seconds. The exact attack speed depends on the timing parameters chosen for the implementation and on the attacked phone.

As we did not send any call related messages on the C-channel, the phone did never ring or play any other sounds. Also, the content of the phone display did not change during the attack. Only when we had to restart our base station, the phone display indicated that it has lost the link to its base station and was now scanning for the base station. This makes it very unlikely that the attack is detected by a user by looking at the phone. Of course a user with a DECT sniffer could detect the presence of an attacker.

9 Improvements and Countermeasures

Chapter 5 has shown that many DECT devices can be attacked due to mistakes made by the manufacturers. Chapters 6, 7, and 8 have shown that the DECT standard itself has many weaknesses. To counter these problems, DECT device manufacturers have agreed to improve the security of DECT in 3 major steps. *Step A* is a minor modification of the existing DECT standard and DECT devices implementing these modifications are immune to many attacks described in this thesis. However not all weaknesses are fixed in *Step A*. To improve the security of DECT further, two additional steps *Step B* and *Step C* were taken, which add two new security algorithms DSAA2 and DSC2 to DECT, replacing DSAA and DSC.

To show what can be expected in the next generation of DECT devices, this Chapter summarizes the changes in *Step A* in Section 9.1 and gives a brief overview what is included in *Step B* and *Step C* in Sections 9.2 and 9.3. Because random number generation is important for DECT devices, this topic is discussed separately in 9.4. Section 9.5 shows that there is still a remaining problem in DECT when it comes to integrity protection, that is not easy to solve. This Chapter contains no new results, instead it just described which improvements have been integrated in the DECT standard.

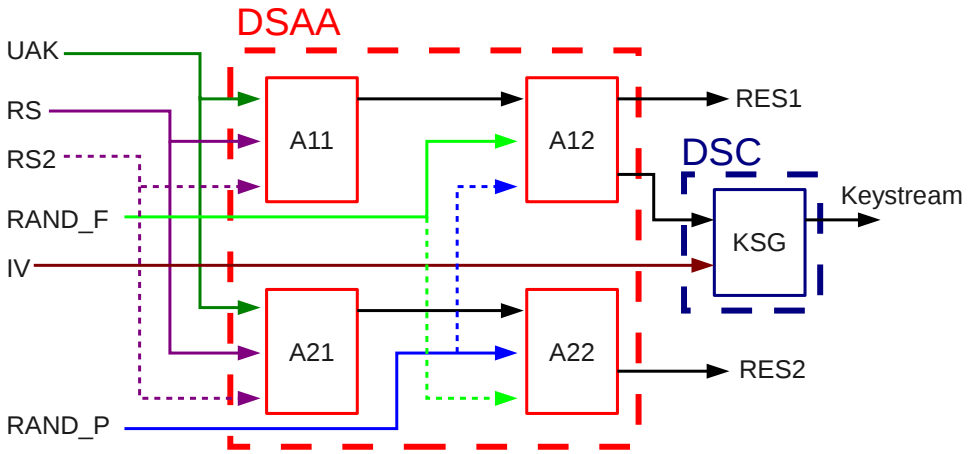
9.1 Step A

In the first step, the existing DECT standard is slightly modified to counter most of the attacks. In detail, the following changes have been integrated:

- A DECT base station is only allowed to go into pairing mode for 120 seconds at most. If no handset has been paired in this time span, the base station switches back to normal mode. Also, the pairing mode needs to be set intentionally by a user. Going into pairing mode automatically at startup is not allowed.
- Phones and base stations need to support encryption, and a base station has to activate encryption automatically. This prevents the attack described in Section 5.1.
- Phones and base stations must support the on-the-air key allocation feature defined in DECT GAP [19] and described in Section 2.4.3.
- If a base station does not enable encryption in time, the phone must automatically drop the connection. This prevents the attack described in Section 5.4.
- Encryption must be enabled early during call setup. This prevents the attack described in Section 5.2.
- In call re-keying is now supported with a default interval of 60 seconds. This prevents the key recovery attack on DSC described in Chapter 7.

All of these changes have been standardized in a new release of the DECT GAP standard [19], and can be integrated into existing DECT products.

Figure 17.: Security algorithms in DECT with DSAA2



However, the attack described in Chapter 8 is still possible and the overall security level of DECT is still limited to 64 bit. To improve the security of DECT further, two additional steps have been taken:

9.2 Step B

In the next step, a new authentication algorithm named DSAA2 has been standardized[21], shown in Algorithms 13 and 14, that is based on the AES. Because DSAA is usually implemented in firmware and not in hardware, DSAA2 can be easily integrated in existing DECT devices, without having to design new baseband chipsets. To counter the attack from Section 8, random numbers from both sides are used during authentication and key derivation. Figure 17 gives an overview of the new security design:

DSAA2 can be summarized as follows:

Algorithm 13 $\mathrm{DSAA2}-1(\mathbf{d1} \in \{0,1\}^{128}, \mathbf{d2} \in \{0,1\}^{64}, \mathbf{d3} \in \{0,1\}^{64}$

1: $\mathbf{r1} \leftarrow \mathrm{aes128}(\mathbf{d1}, \mathbf{d2}||\mathbf{d3})$
2: $\mathbf{r2} \leftarrow \mathrm{aes128}(\mathbf{d1}, \mathbf{r1} \oplus (00)^{16})$
3: **return** $\mathbf{r2}$

Algorithm 14 $\mathrm{DSAA2}-2(\mathbf{d1} \in \{0,1\}^{128}, \mathbf{d2} \in \{0,1\}^{64}, \mathbf{d3} \in \{0,1\}^{64}$

1: $\mathbf{r1} \leftarrow \mathrm{aes128}(\mathbf{d1}, \mathbf{d2}||\mathbf{d3})$
2: $\mathbf{r2} \leftarrow \mathrm{aes128}(\mathbf{d1}, \mathbf{r1} \oplus ((00)^{15}||01))$
3: **return** $\mathbf{r1}[0\ldots31], \mathbf{r2}$

The algorithm DSAA2-1 replaces the algorithms A11 and A21, and the algorithm DSAA2-2 replaces the algorithms A12 and A22. The generated cipher key is always of 128 bit length. If only 64 bit are required, the output is simply cropped. Please note that the FP and PP algorithms (A11 and A21) are now symmetric, which might help to design reflection attacks against the protocol. The authentication

procedures described in Sections 2.4.1 and 2.4.2 need to be updated as well as shown in Figures 18 and 19.

Figure 18.: Authentication of a DECT PP using DSAA2

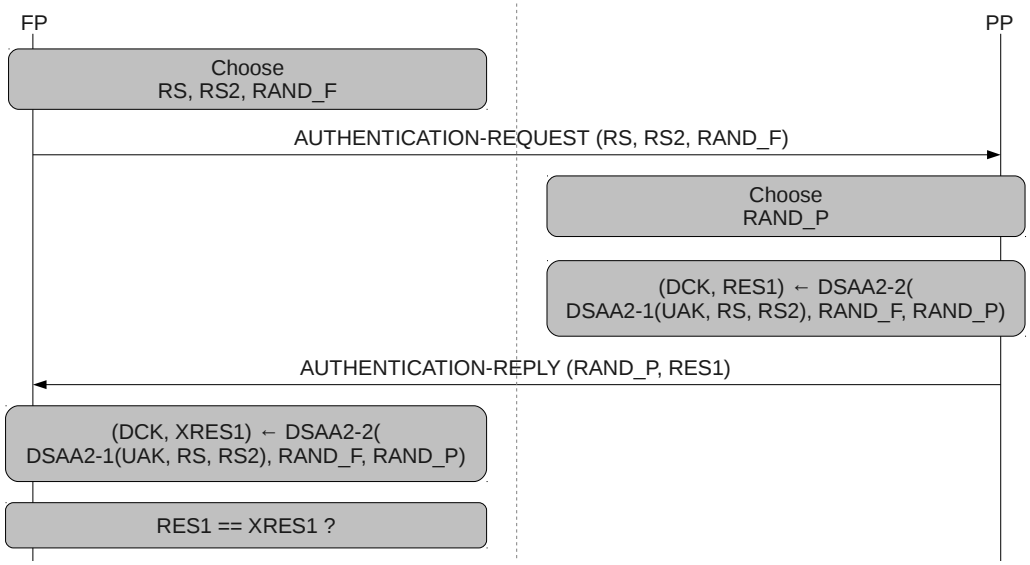
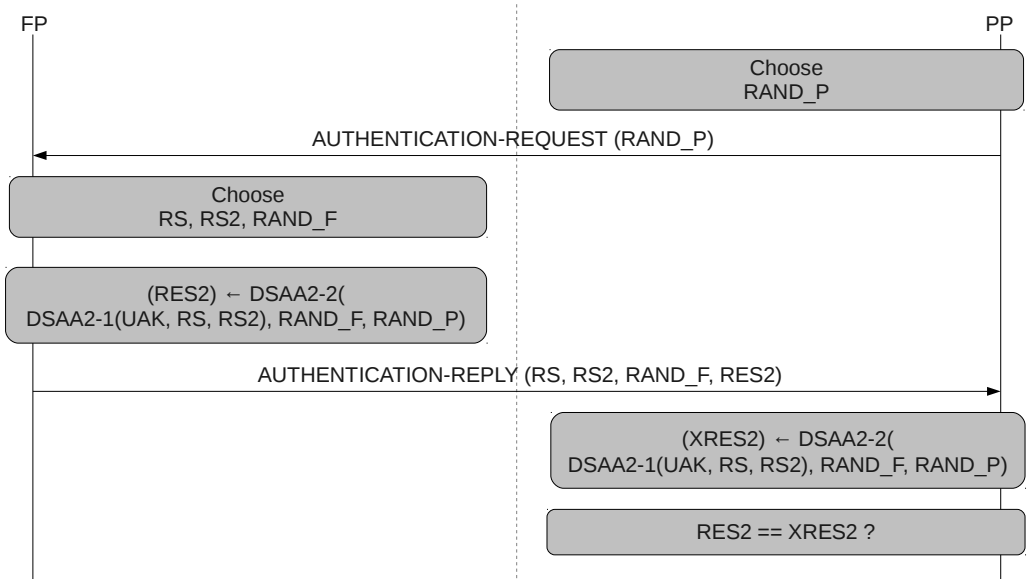


Figure 19.: Authentication of a DECT FP using DSAA2



9.3 Step C

Finally, DSC is replaced with a new stream cipher DSC2[21], shown in Algorithm 15, that is based on AES. The function zero_extend extends the counter **i** to 64 bit length, and () initializes KSS with a zero length string. So DSC2 is just AES128 in counter-mode, using the IV as the start counter. Notably, DSC2 does not provide any features to protect the integrity of the encrypted data and does not expand the plaintext.

Algorithm 15 $\mathrm{DSC2}(\mathsf{CK} \in \{0,1\}^{128}, \mathsf{IV} \in \{0,1\}^{64}, \mathbf{l} \in \mathbb{N}$

1: $\mathsf{KSS} \leftarrow ()$
2: **for** $\mathbf{i} \leftarrow 0 \ldots l-1$ **do**
3: $\quad \mathsf{KSS} \leftarrow \mathsf{KSS} || \mathrm{aes128}\left(\mathsf{CK}, \mathsf{IV} || \mathrm{zero\_extend}(\mathbf{i})\right)$
4: **end for**
5: **return** KSS

9.4 Random Number Generators on DECT Phones

In Section 5.3, I have shown that some DECT devices implement very weak Pseudo Random Number Generators (PRNGs). This issue is known to the device manufacturers, but is hard to fix in a new release of the DECT standard. All additional requirements described in Step A, B, and C are easy to test. Using a semi-automatic setup, one can check if a phone really supports a new authentication mechanism, really enforces encryption of all calls and only goes to pairing mode for 120 seconds, using just black box tests.

In contrast to that, bad random number generators are hard to find. While it might still be possible to detect a specific bad random number generator, a generic test on random number generators that will rapidly and efficiently distinguish bad generators from good ones seems to be impossible.

Specifying a pseudo random number generator, all DECT devices must implement, would not be a perfect solution for the problem for at least two reasons:

- Some DECT devices use very restrictive hardware like an 8 bit micro controller, other DECT devices use a 32 bit ARM embedded CPU. Depending on what hardware is available, one should be able to implement a PRNG that is suitable for this hardware.
- A good PRNG is only a part of a solution. Besides the PRNG itself, a good entropy source is required too. If the PRNG specified is slow on the target hardware, a device manufacture could choose not to use many bits of real entropy, to speed up the seeding process.

I think a DECT device manufacture should be allowed to use a PRNG algorithm, that fits best on his hardware. However, as much real entropy as possible should be used to seed and reseed the algorithm. Fortunately, there are various sources of entropy on a DECT device:

- A good source for entropy is the radio itself, that is available in every DECT device. One could simply start sampling data in an time slot that is not occupied. A DECT base station does this

anyway to check which time slots are not occupied by other base stations. Alternatively, CRC-Errors or similar effects can be used too.

- On DECT handsets, there is a microphone with a analog-digital converter. When the microphone is not used (no call is active), the handset can start sampling audio from the microphone and use it as a source for entropy.
- DECT base stations are usually connected to an analog landline. Sampling signals from the landline, when no calls are active can be used as entropy too.
- DECT devices that operate on battery power are sometimes equipped with a sensor that checks the remaining battery power. This sensor can be used as a source of entropy too.
- On all devices, that have buttons, the timing of the button presses and releases can be used as a source of entropy.
- If a device implements a light sensor to automatically adjust the intensity of the backlight, this sensor can be a source for entropy too.

These sources have been identified by the authors of [23] and [29], but have not been separately published.

9.5 Remaining Problems

After all these changes have been made, a single problem remains. At the moment, DECT provides no cryptographic protection of the payload, and uses only CRC checksums to detect transmission problems. Fixing this problem is hard for two reasons:

- Using any kind of cryptographic protection like a MAC requires additional space in every packet. However, due to the TDMA structure described in Section 2.2, adding additional bits to a packet would increase the time required to send this packet and could influence the packet send in the next time slot. This would require a big change to the DECT radio protocol, or would double the bandwidth usage of a DECT call, if two consecutive time slots would be used for a single packet.
- Even if this problem would be solved and changes to a packet could be detected, it would be hard to handle this in the upper layers. DECT in contrast to WLAN is a telephone system, where a continuous stream of data is expected and a frame that has been altered cannot just be dropped as in a WLAN.

However, no such attack has yet been published.

10 Conclusion and Thanks

In this thesis, I have shown that DECT has many security problems. In Chapters 3 and 4, I have described the two formerly secret security algorithms DSAA and DSC. In Chapter 5, I have shown that many implementations have severe problems, that make them easy to attack or no security features of DECT are implemented at all. In Chapter 6, I have shown that the authentication mechanism does not provide 128 bit security for the long term keys and the building blocks are flawed. In Chapter 7, I have shown that the encryption algorithm used in DECT is flawed and a key recover against the cipher can be performed on a standard PC. In Chapter 8, I have shown that even without these problems, the DECT protocol itself is insecure, and calls can be decrypted without exploiting special properties of DSAA or DSC.

To counter these problems, a new release of the DECT standard is now standardized. I have outlined what kind of improvements in DECT can be expected in the next generations of DECT devices in Chapter 9. This shows how important the research outlined in this thesis has been, and that DECT will still be a major industry standard in the next decade.

10.1 Thanks and Acknowledgments

I would like to thank my thesis adviser Prof. Johannes Buchmann, who allowed me to write this thesis and use parts of my working time and infrastructure of the university for it. I would also like to thank all my co-authors of my publications about DECT, namely Patrick McHardy, Stefan Lucks, Hans Gregor Molter, Karsten Nohl, Kei Ogata, Andreas Schuler, Michael Weiner, Ralf-Philipp Weinmann, and Matthias Wenzel. Their help made these results possible. In addition to that, Alexandra Mengele and Kei Ogata supported this research with their diploma thesis about DECT. I also had a lot of additional supporters as Jan Krissler and many more, whose names are not mentioned here. I would also like to thank the DECT Forum members and ETSI, who are now in progress of releasing a new version of the DECT standard to counter the problems we discovered. I would also like to thank the reviewers of my publications, wo gave me valuable hints how to improve my papers.

To finalize this thesis, this document has been proof-read by Stefanie Bartsch, Johannes Buchmann, Nadja Kokic, Natalia Kutepow, Stefan Lucks, Bettina Roth, and Phuong Thao Do. All of them helped me to improve this thesis.

During my four years of research, I was supported by my parents and a lot of other friends. They allowed me to a lot of time into this thesis to improve it's structure and content.

A Acronyms

Name	Description	Section
A11	Algorithm in DSAA	2.4
A12	Algorithm in DSAA	2.4
A21	Algorithm in DSAA	2.4
A22	Algorithm in DSAA	2.4
AUTHENTICATION-REQUEST	Message to request authentication	2.4
AUTHENTICATION-RESPONSE	Response to an authentication request	2.4
C-Channel	A channel for control traffic	2.2
CIPHER-REQUEST	Message to request start of ciphering	2.5
CK	Cipher Key	2.4
DCK	Derived Cipher Key (see CK)	2.4
DSAA	DECT Standard Authentication Algorithm	2.1, 2.4
DSC	DECT Standard Cipher	2.1, 2.5
FP	DECT base station	2.1
frame	Time period of 10 ms	2.2
IV	Initialization Vector	2.5
LCE.01	Timer which starts when a connection is not required anymore	2.2
LCE-PAGE-REQUEST	Message to request a phone to start a connection	2.2
multiframe number	Most significant bits of a frame number	2.2
P00	Packet format with no B-field	2.2
P80j	Packet format with 80 bits B-field	2.2
P32	Standard packet format with 320 bits B-field	2.2
packet	A single data burst	2.2
PP	DECT phone	2.1
RAND_F	Random number chosen by a base station	2.4
RAND_P	Random number chosen by a phone	2.4
RES1	Response to a challenge during authentication of a phone	2.4
RES2	Response to a challenge during authentication of a base station	2.4
RS	Random number chosen by a DECT network	2.4
UAK	User Authentication Key (shared by phone and base station)	2.4

B Reference Implementations

B.1 DSAA

To verify the pseudocode in our paper, DSAA has been reimplemented in C by Matthias Wenzel, and is now part of libdect written by Patrick McHardy:

```
1  /*
2   * DECT Standard Authentication Algorithm
3   *
4   * This program is free software; you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License version 2 as
6   * published by the Free Software Foundation.
7   *
8   * Copyright (c) 2009 Matthias Wenzel <dect at mazzoo dot de>
9   *
10  */
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <stdarg.h>
15 #include <stdint.h>
16
17 #include <dect/auth.h>
18 #include <libdect.h>
19 #include <utils.h>
20
21 static const uint8_t sbox[256] = {
22         0xb0, 0x68, 0x6f, 0xf6, 0x7d, 0xe8, 0x16, 0x85,
23         0x39, 0x7c, 0x7f, 0xde, 0x43, 0xf0, 0x59, 0xa9,
24         0xfb, 0x80, 0x32, 0xae, 0x5f, 0x25, 0x8c, 0xf5,
25         0x94, 0x6b, 0xd8, 0xea, 0x88, 0x98, 0xc2, 0x29,
26         0xcf, 0x3a, 0x50, 0x96, 0x1c, 0x08, 0x95, 0xf4,
27         0x82, 0x37, 0x0a, 0x56, 0x2c, 0xff, 0x4f, 0xc4,
28         0x60, 0xa5, 0x83, 0x21, 0x30, 0xf8, 0xf3, 0x28,
29         0xfa, 0x93, 0x49, 0x34, 0x42, 0x78, 0xbf, 0xfc,
30         0x61, 0xc6, 0xf1, 0xa7, 0x1a, 0x53, 0x03, 0x4d,
31         0x86, 0xd3, 0x04, 0x87, 0x7e, 0x8f, 0xa0, 0xb7,
32         0x31, 0xb3, 0xe7, 0x0e, 0x2f, 0xcc, 0x69, 0xc3,
33         0xc0, 0xd9, 0xc8, 0x13, 0xdc, 0x8b, 0x01, 0x52,
34         0xc1, 0x48, 0xef, 0xaf, 0x73, 0xdd, 0x5c, 0x2e,
35         0x19, 0x91, 0xdf, 0x22, 0xd5, 0x3d, 0x0d, 0xa3,
36         0x58, 0x81, 0x3e, 0xfd, 0x62, 0x44, 0x24, 0x2d,
37         0xb6, 0x8d, 0x5a, 0x05, 0x17, 0xbe, 0x27, 0x54,
38         0x5d, 0x9d, 0xd6, 0xad, 0x6c, 0xed, 0x64, 0xce,
39         0xf2, 0x72, 0x3f, 0xd4, 0x46, 0xa4, 0x10, 0xa2,
40         0x3b, 0x89, 0x97, 0x4c, 0x6e, 0x74, 0x99, 0xe4,
41         0xe3, 0xbb, 0xee, 0x70, 0x00, 0xbd, 0x65, 0x20,
42         0x0f, 0x7a, 0xe9, 0x9e, 0x9b, 0xc7, 0xb5, 0x63,
43         0xe6, 0xaa, 0xe1, 0x8a, 0xc5, 0x07, 0x06, 0x1e,
44         0x5e, 0x1d, 0x35, 0x38, 0x77, 0x14, 0x11, 0xe2,
45         0xb9, 0x84, 0x18, 0x9f, 0x2a, 0xcb, 0xda, 0xf7,
46         0xa6, 0xb2, 0x66, 0x7b, 0xb1, 0x9c, 0x6d, 0x6a,
47         0xf9, 0xfe, 0xca, 0xc9, 0xa8, 0x41, 0xbc, 0x79,
48         0xdb, 0xb8, 0x67, 0xba, 0xac, 0x36, 0xab, 0x92,
49         0x4b, 0xd7, 0xe5, 0x9a, 0x76, 0xcd, 0x15, 0x1f,
50         0x4e, 0x4a, 0x57, 0x71, 0x1b, 0x55, 0x09, 0x51,
51         0x33, 0x0c, 0xb4, 0x8e, 0x2b, 0xe0, 0xd0, 0x5b,
52         0x47, 0x75, 0x45, 0x40, 0x02, 0xd1, 0x3c, 0xec,
53         0x23, 0xeb, 0x0b, 0xd2, 0xa1, 0x90, 0x26, 0x12,
54 };
55
56 static void bitperm(uint8_t start, uint8_t step, uint8_t *key)
57 {
```

```
58          uint8_t copy[8];
59          unsigned int i;
60
61          memcpy(copy, key, 8);
62          memset(key, 0, 8);
63
64          for (i = 0; i < 64; i++) {
65                  key[start/8] |= ((copy[i / 8] & (1 << (i % 8))) >>
66                                   (i % 8)) << (start % 8);
67                  start += step;
68                  start %= 64;
69          }
70  }
71
72  static const uint8_t mix_factor[3][8] = {
73          { 2, 2, 2, 2, 3, 3, 3, 3},
74          { 2, 2, 3, 3, 2, 2, 3, 3},
75          { 2, 3, 2, 3, 2, 3, 2, 3},
76  };
77
78  static const uint8_t mix_index[3][8] = {
79          { 4, 5, 6, 7, 0, 1, 2, 3},
80          { 2, 3, 0, 1, 6, 7, 4, 5},
81          { 1, 0, 3, 2, 5, 4, 7, 6},
82  };
83
84  static void mix(uint8_t start, uint8_t alg, uint8_t *key)
85  {
86          uint8_t copy[8];
87          unsigned int i;
88
89          memcpy(copy, key, 8);
90          for (i = 0; i < 8; i++)
91                  key[i] = copy[mix_index[alg][i]] + mix_factor[alg][i] * copy[i];
92  }
93
94  static void mix1(uint8_t *key)
95  {
96          mix(4, 0, key);
97  }
98
99  static void mix2(uint8_t *key)
100 {
101         mix(2, 1, key);
102 }
103
104 static void mix3(uint8_t *key)
105 {
106         mix(1, 2, key);
107 }
108
109 static void sub(uint8_t *s, uint8_t *t)
110 {
111         unsigned int i;
112
113         for (i = 0; i < 8; i++)
114                 s[i] = sbox[s[i] ^ t[i]];
115 }
116
117 /* return in s */
118 static void cassable(uint8_t start, uint8_t step, uint8_t *t, uint8_t *s)
119 {
120         unsigned int i;
121
122         for (i = 0; i < 2; i++) {
123                 bitperm(start, step, t);
124                 sub(s, t);
125                 mix1(s);
126
127                 bitperm(start, step, t);
128                 sub(s, t);
129                 mix2(s);
```

```
130
131                 bitperm(start, step, t);
132                 sub(s, t);
133                 mix3(s);
134         }
135 }
136
137 /* return in rand, modifies key */
138 static void step1(uint8_t *rand, uint8_t *key)
139 {
140         uint8_t tmp[8];
141
142         memcpy(tmp, rand, 8);
143
144         cassable(46, 35, tmp, key);
145         cassable(25, 47, key, rand);
146
147         memcpy(key, rand, 8);
148 }
149
150 static void step2(uint8_t *rand, uint8_t *key)
151 {
152         uint8_t tmp[8];
153
154         memcpy(tmp, rand, 8);
155
156         cassable(60, 27, tmp, key);
157         cassable(55, 39, key, rand);
158
159         memcpy(key, rand, 8);
160 }
161
162 static void rev(uint8_t *v, uint8_t n)
163 {
164         unsigned int i;
165         uint8_t tmp;
166
167         for (i = 0; i < n / 2; i++) {
168                 tmp          = v[i];
169                 v[i]         = v[n - i - 1];
170                 v[n - i - 1] = tmp;
171         }
172 }
173
174 static void dsaa_main(const uint8_t *k, const uint8_t *r, uint8_t *e)
175 {
176         uint8_t key[16];
177         uint8_t rand[8];
178         uint8_t a[8];
179         uint8_t b[8];
180
181         memcpy(key, k, sizeof(key));
182         rev(key, 16);
183
184         memcpy(&rand, r, sizeof(rand));
185         rev(rand, 8);
186
187         step1(rand, key + 4);
188         memcpy(a, key + 4, 8);
189
190         memcpy(key + 4, key + 12, 4);
191         memcpy(b, a, 8);
192         step2(b, key);
193
194         rev(a, 8);
195         rev(key, 4);
196         rev(key + 4, 4);
197
198         memcpy(e, key + 4, 4);
199         memcpy(e + 4, a, 8);
200         memcpy(e + 12, key, 4);
201 }
```

```
202
203 const struct dect_aalg dect_dsaa_aalg = {
204         .type           = DECT_AUTH_DSAA,
205         .d1_len         = 16,
206         .d2_len         = 8,
207         .calc           = dsaa_main,
208 };
```

B.2 DSC

As an additional help, a reference implementation of DSC written in C is presented here.

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  #define R1_LEN                  17
8  #define R2_LEN                  19
9  #define R3_LEN                  21
10 #define R4_LEN                  23
11
12 #define R1_MASK                 0x010020 /* tap bits: 5, 16 */
13 #define R2_MASK                 0x04100C /* tap bits: 2, 3, 12, 18 */
14 #define R3_MASK                 0x100002 /* tap bits: 1, 20 */
15 #define R4_MASK                 0x400100 /* tap bits: 8, 22 */
16
17 #define R1_CLKBIT               8
18 #define R2_CLKBIT               9
19 #define R3_CLKBIT               10
20 #define R1_R4_CLKBIT            0
21 #define R2_R4_CLKBIT            1
22 #define R3_R4_CLKBIT            2
23
24 #define OUTPUT_LEN              90 /* 720 bits */
25 #define TESTBIT(R, n)           (((R) & (1 << (n))) != 0)
26
27 uint32_t clock(uint32_t lfsr, uint32_t mask) {
28   return (lfsr >> 1) ^ (-(lfsr&1)&mask);
29 }
30
31 uint32_t combine(uint32_t c, uint32_t r1, uint32_t r2, uint32_t r3) {
32   uint32_t x10, x11, x20, x21, x30, x31;
33
34   x10 = r1 & 1;
35   x11 = (r1 >> 1) & 1;
36   x20 = r2 & 1;
37   x21 = (r2 >> 1) & 1;
38   x30 = r3 & 1;
39   x31 = (r3 >> 1) & 1;
40
41   return ((x11&x10&c) ^ (x20&x11&x10) ^ (x21&x10&c) ^ (x21&x20&x10) ^
42           (x30&x10&c) ^ (x30&x20&x10) ^ (x11&c) ^ (x11&x10) ^ (x20&x11) ^
43           (x30&c) ^ (x31&c) ^ (x31&x10) ^ (x21) ^ (x31));
44 }
45
46 void dsc_keystream(uint8_t *key, uint32_t iv, uint8_t *output) {
47
48   uint8_t input[16];
49   uint32_t R1, R2, R3, R4, N1, N2, N3, COMB;
50   int i, keybit;
51
52   memset(output, 0, OUTPUT_LEN);
53
54   input[0] = iv&0xff;
55   input[1] = (iv>>8)&0xff;
56   input[2] = (iv>>16)&0xff;
57   for (i = 3; i < 8; i++) {
58         input[i] = 0;
59   }
60   for (i = 0; i < 8; i++) {
61         input[i+8] = key[i];
62   }
63
64   R1 = R2 = R3 = R4 = COMB = 0;
65
66   /* load IV || KEY */
```

```
67    for (i = 0; i < 128; i++) {
68      keybit = (input[i/8] >> ((i)&7)) & 1;
69      R1 = clock(R1, R1_MASK) ^ (keybit<<(R1_LEN-1));
70      R2 = clock(R2, R2_MASK) ^ (keybit<<(R2_LEN-1));
71      R3 = clock(R3, R3_MASK) ^ (keybit<<(R3_LEN-1));
72      R4 = clock(R4, R4_MASK) ^ (keybit<<(R4_LEN-1));
73    }
74
75    for (i = 0; i < 40 + (OUTPUT_LEN*8); i++) {
76      /* check whether any registers are zero after 11 pre-ciphering steps.
77       * if a register is all-zero after 11 steps, set input bit to one
78       * (see U.S. patent 5608802)
79       */
80      if (i == 11) {
81        if (!R1) R1 ^= (1<<(R1_LEN-1));
82        if (!R2) R2 ^= (1<<(R2_LEN-1));
83        if (!R3) R3 ^= (1<<(R3_LEN-1));
84        if (!R4) R4 ^= (1<<(R4_LEN-1));
85      }
86
87      N1 = R1;
88      N2 = R2;
89      N3 = R3;
90      COMB = combine(COMB, R1, R2, R3);
91
92      if (TESTBIT(R2, R2_CLKBIT) ^ TESTBIT(R3, R3_CLKBIT) ^
93          TESTBIT(R4, R1_R4_CLKBIT))
94        N1 = clock(R1, R1_MASK);
95      if (TESTBIT(R1, R1_CLKBIT) ^ TESTBIT(R3, R3_CLKBIT) ^
96          TESTBIT(R4, R2_R4_CLKBIT))
97        N2 = clock(R2, R2_MASK);
98      if (TESTBIT(R1, R1_CLKBIT) ^ TESTBIT(R2, R2_CLKBIT) ^
99          TESTBIT(R4, R3_R4_CLKBIT))
100       N3 = clock(R3, R3_MASK);
101
102     R1 = clock(clock(N1, R1_MASK), R1_MASK);
103     R2 = clock(clock(N2, R2_MASK), R2_MASK);
104     R3 = clock(clock(N3, R3_MASK), R3_MASK);
105
106     R4 = clock(clock(clock(R4, R4_MASK), R4_MASK), R4_MASK);
107
108     if(i >= 40) {
109       output[(i-40)/8] |= ((COMB) << ( 7-((i-40)&7)));
110     }
111   }
112 }
113
114 int main(int argc, char**argv) {
115   uint8_t key[8];
116   uint8_t output[OUTPUT_LEN];
117
118   if (argc != 2) {
119     fprintf(stderr, "usage: %s iv\n", argv[0]);
120     exit(1);
121   }
122
123   if (read(STDIN_FILENO, key, 8) < 8) {
124     fprintf(stderr, "short read\n");
125     exit(1);
126   }
127
128   dsc_keystream(key, atoi(argv[1]), output);
129
130   write(STDOUT_FILENO, output, 2048);
131   return 0;
132 }
```

Bibliography

[1] 3rd Generation Partnership Project. 3GPP specification, 2011. http://www.3gpp.org/specifications.

[2] Alcatel. Data ciphering device. U.S. Patent 5,608,802, 1994.

[3] E. Barkan and E. Biham. Conditional estimators: An effective attack on A5/1. In B. Preneel and S. E. Tavares, editors, *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2006.

[4] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In A. Menezes and S. A. Vanstone, editors, *CRYPTO 1990*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1991.

[5] Bundesamt für Sicherheit in der Informationstechnik. Drahtlose Kommunikationssysteme und ihre Sicherheitsaspekte, 2006.

[6] Bundesamt für Sicherheit in der Informationstechnik. Drahtlose Kommunikationssysteme und ihre Sicherheitsaspekte, 2009.

[7] DIRECTION CENTRALE DE LA SÉCURITÉ DES SYSTÈMES D'INFORMATION. Catalogue 2004, 2004. http://web.archive.org/web/20050423205939/http://www.formation.ssi.gouv.fr/formation/catalogue2004.pdf.

[8] P. Ekdahl and T. Johansson. Another attack on A5/1. *IEEE Transactions on Information Theory*, 49(1):284–289, 2003.

[9] M. Ettus. *USRP user's and developer's guide*. Ettus Research LLC, Feb. 2005.

[10] European Telecommunications Standards Institute. ETSI EN 300 444 V2.1.1: Digital Enhanced Cordless Telecommunications (DECT); Generic Access Profile (GAP), Oct 2008.

[11] European Telecommunications Standards Institute. Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 1: Overview, June 2010.

[12] European Telecommunications Standards Institute. ETSI EN 300 175-2 V2.3.1 (2010-06) - Common Interface (CI) Part 2: Physical Layer (PHL), June 2010.

[13] European Telecommunications Standards Institute. ETSI EN 300 175-3 V2.3.1 (2010-06) - Common Interface (CI) Part 3: Medium Access Control (MAC) layer, June 2010.

[14] European Telecommunications Standards Institute. ETSI EN 300 175-4 V2.3.1 (2010-06) - Common Interface (CI) Part 4: Data Link Control (DLC) layer, June 2010.

[15] European Telecommunications Standards Institute. ETSI EN 300 175-5 V2.3.1 (2010-06) - Common Interface (CI) Part 5: Network (NWK) layer, June 2010.

[16] European Telecommunications Standards Institute. ETSI EN 300 175-6 V2.3.1 (2010-06) - Common Interface (CI) Part 6: Identities and addressing, June 2010.

[17] European Telecommunications Standards Institute. ETSI EN 300 175-7 V2.3.1 (2010-06) - Common Interface (CI) Part 7: Security features, June 2010.

[18] European Telecommunications Standards Institute. ETSI EN 300 175-8 V2.3.1 (2010-06) - Common Interface (CI) Part 8: Speech and audio coding and transmission, June 2010.

[19] European Telecommunications Standards Institute. ETSI EN 300 444 V2.2.1: Digital Enhanced Cordless Telecommunications (DECT); Generic Access Profile (GAP), June 2010.

[20] European Telecommunications Standards Institute. ETSI TETRA (Terrestrial Trunked Radio) technology page, 2011. `http://www.etsi.org/website/Technologies/TETRA.aspx`.

[21] European Telecommunications Standards Institute. ETSI EN 300 175-7 V2.4.1 (2012-04) - Common Interface (CI) Part 7: Security features, April 2012.

[22] S. Golomb, L. Welch, R. Goldstein, and A. Hales. *Shift register sequences*, volume 78. Aegean Park Press Walnut Creek, CA, 1982.

[23] S. Lucks, A. Schuler, E. Tews, R. Weinmann, and M. Wenzel. Attacks on the DECT authentication mechanisms. *Topics in Cryptology–CT-RSA 2009*, pages 48–65, 2009.

[24] A. Maximov, T. Johansson, and S. Babbage. An improved correlation attack on A5/1. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography – SAC 2004*, volume 3357 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2004.

[25] P. McHardy, A. Schuler, and E. Tews. Interactive decryption of DECT phone calls. ACM, 2011.

[26] A. Mengele. Digital Enhanced Cordless Telecommunication (DECT) devices for residential use. Diploma thesis, Technische Universität Darmstadt, 2009.

[27] H. Molter, K. Ogata, E. Tews, and R. Weinmann. An Efficient FPGA Implementation for an DECT Brute-Force Attacking Scenario. In *2009 Fifth International Conference on Wireless and Mobile Communications*, pages 82–86. IEEE, 2009.

[28] K. Nohl, D. Evans, Starbug, and H. Plötz. Reverse-Engineering a Cryptographic RFID Tag. In P. C. van Oorschot, editor, *USENIX Security Symposium 2008*, pages 185–194. USENIX Association, 2008.

[29] K. Nohl, E. Tews, and R. Weinmann. Cryptanalysis of the DECT standard cipher. In *Proceedings of the 17th international conference on Fast software encryption*, pages 1–18. Springer-Verlag, 2010.

[30] K. Ogata. Implementierung eines Brute-Force-Kodebrechers, Jan 2009.

[31] M. Weiner, E. Tews, B. Heinz, and J. Heyszl. FPGA Implementation of an Improved Attack Against the DECT Standard Cipher. In *ICISC 2010 - 13th Annual International Conference on Information Security and Cryptology*, LNCS. Springer, Nov 2010.

Akademischer Lebenslauf

1983 Geboren in Lauterbach (Hessen), Deutschland

2002 Abitur am *Alexander-von-Humboldt Gymnasium* in Lauterbach (Hessen)

2003 Beginn Studium Informatik an der *Technischen Universität Darmstadt*

2006 Abschluss *Bachelor of Science*, Thema der Bachelorarbeit *Entwicklung von MicroTLS als sichere Ende-zu-Ende-Verbindung über Bluetooth und TCP/IP* war die Entwicklung eines *SSL/TLS Client Stacks* für mobile Endgeräte.

2007 Abschluss *Diplom-Informatiker*, Thema der Diplomarbeit *Attacks on the WEP protocol* war die Entwicklung von einem neuen Angriff gegen *WEP* gesicherte *WLAN Netzwerke* und die Stromchiffre *RC4*.

2011 Erfolgreiche Verteidigung dieser Dissertation *Security Analysis of DECT*.

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30. Mai 2012

(Erik Tews)