Radboud University Nijmegen

Institute for Computing and Information Sciences

Master's Thesis

Press to unlock: Analysis, reverse-engineering and implementation of HITAG2-based Remote Keyless Entry systems

Author:
ing. Aram Verstegen
Student number:
s4092368

University supervisor:
dr. Peter Schwabe (Assistant professor, Radboud Digital Security Group)
Staff number:
u755144

External supervisor:
Iskander Kuijer (Director, Car Lock Systems B.V.)

Daily supervisor:
dr. ing. Roel Verdult (Research director, FactorIT B.V.)

August 17th 2018

Contents

Chapter 1

Introduction

1.1 Acknowledgements

I would like to thank:

- **Iskander Kuijer** and **Jasper van Herwijnen**, the directors of Car Lock Systems, for granting me the chance to work on a thrilling research project and allowing me to have a look inside the aftermarket car key business;
- The other colleagues from Car Lock Systems (CLS) with whom I could exchange interesting ideas and gain even more insight into the business and technology, in particular **Thijs Breman**, **Joep Brunsveld**, **Armand ten Doesschate**, **Frank Bouman**, **Erwin van der Velden**, **Carolina Kuijer** and **Koen Jonker**;
- Special thanks to **Tom van Biene** for taking most of the pictures in this thesis;
- **Pascal Schiks**, **Roel Jansen**, **Rudy Hardeman** and **Wijnand Modderman-Lenstra** from the Dutch HAM-radio community for accepting a newbie into their midst;
- **Peter Schwabe** for supervising the project and providing inspiring insights.

I would especially like to thank **Roel Verdult** for getting me involved in the project in the first place. I was guided by his daily supervision, mostly administered in person during the car rides he graciously took me on for our semi-weekly trips to Car Lock Systems, as well as through e-mail, via telephone and by text or chat. Besides that I was greatly helped by his many clever ideas and overall guidance over the project's focus. His enthusiasm, professional attitude and the wealth of experience he could rely on have helped me to move across many obstacles we encountered along the way.

I am most grateful to be given a chance to learn by emulation Roel's systematic approach to subjects like reverse-engineering, protocol analysis and code breaking. As I am sure is the case for many other students, the widely publicized dismantling of Mifare Classic by him and his co-authors was a key reason I chose to pursue a Master's degree in Computer Science at the University of Nijmegen in the first place.

That leads me to thank the rest of the people involved with the Digital Security Group past and present, as well as the esteemed faculty members at our sister universities in Eindhoven and Twente under the Kerckhoffs master program.

Thank you all, my time in Nijmegen will always mean a lot to me.

1.2 Electronic security systems in cars

The automotive industry, having once itself been at the very forefront of engineering, has continually incorporated computer technology into their design and manufacturing processes. There is also a persistent trend within the industry to integrate mechanical systems with electronic ones for a variety of practical reasons like efficiency and safety.

Car manufacturers have employed digital electronics for all kinds of control systems in ever more complex and interesting ways over the last few decades. A modern car is now a hotbed of electronic systems, providing everything from engine control to passenger entertainment systems. This means a modern car is not just a computer, but a whole network of embedded computer systems that interact with each other using interconnection buses like the Controller Area Network (CAN) bus system.

Following the trend towards computerization, the vehicle security systems have also gone digital. To prevent hot-wiring of the ignition system, modern cars make use of vehicle *immobilizer* systems that authenticate car keys

electronically, by means of Radio Frequency Identification (RFID) technology. While prevalent, this security mechanism is not one consumers are commonly aware of, as it takes place invisibly, beneath the surface of the locking mechanism.

Drivers will be more acutely aware of Remote Keyless Entry (RKE) systems, electronic locking mechanisms usually exposed though buttons on the car key fob. These devices allow the driver to lock or unlock a car at the press of a button which engages the transmission of a radio signal containing a coded message for the car, in order to actuate its electronic door locks. While having similar security requirements, the type of radios used in immobilizer systems and RKE systems are entirely different, owing to differing requirements in, for example, power consumption and transmission range. RKE systems are built on a one-way connection where one party can only send and the other can only receive over a large distance, while immobilizers universally make use of two-way communication where both parties can send and receive over short distances.

Several researchers have already taken to the automotive technology subject and have proven it is fertile grounds for applied security research. In particular the immobilizer systems are well-studied, and these studies have uncovered serious security weaknesses in such systems [1, 2, 3]. In fact some older immobilizer systems are so flawed their security value has effectively been reduced to zero.

Academic criticisms of their effective security aside, immobilizer systems in general have proven extremely effective against car theft when combined with central door locking [4]. In fact the technology has proven to be so effective that some manufacturers have completely done away with mechanical keys in favor of a 'key card' that can be inserted into the dashboard to disengage the engine immobilizer and start the car. With old-fashioned mechanical keys being phased out, defeating the security of both the keyless entry and immobilizer systems would allow an attacker to both enter and start a modern car through purely electronic means, leaving little to no (physical) traces. That means that, for some types of cars, the door-locking mechanism is effectively its only line of defense against theft.

In this thesis we will contribute to the research of RKE systems, to complement the existing research on what goes on inside car keys. Several such RKE systems are available on the commercial market for systems integrators, however the precise workings of these systems almost exclusively fall under stringent non-disclosure agreements as is commonplace within the automotive industry.

1.3 Focus

Research in this area has thrilling real-world implications reminiscent of Hollywood tropes, i.e. electronically manipulating or stealing cars. It appears that real-life car thieves use much more crude methods like stealing or extorting the keys from the legitimate owners [5]. One of the most sophisticated approaches that seems to occur in the real world to defeat immobilizers involves reconfiguration of the on-board electronics to program in new keys using dealer equipment after gaining entry to the car electronics using 'conventional' means [6]. (This can also involve exposed connectors in unexpected places, such as behind a mirror or head light.) [1]

Another real-life example of a crude but effective attack that targets Passive Keyless Entry (PKE) systems (a feature mostly found in the newest or most high end car models) is reported to be actively exploited by criminals at the present time. This attack involves relaying the Low Frequency (LF) RFID signal from the car's immobilizer system to the key fob over a greater distance than it can normally reach, thereby violating one of the security assumptions inherent in the system's design: that a received signal implies proximity. The return channel from the key to the car can be traversed as normally via one-way Ultra High Frequency (UHF) radio. Once this assumption can be violated the attacker can trigger the key fob's radio transmission at greater distances, thereby getting access to the car outside the owner's view [7].

For now manufacturers have been able to argue that the risk to older cars' immobilizer systems is manageable because physically invasive, on-site attacks are still required before the car can be electronically hot-wired from the inside. Devising practical attacks against the RKE systems found in older cars as we aim to do in this research may help to press the growing urgency of the problem that was uncovered in earlier research once more. But regardless of the exciting possibilities to enter cars by attacking their cryptography, we expect effective real-world attackers to still focus on the weakest link in a security system, such as breaking door locks or windows.

While there are many different types of car keys out there, we know that system integrators for car manufacturers use commercially licensed microcontrollers to implement the immobilizer and keyless entry systems. A very commonly used immobilizer system is HITAG2, manufactured by NXP [8]. We also know that NXP markets *hybrid* variants of the HITAG2 key fob chips which use the same cryptographic primitives to authenticate the key to the car in both the immobilizer and keyless entry systems [1]. We will therefore focus our analysis on car keys which we know to use a HITAG2-based immobilizer system.

[1] This is possible because the networks within the modern car are mostly *bus* systems: systems in which all communications are shared between participants on the shared bus. There is normally no filtering to speak of.

Within the subcategory of HITAG2-based remotes, our selection of specific remotes for thorough investigation in our case studies (Chapter 4) is determined by what is commercially interesting.

1.4 Information sources

We have limited information to go on about the HITAG2 keyless entry systems, which is an unfortunate but common situation when researching proprietary security systems. Luckily, previous research into immobilizer systems has already uncovered many relevant aspects of HITAG2 transponders which allows us to verify our results.

Next to being the most prevalent, HITAG2 is also one of those immobilizer systems whose digital security has effectively been rendered moot as a result of earlier research. This means we can also investigate the binary Electronically Erasable and Programmable Read-Only Memory (EEPROM) contents of transponders for which we should normally not know the cryptographic key.

Through Car Lock Systems we have access to many different kinds of keys and cars with RKE feature, and can make use of their production facility which includes the metal working equipment to cut keys, as well as the diagnostic tools to teach new keys to cars. We also have access to special-purpose hardware and software to work with the RFID transponder chips within the keys without the use of a car, (Proxmark[2], Tango Programmer [3], MiraClone [4]). Beyond that we also have the means to record UHF radio signals through software-defined radios (Ettus USRP, Nuand BladeRF, Great Scott HackRF or RTL-SDR) and accompanying software (GNURadio [9]).

With these resources we should have all the parts required to develop a process to recover the inner workings of the keyless entry systems under investigation.

1.5 Research questions

While the immobilizer chips inside car keys have been thoroughly investigated, there is little publicly known about the composition of HITAG2-based remote keyless entry solutions. Our research question is therefore:

> "What is the protocol employed by various RKE devices and do they provide practical security beyond obscurity given what we know about the HITAG2 cipher?"

with sub-questions:

1. "How do we recover the digital data transmitted by RKE systems over their Ultra High Frequency (UHF) radios?"
2. "Can we learn the protocols used by analyzing their messages?"
3. "Once understood, what security do the recovered protocols provide?"
4. "Is it possible to emulate the devices under investigation?"
5. "Which manufacturers use a HITAG2-based keyless entry system, and how many other keyless systems besides HITAG2 can we identify?"

As a student in the TRU/e security specialization my interests lie in the communication and security aspects of the modern car, in particular their authentication protocols and their application of cryptographic primitives like stream ciphers such as the HITAG2 cipher. In earlier work with Roel Verdult [10] I have gained an in-depth understanding of this cipher and this was my strongest connection to the automotive industry in the larger sense before we started this research. From this earlier work I have access to my own implementation of an efficient HITAG2 brute-force cracking tool as used in this research, which I hope to explain in detail in a forthcoming paper [11].

[2]Proxmark 3 open-source project http://proxmark.org/
[3]Tango Programmer by Scorpio-LK http://www.scorpio-lk.com/
[4]MiraClone by LockDecoders https://www.lockdecoders.com/

1.6 Legal matters

All the documents and software resources referenced during the course of this research were acquired from the public domain, no copy protections were broken, no illegitimate access was attained to trademarked information of any sort. All the third-party hardware and software we have employed during our research was either freely available on the commercial market or is out right open source. We have tried to avoid the use of commercial, closed-source tools and have in fact replaced them wherever viable with open source alternatives. Every step of how we recovered the precise inner workings of the RKE systems in our case studies is documented in this thesis, including references to all papers and presentation slides, notable educated guesses, brute-force solutions, and the final confirmation of testing it on a real car. Finally, we explicitly distance ourselves from any subversive, infringing or illegal use of the research presented in this thesis.

Chapter 2

Background

The radios employed by RKE systems normally transmit in the Ultra High Frequency (UHF) radio band. Before we start our analysis of the radio spectrum we will first detail the general workings of RKE systems, reference the related earlier work in the context of HITAG2, explain the inner structure of the HITAG2 cipher while summarizing the relevant theory and include an example of a cyclic redundancy check (CRC) computation.

2.1 Keyless Entry Systems

Keyless entry systems in cars replace conventional mechanical door and trunk locks by sending a one-way radio signal at the press of a button from a handheld transmitter to a receiver in the car, which actuates a mechanical lock with an electromagnet to lock or unlock the doors or trunk. This has been available as a convenience feature on cars since the 1980's, and similar systems are also used in garage-door openers and alarm systems [12]. The earliest such systems often used infrared signals instead of radio signals, but the principle has remained entirely the same.

The RKE signals contain coded digital information which the receiver can use to determine if the message was sent by the correct (authentic) transmitter. The simplest and earliest example of such a coded message is just a static identifier, for example an encoded command '0' for close and '1' for open. Such a system would not work well in practice when multiple devices are deployed within the range of competing radio signals. Remote control systems should therefore make the messages unique to every device, for example by transmitting a device serial number along with the command. These devices can be characterized as belonging to the 'fixed-code' category.

While somewhat more unsusceptible to interference from other transmitters, a fixed-code type of system still easily falls victim to a *replay attack*, in which an attacker records and later retransmits the (modified) message to operate the system at their convenience. Replay attacks were and remain a real risk, to which the industry responded by sending unique messages for every transmission, and to let the receiver (i.e. the car) reject messages that have already been observed (what is known as a 'rolling-code' or 'hopping-code' mechanism).

Rolling codes for such a system could for example be constructed by using a simple Pseudo-random number generator (PRNG) based on a known *seed* value, but modern systems often involve a cryptographic proof similar to a Message Authentication Code (MAC) provided over an incrementing counter using a cryptographic key shared between the sender and receiver. Ideally this would spawn a sequence of codes which are completely unpredictable to any outside observer that does not have the secret cryptographic key.

Finally, the most complete security protocols can be implemented using two-way radio connections in which cryptographic proofs can be exchanged in both directions ('challenge-response'). We know these kinds of connections from computer networks like the Internet, so we know intuitively there is practically no limit to their scope and sophistication. These protocols could provide the highest security assurances, but are only employed in Passive Keyless Entry (PKE) systems, which fall outside the scope of this research. [1]

Fixed code	A static code for every transmission
Rolling code	A new code for every transmission
Challenge-response	New codes for every transmission in both directions

Table 2.1: Different categories of Keyless Entry systems protocols.

According to the automotive experts at Car Lock Systems most modern cars have an RKE system, but the PKE

[1] These systems let cars communicate to keys through its LF field which ranges outside the car for a meter or two.

feature is not yet commonplace among cars currently on the road. It is still mostly found on newer or more luxury cars.

We are focussing our efforts on recovering the inner workings of RKE car keys employing *hybrid* HITAG2 designs, which use a single HITAG2 hardware chip to authenticate the key in both the immobilizer and the keyless entry system domains [13]. While the bulk of the work can be applied to other RKE systems, we choose to focus specifically on such HITAG2 systems because its cryptography is already well understood, but there are still more protocol details to uncover.

2.2 Relation to immobilizer systems

The immobilizer systems found in modern cars make use of low frequency (LF) Radio Frequency Identification (RFID) technology to achieve mutual authentication. The RFID chips inside these keys are called *transponders*, short for transmitter-responder (as they can only respond to transmissions and do not initiate their own transmissions). Through the LF radio interface the transponder chip exposes a cryptographically secured communication channel to read and program a small amount of EEPROM that can be used to store application-specific information [8]. The data that can be shared between the immobilizer and RKE domains through the EEPROM normally involves its Unique Identifier (UID), cryptographic keys, device settings, rolling-code counters et cetera. These settings except the UID are normally re-programmed on the key via the car's RFID reader by operating its Engine Control Units (ECUs) in diagnostic mode using brand-sanctioned or aftermarket diagnostic equipment connecting over the standardized On-Board Diagnostics (OBD) port. [2]

One piece of information in particular, the HITAG2 transponder's UID, would be particularly useful for an attacker to recover, considering immobilizer protocols require the key to transmit this identifier when prompted before continuing the authentication procedure. Learning it over the air (by sniffing the RKE radio channel) would allow an attacker to initiate the authentication procedure in the HITAG2 immobilizer while impersonating the legitimate (whitelisted) car key, allowing them a foothold to mount practical attacks as explained in earlier work [1, 2, 3].

2.3 Earlier work

Cryptographic research of relevance has documented the construction of the HITAG2 cipher [14, 15], the communication channel and protocol used by HITAG2-based transponders for immobilization, along with various attacks against this protocol [1, 2, 3] and its cryptographic cipher [16, 17, 18, 19, 20], some of which we have implemented in earlier work. We summarize the results in attacking HITAG2 in Subsection 2.5.9 after explaining the structure of the cipher.

2.3.1 Previously reverse engineered HITAG2 RKE system

From earlier research [21] we know what rolling-code protocol is used in UHF transmissions by some HITAG2-based RKE devices. Reproducing this research would be a good first step to get familiar with the tools at our disposal, and verify the authors' claims. This is the first subject of our case studies in Chapter 4 where we apply it to the Opel Meriva B key.

This same topic came up in a more recent publication where the subtle nuances of the system were investigated [22]. We were also able to verify the claims in this research and expand on their results in Chapter 4 and Chapter 8.

2.3.2 HITAG2 device type classification

In related work, Kasper has documented how different types of HITAG2 transponder devices can be identified [23]. He details how the device's UIDs contain the device type encoded in the second to last hexadecimal nibble as seen in Table 2.2.

XX XX XX 1X	PCF7936
XX XX XX 4X	PCF7942
XX XX XX 2X	PCF7946
XX XX XX 7X	PCF7962

Table 2.2: UID to model mapping as recovered in earlier work.

[2] Some car makers employ immobilizer systems and transponders that are pre-configured with security settings by the licensed manufacturer and thus cannot be replaced without manufacturer involvement. This normally means only licensed resellers can order replacement keys, and must specify the vehicle chassis number.

Unfortunately this is not the whole story, as newer generations of transponders can emulate the behaviour of their predecessors. These newer devices contain Flash memory that is user-programmable, allowing system integrators to augment the device's (RKE) features with custom code. This nibble therefore only serves as a broad indication of the feature set provided by the transponder interface of the device. Any HITAG2 RKE systems we investigate will need to be classified by their UHF signals and not just their behavior as a transponder.

2.4 Notation

We represent binary values as we interact with them in software with the most significant bit on the left, and index bits starting from zero at the least significant bit.

We will use $n(0x\texttt{hh})$ to denote a string of n repeated bytes in hexadecimal notation '`hh`'.

Furthermore we will use subscript to denote bit indexing, $\|$ to indicate bitwise concatenation, $\vee$ to mean logical OR, $\wedge$ for logical AND, $\neg$ for negation and $\oplus$ to denote the exclusive-or (XOR) operation (addition modulo two).

2.5 The HITAG2 cipher

A major part of this research centers around use of the HITAG2 cipher as used in RKE systems based on the PCF79XX family of transponders. The security guarantees that can be made about these systems primarily depend on the cryptographic strength of the cipher construct. The HITAG2 cipher was developed in the mid 1990's by Mikron electronics, later acquired by Philips semiconductors which then became NXP semiconductors. The structure of the cipher was recovered through reverse-engineering in the last decade, which then prompted the academic research community to investigate weaknesses in its design.

2.5.1 Cryptography & ciphers

A cipher is the logical construct which is used to *encrypt* and *decrypt*, that is: to translate readable information (*plaintext*) to something unreadable (*ciphertext*) and vice-versa. The ideal is to make these ciphertexts indistinguishable from random noise, so that an observer cannot learn the contents of the messages. The only way such transformation of the information adds any cryptographic security is if there is a secret part to the deterministic algorithm used to encrypt and decrypt.

Conceptually this secret part relates to the rest of the cipher in the same way a key does to a lock, hence we call it a *key*. There are two main branches in cipher design, namely *symmetric* (or secret key) and *asymmetric* (or public-key) cryptography. The HITAG2 cipher is a symmetric cipher, which means that the same key that was used to encrypt a message is required to perform successful decryption. [3] Within the category of symmetric ciphers, HITAG2 is a *stream* cipher: it produces a *keystream* of bits that are used to encrypt/decrypt individual data bits using the XOR operation, rather than encrypting/decrypting multiple input bits together as a block of data (the approach taken by *block ciphers*).

[3] Asymmetric cryptography has the interesting and useful property that these operations can be performed using *different* keys, but despite being interesting that has no direct relevance to this research.

2.5.2 Exclusive or (XOR) and OTP

An operation commonly found in symmetric cryptography is the XOR operation. This binary operator yields true for two single bits if they are differently valued and false otherwise. This implies the operation can be reversed by re-applying the result with one of the operands to get back the other operand; it is effectively returning the difference. As such, XORing two equal values will always yield zero as a result.

The XOR operator can be used for cryptography as shown in Equation 2.1, in a system once patented as the 'Vernam Cipher'.

The XOR operator can be used to perform a One-time pad (OTP) encryption. Provided the secret key is chosen randomly and used only once, OTP provides provable *perfect secrecy*. Its name may sound too good to be true, but this perfect secrecy principle merely comes down to the property that a given ciphertext bit can be decrypted to resemble any plaintext bit depending on a chosen key bit. In other words: when the key is assumed to be random any possible fixed-length plaintext is equally likely to have resulted in the observed ciphertext (or vice versa). A good OTP key should therefore look entirely random as this will make the encrypted data appear entirely random as well.

$$\begin{aligned} \text{message} &= M \\ \text{secret key} &= S \\ \text{ciphertext} &= M \oplus S \\ \text{decrypted message} &= \text{ciphertext} \oplus S \\ &= M \oplus S \oplus S \\ &= M \end{aligned} \tag{2.1}$$

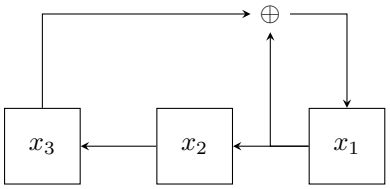
The most obvious drawback to this is that communicating parties need to agree on a strong new key for every message, and this key needs to be at least as long as the message. [4] Symmetric ciphers therefore invent ways to enable a single key of a given fixed length to secure multiple messages which are longer than the length of the key. This is also the role of the HITAG2 cipher primitive; after initialization its stream of output (*keystream*) bits effectively constitutes the OTP key, which is XORed with the plaintext or the ciphertext to encrypt or decrypt.

2.5.3 Linear Feedback Shift Register

In cryptography it is often useful to extend a true random value to a longer sequence of *pseudorandom* values. A simple example of a PRNG that performs this function can be found in the Linear Feedback Shift Register (LFSR) primitive. A *shift register* is a fixed-size vector of n bits where all bits can be shifted by one position (left or right). The last bit on the direction we are shifting in will be expunged as output, while on the opposite side we will have created an unused position for a new input bit. Such a construct can be made to iterate over the elements within the underlying (binary) number field in a deterministic but nontrivial sequence of traversed elements by using *linear feedback*.

By designing a feedback function from the state bits to the next input position we can influence the order in which elements are traversed. These sequences will eventually form *cycles*, where the last traversed element leads back to the initial element through the feedback function. The idea is now to make this cycle of traversed elements have a maximal length before repeating, thereby stretching our initial state to produce a maximal number of pseudorandom ones. If we were ever to land on zero there would be no more feedback bits and we would have broken the cycle, so because zero can never be in a cycle, $2^n - 1$ steps is the maximal cycle length.

An example of an LFSR in the field $\mathbb{F}_{2^3}$ can be seen in Figure 2.1. This LFSR traverses a cycle of length $2^3 - 1 = 7$, which is the maximal cycle length for this field.



x_3	x_2	x_1	Value
0	1	0	2
1	0	0	4
0	0	1	1
0	1	1	3
1	1	1	7
1	1	0	6
1	0	1	5
0	1	0	2

Figure 2.1: On the left we see an LFSR schematic and its outputs are shown on the right. These traverse the finite field $\mathbb{F}_{2^3}$ using the feedback polynomial $x^3 + x + 1$.

[4] For OTP encryption to actually be secure, encrypted messages need to be of fixed length.

Using the underlying mathematics we can construct a feedback function that ensures the cycles are of maximal length for registers of any length, so this principle can be scaled up to larger bit vectors which generate larger cycles. For example, the 48-bit LFSR used in the HITAG2 cipher has a feedback function that generates a cycle of maximal length, ensuring a cycle period of $2^{48} - 1 = 281474976710655$ steps.

2.5.4 Non-linear function

After initialization, the LFSR in the HITAG2 cipher constitutes its internal state and determines the exact sequence of output (keystream) bits. Like in the similar Crypto-1 cipher [24, 25], keystream bits in the HITAG2 cipher are generated by passing bits from the secret 48-bit internal state register through a nonlinear *filter* function.

The filter function performs a nonlinear mapping from 20 input bits to one output bit. Computing the transformation requires several boolean logic steps to compute, the best known solution for which is shown in Equation 2.2 (sourced from the HITAG2 cipher implementation pseudonymously released by Wiener [15]). To optimize for speed, most software implementations implement these functions using lookup tables as shown in Figure 2.2.

$$\begin{aligned} f20a(a,b,c,d) &= \neg(((a \vee b) \wedge c) \oplus (a \vee d) \oplus b), \\ f20b(a,b,c,d) &= \neg(((d \vee c) \wedge (a \oplus b)) \oplus (d \vee a \vee b)), \\ f20c(a,b,c,d,e) &= \neg((((((c \oplus e) \vee d) \wedge a) \oplus b) \wedge (c \oplus b)) \oplus (((d \oplus e) \vee a) \wedge ((d \oplus b) \vee c))). \end{aligned} \tag{2.2}$$

2.5.5 The big picture

Each time a keystream bit is generated, the state is permuted by the Linear Feedback Shift Register (LFSR) to iterate to a different state so that the cipher will generate the next keystream bit using an entirely different valuation of the filter inputs.

Twenty bits in the state lead to the two-stage nonlinear filter function (*f20*) to produce one keystream bit. Simultaneously sixteen bits of the state are XORed together to produce a new LFSR feedback bit, which is shifted in to reach the next state. The whole cipher is visualized in Figure 2.2.

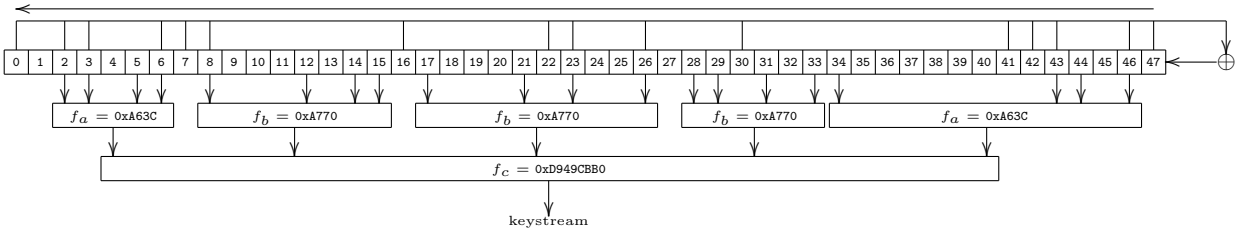


Figure 2.2: Structure of the HITAG2 stream cipher [1]. The hexadecimal values pictured in the 2-layer filter represent binary lookup tables for the boolean functions in Equation 2.2.

2.5.6 Initialization

Knowing the properties of these building blocks, *f20* and the *LFSR*, we need only let communicating parties agree on the initial secret state to be constructed before they can create identical keystreams. Next to sharing the public UID, the last requirement for this step is the generation and exchange of a public number used only once (Nonce), used to ensure the initialized cipher state is different for every authentication. Before keystream bits are generated, the 48-bit HITAG2 cipher state is then initialized as follows:

1. The 32-bit UID is shifted highest-bit-first into the state from the MSB (left) side.
   `uid_lo` ‖ `uid_hi` ‖ 2(0x00)
2. The highest 16 bits of the key are shifted into the state in the same way.
   `key_hi` ‖ `uid_lo` ‖ `uid_hi`
3. The 32-bit nonce is shifted into the state where each nonce bit is encrypted by XORing it with the output of the filter function.
   `enc_nonce_lo` ‖ `enc_nonce_hi` ‖ `key_hi`

Note that for every initialization with the same key, 16 of the 48 bits in the state are initialized the same way regardless of the nonce. The UID and nonce are public information, the only secret information required by the parties is the key. The default key, used to authenticate to HITAG2 transponders before their key has been personalized by the car, is made up of the 48 bits in the the ASCII characters 'MIKRON', which translates to 'ONMIKR' after swapping two 32-bit words, which is `0x4f4e4d494b52` in hexadecimal notation.

2.5.7 Encrypted communication

With all these pieces in place, communicating parties can encrypt and decrypt messages they send to each other knowing only the secret key and the public UID and nonce. The initiator can send the nonce in plain text, which the receiver will then use to initialize the cipher on their end. The secret state is then initialized on both sides using partly public and partly secret information. Once the same keystream can be generated on either side, both parties can encrypt and decrypt as long as their respective cipher states remain synchronized as shown in Figure 2.3.

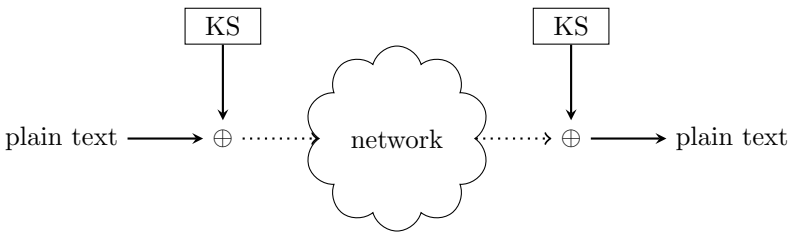


Figure 2.3: Stream ciphers on each side generate the same keystream which is used to encrypt or decrypt plain text using the XOR operation.

2.5.8 Message Authentication Codes

A Message Authentication Code (MAC) is a cryptographic primitive that can be used to authenticate messages using a secret key that is shared between communicating parties. Among other designs, these functions can be based on ciphers (CMAC) or hash functions (HMAC) and both work on the principle of performing a one-way function using a secret key and the message to construct a proof that could (ideally) only have been generated by those who possess the secret key.

The HITAG2 cipher is also used like this for authentication protocols: a sender will generate a fresh nonce and transmit this along with the UID and a sample of keystream that was constructed using these public values and the secret key. This allows a receiver who knows the secret key to recreate the same keystream and verify that the sender must indeed have known the secret key, and is therefore authentic. To avoid confusion with more effective MAC schemes used in modern security solutions I will refer to HITAG2's use of keystream in this way as HITAG2 *cryptograms*.

2.5.9 HITAG2 attacks

The HITAG2 cipher's internal state is only 48 bits, which by modern standards in cryptography is too small to be considered secure. Like with any stream cipher, XORing a pair of related ciphertext and plaintext samples will yield a sample of keystream (see Equation 2.1). Given the keystream, the associated nonce and the UID we can exhaust the 2^{48} internal states and see which of these generate the same keystream we uncovered. The more keystream samples we have, and the longer these samples are, the more internal states we can eliminate (faster) until we recover the one correct internal state. Once the secret internal state is known (along with the other public information) the secret key can be retrieved by reversing the initialization procedure. Earlier results of practical and theoretical attacks are presented in Table 2.3.

Attack	Description	Practical	Computation	Samples	Time
[19]	brute-force	yes	2102400 min	2	4 years
[16]	SAT-solver	yes	2880 min	4	2 days
[22]	brute-force	yes	1080 min	2	18 hours
[18]	brute-force	yes	660 min	2	11 hours
[20]	SAT-solver	no [5]	386 min	N/A	N/A
[26]	cube	no [6]	1 min	500	N/A
[1]	cryptanalytic	yes	5 min	136	6 min
[21]	correlation	yes	1-10 min	4-8	1-10 min

Table 2.3: Comparison of attack times and requirements. (Sourced from Verdult et al. [1] and expanded.)

Only two of the earlier works specifically looked at the use of the HITAG2 cipher in the context of RKE, where samples contain only 14 of the 32 nonce bits and also contain only 32 bits of HITAG2 keystream output. This problem identified by Garcia et al. [21] was solved in their research by simply guessing the missing information, but Benadjila et al. [22] found that this data is randomized in newer cars to attempt to defeat such guesses.

To at least partially recover the HITAG2 cipher's internal state from an RKE transmission we made use of our own implementation of a practical attack that stemmed from a separate research project [10, 11].

Once recovered, this internal state allows the same possibilities that Benadjila et al. explored to demonstrate a practical attack, even while bits in the real key and nonce remain unknown. This idea is explained in more detail in Chapter 8.

2.6 Cyclic Redundancy Check

The same mathematical principles that are used in LFSRs can be used to create error-checking codes, usable to verify correct reception of the data. This is not part of the HITAG2 cipher itself, but a part of the RKE protocols we will encounter in Chapter 4.

By taking a characteristic polynomial of n bits over a series of bits in the finite field we will end up with an n-bit remainder which we can append to the message. The remainder contains some redundant information about the message. The receiver can then perform the same modular reduction on the message and verify that they have the same remainder. Any single error the received message would incur a cascade of differing bits in the remainder. In such a checksum value we have a probability of only $\frac{1}{2^n}$ of a false positive, that is *not* detecting the error. An example of a 3-bit CRC computation is given in Figure 2.4.

[5] Soos et al. require 50 bits of consecutive keystream
[6] Sun et al. require control over the nonce

```
10101010 000
1011
00011010 000
   1011
   01100 000
    1011 000
    0111 000
     101 1
     010 100
      10 11
         010
```

	x^{10}	$+x^8$		$+x^6$		$+x^4$			
$-$	x^{10}	$+x^8$	$+x^7$						
			x^7	$+x^6$		$+x^4$			
$-$			x^7		$+x^5$	$+x^4$			
				x^6	$+x^5$				
$-$				x^6		$+x^4$	$+x^3$		
					x^5	$+x^4$	$+x^3$		
$-$					x^5		$+x^3$	$+x^2$	
						x^4		$+x^2$	
$-$						x^4		$+x^2$	$+x$
									x

Figure 2.4: The computation of a 3-bit Cyclic Redundancy Check (CRC) from an 8-bit value using the polynomial $x^3 + x + 1$.

Chapter 3

Analysis

Bringing our existing knowledge of HITAG2-based systems to the research of RKE systems requires us to bridge the gap from the ether to the software domain. To investigate the UHF spectrum we can make use of Software-defined radio (SDR) devices and custom software to receive, demodulate and decode these signals into strings (or *frames*) of bits.

3.1 Radio signals

Radio signals propagate through the electromagnetic spectrum at the speed of light. Transmitters can excite the field by using an oscillating antenna with alternating current at a given frequency, and receivers can pick up these excitations at a distance, limited by the power with which the transmitter is exciting the field.

Because radio transmissions can interfere with each other they are governed by strict government regulations on a worldwide scale; in the Netherlands this regulation is performed by Agentschap Telecom. Certain bands are strictly reserved for radio and television, others for mobile telephones, and there is a specific band which is reserved for purposes other than telecommunications. These are called the Industry, Science and Medical (ISM) bands and they have been defined by the International Telecommunication Union (ITU) [1] Radiocommunication Sector (ITU-R). The devices we are interested in have been relegated to these ISM bands and in our part of the world (ITU Region 1) they are tuned at 433.920 MHz center frequency (with a maximum bandwidth of 1.74 MHz), but we can also expect to see transmitters operating around 315 MHz for overseas cars.

3.2 Antennas

The frequency and bandwidth of radio signals that an antenna can transceive is determined by its shape, most importantly its physical length. The fundamental frequency f that an antenna is tuned to is directly related to its length λ through the speed of light c.

$$\begin{aligned} \lambda &= \frac{c}{f} \\ &= \frac{299792458 \text{ meters per second}}{433920000 \text{ waves per second}} \\ &\approx 0.69 \text{ meters per wave} \end{aligned} \tag{3.1}$$

In Equation 3.1 we can see that a full-wavelength wire antenna tuned to 433.92 MHz would be almost 70 centimeters long. This is not convenient for small radios in practice, so most systems actually tune the antenna to one of the harmonics of the signal's baseband frequency. These harmonics occur at one half, one third, one quarter wavelengths and so on with exponentially less power at every harmonic (see Figure 3.1).

The UHF radios found in car keys often use circular ('loop') antennas of $\frac{1}{16}$th wavelength (4.3 cm) or circular antennas of $\frac{1}{32}$nd wavelength (2.1 cm). We receive their signals using wire ('whip') antennas of various lengths ($\frac{1}{4}$th, $\frac{1}{16}$th).

[1] This United Nations specialized agency tasks itself with drafting standards and regulations to achieve international cooperative use of the spectrum. The organization spans 193 member countries and over 700 industry and academic partners.

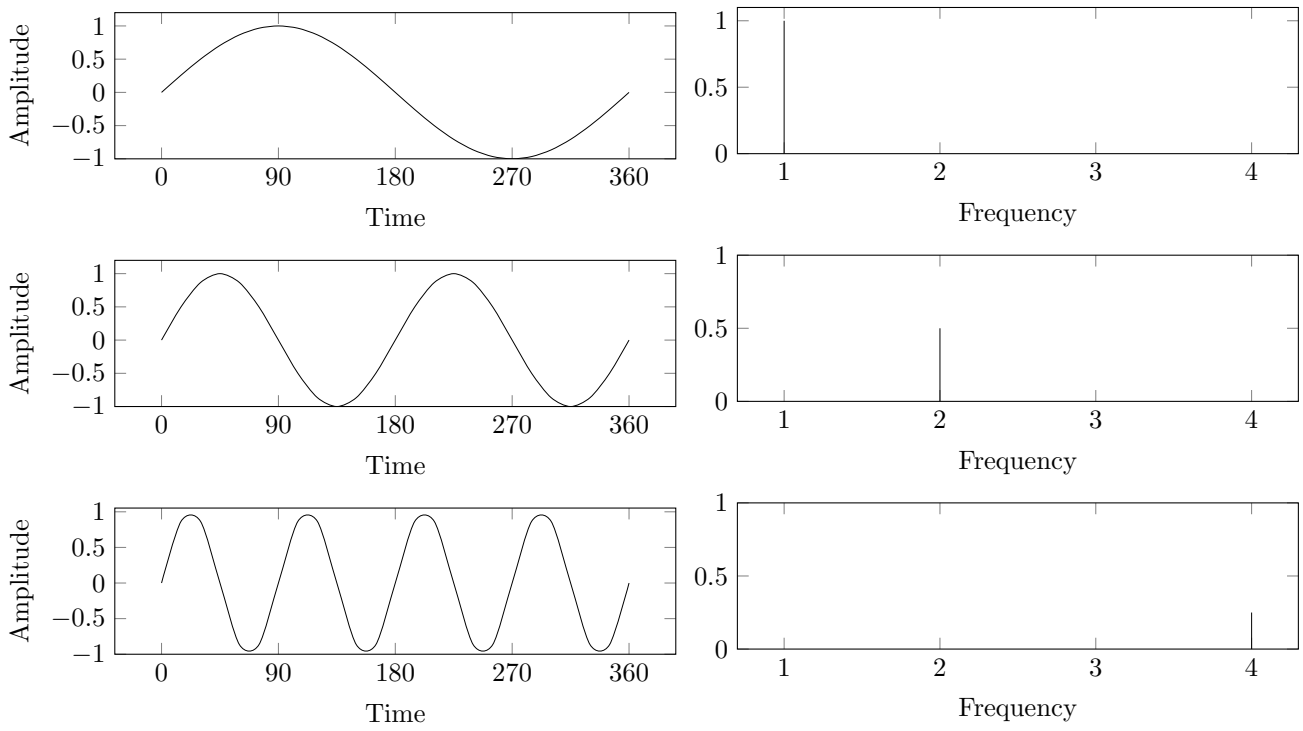


Figure 3.1: First, second and fourth harmonic in the time (left) and frequency (right) domain, where amplitude is adjusted for an antenna tuned to the first harmonic.

3.3 Modulation

From the excitement of the electromagnetic field it is a small step to transmitting information. This excitation can be represented as a waveform graph which represents the intensity (amplitude) and polarity (phase) of the field excitation over time at a certain frequency. When the transmitter is switched on it excites the field at and around its *baseband* frequency, that is, its tuning frequency determined by the antenna length. In the case of the key fobs we are investigating, the baseband frequency is almost always 433.92 MHz.

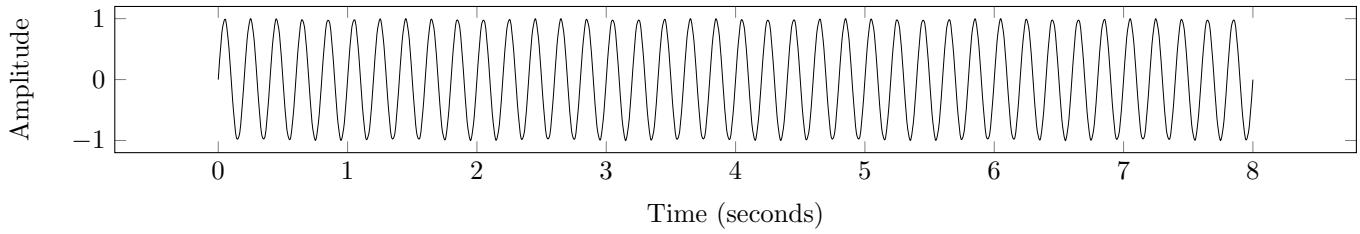


Figure 3.2: Plot showing baseband signal.

In Figure 3.2 we schematically show an excitation of the *baseband* at 5 Hz for a duration of 8 bit-periods at a rate of 1 bit per second, i.e. a radio signal that lasts for 8 seconds. To transmit a data signal over the *baseband* we can vary (modulate) the *amplitude* with which we excite the field or the *frequency* at which we do so. As an analogy, the modulation of the baseband can be thought of as a flickering of light of a given color (frequency) to encode bits at a certain rate, and it is this flickering pattern we are interested in, rather than the color (baseband frequency) of the light itself.

In digital electronics like the key fobs where digital data is modulated, we speak of *keying* schemes, such as Amplitude Shift Keying (ASK) also known as On-Off Keying (OOK), Frequency Shift Keying (FSK) and Phase Shift Keying (PSK). The following figures exaggerate the modulation characteristics; in practice it is almost impossible to tell frequency-modulated (FM) and phase-modulated (PM) signals apart by visual inspection.

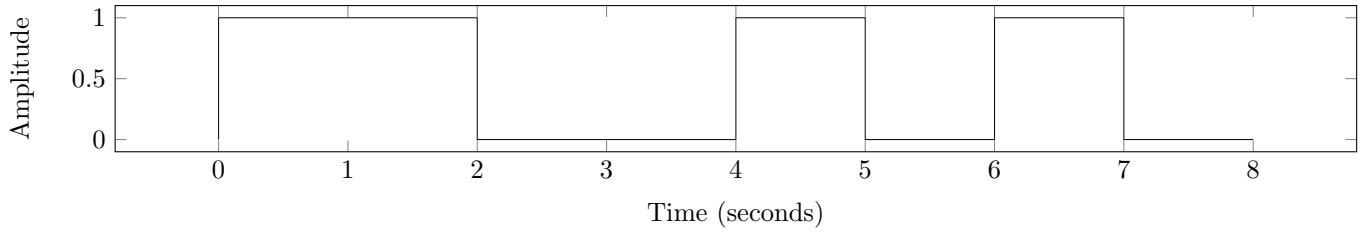


Figure 3.3: Plot showing a data signal with 8 bits (11001010).

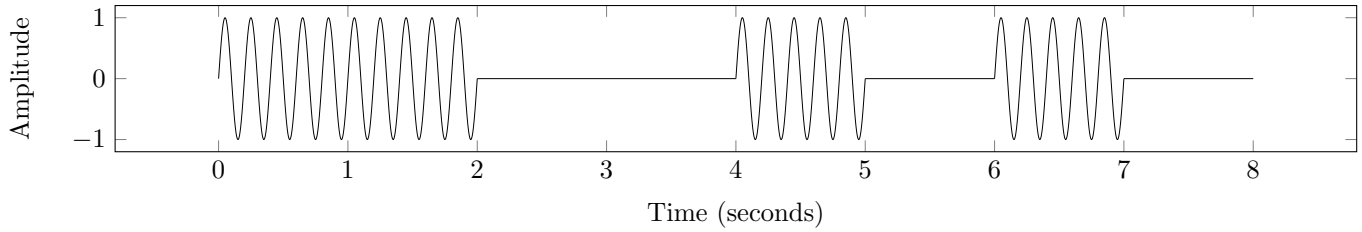


Figure 3.4: Plot showing AM/ASK/OOK modulated data signal with 8 bits (11001010).

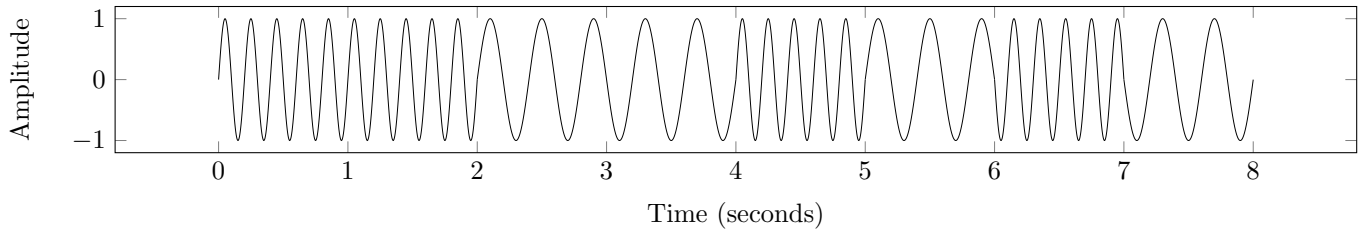


Figure 3.5: Plot showing FM/FSK modulated data signal with 8 bits (11001010).

Another thing we could use is the *phase* of the excitation.

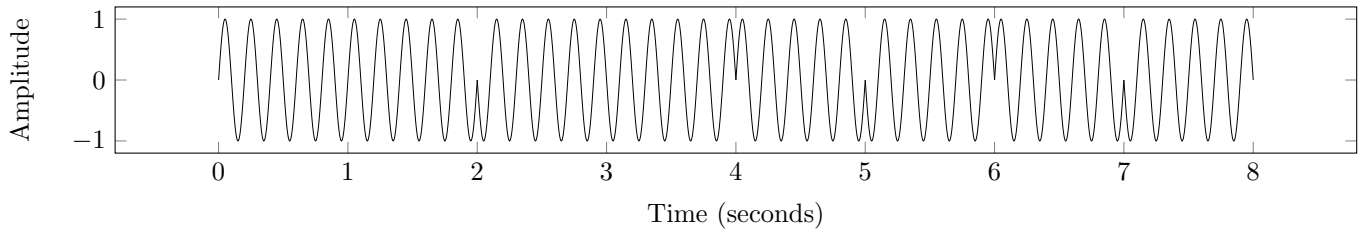


Figure 3.6: Plot showing PM/PSK modulated data signal with 8 bits (11001010).

3.4 Software-defined radio

Given that the data signal we are interested in is modulating the baseband signal at a much lower rate than the actual excitation frequency of the electromagnetic spectrum, it makes sense to limit the precision with which we actually take this information into a computer. The translation from the analog to the digital domain is where SDR plays a role. Please note that a software defined radio is not a requirement to investigate the UHF radio signals we investigate here, it is just the most practical approach because of its versatility and affordability.

Just as we have seen (embedded) software and digital signal processing permeate the automotive industry, software has also taken up the role of signal-processing components which are traditionally analog such as filters, mixers, modulators and demodulators. Ideally we could just connect an antenna to an analog-to-digital converter (ADC) and start measuring, but high-frequency signals such as the ones we are investigating are actually quite hard to capture in the digital domain with common off-the-shelf hardware and software.

Because radio signals exist in the analog domain (where signals are continuous) and we want to translate them to the digital domain (where signals are discrete) we need to employ *sampling*. As when sampling sound waves to digital audio we have to define how precisely we will measure the excitation of the field using a given number of bits (*bit depth*) and how often we will take this measurement over time (*sample rate*). These aspects together determine the signal's *bandwidth*.

With higher *sample rate*s we can demodulate faster data rates, and with greater *bit depth* we can pick up more subtle deviations in the excitation of the field (see Figure 3.8). The question is then: how much resolution do we need in each aspect for our research?

Modern SDR hardware exposes a general-purpose computer to a radio front-end, which tunes to the baseband frequency, samples the signal to a given *bit depth* using an ADC with a given sampling frequency (leading to the required/desired *bandwidth*), applies filtering and passes discrete samples to the host at this rate. In most modern SDRs this front-end is largely implemented in a programmable logic chip like an FPGA or CPLD to also allow software control over such sampling and filtering. The general-purpose PC hardware is then tasked with processing the actual information contained in the signal within the digital domain.

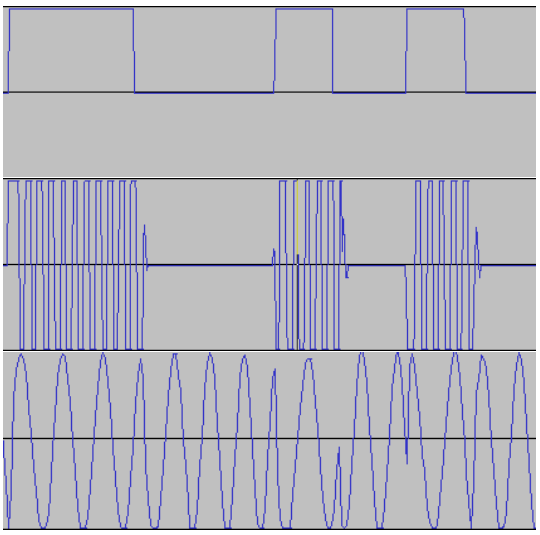


Figure 3.7: A demodulated 9600 baud data signal with 8 bits (11001010) with corresponding AM and FM signals as they were recorded in practice from different UHF remotes, rendered and aligned in our .wav file editor Audacity.

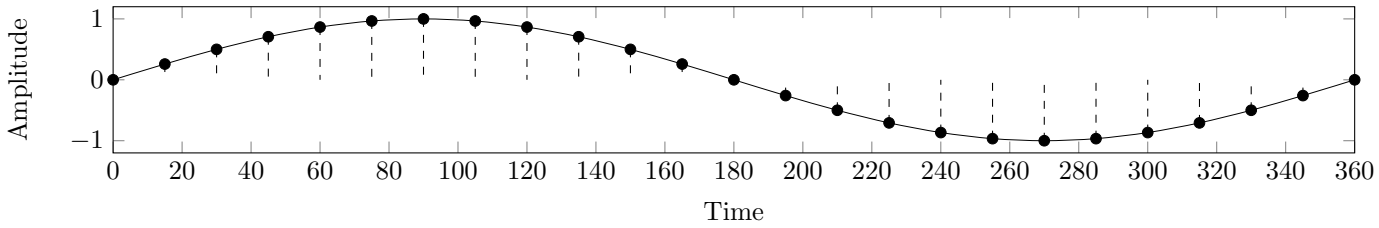


Figure 3.8: Sampling a continuous waveform to the digital domain requires us to limit the information in the signal to discrete values. We must *quantize* the sampled amplitude values to a fixed number of bits (which relates to their relative power in dB) and take these samples at a *rate* of more than twice that of the signal we want to record.

3.4.1 Bit depth

The bit depth required depends on the strength of the signal in the face of surrounding noise. The amplitude samples must be scaled to match the precision available in the given bit depth to translate from a continuous power sample to a discrete number of bits – a process known as *quantization*. This means we will need to round values leading to *quantization errors*. With more bits available we can transcribe our amplitude samples with higher precision, and incur smaller quantization errors. Through this consideration bit depths directly relate to *dynamic range*, measured in decibels (dB). Decibels are not actually a unit of measure, but express a ratio between two quantities. We can use the decibel to express the precision of amplitude samples relative to surrounding noise – the Received-signal-strength indication (RSSI), or the total dynamic range available in a binary system of n bits.

$$\begin{aligned} 10 \cdot \log_{10}(\text{ratio}^2) &= 20 \cdot \log_{10}(\text{ratio}) && \text{(To convert amplitude ratios to dB.)} \\ \text{ratio} &= \frac{2^n}{1} \\ 20 \cdot \log_{10}(2) &= 6.02\,\text{dB} && \text{(So about 6.02 dB per bit.)} \\ 20 \cdot \log_{10}(2^8) &= 48.16\,\text{dB}. \end{aligned} \tag{3.2}$$

We can compute the dynamic range in decibels for a given bit depth using Equation 3.2, which assumes the 'worst case' for the sampled amplitudes: a uniform distribution. We can use the laws of exponents to take out the powers inside the logarithm and take them as leading coefficients. Thus we can compute the dynamic range of a given bit depth on a linear scale where each bit gives us roughly 6 dB of dynamic range. 8-bit values already give us approximately $6 \cdot 8 = 48$ dB of dynamic range in the amplitude.

Unlike in the analog domain, in the digital domain we have an absolute maximum loudness, so we use the decibels relative to full scale (dBFS) as the scale for our amplitude samples. We use a fully 'on' string of bits to represent 0 dBFS, half that value (MSB set to off) as -6.02 dBFS and so on. Using this scale we can express a signal's peaks relative to an absolute maximum, and the minimum is only defined by how many bits we can transcribe into.

Decibel values are also used to give an indication of a signal's relative power. Signal and noise strengths (powers) are normally expressed in Watts, but can be expressed in decibels per Watt (dBW) where the value captures the power ratio of the signal to a 1-Watt reference. Actually, we normally measure the power of radio signals in milliwatts, leading to the decibel per milliwatt (dBm) measure of signal strength.

3.4.2 Sample rate

The sample rate we need is determined by the bandwidth of the signal we want to capture, which is in turn determined by its data rate and modulation method. We have to take into account the Nyquist-Shannon sampling theorem, which states that the sampling rate required to accurately sample a signal of a given bandwidth must be more than double that of the original signal. When measuring the amplitude of a waveform we only have discrete samples indicating the amplitude of the wave, but not its actual movement which can be captured in vectors or complex numbers. This can lead to misinterpretation of the sampled signal, called *aliasing* (see Figure 3.9).

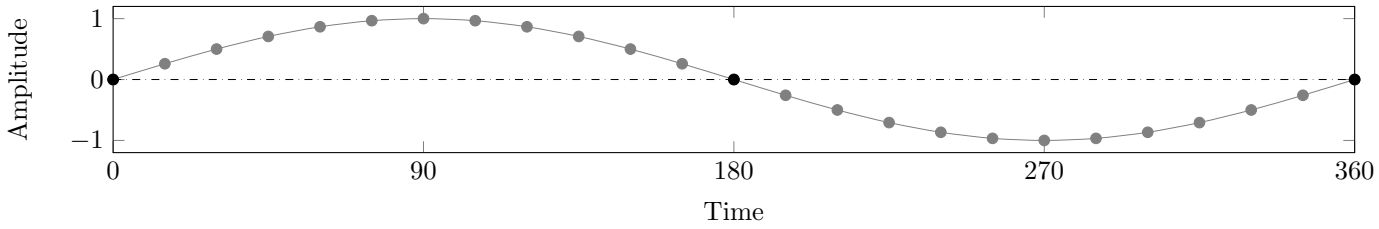


Figure 3.9: Sampling a signal at exactly twice the signal's bandwidth can still result in *aliasing*.

By sampling at more than twice the signal's frequency we can ensure we know not just the position but also the trajectory of the wave between every two samples to allow for full reconstruction using interpolation (see Figure 3.10).

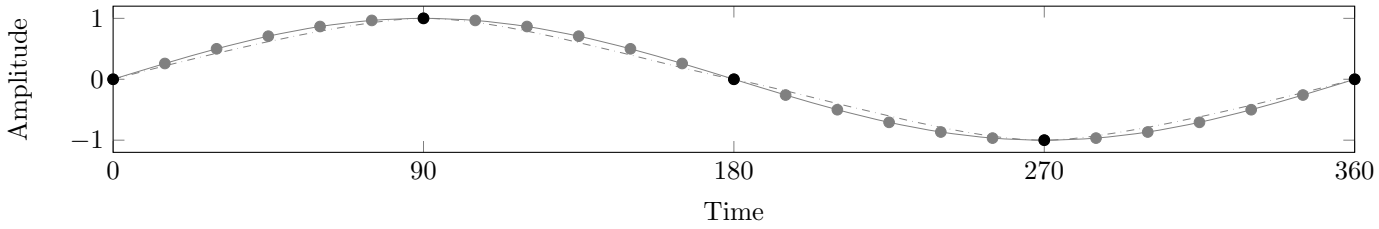


Figure 3.10: Sampling a signal at more than twice the signal's bandwidth allows for a full reconstruction using *interpolation*.

In practice we will have to use a much higher sample rate than merely twice the signal's bandwidth because of signal loss incurred at every stage of the process due to noise. From early experiments we learned that a bandwidth of 512.000 samples per second appears to be adequate for most transmitters we have researched. In ideal circumstances this would give us about 53 amplitude samples per bit in a signal transmitted at 9600 bits per second, which is significant enough to filter out any noise in practice. But, to maintain some margin for error in case we encounter higher digital data rates in our analysis, we would advise a sample rate of 2.048 million samples per second.

3.5 GNURadio

The GNURadio toolkit contains software implementations of a collection of algorithms that mimic the behaviour of hardware components used in (radio) signal processing, implementing them as interconnectable signal-processing *blocks*. The cores of these software blocks are implemented in native C/C++ code to deliver high performance, and these cores are then exposed to the Python scripting language through binding wrappers generated using the Simple Wrapper Interface Generator (SWIG) [27].

One convenient side-effect of this block-based model is that because it is so regular and completely object-oriented, the processing blocks can be dealt with as editable nodes in a runnable signal-chain graph. This idea is realized in

the GNU Radio Companion (GRC) software, a graphical front-end to the toolkit which allows users to draw signal-processing schematics using these blocks, which can be compiled to runnable Python code at the click of a button. Because of its convenience we used this feature extensively to experiment with recording and processing the signals. Once working as desired and compiled to Python the schematic can be used standalone just like any other Python code, which is a convenient way to rapidly prototype standalone applications making use of the GNURadio toolkit.

3.5.1 Hardware

We have used the following commercially available SDR USB peripherals supported by the open-source GNURadio platform to make recordings of the UHF signals:

USRP B100

The USRP B100 manufactured by Ettus Research is a flexible and easy-to-use device to work with digital radio signals in various spectra. Our model is equipped with a WBX daughterboard which covers the spectrum from 50 to 2200 MHz. The model is discontinued; its retail price was 620 $ at introduction in 2011. Since then community support for the drivers required to use this device has fallen behind compared to the other options listed here.

BladeRF x40

The BladeRF x40 manufactured by Nuand LLC is a much more modern device that aims at filling the same role as the USRP. It is also considerably cheaper and seems to be aimed at the advanced hobbyist market. The tuning range of this hardware spans the spectrum from 300 to 3800 MHz. Its retail price is 420 $ and the community driver support is good.

HackRF

The next step down for an affordable SDR is the HackRF by Great Scott Gadgets. It is an open source hardware/-software design which has an active community. The tuning range runs from 100 to 6000 MHz. Its retail price is 300 $ and community driver support is excellent.

RTL-SDR

By far the most accessible option to explore the use of SDRs is to repurpose a common DVB-T receiver USB stick based on Realtek RTL2838U chipsets. It was discovered by members of the Osmocom project [28] that their design allows the USB host to bypass its built-in (QAM) demodulator and MPEG decoder by only configuring the front-end filter and tuner, after which the host can receive 'raw' digital samples from it. The tuning range of this hardware depends on the tuner component used, most span the spectrum from 300 to 3800 MHz. Note that it can only receive and does not have transmitter hardware. Its retail price: < 10 $ and community driver support is excellent.

3.5.2 Recording

First we record the signal using no additional preprocessing besides a squelch filter that will prevent the rest of the signal-chain from being triggered while the power detected in the input signal is below a certain threshold. The signal is then converted from the complex domain to the discrete (floating-point) domain where both components of the complex values are transcribed into separate, synchronized `.wav` files at 512.000 samples per second using 8-bit samples (see Figure 3.11).

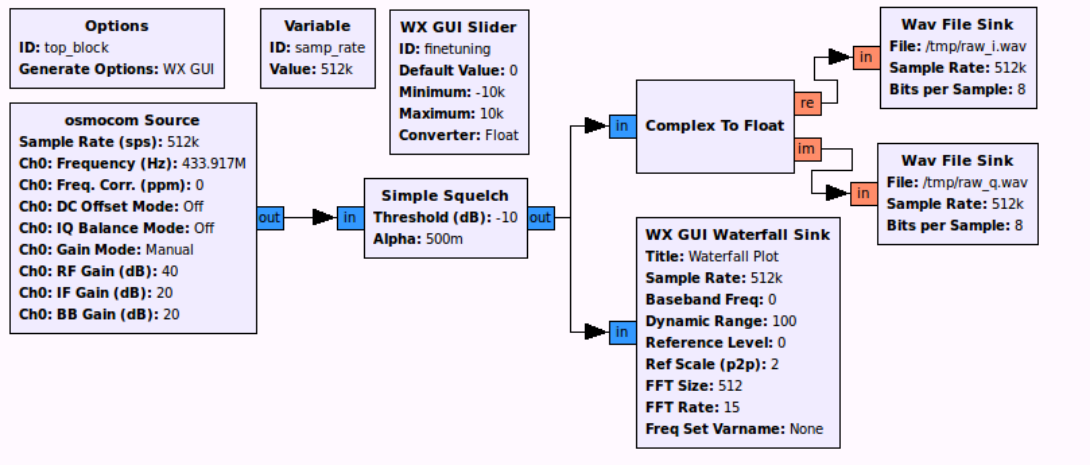


Figure 3.11: An example sketch to capture data in GNU Radio Companion (GRC) with separate files to store both the real and imaginary parts of the complex values in the continuous signal.

3.5.3 Demodulation

After recording a signal we can demodulate it using multiple demodulators with differing settings, as shown in Figure 3.12. ASK signals are simple in that no further settings besides tuning are required for a correct demodulation, but to demodulate FSK signals we need to configure the frequency deviation used. [2]

With these sketches we can record and demodulate UHF signals, and visually identify that they indeed contain usable data (see Figure 3.7) in the sound file editor Audacity [29]. The last step to get back to the software domain requires decoding these wave shapes to bits.

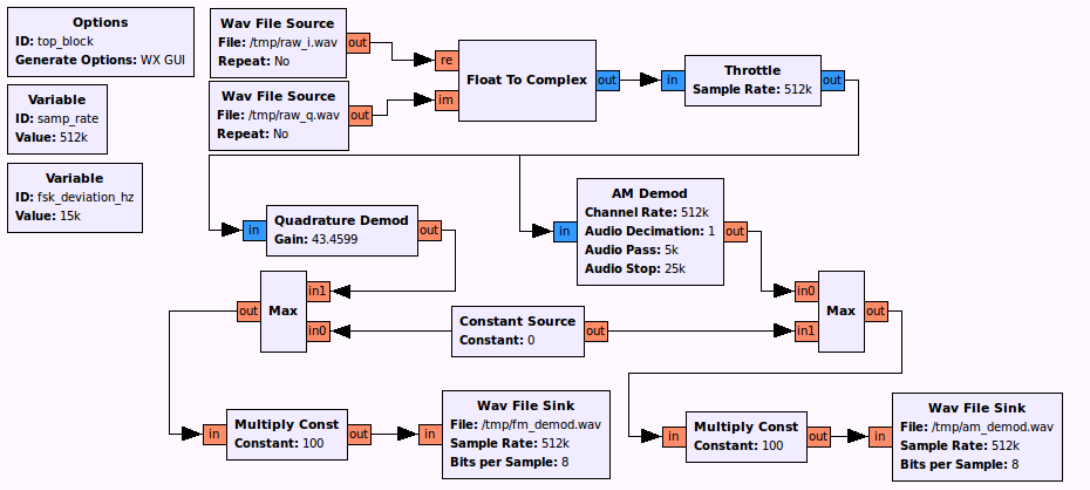


Figure 3.12: A GNU Radio Companion (GRC) sketch to demodulate AM and FM signals simultaneously after reconstructing them from .wav files containing I/Q samples.

3.6 Coding and decoding

Within the signals we have so far demodulated we can expect to see some sort of coding mechanism that allows for synchronization between the sender and the receiver. A preamble in the transmission with a known pattern allows the receiver to do just that synchronization, after which the decoder processes data at the detected rate.

There are several line-coding mechanisms we can expect to see: Manchester, Biphase-M (Mark) and Biphase-S (Space) coding.

[2] Recovery of the missing variables like baseband fine-tuning and FSK deviation can be automated by using simple statistics in the frequency domain after a fast Fourier transform (FFT) of the signal.

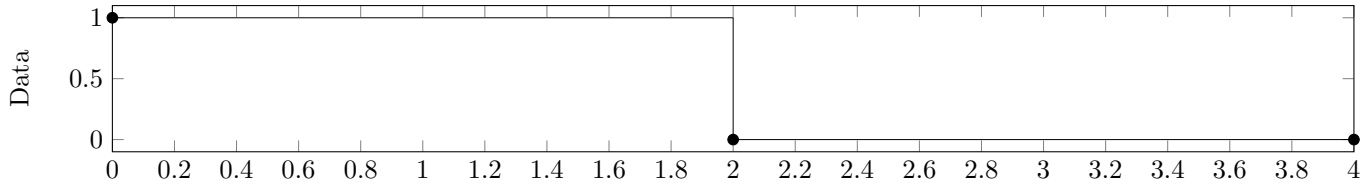


Figure 3.13: Data signal with two bits (10).

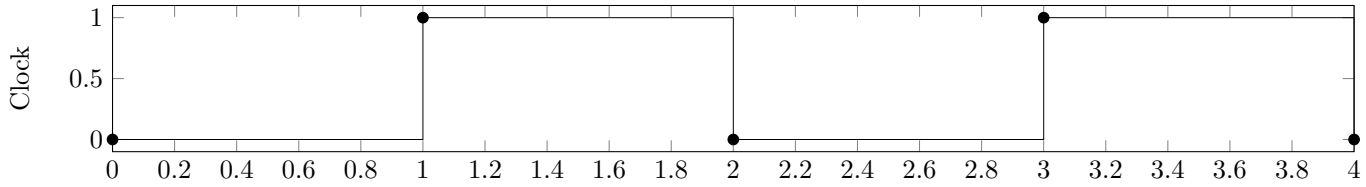


Figure 3.14: Clock signal with two pulses per bit period.

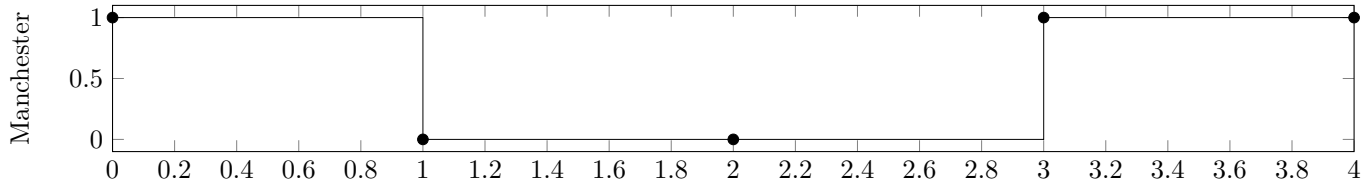


Figure 3.15: Manchester-encoded data signal with two encoded bits (10).

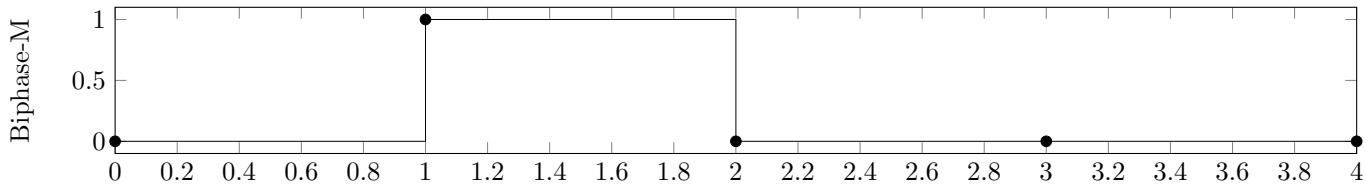


Figure 3.16: Biphase-M-encoded data signal with two encoded bits (10).

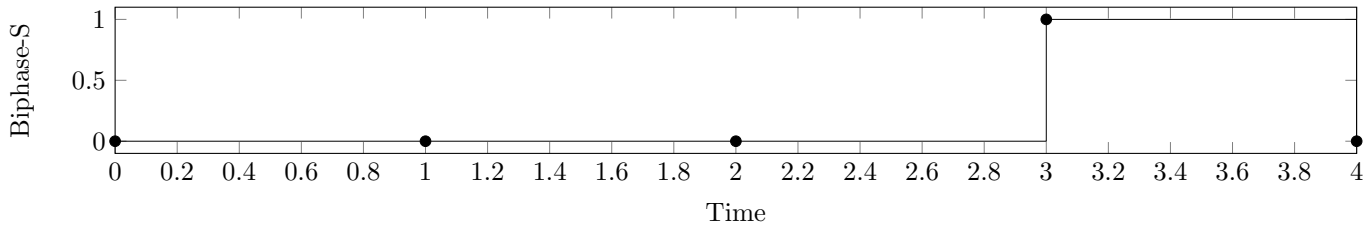


Figure 3.17: Biphase-S-encoded data signal with two encoded bits (10).

Encoded data	Manchester (Figure 3.15)	Biphase-M (Figure 3.16)	Biphase-S (Figure 3.17)
00	Invalid	0	1
11	Invalid	0	1
01	0	1	0
10	1	1	0

Table 3.1: The encoded signals can be decoded using 2-bit lookup tables.

In Table 3.1 we can see why Manchester coding is likely to be a popular choice among manufacturers. It allows for detection of transmission errors; but what is really convenient from an electrical engineering perspective is that

Manchester coding and decoding can be performed by XORing the data signal with the clock signal. The clock of a receiver then merely needs to synchronize with the data stream, which is achieved using a preamble with a known pattern. Because this encoding uses two modulated bits to transmit a single bit of information, we have to differentieate between the modulation rate of symbols (*baudrate*) and the rate of transmitted data (*bitrate*), which is half the baudrate when using Manchester coding.

Figure 3.18 shows an annotated waveform of a demodulated Manchester-encoded signal from a remote with UID `0x0c15af38`. Because the received data contains the device's 4 UID bytes (after the constant 2-byte preamble) we can assume this is a correct reception and decoding.

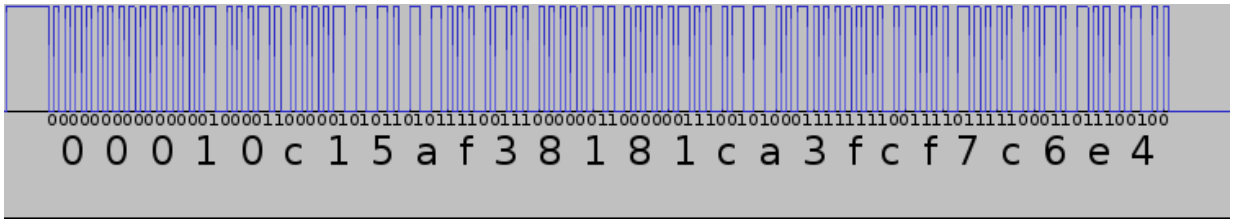


Figure 3.18: The Manchester-coded data in a demodulated transmission from a 19200 baud FSK remote, decoded.

3.6.1 Automatic decoding

We drafted a custom tool which decodes the recorded and demodulated signals output by our GRC sketches. We are looking for wavelengths of one or two *symbol lengths*, and as not all the wavelengths are of exactly equal length should allow for some deviation.

Decoding tables as shown in Table 3.1 allow us to write one routine that decodes all encodings simply by switching out the decoding table, and therefore it is the approach we took to decode the recorded signals programatically. The algorithm detects edges (on or off) in the signal and counts the number of samples between them to produce a bit-period length, which is measured to see if this length contains one, two or no data symbols. Once a bit-period length fits within our window of one or more expected symbol lengths we transcribe as many 'on' or 'off' bits to a buffer that is then decoded by our table-based decoding routine. The data is then represented as a string of bits and a hexadecimal encoding.

Different types of RKE frames are identifiable by virtue of their indicative length. In cases where we know the frame layout, the frame is also decoded and split up into its constituent fields (UID, button, nonce, cryptogram, CRC). An example of the output can be seen in Figure 3.19. More context about the frame contents can be found in Chapter 4.

```
$ ./decode_wav /tmp/am_demod.wav 9600 -m manchester
Mode: manchester
Decoding with 26.666667 samples per symbol
Frame length: 104
0000000000000001000011000001010110101111001110000001100000011100101000111111110011110111111000110111001 00
00010c15af38181ca3fcf7c6e4
0c15af38 | 1 | 207 | 28ff3df1 | e4
```

Figure 3.19: A run of our custom tool to decode .wav files using various line codings and baud-rates.

3.6.2 Automatic baud-rate detection

Before we add automatic baud-rate detection to our tool, we can visually identify the baud-rate of the data we need to decode by using Audacity as shown in Figure 3.20.

We are using a sample rate of 512 kHz, that is 512000 samples per second. We see a wavelength of about 56 samples for a given waveform, which is thus $\frac{56}{512000}$ of a second in length. The frequency of the data is its inverse, $\frac{512000}{56} \approx 9142$ baud. This should be interpreted to likely be a 9600 baud signal.

This method is of course tedious but gives a good illustration of the process we want to automate. We have implemented simple baud-rate detection by counting the frequency of observed bit-period lengths and using the most-often seen one.

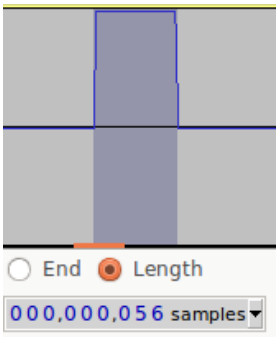


Figure 3.20: With the selection tool in our .wav file editor Audacity we can see the number of samples in a waveform allowing us to identify the baud-rate of the signal.

The most straightforward and lightweight way to implement this seems to simply count the occurrences of the various wavelengths and take the smaller of the two most common symbol lengths. Visual inspection not only tells us the baud-rate, but can also show us whether the correct demodulation was used and if the signal was recorded correctly in the first place. Also automating these tasks requires a more focussed effort than we were willing to invest on this sub-topic, and we therefore accepted this imperfect but mostly correct automatic baud-rate detector as a workable solution. The tool was later used with a set of specific baud-rates to try, its output then used to identify a correct decoding and establish the baud-rate as such (see Chapter 7).

Chapter 4

Case studies

Opel and Renault keys are high on the list of interesting research targets because of their market share. The task at hand is to create software emulations of these devices through reverse engineering up to a full understanding of the RKE protocols used.

4.1 Opel Meriva B (sanity test)

Before we begin analyzing our targets we test our toolset on a type of RKE system that has been reverse-engineered as part of earlier work [21, 22]. The framing used is shown in Figure 4.1. The Opel Meriva B remote is known to have the same behaviour and can be used to verify their results.

Reading out the device with a HITAG2 RFID transponder reader gives us the UID, `f47ef76a`. From this UID we can assume the device to be configured as a PCF7941 device (ID46 Philips CRYPTO2), based on Timo Kasper's findings as summarized in Table 2.2 [23].

With some trial and error we soon determined that the device uses Amplitude Shift Keying (ASK) to transmit 16 identical 104-bit frames at 4800 bits per second. Chronologically, this is where the bulk our research into software defined radio (Chapter 3) took place.

The nonce used to initialize the state with which the cryptogram is computed has the form: $counter_{27..0} \| button_{3..0}$. After transmitting an initial synchronization pattern, while the button remains pressed a number of repetitions of the authentication frame are transmitted. When the button returns to the non-pressed state, or the device-specific maximum amount of repeated messages has been transmitted, a final message is transmitted in which the button bits are cleared (zero).

Recreating the existing research proved a good exercise, as we soon found other remotes that employed the exact same protocol at different data rates or with a different modulation scheme.

1 time	Header 64(0xff) (512 bits)						
repeated	Header 0x0001 (16 bits)	UID (32 bits)	Button (4 bits)	Counter (10 bits)	Cryptogram (32 bits)	0x2 (2 bits)	CRC (8 bits)
repeated	Header 0x0001 (16 bits)	Counter (10 bits)					
1 time	Header 0x0001 (16 bits)	UID (32 bits)	0x0 (4 bits)	Counter (10 bits)	Cryptogram (32 bits)	0x2 (2 bits)	CRC (8 bits)

Figure 4.1: The 104-bit *PCF7946* HITAG2 keyless entry frame as recovered by Garcia et al. [21] and further explored by Benadjila et al. [22].

4.2 Renault Megane III

The Renault Megane key is of significant commercial interest because legitimate after-market keys can only be bought directly from the manufacturer using their pre-programmed secret keys specific to the car. It is the type of 'key card' that is to be inserted into a slot in the dashboard in order to start the car at the press of a button. It hides a mechanical key inside which is normally only required in emergency situations.

Freeing the printed circuit board in Figure 4.2 from its plastic enclosure requires a fair bit of violence: so much that our first retrieved PCB sustained some harm during the extraction. Because of the cost of these devices it was worthwhile to repair the broken connection using a wire trace and bandage the board with electrical tape. We can see a chip on the PCB marked PCF7947AT with the NXP logo and another marked 5100 B3.

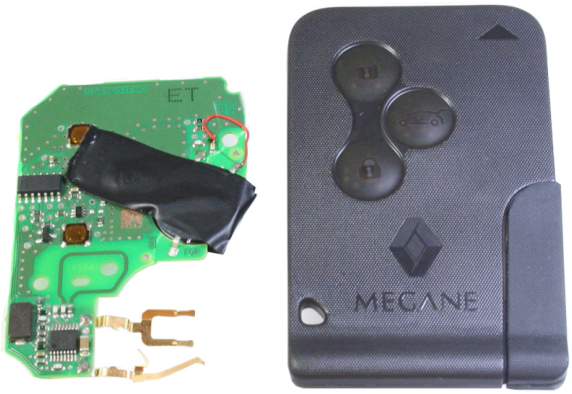


Figure 4.2: The Megane III key under investigation.

4.2.1 Determining modulation and data rate by oscilloscope

When investigating the Renault remote, we recognized one of the chips soldered on the PCB to be a TDK 5100 manufactured by Infineon: a commonly-used component for UHF transmission that is well understood and for which documentation is publicly available.

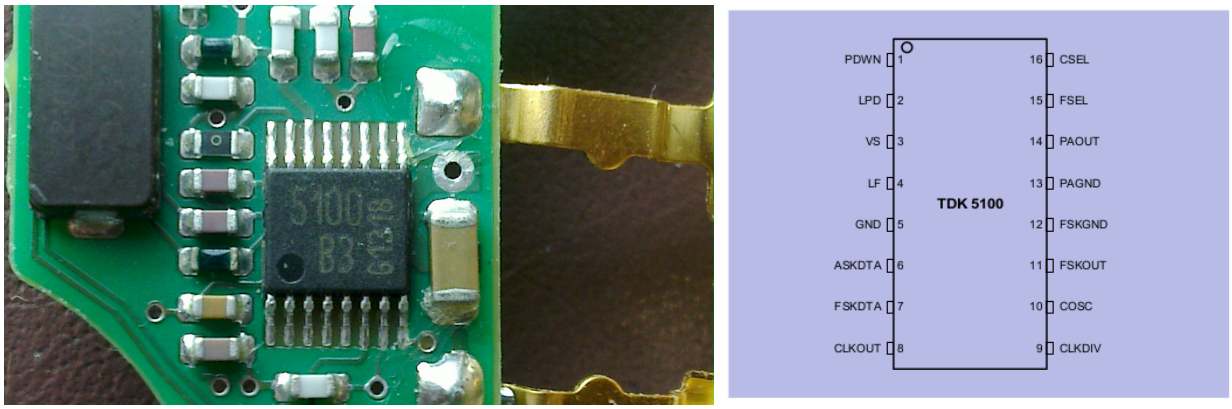


Figure 4.3: A picture of the TDK chip responsible for UHF transmission on the Renault Megane III remote (left) along with its pinout diagram (right).

The chip has two different data input pins which lead to modulation circuits for ASK or FSK modulation. While probing the exposed data input pins on the chip, we used an oscilloscope to measure the digital signals sent in by the microcontroller when we press a button. We learned that the modulation used by this device is FSK, and the data rate with which this chip is transmitting appears to be 9600 bits per second.

4.2.2 Recording and decoding data

Now that we know FM is the the modulation used we will need to test a new part of our signal processing chain, the FM demodulator. Having determined the bit-rate we can be confident in the knowledge that the only variable that remains to be determined is the frequency deviation used. While frequency analysis using plots could help here, it came down to trying a few values until we got a successful decoding (where we see multiple transmissions with the same frame lengths). After adjusting our demodulation chain for FM demodulation by setting the remaining variable for FSK deviation, we are able to verify that the Renault Megane III keys use the known *PCF7946* framing shown in Figure 4.1.

4.3 Opel Astra H

Another remote that is of commercial interest is that of the Opel Astra H. While we have successfully decoded Opel remotes using the PCF7946 HITAG2 framing for the previously known keyless entry protocol with 96 or 104 bits per frame, this device sends a quite different series of frames. This remote has UID `0xe03a236e` and should therefore behave like a PCF7941 (ID46 Philips CRYPTO2) over the transponder interface.

4.3.1 Recording, demodulation and decoding

We use our crafted toolset to record and demodulate the signal. After multi-demodulation we can examine the generated `.wav` files in an audio editor and quickly establish through visual means which of the demodulation methods was successful. In the case of the Astra H remote, there is data in the AM demodulated .wav file.

4.3.2 Frame slicing

The goal is to isolate all fields of data within the decoded bitstrings so that we can interpret the transmissions. To recover the composition of the received frames, we attempt to identify known values in our decoded transmissions. We expect the transmission to contain the UID, button, counter and a HITAG2 cryptogram of 32 bits, just like the previous protocol did.

Transponder readout

Figure 4.4: We use the commercial MiraClone tool to read the remote secret key and edit the counter pages in the transponder EEPROM.

Because we expect to have received a 32-bit HITAG2 cryptogram, we decide to examine the cipher parameters that were used for its generation. The PCF7941 has an extended EEPROM memory with four extra 32-bit pages which contain the remote secret key. By making use of earlier work we were able to recover the secret key from the transponder by performing an offline attack after recording a transmission from the immobilizer system of a car to the key. Once the transponder key was thereby recovered we could read out the 48-bit remote secret key as used for the generation of UHF cryptograms by reading the transponder using a commercial HITAG2 reader as shown in Figure 4.4.

While experimenting with the transponder we noticed the User page 0 and User page 2 are used to keep track of a 32-bit counter value that increments when we press a button.

Received UHF data

We have received multiple frames of various lengths, some of which are repeated, as shown in Figure 4.5. The transmission starts with a frame preceded by a very long frame header (used for synchronization), the next frame contains the same last 128 bits from this transmission with a shorter header, then we see a repeated message of 34 bits and finally a different message of 128 bits. The first thing we notice about the two consecutive 128-bit messages is that they both contain the 32-bit UID directly after what appears to be the frame header. It seems that each of the frames starts with a number of `0xe0` bytes followed by a `0x69` to denote the start of a frame. Next to the UID field we can identify two 70-bit fields which appear to contain the cryptogram.

```
Frame length: 1368
e0e0e0e ... 0e0e0e0e069e03a236e081c41324500b5
Frame length: 127
e0e0e0e069e03a236e081c41324500b5
Frame length: 34
38381a41c
...
Frame length: 34
38381a41c
Frame length: 128
e0e0e0e069e03a236e001c6cd6c66d9a
```

Figure 4.5: Decoded transmissions received from the Opel Astra H remote with (highlighted) UID `0xe03a236e`.

Nonce finder

Now that we know the UID and remote secret key, there is only one value that remains to be determined that should have generated the expected cryptogram: the 32-bit nonce. From earlier work we know the nonce to likely consist of a 28-bit counter and a 4-bit field used to store the pressed button. By exhaustively searching for all 32-bit nonces we can recover the precise one that led to the observed keystream, and the value of these bits and their layout in the nonce is the final required ingredient to create our own cryptograms.

We have recorded an RKE transmission, which yielded two 70-bit values likely to contain cryptograms. With the known UID and key we can try all possible nonces and generate a 32-bit keystream, then look for this keystream in the received transmissions at any offset. Once we find two nonces which both result in the correct keystream and this keystream is also found at the same offset in both the received transmissions, we can conclude we have found the correct nonces.

We have drafted a tool that iterates through all possible 32-bit nonce values given a UID and key, uses these parameters to generate a 32-bit sample of keystream and checks which of the resulting keystream values occur within a string of bytes. As we can see from the output in Figure 4.6, the cryptogram is constructed as $button_{3..0} \| counter_{27..0}$ - these fields were simply switched! Again like in the PCF7946 framing, we can see a 'final' frame that is constructed in the same way as the other frames, but where the button bits are set to zero.

```
$ ./find_nonce a82fbfae3fe2 e03a236e 081c41324500b5
Enumerating 32-bit nonces for key a82fbfae3fe2 and uid e03a236e
to identify cryptograms in transmission 081c41324500b5
Initializing HITAG2 tables...
Allocated keystream tables in 1024 kB
Generating table with 32-bit sliding window
Searching for match with 64 threads
Nonce 2000001c created keystream 41324500 at index 16
...

$ ./find_nonce a82fbfae3fe2 e03a236e 001c6cd6c66d9a
Enumerating 32-bit nonces for key a82fbfae3fe2 and uid e03a236e
to identify cryptograms in transmission 001c6cd6c66d9a
Initializing HITAG2 tables...
Allocated keystream tables in 1024 kB
Generating table with 32-bit sliding window
Searching for match with 64 threads
Nonce 0000001c created keystream 6cd6c66d at index 16
...
```

Figure 4.6: Brute-forcing nonces from 2 RKE transmissions to recover their construction from button and counter bits and locating the cryptogram in the frame.

The design of the tool is simple but effective. We load the given bitstring into a table of 32-bit values using a 32-bit moving window. (That is, we first take the $[0, 31]$ bits, then the $[1, 32]$ bits and so on as our table values until the bitstring is exhausted.) We also note the original offset within the bitstring at which these values originally occurred. To increase the search speed for the many table-lookups that are about to follow we sort the table once at this point. To ensure the even more prevalent cryptographic operations that will follow are also performed with minimal latency we precompute tables of `f20` (the HITAG2 keystream generation function) output indexed by the input. Then we evenly divide the domain of 2^{32} into as many threads as are available on the target machine. Each thread then walks its respective subdomain of nonces and generates 32 bits of keystream for each of them after which a table search for this 32-bit value is performed. For each of the resulting matches, the relevant data is printed to the standard output. Exhausting this 32-bit search space takes about 20 minutes using 4 threads on my 2016 Intel i7 laptop.

CRC recovery

We suspect a CRC at the end of each message as in the other protocol. We can use the free software tool CRC RevEng [30] to automatically recover the structure of the used CRC by feeding it some decoded transmissions as an example (see Figure 4.7).

```
$ ./reveng -w 8 -s e03a236e08025a32c5f0c0 e03a236e08032853bc643f \
e03a236e0804e732faba0e e03a236e09e643a1d60e42 e03a236e05e6e0c7caee77 e03a236e09e74701c509f3 \
e03a236e05e7a4c57e4c26 e03a236e09e8c235fcadd0 e03a236e05e87d91cfc29b
width=8  poly=0x01  init=0x00  refin=false  refout=false  xorout=0x00  check=0x31  name=(none)
width=8  poly=0x01  init=0x00  refin=true  refout=true  xorout=0x00  check=0x31  name=(none)
```

Figure 4.7: Using the tool RevEng to recover the structure of the CRC algorithm used. We indicate the width of checksum we expect and give a few examples to aid the search.

Frame reconstruction

Now that we know where the offsets for the UID, counter, button, cryptogram and CRC are, as well as the construction of the nonce, filling in the rest of the pieces becomes easy (see Figure 4.8).

1 time	Header 159($0xe0$)$\|\|0x69$ (1280 bits)	UID (32 bits)	Button (6 bits)	Counter (10 bits)	Cryptogram (32 bits)	CRC (8 bits)
repeated	Header 4($0xe0$)$\|\|0x69$ (40 bits)	UID (32 bits)	Button (6 bits)	Counter (10 bits)	Cryptogram (32 bits)	CRC (8 bits)
repeated	Header 2($0xe0$)$\|\|0x69$ (24 bits)	Counter (10 bits)				
1 time	Header 4($0xe0$)$\|\|0x69$ (40 bits)	UID (32 bits)	0x0 (6 bits)	Counter (10 bits)	Cryptogram (32 bits)	CRC (8 bits)

Figure 4.8: The HITAG2 keyless entry frame used by the Opel Astra H.

4.4 Renault Laguna II

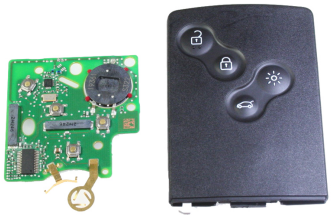
This remote (pictured in Figure 4.9a) is interesting for the same reason as the Renault Megane III remote: they have to be ordered pre-programmed from the manufacturer by special request. Decoding the data is by now straightforward: 9600 bits per second ASK with the default PCF7946 framing as shown in Figure 4.1.

(a) Two Renault Laguna II remotes. The one on the left is genuine, while the right one is a Chinese clone.

4.5 Renault Clio IV

The remote in Figure 4.9b has a PCF7939MA transponder, which is a device we don't know much about and does not seem to behave like a normal HITAG2 transponder. We can see its UID is `0xb7eadc60`, but cannot read further information from the transponder. With its 3 one-dimensional LF antennas this design looks to be intended for Passive Keyless Entry (PKE) usage.

It transmits the following FM-modulated data at 9600 bits per second.



(b) A Renault Laguna II key with PKE feature.

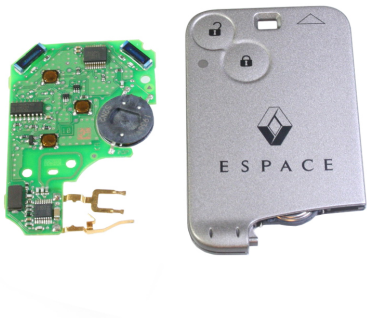
```
Frame length: 424
0000 ... 0000
Frame length: 120
000126b7eadc600804d4f4cc2a8885
Frame length: 120
000126b7eadc600808d4f4cc2a8889
Frame length: 120
000126b7eadc60080cd4f4cc2a888d
Frame length: 424
0000 ... 0000
Frame length: 120
000124b7eadc600810d4f4cc2a8891
Frame length: 120
000124b7eadc600814d4f4cc2a8895
Frame length: 120
000124b7eadc600818d4f4cc2a889b
Frame length: 78
0004b2dfab7180006d53
```

Figure 4.10: Decoded transmission from a Renault Clio IV key with PKE feature with (highlighted) UID `0xb7eadc60`.

Next to the UID field we can identify what appears to be a 10-bit counter field which increases with every transmission, next to what seems like an 8-bit frame counter. It appears as though the final byte is a CRC as only one bit differs for each differing bit in the preceding bytes. This exactly marks out a 32-bit field which is likely to be a cryptogram. As we cannot read out the pre-configured, locked transponder we cannot recover the remote secret key to verify how the nonce is constructed.

4.6 Renault Espace

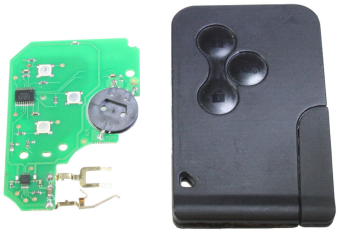
We test a key with the Passive Keyless Entry feature shown in Figure 4.11a, which is otherwise identical to the Megane III remote and employs the *PCF7946* framing using 9600 bits per second FSK modulation.



(a) A Renault Espace key

4.7 Chinese Renault clone

An interesting development in the after market car key industry is the availability of cheap cloned remote keys from China. We have access to one of these clones (shown in Figure 4.11b) and can conclude from our experiments that while it has equivalent UHF transmissions, the signal generated by the device is quite noisy compared to an officially licensed remote: its signal is less sharply tuned to the center frequency and thus leaks energy around it, which is especially apparent when faced with environmental noise.



(b) A Chinese clone of Renault's design with no branding

4.8 Opel Astra J

We can read out the transponder using MiraClone and see that it is recognized as an unknown HITAG2 transponder type with UID `0xc57c5bb5`. These transponders send ASK-modulated frames of 144 bits at 4800 bits per second. We have begun preliminary analysis but cannot provide a complete description yet.

4.9 Overview

The results of our case studies focusing on HITAG2-based RKE systems are shown in Table 4.1. This table shows that while there exists a wide variety of different devices in multiple shapes and sizes, these devices largely share the same behaviour. A notable exception to the list is the Astra H remote framing, which required some additional analysis to identify a new variant of HITAG2 RKE cryptograms we call *v2*. We have shown that a grey-box approach to reverse engineering such variants is sufficient to recover their workings with our custom tools. Additionally, our tools also provide some foothold to investigate newer variants in which the used cipher is still unknown.

Brand	Car	Bit-rate	Modulation	UHF Framing	Tested UID	Transponder
Opel	Meriva B	4800	ASK	HITAG2 PCF7946	`0xf47ef76a`	PCF7941
Opel	Astra H	9600	ASK	HITAG2 v2	`0xe03a236e`, `0x30b5e66d`	PCF7941
Renault	Megane III	9600	FSK	HITAG2 PCF7946	`0x0c15af38`	PCF7941 (PCF7936 mode)
Renault	Laguna II	9600	ASK	HITAG2 PCF7946	`0x492b595a`	PCF7943AT
Renault	Espace	9600	FSK	HITAG2 PCF7946	`0x492b595a`	PCF7943AT
Chinese clone	unknown	9600	FSK	HITAG2 PCF7946	`0x2850c531`	PCF7947AT clone
Opel	Astra J	4800	ASK	unknown 144 bits	`0xc57c5bb5`, `0xd4e534b5`	PCF7939EX
Renault	Clio IV	9600	FSK	unknown 120 bits	`0xb7eadc60`	PCF7939MA

Table 4.1: Overview of (shared) device characteristics uncovered by the case studies in this chapter.

Chapter 5

Software emulation of key fobs on general purpose hardware

As a sub-goal of our research we aim to develop a remote keyless entry solution which accurately emulates one or more of the analysed remotes. Now that we have recovered some of their designs through reverse-engineering we can draft software implementations which can be run on a general-purpose micro-controller.

5.1 Atmel automotive hardware

Through Car Lock Systems we have access to several automotive hardware evaluation kits from micro-chip manufacturer Atmel. We can use the generic ASTK600 development board to interface with the automotive evaluation kits: the ATAK51001-v1 and ATAK51002-v2. These provide radio modules which can emulate the car, providing low-frequency and high-frequency radio communication through daughterboards.

For our project we will focus on the immobilizer key fob development board included in the development kits. For most of these devices some example code is available from the manufacturer website. For our prototype design we chose the ATA5795 development board, for which there is such example code available. It is equipped with UHF and LF modules allowing us to experiment with the UHF feature while leaving room to implement the LF immobilizer chip features later on. It also has a LED and three buttons, which can be used as minimalistic sources of input and output during debugging. The ATA5790 is also compatible with the example codebase provided by the manufacturer, and in addition offers a 3D LF antenna which can be used for the later development of passive keyless entry applications. Unfortunately there is no example code available for this feature.

The key fobs are built around an `avr5` (ATmega) core, which can drive surrounding peripherals at clock speeds of up to 4 MHz. Some of the peripherals available to the core are unique to RKE and immobilizer applications, for instance a modulation and encoding chain. These are connected in such a way that they can operate independently of the core; they merely rely on the core to re-configure them, to send or receive data, and to wait for interrupts from the peripheral in between (see Figure 5.1).

We also have access to a general-purpose programming and debugging interface for AVR micro-controllers, the Atmel Dragon device. The ATA5795 device does not come with a standard JTAG connector, but exposes a six-pin in-system programming (ISP) header using a custom but publicly documented protocol based on the serial peripheral interface (SPI) protocol. These connect the key fobs to the programmer device (Dragon) which is then interfaced with from the host PC over USB. The devices are shown connected together in Figure 5.2.

Our programmer device and the target device are both supported by Atmel's *AVR studio* for Windows, a typical Integrated Development Environment (IDE) with support for many Atmel devices. AVR studio is only available for Windows and the latest version (required for debugging on this device) is rather sluggish even on modern hardware. Rather than reinventing the wheel we sought out to add support for existing open source tools that can take on the role of compiler, programmer and debugger.

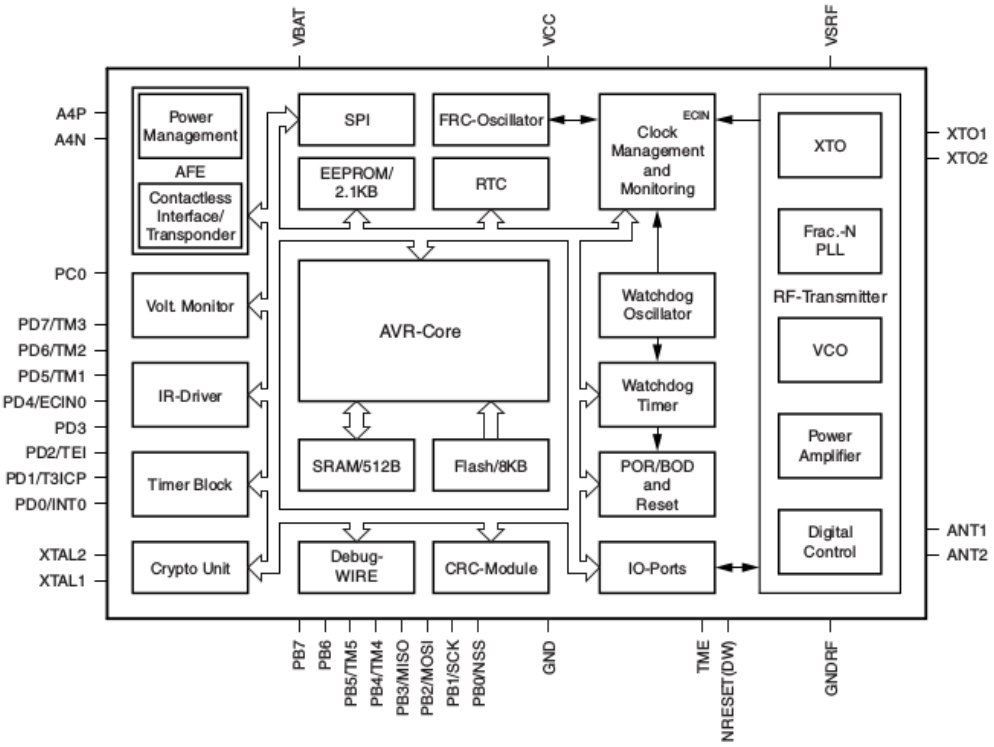


Figure 5.1: The ATA5795 MCU core with surrounding peripherals.

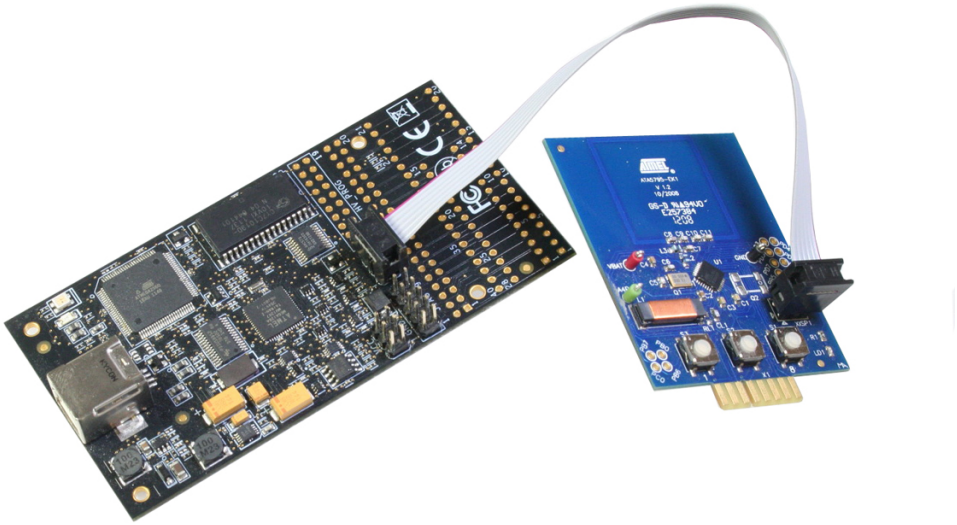


Figure 5.2: The ATA5795 prototype board (right) connected to the AVR Dragon programmer (left) via the ISP connector.

5.1.1 Sourcing and licensing

Note that all of the hardware was purchased on the free market and all of the software and available documentation were sourced from Atmel's website. We do not have the full documentation, but we have software and hardware as well as some support files. We will try to build upon these sources to recover a complete understanding of the device while avoiding any licensing agreements. This means we are not bound by any terms of non-disclosure regarding what we have discovered about these devices up to this point. While we may recover the workings of the transponder subsystem, we will explicitly not cover this as it is outside the scope of our research in this thesis.

5.2 Going GNU

5.2.1 AVR compilers and toolchains

The example code that is available through Atmel is targeting the commercial IAR embedded C compiler, for which we only have a restricted trial license. We can not justify investing in a full license when there is a free and open source alternative available in the form of GCC, the GNU compiler collection. GCC has target support for all of the known AVR cores, and this support is fully realized in a non-GNU runtime library that supports many devices, AVR-libc. This C run-time library provides a rather complete standard library which allows us to write more readable code faster. This can be achieved by making use of the included APIs for features like power management, sleep modes, EEPROM access and interrupt handling among many other things. Together this toolchain is colloquially known as AVR-GCC, which includes the C/C++ compilers, assembler, debugger, linker et cetera next to the standard library.

Atmel studio also uses AVR-GCC as a compiler which is why porting code between them is normally not an issue, but the sample code for the IAR compiler is notably different in a few minor but important ways. Because we are interested in developing our solution targeting only AVR-GCC, we modify the provided source code to adhere to the GNU-style. This mainly involves device-specific features like how to define interrupts, or simply replace some of the port manipulation commands with more expressive code using the GNU AVR-libc features defined for our platform. There is also partial support for these IAR-conventions through the *iar.h* header in AVR-libc, but since we will not be using both compilers and fear its deprecation we decided to go for the consistency of the GNU style.

See Figure 5.3 for the patches required to the example codebase, sourced from the AVR-libc user manual [31]. Patches for style and readability are much more extensive and subjective, but these patches are essential when switching compilers.

IAR notation	GNU notation	Meaning
`__enable_interrupts();`	`sei();`	Enable global interrupt
`__disable_interrupts();`	`cli();`	Disable global interrupt
`#pragma vector = INTERRUPT_name`		
`__interrupt void foo {}`	`ISR(INTERRUPT_name){}`	Define interrupt routine
`__sleep();`	`sleep_cpu();`	Put the device in sleep mode
`__delay_cycles(10);`	`__builtin_avr_delay_cycles(10);`	Delay execution for a number of cycles
`__regvar __no_init`		
`volatile unsigned int foo @ 28`	`register unsigned int foo asm("r28");`	Declare register value

Figure 5.3: Translations to convert from IAR to GNU style notation.

5.2.2 AVR compilation

Covering the inner workings of a compiler is outside the scope of this chapter, but it should be mentioned that these compilers are doing *cross-compilation*. The source code is compiled for a different target than the host machine, namely for 8-bit *avr* devices from a 32 or 64-bit *x86* architecture.

Using the GNU compiler collection (GCC) to build our source code requires us to specify which microcontroller is targeted through the `-mmcu` option to the compiler, assembler and linker. This tells the AVR-libc component how peripheral I/O ports and interrupt vectors are mapped into memory and should be used, and the GCC compiler backend will interpret this to learn the type of core targeted and the configuration of the SRAM.

Compilation will yield an ELF binary (similar to an EXE on windows) which holds the compiled code and data, but this still needs an extra processing step. The ELF file must be split up in Flash and EEPROM files and converted to a hex-encoded format with CRC checksums before they can be uploaded to the AVR device memories separately. Including these checksums allows the target device to ensure that the code that is programmed is what was sent to it through the programmer interface. These `.hex` and `.eep` files can then be uploaded to the ATA5795 device.

After having ported the source code to conform with the GNU style we are able to test the resulting firmware by uploading it using the AVR Studio tool via the AVRDragon device. Neither AVR-GCC nor the IAR compiler provide any features to load the resulting code onto the device yet. For that we need to use either the Atmel Studio IDE or a GNU alternative called AVRDUDE.

5.2.3 Adding device support to AVRDUDE

Now that we have a working build using open source tools, it is time to create a custom configuration file for the AVRDUDE (AVR Downloader/UploaDEr) tool. The support for a new device is added by configuring the device class ID, the size of available storage media (Flash, static RAM and EEPROM memories), and most importantly a few strings of JTAG command bytes that allow reprogramming of these memories. These settings were retrieved from the `.xml` files included with the freely available software on Atmel's website.

5.2.4 Adding device support to AVaRICE

The aforementioned ISP port can also be used by AVR Studio to do on-chip-debugging (OCD) on our target device, allowing us to single-step through the program while analyzing and modifying the state of the machine with full control. That feature is very useful when developing software for a micro-controller because the software can continuously be tested on real hardware, and we are not working 'blindly' on the code.

The normal way of debugging programs in the GNU environment is through the GDB tool, which has the capability to connect to a remote target that provides a software debugging bridge. This debugging bridge is normally created around a UNIX process with *gdbserver*. The same interface is adhered to by AVaRICE while interfacing on the other end with various JTAG USB debuggers for AVR devices, among which our AVR Dragon device.

Before we can actually use that feature it is required to extend AVaRICE to support the ATAK5795. The modifications required are effectively the same as those required for AVRDUDE, albeit slightly more in-depth because the actual software needs modification rather than a mere configuration file.

Once patches are in-place we can start both tools and get a familiar GDB debugging shell and debug as normally.

5.2.5 Emulation using SimulAVR

Using the open source SimulAVR tool we are able to perform simulated benchmarks of our mixed C/assembly code to ensure unit tests pass and to measure cycle-accurate timing of our hand-optimized code, which we make use of in Chapter 6 when measuring our performance of HITAG2 cipher implementations. SimulAVR exposes the same software debugging bridge interface to GDB to allow remote debugging inside the simulator. While not all the I/O ports and certainly not the special radio hardware are simulated, we can use the tool to test and verify parts of our code by instructing it to simulate an included ATMega CPU model, which is similar to our ATA5795's CPU core. This simulator was also used to generate the speed measurements in Chapter 6. We can connect a debugger like GDB to the simulator to examine how the code is simulated on a low level.

The simulator has support for an emulated UART device which can be connected to the standard output. This allows printing status information to the terminal while the program is simulated.

5.2.6 Programming fuses

Before we can make use of the DebugWire (DW) feature we must enable it through the writing of *fuse* bits in the device. This changes the device from the default SPI programming mode to use the DW protocol. The SPI and DW modes are mutually exclusive. Enabling both causes DW mode to take precedence, while disabling both could result in a device that is no longer programmable with either and must have its fuses reset via high-voltage programming (a feature not available on our development board), so it is rather important we do not make mistakes here.

The logic of the fuse bits is inverted, meaning that setting a pin low enables this fuse. The table of fuses enabled in the single fuse byte we have on our device is as listed in Table 5.1.

We decide on the fuse mask `0x37` to divide the clock by 8 to save power, enable debugWire and save EEPROM contents between chip erases. The fuse mask to disable DebugWire is the same but has the corresponding bit disabled while SPI is enabled, and it is thus `0x57`. The device refuses to accept the 'save EEPROM contents between chip erases' fuse while enabling DebugWire, instead reporting its fuse as `0x17`. We accept this as a previously unknown feature or limitation.

Bit	Meaning
7	Divide clock by 8
6	DebugWire enable
5	SPI enable
4	WDT always on
3	Save EEPROM contents during chip erase
2	Reserved (must be set)
1	Crystal oscillator is enabled after reset
0	External clock enable

Table 5.1: The fuse bits available on the ATAK5795.

5.2.7 Interrupt vector table

While comparing our builds to the IAR builds we noticed that the interrupt vector table was not getting mapped correctly in the compiled firmware. This table describes the addresses of interrupt handlers, similar to signal handlers on UNIX, that are called when certain interrupts are triggered. The offsets for interrupt vectors are defined using macros which account for the size of the interrupt vector.

The part specification for C runtime library, AVR-libc, does not account for this pre-defined size and maps the interrupt vectors too far apart. The corresponding header file for our device needs a patch to make use of this feature, which is then compiled into the C runtime (crt0.o) file. Once we have corrected the layout the vector table is identical to the IAR builds and the interrupts we configure are executed as expected.

5.3 RKE software implementation

5.3.1 Power management and sleep modes

The device has a number of peripherals which have individually controllable power supplies. Next to power management there are several levels of 'sleep' during which more or fewer interrupts are enabled. These features are supported by AVR-libc, which allows us to rework all of the power management and sleeping code to be more legible. Both of these features are important to get right to avoid draining the battery of the device. It is important to note that it appears the DebugWire fuse setting overrides some of the power management settings, a fact we uncovered while resolving the then-mysterious reason behind some interrupts not firing.

5.3.2 Handling button presses

While waiting for a button to be pressed, the device should be in deep sleep mode to save power. An I/O pin like a button can be used to trigger an external hardware interrupt which will wake up the CPU after which we can resume control. External hardware interrupts have the highest priority, and can thus wake up the device from the deepest sleep mode.

Determining which button was pressed is done by testing the button I/O port against a mask representing each button as shown in Figure 5.4.

```
#define BUTTON1 4
#define BUTTON2 5
#define BUTTON3 6
if((BUTTON_PIN & (1<<BUTTON1)) == 0 ){
    return 1;
} else if((BUTTON_PIN & (1<<BUTTON2)) == 0 ){
    return 2;
} else if((BUTTON_PIN & (1<<BUTTON3)) == 0 ){
    return 3;
}
```

Figure 5.4: Code to read out buttons consecutively mapped to pins 4-6.

5.3.3 Sending UHF signals

The UHF transmitter can be configured to use a certain baud-rate and modulation by writing a packed array of configuration values to some of the transmitter registers over Synchronous Serial Interface (SSI). After this configuration the data can be directly written to any of 3 (Manchester, Biphase or 'raw') I/O ports in 4-bit chunks, which leads to the pre-configured encoding/modulation/transmission chain.

In the example UHF code we find several routines to split off into a library. We have the initialization routine, which enables power to the required subsystems, configures the I/O ports, sets up the timer to fire interrupts at the configured baud-rate and transfers the radio settings to the radio subsystem via SSI. We adapt the original start-continue- and stop-transmission routines from the example code which can be used from calling code to transmit an array of bytes. The reason we would like to have routines transmit data in this piecewise way is the that the target AVR device only has a small amount of available SRAM which may prevent us from creating long data buffers in memory. The code is reworked to be more legible, optimized for size and refactored for re-use.

5.3.4 LF transponder emulation

While developing and testing the ported UHF functionality we have used the DebugWire interface to assure the code behaved as expected. When incorporating the LF functionality, we noticed the device does not reset in response to a detected LF field while in DW mode. This can be explained by the fact that the DW protocol uses the external reset I/O line - the same one that the LF coil needs to drive in order to reset the chip, allowing detection of the LF field and appropriate response.

Unfortunately this means that the LF features cannot be completely integrated while using DW mode. We were able to confirm the LF functionality works and should allow us to implement emulation of LF transponders.

5.3.5 Making use of EEPROM

The ATAK5795 has an interesting feature which most AVR devices don't have: the EEPROM memory is segmented to make some of the memory pages inaccessible through normal read/write operations. This feature to lock segments of the EEPROM can help protect sensitive data like cryptographic keys by mitigating the risks posed by attacks on the EEPROM control logic inside the chip. Such attacks might use software exploitation or invasive hardware attacks to direct EEPROM operations at the level of the CPU.

We instruct the linker to reserve a section of the mapped binary file to fall within these designated lockable sections as shown in Figure 5.5.

```
LDFLAGS += -j .eep.sec0 --set-section-flags=.eep.sec0="alloc,load" --change-section-lma .eep.sec0=0x00
```

Figure 5.5: Instructing the linker to add an EEPROM section mapping that is addressable by name

We should take care not to use names deriving from 'eeprom' for these sections as the linker will try to override these settings to coerce all the section contents into an unpartitioned EEPROM section. Once the secure EEPROM area section is defined for the linker we can place EEPROM data in it as shown in Figure 5.6.

```
#define EESEC0 __attribute__((section(".eep.sec0")))

EEMEM hitag2_page_t serial             = {.bytes = {0x1a, 0xb5, 0x5f, 0x69}};
EESEC0 hitag2_page_t key_lo            = {.bytes = {0xe9, 0x91, 0x77, 0x11}};
EESEC0 hitag2_page_t key_hi            = {.bytes = {0x00, 0x00, 0xa3, 0x03}};
EESEC0 hitag2_page_t conf_and_password = {.bytes = {0x00, 0x00, 0x00, 0x00}};
EEMEM hitag2_page_t userpage0          = {.bytes = {0x01, 0x02, 0x03, 0x04}};
EEMEM hitag2_page_t userpage1          = {.page =  0x3ff};
EEMEM hitag2_page_t userpage2          = {.bytes = {0x11, 0x12, 0x13, 0x14}};
EEMEM hitag2_page_t userpage3          = {.bytes = {0x21, 0x22, 0x23, 0x24}};

hitag2_page_t *hitag2_config[8] = {
   &serial,
   &key_lo,
   &key_hi,
   &conf_and_password,
   &userpage0,
   &userpage1,
   &userpage2,
   &userpage3,
};
```

Figure 5.6: We can confine variables to the segmented EEPROM sections in C using a macro. The literal values will be reserved and linked into our EEPROM contents file to be flashed onto the device. Our software can make use of logical mappings to these EEPROM pages through an array of page pointers.

5.4 Unlocking cars

5.4.1 8-bit HITAG2 implementation

The existing HITAG2 cipher implementations make use of 64-bit native integer types to represent a 48-bit HITAG2 state. Because we are confined to an 8-bit machine on which such registers can only be emulated at considerable overhead, we chose to implement a version of the HITAG2 cipher in assembly that makes optimal use of the available processing power. We are taking special care to use as little code as possible because we only have 8 kilobytes of flash memory available for code. Complete details for our implementation can be found in Chapter 6.

The assembly code is exposed to the rest of the C code with some assembly 'glue' that translates between the two contexts. We have drafted specific routines to construct nonces from a counter and button in the two different configurations we have uncovered in Chapter 4 and use these to immediately initialize the cipher and compute a cryptogram. This strategy helps to minimize the overhead incurred in the naive approach, where we would use general-purpose HITAG2 functions and construct the nonce externally to the initialization/keystream function. We can then, based on our configuration settings, use either method to construct a nonce and corresponding cryptograms which are transmitted over UHF in their respective protocol framing, modulation and baud-rate.

5.4.2 Constructing RKE authentication frames

Now that we have all the pieces working, we can start sending some messages to a real car. With our emulated HITAG2 transponder memory in EEPROM, we temporarily unlock our key pages and read out the key, counter and UID. Depending on the configuration we send either of the two different types of UHF transmissions we have uncovered so far. The cryptogram is constructed using the appropriate nonce and then transmitted according to the specific framing required, including CRC checksums. These frames are transmitted a number of times over UHF in the applicable baud-rate and modulation. We are happy to report this allows locking and unlocking an actual car, an Opel Astra H, using our ATA5795 prototype board.

Chapter 6

8-bit HITAG2 implementations

We can present speed and size-optimized implementations of the HITAG2 cipher for 8-bit hardware such as the AVR. We have implemented C, Python and AVR assembly versions for study and comparison.

6.1 Representation

We first look at the representation of the 48-bit HITAG2 internal state, 48-bit key and 32-bit UID. We use a representation to store these values in multiple bytes where each byte holds 8 consecutive bits of the total value.

In both sections we will examine the nonlinear filter function which generates a single keystream bit from a given state and the LFSR which permutes the state afterwards.

In HITAG2 the first layer of the filter function is fed with 20 bits of the state that lead to 5 instances of two kinds of 4-bit filter subfunctions whose output is then fed into a third 5-bit filter function. The fastest way of computing the non-linear filter function is done through table lookups, which requires storing tables in memory at the cost of available code size. To avoid this impact on code size we implement these subfunctions using boolean logic in both the speed and size-optimized implementations, which were proposed as a means of optimizing the cipher using bitslicing in earlier work [15]. This has the desirable side-effect of ensuring constant-time operation for these functions [1]. We establish this constant-time property with regards to the cycle time specified for instructions executing on the AVR architecture. Each assembly instruction takes a specified amount of time expressed as the number of CPU clock ticks, and we ensure that each path through a given block of assembly code takes the same amount of cycles to complete, regardless of data input to the cipher steps through its internal state.

We reserve 5 registers (`I4_`i) to function as the inputs and outputs for the first layer and input for the second layer.

6.2 Speed-optimized version

For a fast implementation we directly adress bits in the state using the `sbrs` and `sbrc` instructions. See Figure 6.1 and Figure 6.2 for examples of how this applies to parts of the f20 function and the LFSR function. These blocks each occur several times with minimal variation except for the bits that are addressed.

We see that in both the f20 and LFSR subroutines the reading of individual bits from the 48-bit state requires many similar steps. These constructs could be 'rolled' into loops that should determine which bits are to be read and accumulate them as appropriate.

[1] Execution timing differences dependent on secret information would introduce a timing side channel in the implementation. Precise measurement of the timing would leak information about the secret.

```
#ifdef QUANTIZE_TIMING
  sbrs STATE5, 2
    mov I4_A, ZERO
  sbrs STATE5, 3
    mov I4_B, ZERO
  sbrs STATE5, 5
    mov I4_C, ZERO
  sbrs STATE5, 6
    mov I4_D, ZERO
#else
  rcall clear_i4_regs
#endif
  sbrc STATE5, 2
    mov I4_A, ONE
  sbrc STATE5, 3
    mov I4_B, ONE
  sbrc STATE5, 5
    mov I4_C, ONE
  sbrc STATE5, 6
    mov I4_D, ONE

  rcall f20a
  push TEMP_BIT
```

Figure 6.1: The speed-optimized version uses direct bit-tests to prepare the inputs to subfilter functions. Similar code is repeated 5 times to process the 20 filter input bits into 5 output bits, which are then fed to the second layer subfilter function.

```
  clr OUTPUT_BIT
  ; bits 0, 2, 3, 6
  sbrc STATE5, 0
    eor OUTPUT_BIT, ONE
  sbrc STATE5, 2
    eor OUTPUT_BIT, ONE
  sbrc STATE5, 3
    eor OUTPUT_BIT, ONE
  sbrc STATE5, 6
    eor OUTPUT_BIT, ONE
#ifdef QUANTIZE_TIMING
  sbrs STATE5, 0
    eor OUTPUT_BIT, ZERO
  sbrs STATE5, 2
    eor OUTPUT_BIT, ZERO
  sbrs STATE5, 3
    eor OUTPUT_BIT, ZERO
  sbrs STATE5, 6
    eor OUTPUT_BIT, ZERO
#endif
```

Figure 6.2: The speed-optimized version uses direct bit-tests to compute the LFSR bit. Similar code is repeated 4 times to process the 16 LFSR inputs bits to one output bit.

6.3 Size-optimized version

The illustrated routines are straightforward but contain a lot of redundant code. We should investigate ways to make the code more compact by drafting re-usable routines that perform these blocks in fewer instructions.

We draft a routine to transcribe bits from the bytes in the 48-bit state into temporary registers following an 8-bit mask, which are then used by the subfilter functions. We extract bits from these state bytes by defining masks, iterate these using a shifting operation and transcribe the matches into the `I4_`i registers using the routine shown in Figure 6.3, which is called as shown in Figure 6.4. We can easily see which of the 48 bits in the state fall into which 8-bit value, but we don't always need to read 4 bits from a state byte. How many bits are to be transcribed is defined by how many bits are set in the mask. We therefore keep track of the pointer to our `I4_`i registers independently of the transcription function. By addressing these registers indirectly using the pointer we can write to them without requiring an explicit instruction for each register. This was based on an idea explored in earlier work on implementing cryptography for AVRs [32]. This optimization strategy allows the transcription procedure to increment the pointer when transcribing bits, and allows the calling code to reset it as needed.

This can be made more efficient by not resetting the pointer to the `I4_`i registers in between calls to consecutive subfilter function calls that operate on the same state byte. We can re-use the already partially rotated state bytes if we need more bits from the same byte. So, as an optimization, we keep partially processed state bytes in their rotated positions in between calls to the bit transcribing routine. Our masks (shown in Table 6.1) are tweaked to support this by adding a bitwise offset of the number of previous rotations of the state byte.

Filter	f20a	f20b	f20b	f20b	f20a
Bits	46, 44, 43, 34	33, 31, 29, 28	26, 23, 21, 17	15, 14, 12, 8	6, 5, 3, 2
State bytes	0 &1	1 & 2	2 & 3	4	5
Masks	0x58 & 0x4	0x2 & 0x58	0x4 & 0xa2	0xd1	0x6c
Optimized masks	0x58 & 0x1	0x2 & 0xb	0x4 & 0xa2	0xd1	0x6c

Table 6.1: The HITAG2 cipher state's inputs to 5 filters using standard 8-bit representation.

```
transcribe_input_bits:
  transcribe_bit_loop:
    sbrs TEST_BITS, 0
      rjmp transcribe_nobit  ; speed optimization
    sbrc DATA_BITS, 0
      st Y+, ONE
    sbrs DATA_BITS, 0        ; timing quantization
      st Y+, ZERO
    transcribe_nobit:
      lsr DATA_BITS
      lsr TEST_BITS
    brne transcribe_bit_loop ; break when TEST_BITS becomes zero
  ret
```

Figure 6.3: A routine to transcribe bits from state bytes to memory following a mask.

```
  ; bits 2, 3, 5, 6
  clr YL
  ldi TEST_BITS, 0x6c
  mov DATA_BITS, STATE5
  rcall transcribe_input_bits

  rcall f20a
  push TEMP_BIT
```

Figure 6.4: An example of how the transcription procedure is used to prepare the inputs for subfilter f20a.

The next step is to apply the LFSR step to the state register, where 16 of the state bits are XORed together to produce the next bit which is shifted in.

We employ a similar construction using masks to define which bits of the states to examine, and rather than transcribing them we simply keep updating the result value with XOR operations. In this procedure (shown in Figure 6.5) we always increment the pointer by one byte so we need only to set the masks from Table 6.2 between consecutive calls (as shown in Figure 6.6).

```
lfsr_xor:
  ld DATA_BITS, Y+
  lfsr_bit_loop:
    sbrs TEST_BITS, 0
      rjmp lfsr_nobit
    sbrc DATA_BITS, 0
      eor OUTPUT_BIT, ONE
    sbrs DATA_BITS, 0  ; timing quantization
      eor OUTPUT_BIT, ZERO
    lfsr_nobit:
      lsr DATA_BITS
      lsr TEST_BITS
    brne lfsr_bit_loop ; break when TEST_BITS becomes zero
  ret
```

Figure 6.5: A routine to XOR bits from state bytes together following a mask.

```
clock_lfsr:
    clr OUTPUT_BIT
    ldi TEST_BITS, 0x4d
    rcall lfsr_xor
```

Figure 6.6: An example of how to use our XOR routine to XOR bits of the state together.

State byte	0	1	2	3	4	5
LFSR Bits	47, 46, 43, 42, 41		30, 26	23, 22, 16	8	7, 6, 3, 2, 0
Mask	0xce		0x44	0xc11	0x1	0xcd

Table 6.2: The HITAG2 cipher state's inputs to the LFSR using standard 8-bit representation.

This covers the core techniques used in our design of the 8-bit HITAG2 cipher for AVR to make our initial implementation smaller. The drawback in speed was a concern, so we investigated ways to optimize the implementation without adding to the code size.

6.4 Bitwise reversal

We noticed the HITAG2 cipher initialization routine (summarized in Subsection 2.5.6) could benefit from a bitwise reversed representation of the whole cipher state to achieve a faster initialization and decided to pursue it. That benefit can occur because the existing versions initialize the cipher using a bitwise traversal that places the the highest key bits in the lowest key register in bitwise reversed order. Thus we implement the cipher using the same constructs but employ different masks as shown in Table 6.3 and Table 6.4 and optimize the initialization routine. Hereby we get a significant speedup in the initialization routine at no cost to the keystream generation procedure.

Note that it is possible to reverse the order in which masks are read (and exhausted by shifting) in order to achieve faster processing here. Keeping the original right-to-left mask reading routine allowed for bigger jumps in our use of the optimized masks that re-use rotation ($2 + 3$ in the original representation bits versus $4 + 6$ in the reversed representation).

6.5 Results

We can present some numbers for the different implementations we have drafted to compare the size and cycle count (see Table 6.5).

Filter	f20a	f20b	f20b	f20b	f20a
Bits	45, 44, 42, 41	39, 35, 33, 32	30, 26, 24, 21	19, 18, 16, 14	13, 4, 3, 1
State bytes	0	1	2 & 3	3 & 4	4 & 5
Masks	0x36	0x8b	0x45 & 0x20	0xd & 0x40	0x20 & 0x1a
Optimized masks	0x36	0x8b	0x45 & 0x2	0xd & 0x1	0x20 & 0x1a

Table 6.3: The HITAG2 cipher state's inputs to 5 filters using bitwise reversed 8-bit representation.

State byte	0	1	2	3	4	5
LFSR Bits	47, 45, 44, 41, 40	39	31, 25, 24	21, 17		6, 5, 4, 1, 0
Mask	0xb3	0x80	0x83	0x22		0x73

Table 6.4: The HITAG2 cipher state's inputs to the LFSR using bitwise reversed 8-bit representation.

Reflecting on this table, we note that the bitwise reversal of representation only affects the cycle count of the cipher initialization phase. Also the bitwise reversal does not affect the size of the code. The speed-optimized version can benefit from the same speedup in initialization as the size-optimized version has, which improves both the speed and size of the implementation.

It is noteworthy that the constant-time versions are faster at initialization (because we use fast bitwise instructions rather than a procedure call to reset the f20 inputs) but are slower to generate a byte of keystream (because we require extra XOR operations in the LFSR). The speed-optimized version still leaves room for improvement, but a larger code size starts to become impractical for our target device. We could investigate techniques to pre-compute parts of the algorithm such as the subfilters and perform fast lookups in 4-bit tables. The hotspots in the algorithm are certainly the f20 filter function and the LFSR procedure. It feels like we are pushing the limits for the size-optimized version while also remaining time-constant.

To provide an identical interface between different implementations, we make use of a C wrapper for all assembly functions (see Table 6.6).

This wrapper also includes routines to compute the two known variants of 32-bit cryptograms. which are optimized to not spill any of the cipher state registers to SRAM between calls to generate a byte of the keystream and therefore have a low overhead. When emulating an RKE device there is no need to have wrappers for the primitives, we only need a wrapper for the cryptogram function which takes the key, UID, button and counter as inputs and produces a 32-bit cryptogram. The total size of the final binary is therefore determined by how the primitives are composed and how much of the library is used. For our implementation we are only concerned with constructing 32-bit cryptograms but don't need a general purpose HITAG2 library. We only need the small 'cryptogram' wrappers that call out to the assembly functions.

It is worth noting that the feature to operate in constant time only contributes very marginally to the cycle count. The optimization to bitwise reverse the representation provides only a fractional speedup in the grand total. We are left with some options, but choose to include the fastest version in our AVR prototype as it decreases power usage and thus extends battery life.

Version	Size (bytes)	Cycles (initialization)	Cycles (keystream byte)
C	1476	30936	6443
ASM (speed-optimized)	380	8649	2175
ASM (speed-optimized, time-constant)	504	8169	2311
ASM (speed-optimized, reversed)	372	7845	2175
ASM (speed-optimized, reversed, time-constant)	496	7365	2311
ASM (size-optimized, time-constant)	334	19497	7191
ASM (size-optimized, reversed, time-constant)	324	18245	7071

Table 6.5: Comparison of multiple implementations of the HITAG2 cipher for the AVR architecture without wrappers.

Function	Size (bytes)	Cycles
Load state	14	16
Save state	14	16
Init	92	223
Byte	42	175
Cryptogram 1	184	473
Cryptogram 2	142	226

Table 6.6: Overhead of assembly wrapping functions.

Version	Cycles (PCF7946)	Cycles (v2)
ASM (speed-optimized)	17843	17596
ASM (speed-optimized, time-constant)	17907	17660
ASM (speed-optimized, reversed)	17039	16792
ASM (speed-optimized, reversed, time-constant)	17103	16856
ASM (size-optimized, time-constant)	48755	48508
ASM (size-optimized, reversed, time-constant)	47023	46776

Table 6.7: Overview of the various sources as used in two different cryptogram functions.

Chapter 7

Bulk analysis

Using the tools we have developed we were able to do some bulk analysis of UHF keyless entry signals.

7.1 Automated data collection

After getting some familiarity with the signals, we went on to develop an automated data collection system based on the BladeRF device to be used in the CLS warehouse. After some experimentation with the hardware we became confident that a sampling rate of 10 times the maximum expected bit-rate (which we established as twice the highest bit-rate we've seen used in our test devices, $2 \cdot 9600 = 19200$) should allow us to reconstruct the data signal from any recorded key fob for analysis.

The application consists of a Python script which was adapted from a GNU Radio Companion sketch, itself wrapped in a bash script that allows the user to signal when starting and stopping the recording process. Each recorded signal can be cataloged by its product code, allowing us to build up a preliminary database of radio transmissions for all remote controls in the CLS inventory.

```
Voer productcode in en druk op <enter>.
12345TEST
Wacht (ongeveer) 5 seconden tot je een ruis hoort.
Druk drie keer dezelfde knop van de handzender in
Druk op <enter> om te stoppen en deze opname te bewaren.

Opgeslagen in 12345TEST.wav

**************************************************

Voer productcode in en druk op <enter>.
```

Figure 7.1: A transcript of the custom software to record UHF signals from car key fobs.

7.2 Fixing .wav file headers

The recordings we made with the HackRF were partially corrupted when the recording script was interrupted with a SIGTERM signal from our text-based user interface. The corruption occured in the `.wav` file header, in two fields which relate to the number of samples (WAV frames) within the file. These values can be computed from the filesize, and are the two fields ChunkSize and SubChunk2Size (see Table 7.1).

Byte index	Field	Fix
0	ChunkID	
4	ChunkSize	filesize − 8
8	Format	
12	SubChunk1ID	
16	SubChunk1Size	
20	AudioFormat	
22	NumChannels	
24	SampleRate	
28	ByteRate	
32	BlockAlign	
34	BitsPerSample	
36	SubChunk2ID	
40	SubChunk2Size	filesize − 44
44	data	

Table 7.1: .WAV file header with necessary fixes.

7.3 Semi-automated categorization of tested devices

After gathering this data in bulk and fixing the headers, we are tasked with sorting out several hundred recordings. In our preliminary analysis over 500 remote controls were recoded in which some non-noise RF signal can be identified. We have used the parallel AM/FM demodulator to create two different `.wav` files for each recording which are then fed to our Manchester-decoding script using various bit-rates.

From here on we used manual verification on all the demodulated signals to verify the recording was valid and the demodulation was successful. We use manual verification to determine the correct bit-rate and categorize the various identified framings. All this information is stored in a custom database, categorized by product code. As each recorded wave file is named with the product code as used in-house, all the information later analysis on the data can bring is linkable to the product inventory database used at CLS.

Recorded remotes	545
AM remotes	205
FM remotes	276
Unknown modulation / failed recording	64
Recovered bit-rate & frame length	393
Unknown bit-rate & frame length	152
Unique types of framings identified by frame lengths	36

Table 7.2: Categorized results from bulk analysis.

Chapter 8

Security assessment

The devices we have investigated have the common characteristic that they employ the HITAG2 cipher to construct a cryptographic proof with which to authenticate to the car. In this section we will consider the protocol security of the three protocols we have uncoverd throughout our research, and provide a generalized formal security proof.

8.1 Protocols under investigation

The HITAG2 *PCF7946* protocol shown in Figure 8.1 was explained in earlier research [21, 22], which we have reproduced in our experimentation with the Renault and Opel remotes in Chapter 4. Then, during experimentation with the Opel Astra H remote we uncovered another variant, which we will call the *v2* protocol (shown in Figure 8.2). Although it's not fully understood yet, from our analysis of the unknown protocol (shown in Figure 8.3) it appears to share the same information as the others do, albeit with different framing.

The validation step performed by the car implies checking the received information against the expected information, and as a result locking or unlocking the doors or trunk of the car.

$$\begin{aligned} Key \rightarrow Car &: (ID, Btn, Ctr, \text{HITAG2}(Key, ID, Ctr \| Btn)) \\ Car &: \text{validation} \end{aligned}$$

Figure 8.1: HITAG2 one-way authentication protocol variant one (PCF7946).

$$\begin{aligned} Key \rightarrow Car &: (ID, Btn, Ctr, \text{HITAG2}(Key, ID, Btn \| Ctr)) \\ Key \rightarrow Car &: (ID, Btn, Ctr, \text{HITAG2}(Key, ID, Ctr)) \\ Car &: \text{validation} \end{aligned}$$

Figure 8.2: HITAG2 one-way authentication protocol variant two (Opel Astra H). The counter Ctr is the same in the initial frame as in the final frame.

$$\begin{aligned} Key \rightarrow Car &: (ID, Btn, Ctr, \text{Unknown}(Key, ID, Btn, Ctr)) \\ Car &: \text{validation} \end{aligned}$$

Figure 8.3: Unknown one-way authentication protocol (Renault Clio IV).

```
(* ProVerif code for keyless entry protocol:

   key -> car: key_uid, counter, button, {0, key_uid, counter, button}_psk

*)

(* HITAG2 keystream *)
fun hitag2_ks/3.

(* The remote's key *)
private free psk.

(* Our secrecy queries *)
(* The attacker shouldn't learn the symmetric key *)
query attacker : psk.
(* The attacker should learn this dummy flag as a sanity check *)
query attacker : key_authenticated.

(* Avoid replay attacks *)
query evinj:end_key_auth(k, co, b, cr) ==> evinj:begin_key_auth(k, co, b, cr).

(* The attacker should not be able to distinguish the same message with the same key *)
noninterf psk.

(* This secret should not be vulnerable to offline attacks *)
weaksecret psk.

(* Key, the initiator *)
let key = new counter;
          event begin_key_auth(key_uid, counter, button, hitag2_ks(psk, key_uid, (counter, button)));
          out(net, (key_uid, counter, button, hitag2_ks(psk, key_uid, (counter, button))));
          0.

(* Car, the responder *)
let car = in(net, (=key_uid, =counter, button, cryptogram));
          let =cryptogram = hitag2_ks(psk, key_uid, (counter, button)) in
            event end_key_auth(key_uid, counter, button, cryptogram);
            out(net, key_authenticated);
            0.

process
  !key | !car
```

Figure 8.4: ProVerif proof for the HITAG2 RKE protocols.

8.2 Formal security proof

We have implemented a generalized version of these protocols in the ProVerif modeling language for the formal verification of security protocols [33]. This tool can verify security properties of authentication protocols such as confidentiality and indistinguishability of the messages, insusceptibility of keys to off-line attacks, and injective agreement on authentication decisions. A failure to prove such a property leads to a contradiction which can be used to provide a counterexample. Such a counterexample is effectively a symbolic trace of protocol-conform transmissions that lead to a violation of the security property.

As all the observed authentication protocols are essentially the same, working with the same public and private information while using the cipher, we have decided to abstract them into a single model where the same information is shared and checked. We can use the ProVerif language to construct a formal definition of the security protocol as shown in Figure 8.4, which can then be checked for security properties by the ProVerif tool.

In this formal model we choose to model the cipher abstractly, and only model its use as a keyed MAC function. The model assumes no vulnerabilities exist at all in the MAC function: it is modeled as a perfectly secure building block. Therefore these results are transferable to any similarly designed protocols as it is precisely those protocol flaws, not flaws in their building blocks, that ProVerif exposes. The tool will give us some results to our queries, which are summarized in Table 8.1.

`RESULT evinj:end_key_auth(k_7,co_8,b_9,cr_10) ==> evinj:begin_key_auth(k_7,co_8,b_9,cr_10) is true.` Because the counter is always changing, authentications are injective.
`RESULT not attacker:key_authenticated[] is false.` The key is authenticated to the car.
`RESULT not attacker:psk[] is true.` The attacker cannot learn the cipher key directly.
`RESULT Non-interference psk cannot be proved (bad derivable).` But the key is always the same.
`RESULT Weak secret psk is false.` The wording of how ProVerif presents the result is unfortunate, but it shows that the secret key is only weakly protected in this protocol - it is not resistant to offline attacks.

Table 8.1: Annotated ProVerif results for the HITAG2 RKE protocols.

8.3 Attacks

8.3.1 Theoretical

Theoretical attacks against Passive and Remote Keyles Entry systems are categorized as follows by Alrabady and Mahmud [34]: The *scan attack* consists of the attacker sending messages containing random cryptographic proofs until a successful authentication is bruteforced, and is not very efficient with long cryptograms like in HITAG2. A *replay attack* involves the attacker recording authentic messages with valid cryptographic proofs and replaying them at a later time to gain authentication, which normally only works against fixed-code protocols. In a *relay attack* two attackers work together to relay the messages sent by an authentic remote to a car through their own channel, which works against many security protocols unless they take specific (and costly) countermeasures. For a *forward prediction attack* the attacker observes a few cryptographic proofs and tries to leverage this information to predict the next proof. Finally, a *dictionary attack* can be used against a challenge-response protocol, where the attacker constructs a codebook of valid responses for challenges from observed messages.

8.3.2 Practical offline / forward prediction attack against HITAG2 keyless entry systems

As we can see from the ProVerif results there is indeed the possibility of an *offline* attack, which enables the *forward prediction attack*. To mount a practical forward prediction attack against a given HITAG2 remote, the attacker first captures two or more consecutive messages from it. These messages contain the UID and the partial information of two nonces with two associated 32-bit samples of keystream (cryptograms). Because the internal state of the HITAG2 cipher is determined by 48 bits, having only one 32-bit cryptogram is not enough to identify the one correct internal state.

With two or more nonce/cryptogram pairs the attacker has enough information to recover the state through exhaustive search or more involved cryptanalytic attacks [21, 1]. But because 18 bits of the nonces used to construct the cryptograms are never transmitted over the air, the attacker lacks information to recover the complete key used to form this internal state.

However, it's still possible to exploit HITAG2's vulnerabilities in a purely passive RKE scenario, wherein 18 bits of the nonces used to generate the cryptograms remain unknown. The idea was formalized by Benadjila et al. [22]. As in the other scenario, given two cryptograms from the same key the internal state of the cipher can be determined. At this point the highest 16 bits of the key can be directly read out of the state, and we can recover the lowest 14 bits of the key by rolling back the initialisation function using the observed lower bits of the nonce. The 18 unknown bits of the key that remain correspond directly to the unknown high bits of the nonces, therefore by simply fixing a guess for these bits, any corresponding key can be computed. Such pseudo-keys can be used to generate valid new keystreams of any length for nonces in which the highest 18 bits remain the same. This means such keys remain usable up to the point that a bit is flipped in the highest 18 bits of the nonce, which the attacker can predict by learning/controlling the lower 14 bits of the nonce. As the lower 14 bits of the nonce contain 10 bits used for the counter, we can expect the pseudo-key to remain usable for $\frac{2^{10}}{2} = 512$ button presses on average. Once a bit in the upper 18 bits of the nonce does change, a corrected version of the internal state and corresponding pseudo-nonce can be computed trivially using another observed sample.

After having recovered the (pseudo-)key the attacker can increment the last seen counter and construct new nonces to authenticate new commands. Using this attack a legitimate key can be cloned to unlock the car, but that will cause a desynchronization between the counters in the legitimate key and the car. We have demonstrated these cloned keys work on an actual car (Opel Astra H) using our ATAK5795 hardware emulator.

This vulnerability is practically exploitable by a passive attacker with moderate cryptanalytic/computational capacity. It is feasible to sniff the UHF band continuously while automatically decoding authentication messages that can be categorized into logs by their respective UIDs. Once enough cryptograms are collected a cryptanalytic attack can be started to recover the HITAG2 RKE key in the background. This simple and affordable system gives an attacker the possibility of passively building up a cache of cloned keys with up to date counters to be used at their whim.

8.3.3 Jamming attacks

It is possible to make the radio channels in RKE systems unusable by introducing a jamming signal. This would prevent the car from receiving the authentication message required to lock the doors, and could allow an attacker access to the unlocked car (as long as the driver does not get suspicious by the missing sights and sounds).

Interactively jamming the radio spectrum can prevent a command from being correctly received by the car, while the attacker does learn the message. This approach could also turn the passive attack scenario described above into a partially active one, which allows the attacker to recover the required 2 authentications in a single user interaction.

8.4 Suggested improvements

The investment in a particular platform like HITAG2 is one which system integrators cannot easily forfeit, as it is tightly integrated with other systems in many different models of cars. The manufacturer NXP has already acknowledged the problems with HITAG2 cryptography and urged its customers to make use of its newer immobilizer/RKE platform based on HITAG-AES. From Kerckhoffs's principle, which states that a good cryptographic system should remain secure if all working parts except its secret key are made public, switching from proprietary HITAG2 cryptography to the publicly standardized AES cipher is a step in the right direction.

Although such better alternatives to HITAG2 are available to the industry, they are still not widely adopted in cars on the road. Retrofitting a better security system onto previously sold cars is not good car salesmanship, so we must look for improvements coming out in newer lines of cars.

At the time of writing it seems the industry faces a much more serious problem: newer cars with PKE feature can almost universally be broken into with much less sophisticated means than the attacks theorised in this section by means of a relay attack [7]. As all such systems share the same fundamental flaw independent of the employed cryptography, sadly the cryptographically improved HITAG-AES keys employing PKE are also at risk.

Chapter 9

Conclusion

Due to a unique opportunity we were able to explore the subject of automotive security, specifically modern RKE systems within a company that specializes in after-market car key and lock service. This subject is related to earlier work into RFID-based immobilizer systems (the other main digital security feature on modern cars) because both features come together in the car key.

A family of widely deployed microchips for such car keys are made by NXP, which is known to use the cryptographic cipher HITAG2 to secure the contents of its RFID immobilizer interface and also performs the RKE functionality over a UHF radio interface. This family of devices was our main area of interest during the study, as we were optimistic about finding the same cipher's output appear in transmissions from both radio interfaces.

During this research we have gained the experience and drafted the tools to complete in-depth analyses of some widely deployed HITAG2-based RKE systems, and documented the principles behind them to the point of being able to construct a working emulation. This should be alarming to readers due to the fatal flaws we know to exist in the HITAG2 cipher, because these flaws allow a practical attack against the vulnerable systems that can (partially) recover the secret cryptographic key used by to generate authentication cryptograms by *sniffing* two such authentication messages. Because the behavior of a real RKE key can be cloned to the emulated device using nothing but information passively gathered over the air, we would like to stress the very real risk of abuse.

Existing risks in car diagnostics interfaces and RFID-based immobilizer systems (which prevent a car from being hot-wired after entry) are greatly compounded by this weakened physical security: the attacker can get inside the car to do this while leaving much less physical evidence. Another compounding risk factor in this context is that the hardware to receive and transmit messages to the RKE systems costs in the order of only tens of Euros (depending on desired range, quality, and ease-of-use). Performing the cryptographic attack to recover an equivalent *pseudokey* requires only some sniffed data, a decent desktop computer, and some patience.

As an offshoot of the low-level research, our analysis tools proved useful in mapping the landscape of RKE systems, where we have identified the exact types of HITAG2-based RKE systems we know to be vulnerable by the current state of our research. Other systems which we have not yet investigated in-depth are likely to raise at least a few more equally alarming results.

The result of the study is a broad and deep analysis of the subject crafted over the course of the better part of a year, using every every bit of hardware, software and expertise available within the company that I could desire. However, something they explicitly have not given me access to is confidential information that originated from outside their own company.

We have employed a so-called *gray box* reverse-engineering approach, which in this instance means we knew a few details about the devices in question (i.e. they are based on HITAG2), were aware of the general contents of an RKE message, and made educated guesses along the way to complete our understanding.

This was only possible due to the extensive material and mental support of the team at Car Lock Systems. The research also proved to be a valuable experience. After studying up on the theory and building my own software tools I have become completely confident in further researching both the analog and digital aspects of the topic.

Bibliography

[1] R. Verdult, F. D. Garcia, and J. Balasch, "Gone in 360 seconds: Hijacking with Hitag2," in *21st USENIX Security Symposium (USENIX Security 2012)*, pp. 37–37, USENIX Association, 2012.

[2] S. C. Bono, M. Green, A. Stubblefield, A. Juels, A. D. Rubin, and M. Szydlo, "Security analysis of a cryptographically-enabled RFID device," in *14th USENIX Security Symposium (USENIX Security 2005)*, pp. 1–16, USENIX Association, 2005.

[3] R. Verdult, F. D. Garcia, and B. Ege, "Dismantling Megamos crypto: Wirelessly lockpicking a vehicle immobilizer," in *22nd USENIX Security Symposium (USENIX Security 2013)*, pp. 703–718, USENIX Association, 2013.

[4] G. Farrell and R. Brown, "On the origins of the crime drop: Vehicle crime and security in the 1980s," *The Howard Journal of Crime and Justice*, vol. 55, no. 1-2, pp. 226–237, 2016.

[5] G. Farrell, N. Tilley, A. Tseloni, and J. Mailley, "The crime drop and the security hypothesis," *Journal of Research in Crime and Delinquency*, pp. 147–175, May 2011.

[6] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, *et al.*, "Comprehensive experimental analyses of automotive attack surfaces.," in *20th USENIX Security Symposium (USENIX Security 2011)*, pp. 77–92, USENIX Association, 2011.

[7] A. Francillon, B. Danev, and S. Capkun, "Relay attacks on passive keyless entry and start systems in modern cars.," in *NDSS*, 2011.

[8] "Transponder IC, Hitag2." Product Data Sheet, Nov 2010. NXP Semiconductors.

[9] G. Goavec-Mérou and J. Friedt, "GNURadio as a general purpose digital signal processing environment," *FOSDEM (Bruxelles, 2014)*, 2014.

[10] A. Verstegen and R. Verdult, "Analysis, implementation and attack of HITAG RFID protocols." unpublished, 2014.

[11] A. Verstegen, R. Verdult, and W. Bokslag, "Hitag 2 hell – brutally optimizing guess-and-determine attacks," in *12th USENIX Workshop on Offensive Technologies (WOOT 2018)*, (Baltimore, MD), USENIX Association, 2018.

[12] J. Wetzels, "Broken keys to the kingdom: Security and privacy aspects of RFID-based car keys," *arXiv preprint arXiv:1405.7424*, 2014.

[13] "Security transponder plus remote keyless entry – Hitag2 plus, PCF7946AT." Product Profile, Jun 1999. Philips Semiconductors.

[14] N. T. Courtois and S. ONeil, "FSE rump session – Hitag 2 Cipher (2008)." http://fse2008rump.cr.yp.to/00564f75b2f39604dc204d838da01e7a.pdf, 2008.

[15] I. Wiener, "Software optimized 48-bit Philips/NXP Mifare Hitag2 PCF7936/46/47/52 stream cipher algorithm." http://cryptolib.com/ciphers/hitag2/, 2007.

[16] N. T. Courtois, S. ONeil, and J.-J. Quisquater, "Practical algebraic attacks on the hitag2 stream cipher," in *International Conference on Information Security*, pp. 167–176, Springer, 2009.

[17] N. T. Courtois, "Fast algebraic attacks on stream ciphers with linear feedback," in *Advances in Cryptology – CRYPTO 2003*, vol. 2729 of *Lecture Notes in Computer Science*, pp. 176–194, Springer-Verlag, 2003.

[18] V. Immler, "Breaking Hitag2 revisited," in *Security, Privacy, and Applied Cryptography Engineering*, vol. 7644 of *Lecture Notes in Computer Science*, pp. 126–143, Springer-Verlag, 2012.

[19] P. Štembera and M. Novotný, "Breaking Hitag2 with reconfigurable hardware," in *14th Euromicro Conference on Digital System Design (DSD 2011)*, pp. 558–563, IEEE Computer Society, 2011.

[20] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, vol. 5584 of *Lecture Notes in Computer Science*, pp. 244–257, Springer-Verlag, 2009.

[21] F. D. Garcia, D. Oswald, T. Kasper, and P. Pavlidès, "Lock it and still lose it - on the (in)security of automotive remote keyless entry systems," in *25th USENIX Security Symposium (USENIX Security 2016)*, pp. 929–944, USENIX Association, 2016.

[22] R. Benadjila, M. Renard, J. Lopes-Esteves, and C. Kasmi, "One car, two frames: Attacks on Hitag-2 remote keyless entry systems revisited," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, (Vancouver, BC), USENIX Association, 2017.

[23] T. Kasper, "Embedded security analysis of RFID devices," *Master's thesis, Ruhr Universität Bochum*, 2006.

[24] K. Nohl, D. Evans, Starbug, and H. Plötz, "Reverse engineering a cryptographic RFID tag," in *17th USENIX Security Symposium (USENIX Security 2008)*, pp. 185–193, USENIX Association, 2008.

[25] F. D. Garcia, G. de Koning Gans, R. Muijrers, P. van Rossum, R. Verdult, R. Wichers Schreur, and B. Jacobs, "Dismantling MIFARE Classic," in *13th European Symposium on Research in Computer Security (ESORICS 2008)*, vol. 5283 of *Lecture Notes in Computer Science*, pp. 97–114, Springer-Verlag, 2008.

[26] S. Sun, L. Hu, Y. Xie, and X. Zeng, "Cube cryptanalysis of Hitag2 stream cipher," in *10th International Conference on Cryptology and Network Security (CANS 2011)*, vol. 7092 of *Lecture Notes in Computer Science*, pp. 15–25, Springer-Verlag, 2011.

[27] D. M. Beazley *et al.*, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *4th USENIX Tcl/Tk workshop*, pp. 129–139, USENIX Association, 1996.

[28] Osmocom RTL-SDR project, "RTL-SDR." http://osmocom.org/projects/sdr/wiki/rtl-sdr.

[29] A. Team, "Audacity: free, open source, cross-platform audio software." https://www.audacityteam.org/, 2014.

[30] G. Cook, "CRC-RevEng." http://reveng.sourceforge.net/, 2015.

[31] A. Team, "Avr-libc user manual." http://www.nongnu.org/avr-libc/user-manual/index.html, 2015.

[32] K. Papagiannopoulos and A. Verstegen, "Speed and size-optimized implementations of the PRESENT cipher for tiny AVR devices," in *Radio Frequency Identification*, vol. 8262 of *Lecture Notes in Computer Science*, pp. 161–175, Springer-Verlag, 2013.

[33] B. Blanchet, V. Cheval, X. Allamigeon, and B. Smyth, "ProVerif: Cryptographic protocol verifier in the formal model." http://prosecco.gforge.inria.fr/personal/bblanche/proverif, 2010.

[34] A. I. Alrabady and S. M. Mahmud, "Analysis of attacks against the security of keyless-entry systems for vehicles and suggestions for improved designs," *Vehicular Technology, IEEE Transactions on*, vol. 54, no. 1, pp. 41–50, 2005.