

Master Thesis

Breaking Authentication in the Iridium Network

by

Max Lill

January 26, 2024

University of Kaiserslautern-Landau
Department of Computer Science
Distributed Systems Lab

Examiner: Prof. Dr.-Ing. Jens B. Schmitt
M. Sc. Eric Jedermann

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Breaking Authentication in the Iridium Network“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Kaiserslautern, den 26.01.2024

Max Lill

Max Lill

Abstract

Satellite communication has become an essential technology in an era of increasing digitization and the need for seamless global connectivity. It provides communication services in remote, rural, and maritime regions, as well as in areas where terrestrial infrastructure has been destroyed by war or natural disasters. Given the sensitivity of the transmitted data and the potential consequences of unauthorized access, robust authentication is essential to ensure the integrity and confidentiality of user data. In recent years, several new providers of satellite communications networks have emerged, alongside the established Iridium network, which is known for its unique global coverage, including the polar regions. Iridium was launched in 1998 and initially relied on the now outdated GSM mobile communications standard for authentication. Similar to terrestrial mobile communications, authentication in the Iridium network is carried out using a SIM card, on which a secret key, known only to the network operator, is stored. This thesis demonstrates that Iridium still uses the COMP128-1 algorithm of the GSM standard for authentication, which is no longer considered secure. We were able to successfully extract the secret key of an Iridium SIM card and even clone the SIM card. Since the credentials of an Iridium account is also transmitted unencrypted and can therefore be easily intercepted, complete account takeover is possible, thereby compromising authentication in the Iridium network. Furthermore, this work explores the possibility of account takeover exclusively via radio communication. We concluded that such an attack is conceivable with the necessary resources and laid the groundwork for further research in this area with our code analysis and radio experiments.

Zusammenfassung

In einer Zeit zunehmender Digitalisierung und der Notwendigkeit einer nahtlosen globalen Konnektivität hat sich die Satellitenkommunikation als unverzichtbare Technologie etabliert. Sie stellt Kommunikationsdienste in entlegenen, ländlichen und maritimen Regionen sowie in Gebieten bereit, in denen die terrestrische Infrastruktur durch Krieg oder Naturkatastrophen zerstört wurde. Angesichts der Sensibilität der übertragenen Daten und der potenziellen Folgen eines unbefugten Zugriffs ist eine robuste Authentifizierung unerlässlich, um die Integrität und Vertraulichkeit der Nutzerdaten zu gewährleisten. In den letzten Jahren sind mehrere neue Anbieter von Satellitenkommunikationsnetzwerken auf der Bildfläche erschienen, neben denen sich das etablierte Iridium-Netzwerk, bekannt für seine einzigartige globale Abdeckung einschließlich der Polarregionen, behaupten muss. Iridium wurde schon 1998 in Betrieb genommen und setzte damals bei der Authentifizierung auf den inzwischen veralteten Mobilfunkstandard GSM. Wie beim terrestrischen Mobilfunk wird im Iridium-Netzwerk die Authentifizierung mittels einer SIM-Karte durchgeführt, auf der ein nur dem Netzbetreiber bekannter geheimer Schlüssel gespeichert ist. In dieser Arbeit wurde gezeigt, dass Iridium noch immer den als nicht mehr sicher geltenden Algorithmus COMP128-1 des GSM-Standards für die Authentifizierung verwendet. Dabei konnte der geheime Schlüssel einer Iridium-SIM-Karte erfolgreich extrahiert und die SIM-Karte sogar geklont werden. Da auch die Zugangsdaten eines Iridium-Accounts unverschlüsselt übertragen werden und somit leicht abgefangen werden können, ist eine komplette Account-Übernahme möglich, wodurch die Authentifizierung im Iridium-Netzwerk als gebrochen gilt. Zudem wurde in dieser Arbeit die Möglichkeit einer Accountübernahme ausschließlich über Funk untersucht. Es wurde die Einschätzung getroffen, dass ein solcher Angriff mit den erforderlichen Ressourcen denkbar ist. Durch eine vertiefende Code-Analyse und Funkexperimente konnte eine Basis für weitere Forschungen in diesem Bereich geschaffen werden.

Contents

1. Introduction	1
2. Background	3
2.1. Satellite Communication	3
2.2. GSM	5
2.2.1. Authentication in GSM	5
2.2.2. COMP128-1	7
2.2.3. Collision Attack on COMP128-1	9
2.3. Iridium	13
2.3.1. Authentication in Iridium	14
2.4. gr-iridium and iridium-toolkit	16
3. Related Work	18
3.1. Side-Channel Attacks on Comp-128-1 in GSM	18
3.2. Security in Satellite Systems	19
3.3. Security in the Iridium Network	19
4. Outdoor Field Experimentation	22
4.1. Experimental Setup	22
4.2. Experimental Results	24
5. Key Extraction	28
5.1. Experimental Setup	28
5.2. Experimental Results	30
6. SIM Card Cloning	33
6.1. Experimental Setup	33
6.2. Experimental Results	34
7. Code Analysis of COMP128-1	37
7.1. Experimental Motivation	37
7.2. Experimental Results	38
8. Indoor Radio Experimentation	44
8.1. Experimental Setup	44
8.2. Experimental Results	45

9. Conclusion and Future Work	47
A. Source Code	52
A.1. C Implementation of COMP128-1	52
A.2. Calculation of the Originating Pairs	56
A.3. Statistical Evaluation of Back-Calculation	57

List of Figures

2.1. LEO, MEO, and GEO satellite orbits [4].	3
2.2. Spot beams (1) and footprints (2) of a LEO satellite [5].	4
2.3. GSM authentication process [6].	6
2.4. Input and output of COMP128.	7
2.5. Flowchart of COMP128-1.	8
2.6. Butterfly compression of the hashing function [9].	10
2.7. Collisions in COMP128-1. The pairs $(K_{i_i}, RAND_{i_i})$ and $(K_{i_{i+8}}, RAND_{i_{i+8}})$ of level 0 are combined to calculate level 1. The pairs $(K_{i_i}, K_{i_{i+8}})$ and $(RAND_{i_i}, RAND_{i_{i+8}})$ of level 1 are combined to calculate level 2. $RAND_{i_i}$ and $RAND_{i_{i+8}}$ of the input challenge can be iterated through until a collision occurs.	11
2.8. Inputs of the hashing function for the collision attack; K_i is fix.	11
2.9. Inputs of the hashing function for the collision attack; $RAND$ is fix.	12
2.10. Iridium satellite constellation [13].	13
2.11. Authentication process in the Iridium network.	15
4.1. Interception setup: Iridium antenna connected via a hackRF SDR to a laptop running gr-iridium to intercept $RAND$ and $SRES$ during an authentication process in the Iridium network.	22
4.2. Experimental setup in Weselberg.	23
4.3. .pcap file opened with Wireshark.	24
4.4. Line of sight from Morlautern to the university parking garage roof [30].	25
4.5. Experimental setup in Morlautern.	26
4.6. Displayed log on the phone while sending an e-mail.	27
5.1. Card readers used in our experiments.	29
5.2. Woron Scan executed on Windows 10 with HID OMNIKEY card reader.	30
5.3. Execution of COMP128-1 in both software and hardware using the same challenge. The first command runs <code>comp128-1.c</code> , while the second command executes the algorithm directly on the SIM card.	32
6.1. SIM Scanner v5.15 and Sim Scan v2.01.	33
6.2. Writing SIM MAX card with SIM Scanner v5.15.	34
6.3. Execution of COMP128-1 on the Iridium SIM card (first command) and on the cloned SIM MAX card (second command).	35
7.1. Butterfly compression of the hashing function at level 4 [9].	39

7.2. Butterfly compression of the hashing function. Calculation of SRES starting from level 2 [9].	43
---	----

List of Tables

- 2.1. Compression table structures of the hashing function. 9
- 7.1. Compression table of level 4 used in the hashing function. 39
- 7.2. Statistical evaluation of the 65,536 possibilities of a four-digit block in the output. It was conducted to determine, for each possible output, how many of the corresponding 256 possibilities in level 3 actually generate the respective output. 42

1. Introduction

In an era where global connectivity is paramount, the importance of satellite communications networks has become increasingly prominent due to their ability to provide uninterrupted, worldwide coverage. Thus, satellite communications networks play a crucial role in regions where terrestrial infrastructures are absent or ineffective, such as remote areas, vast oceanic regions and areas affected by natural disasters. The growing reliance on satellite communications networks for a variety of applications, ranging from personal communication to critical data transmission in sectors such as maritime navigation, aviation, environmental monitoring, and global broadcasting, underscores the need for robust security mechanisms. Hence, secure authentication of these networks, becomes vital to ensure the integrity and confidentiality of communication.

While Iridium is a long-established satellite communications provider with more than two decades of operational experience, new entrants like SpaceX's Starlink, OneWeb and Amazon's Project Kuiper have recently emerged. Nevertheless, Iridium has not only demonstrated sustained growth, with a 15% increase in its user base to more than 2.2 million over the past five years, but also continues to be the only network offering connectivity across every point on the globe [1].

Global System for Mobile Communications (GSM), is a standard developed to describe protocols for second generation (2G) digital terrestrial cellular networks used by mobile phones. First introduced in Europe in the early 1990s, GSM became the global standard for mobile communications and revolutionized mobile connectivity. It laid the foundation for various services such as voice calls, SMS (Short Message Service), and basic data communication. The GSM standard, now considered outdated and insecure, has been replaced by newer standards such as 3G, 4G (LTE), and 5G. However, GSM remains a key technology in many parts of the world, particularly in areas where more advanced infrastructure is not available. GSM users authenticate themselves with SIM cards. The authentication algorithm is executed directly on the SIM card and confirms to the provider through the knowledge of a shared secret that the user is legitimate.

Iridium adopted the GSM standard for its authentication process and it appears that Iridium is still using the outdated GSM authentication algorithms today. However, Iridium operates independently of terrestrial GSM networks and uses its own network of satellites to provide global coverage. In the Iridium network, the user-side authentication algorithm is also executed directly on the SIM card, which is inserted

into an Iridium device. The implementations of the GSM authentication algorithms have been kept confidential and Iridium also maintains secrecy about its authentication process. However, over time, the GSM authentication algorithm COMP128 has been reverse engineered and exists in three versions. The original version, COMP128-1, is no longer considered secure, as such SIM cards can be cloned and duplicated, among other vulnerabilities. However, the newer versions, COMP128-2 and COMP128-3, are not yet considered deprecated [2]. At the start of the thesis, we suspected that Iridium had implemented either COMP128-1 or COMP128-2 on its SIM cards, since Veeneman [3] attributes a characteristic to Iridium's authentication algorithm that applies only to COMP128-1 and COMP128-2, but not to COMP128-3.

In this thesis, we delve into Iridium's authentication process to determine which algorithm is used for authentication and to identify potential vulnerabilities. For this purpose, Chapter 2 provides background information on satellite communication, GSM and its authentication process, Iridium, and the libraries used in our experiments. In addition, we explain the exact operation of COMP128-1 and the collision attack, which aims to extract the shared secret of the SIM card. In Chapter 3, we outline related work on other attacks on COMP128-1 and security in satellite networks, in particular the Iridium network. In the subsequent five Chapters, we present our own research and findings. Chapter 4 showcases the results of our field measurements. Chapter 5 focuses on the extraction of the shared secret from Iridium SIM cards, while Chapter 6 delves into the cloning of these SIM cards. In Chapter 7, we examine vulnerabilities of the COMP128-1 algorithm through code analysis. Finally, Chapter 8 presents the results of our shielded radio experiments, offering insights into the practical aspects of security within the Iridium network. In the conclusion, Chapter 9, we summarize our findings and discuss future work, offering insights and recommendations for enhancing the security of the Iridium network based on our analysis.

2. Background

2.1. Satellite Communication

In the field of satellite communication, an integral component of modern telecommunications, the journey began with the launch of Sputnik 1 by the Soviet Union in 1957. This marked the first instance of an artificial satellite orbiting the Earth, setting the stage for future advancements in satellite technology. However, the first active communications satellite, Echo 1, launched by the United States in 1960, truly pioneered the era of satellite communication by successfully transmitting and receiving signals from the ground.

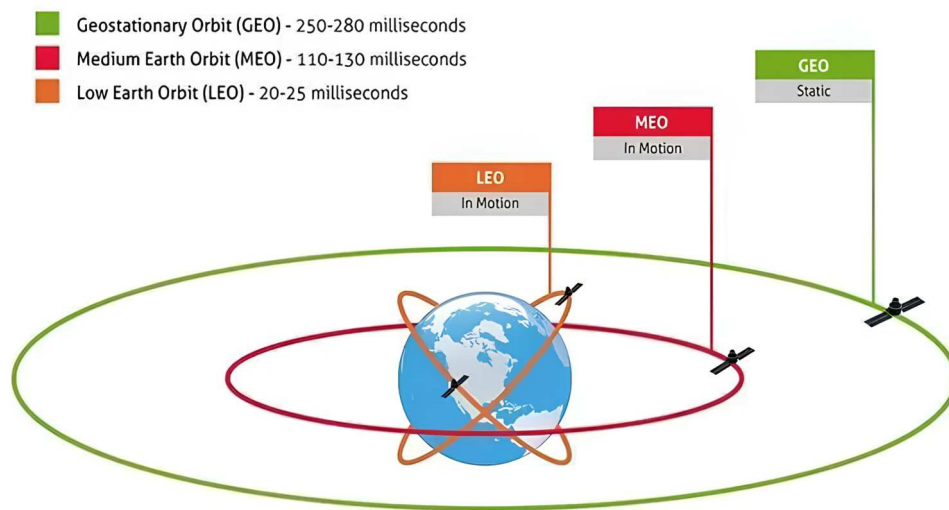


Figure 2.1.: LEO, MEO, and GEO satellite orbits [4].

Satellites today operate in various orbits, each characterized by its unique altitude and purpose. Figure 2.1 shows the most common satellite orbits, including the low Earth orbit (LEO), medium Earth orbit (MEO), and geostationary orbit (GEO). LEO satellites, such as those used by Iridium and Starlink, orbit at altitudes between 160 to 2,000 kilometers, providing lower latency but requiring a larger number of satellites for continuous coverage. MEO satellites, typically found between 2,000 to 35,786 kilometers, strike a balance between coverage area and delay. The GEO satellites,

positioned at approximately 35,786 kilometers, align with the Earth's rotation, offering constant coverage over specific areas, making them ideal for weather forecasting, television broadcasting, and other communications applications.

The architecture of a satellite communications system comprises several key components: the satellite itself, the ground stations, and user terminals. The satellite acts as a relay, receiving signals from one location and transmitting them to another. The transmission of signals from a satellite to Earth is known as downlink, whereas uplink refers to the transmission of signals from Earth to the satellite. Notably, signals can also be relayed from satellite to satellite before being sent down to a ground station. This inter-satellite linking enhances global coverage and communication efficiency, especially in areas without direct ground station visibility. Ground stations, equipped with large antennas and tracking systems, play a critical role in controlling satellite operations and facilitating communication between satellite and network. These ground stations are strategically placed around the world to maintain continuous contact with the orbiting satellites. User terminals, varying in size from small handheld devices to large dishes, enable end-users to send and receive data via the satellite network.

In terms of access protocols, satellite communications systems predominantly use Time Division Multiple Access (TDMA), Frequency Division Multiple Access (FDMA), or Code Division Multiple Access (CDMA). These protocols efficiently manage the allocation of bandwidth and satellite resources, ensuring effective communication among a large number of users.

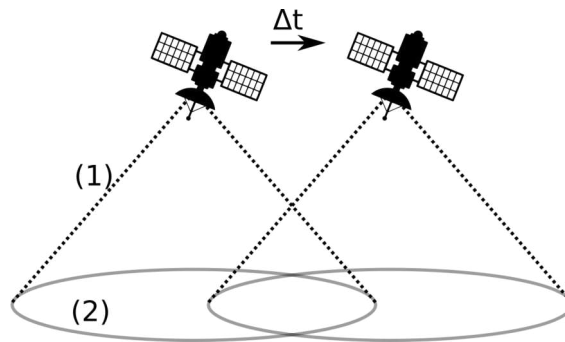


Figure 2.2.: Spot beams (1) and footprints (2) of a LEO satellite [5].

A communications satellite is usually equipped with multiple antennas, each capable of creating a spot beam. Spot beams are essentially concentrated signals directed towards a specific area on Earth. The use of multiple spot beams enables the satellite to increase signal strength within a focused area, thereby enhancing communication quality and efficiency. Additionally, it allows for the reuse of frequencies in neighboring beams, an essential feature for optimizing bandwidth and reducing interference.

The high relative angular velocity of LEO satellites as they orbit the Earth results in a rapid transitory motion across the sky from the perspective of a terrestrial observer.

Communications satellites have an area on the Earth's surface within which their signals can be received. This area is known as the satellite's footprint, and it moves along with the satellite. Figure 2.2 illustrates the relationship between spot beams and footprints of a LEO satellite.

A key design principle in LEO satellite networks is ensuring that a terrestrial user terminal is always within the coverage of at least one spot beam during active communication. The messages transmitted from the satellite to the terrestrial terminal on the downlink are receivable across the entire beam area, due to the broadcast nature of such antennas. This broadcast characteristic allows for a wide distribution of signals, but also presents challenges in terms of security and signal management, particularly in a dynamic environment like that of LEO satellites.

Moreover, the rapid movement of LEO satellites presents significant challenges, including frequent inter-beam and inter-satellite handoffs, as well as constantly changing Doppler shifts.

Notable examples of satellite communications systems include the Iridium network, known for its constellation of LEO satellites providing global coverage. Another example is Starlink, a project by SpaceX, which aims to deliver high-speed internet worldwide through a vast network of LEO satellites. These systems exemplify the diverse applications and technological advancements in the field of satellite communication, underscoring its significance in global connectivity and security.

2.2. GSM

The Global System for Mobile Communications (GSM) is a standard developed to describe protocols for second-generation (2G) digital cellular networks used by mobile phones. Introduced in the 1990s, GSM was a revolutionary technology that replaced first-generation analogue cellular networks. It facilitates mobile communications worldwide by defining various protocols for voice and data services. GSM's widespread adoption is attributed to its robust and efficient mobile communications system, providing higher call quality and security than previous generations.

GSM networks operate in a number of frequency bands and are divided into cells, each served by a base station. These cells provide extensive coverage and capacity by reusing frequencies in non-adjacent cells. GSM supports various data services such as SMS (Short Message Service) and MMS (Multimedia Messaging Service) and has been the backbone for the growth of mobile communications worldwide.

2.2.1. Authentication in GSM

The GSM network authentication process is a critical component in ensuring secure communication between the user and the network. It verifies the identity of the Sub-

2. Background

subscriber Identity Module (SIM) card used in the mobile device, preventing unauthorized access to the network. The user identifies themselves with a PIN (up to 8 characters) on the SIM card. The process then involves a challenge-response mechanism where the network challenges the SIM card with a random number (RAND). The SIM card, which contains a shared secret, the individual secret key (K_i), calculates a response (SRES) using the COMP128 authentication algorithm, also known as A3/A8, and sends it back to the network.

The network provider possesses the shared secret associated with the SIM card and performs the same calculation by a base station. The base station then compares its result with the response from the SIM card to authenticate it and grant the user network access. The sequence of this challenge-response procedure is visualized in Figure 2.3.

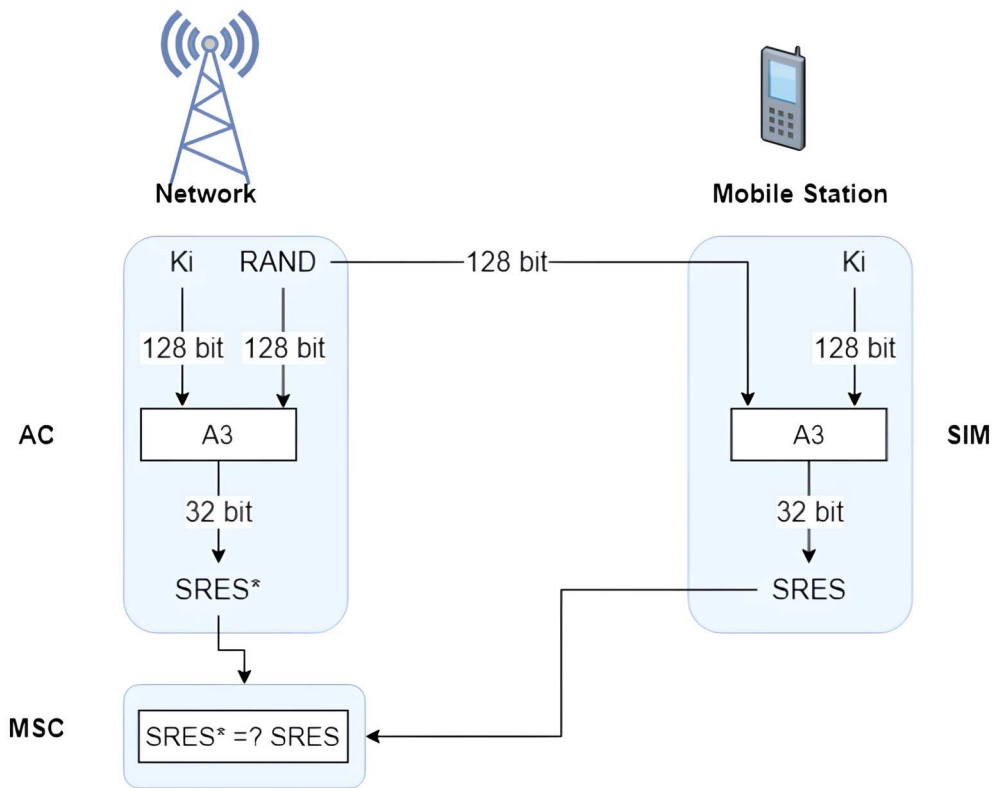


Figure 2.3.: GSM authentication process [6].

To maintain user anonymity and location confidentiality, the International Mobile Subscriber Identity (IMSI) is used minimally, primarily during authentication. After the initial network registration, a Temporary Mobile Subscriber Identity (TMSI) is securely transferred to the SIM card and used for subsequent network interactions, including obtaining a new TMSI. Thus, the GSM authentication process is not only critical for verifying user identity but also for maintaining the privacy and integrity of communication within the GSM network.

2.2.2. COMP128-1

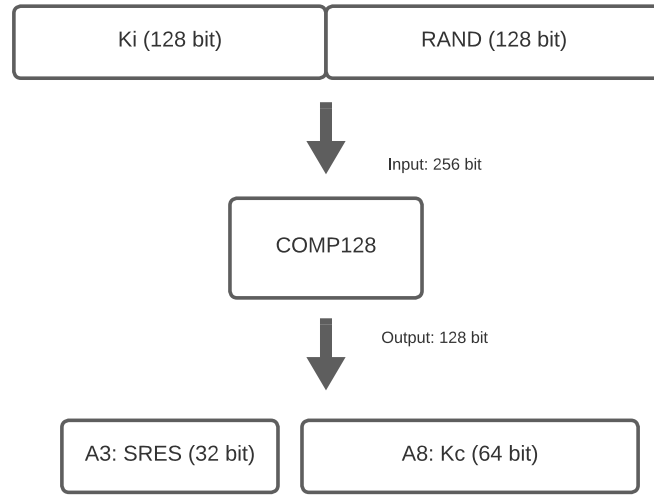


Figure 2.4.: Input and output of COMP128.

On the input side of COMP128-1, there are two 128-bit values: Ki and RAND. Ki is loaded into the most significant (left-most) 128 bits, while RAND occupies the least significant (right-most) 128 bits. COMP128-1 processes 256-bit input to generate a 128-bit output, which explains the name "COMP128". The most significant 32 bits represent the output of the A3 algorithm, the SRES, while bits 74-127, along with 10 appended 0-bits, constitute the output of A8. This 64-bit value generated by A8 forms the ciphering key Kc, which is used as session key of algorithm A5 for encrypting voice and control data. It is worth noting that Kc appears to have been intentionally weakened by a factor of 2^{10} due to the presence of the 10 0-bits. This weakening leads to a reduction in the number of attempts required for a brute force attack by the specified factor.

COMP128-2 and COMP128-3 are successors of the original COMP128-1 algorithm used in GSM authentication. Developed under the "security by obscurity" approach, their designs were initially kept secret. However, open-source implementations of these algorithms have surfaced, suggesting they have been reverse-engineered [7]. The input and output structure shown in Figure 2.4 also applies to COMP128-2 and COMP128-3. COMP128-2 is a more complex algorithm and was developed as a quick fix to the issues present in COMP128-1. However, COMP128-2 also reduces the effective key size of Kc to 54 bits by setting the last 10 bits of the Kc to zero. COMP128-3, essentially a variant of COMP128-2, removes this limitation, utilizing the full 64-bit key size to enhance security. This improvement addresses the deliberate weakening found in COMP128-2 and strengthens the associated A5 ciphering process. [8]

Figure 2.5 depicts the flowchart of COMP128-1, which exhibits a round-based structure. First, Ki and RAND are concatenated in the variable x (256 bits). Then, the

2. Background

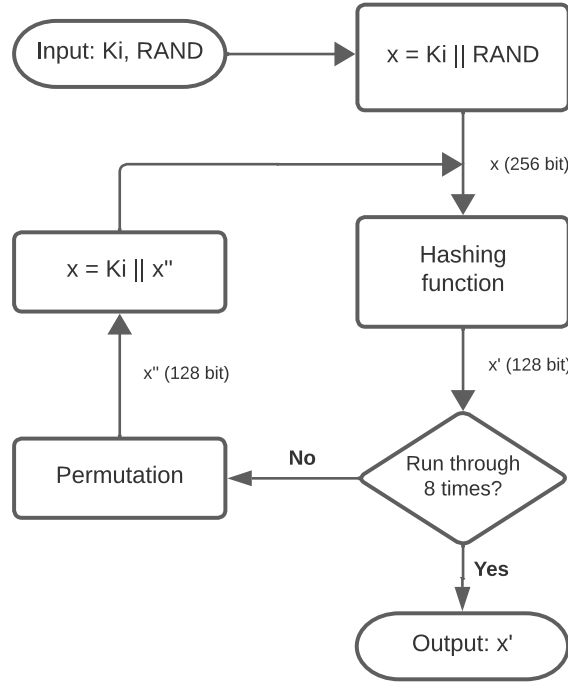


Figure 2.5.: Flowchart of COMP128-1.

hashing function is iterated eight times. After each call to the hashing function (except the last one), x' is produced consisting of 32 4-bit values. Before entering the permutation block, x' is first consolidated into 16 8-bit values. Then, the values of x' are processed through the permutation block, where they are swapped as described later. The resulting x'' (128 bits) should be extended by the preceding K_i to match the required 256 bits of the input for the hashing function. After the eight iterations, x' (128 bits) will form the output of the COMP128-1 algorithm and the most significant 32 bits of the output form the SRES.

Listing 2.1: Pseudocode of the hashing function [9].

```

for j = 0 to 4 do {
  for k = 0 to  $2^j - 1$  do {
    for l = 0 to  $2^{4-j} - 1$  do {
      m =  $1 + k \cdot 2^{5-j}$ ;
      n =  $m + 2^{4-j}$ ;
      y =  $(X[m] + 2 \cdot X[n]) \bmod 2^{9-j}$ ;
      z =  $(2 \cdot X[m] + X[n]) \bmod 2^{9-j}$ ;
      X[m] = T[j][y];
      X[n] = T[j][z];
    }
  }
}

```

Listing 2.1 presents the pseudocode for the hashing function, while Figure 2.6 provides a visualization of the process. In each of the eight rounds, the hashing function undergoes five levels. The hashing function is a surjective function, but not an injective one.

The variable j indicates the level we are currently in. In the current level, the entries at positions m and n are always combined, such that in y , the simple value of position m is added to twice the value of position n , and in z , twice the value of position m is added to the simple value of position n . The value of entry y from a given compression table is then written to position m of the next level. Similarly, the value of entry z from the given compression table is written to position n of the next level.

The compression essentially occurs through the tables. Each level has its own compression table and the number of entries and the value range are halved with each level. The first table, $T[0]$, has 512 entries with values between 0 and 255. The last table, $T[4]$, only has 32 entries with values between 0 and 15. Hence, in each table, each value appears twice. The compression table structures are presented in Table 2.1, while the exact entries can be found in Appendix A.1.

Table	Number of entries	Range of values
$T[0]$	512 (9 bit)	0 - 255 (8 bit)
$T[1]$	256 (8 bit)	0 - 127 (7 bit)
$T[2]$	128 (7 bit)	0 - 63 (6 bit)
$T[3]$	64 (6 bit)	0 - 31 (5 bit)
$T[4]$	32 (5 bit)	0 - 15 (4 bit)

Table 2.1.: Compression table structures of the hashing function.

m and n are calculated in such a way that, initially, the pairs are spaced apart by 16 positions, and this distance is halved with each level until, in level 4, the positions of the 16 pairs are adjacent. Figure 2.6 illustrates this interleaving. The crossover operation is also referred to as an S-box or even as butterfly compression because the cross resembles the wings of a butterfly.

If the eight rounds of COMP128-1 have not yet been completed, the permutation block follows in the block diagram in Figure 2.5. The permutation rearranges the 128-bit value of x' from the hashing function, whereby each bit is shifted individually and the value of each position k is set to the new position $k' = f(k) = (k \cdot 17) \bmod 128$. After the shifts, the resulting 128-bit sequence is divided into bytes, and the order of the bits within each byte is reversed.

2.2.3. Collision Attack on COMP128-1

The collision attack was first carried out by Briceno et al. [10] at the University of California, Berkeley, in 1998. Within 8 hours, they were able to reconstruct the secret key, K_i , using 150,000 requests at a rate of 6.25 requests per second.

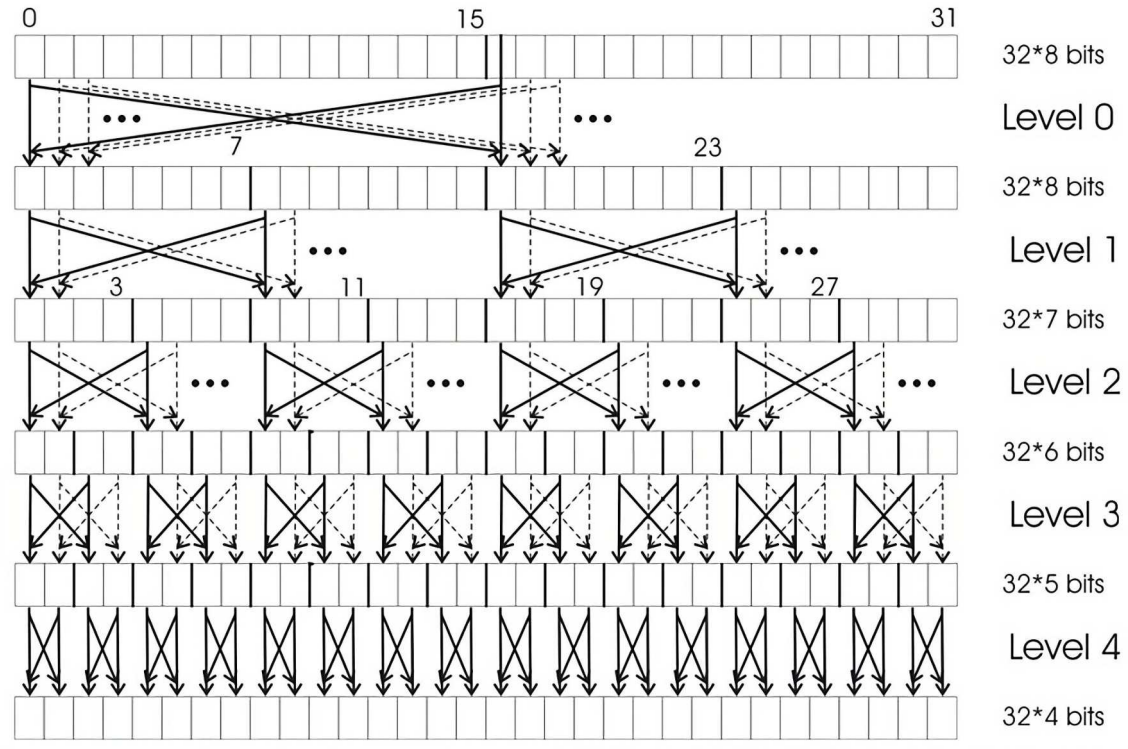


Figure 2.6.: Butterfly compression of the hashing function [9].

Woron Scan and Sim Scan, the programs we use in Chapter 5, implement an improved version of this collision attack. The exact implementation of Woron Scan and Sim Scan is not known.

Collisions happen when the results of the COMP128-1 algorithm, $SRES^1$ and $SRES^2$, are identical for two different values of $RAND^1$ and $RAND^2$. This occurs because the input set is mapped to a smaller output set, causing multiple elements of the first set to be mapped to a single element of the second set. It is important to note that collisions are inevitable in this process.

Collisions can already occur in the second level of the first round. If a collision occurs, it propagates right through all other levels. While collisions could also happen in subsequent levels, it would exponentially increase the effort for each additional level.

As shown in Figure 2.7 for position index $i = 0$, the pairs $(Ki_i, RAND_i)$ and $(Ki_{i+8}, RAND_{i+8})$ are combined, respectively, in the butterfly compression of the first level. Subsequently, in the second level, the pairs (Ki_i, Ki_{i+8}) from the secret key and $(RAND_i, RAND_{i+8})$ from the challenge are combined. After butterfly compression of the second level, the length of the result $(Ki_i, Ki_{i+8}, RAND_i, RAND_{i+8})$ is $4 \cdot 7 = 28$ bits. The input $RAND_i$ and $RAND_{i+8}$ can be freely chosen to create a collision. A collision that occurs here propagates from the second level of the first round all the way to the output of COMP128-1. [11]

2. Background

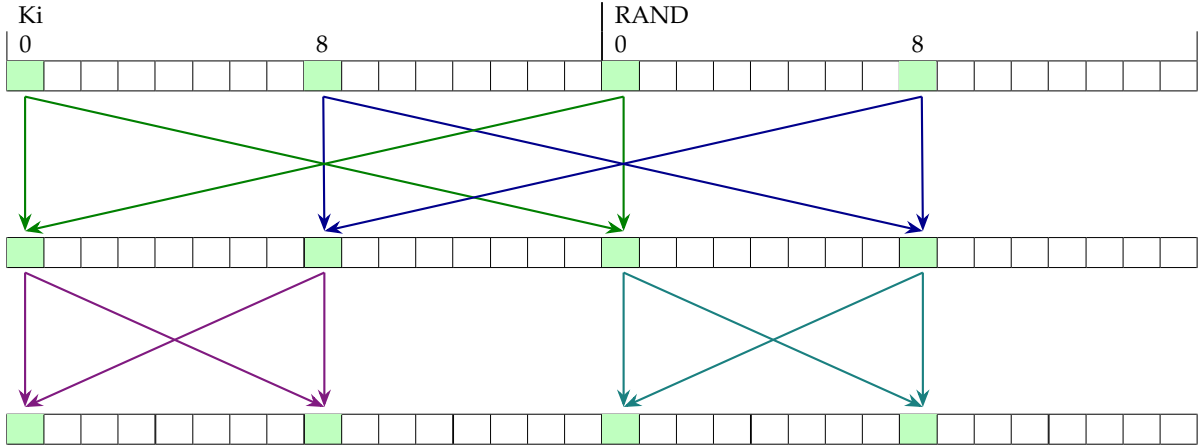


Figure 2.7.: Collisions in COMP128-1. The pairs $(Ki_i, RAND_i)$ and $(Ki_{i+8}, RAND_{i+8})$ of level 0 are combined to calculate level 1. The pairs (Ki_i, Ki_{i+8}) and $(RAND_i, RAND_{i+8})$ of level 1 are combined to calculate level 2. $RAND_i$ and $RAND_{i+8}$ of the input challenge can be iterated through until a collision occurs.

During an attack, $(RAND_i, RAND_{i+8})$ is iterated through, while keeping the other bytes of the RAND challenge constant, until two RAND values are found for which, in combination with the fixed Ki , the same output of COMP128-1 occurs.

This process is repeated eight times for all RAND pairs from $(RAND_0, RAND_8)$ to $(RAND_7, RAND_{15})$. Figure 2.8 visualizes the input structure for $i = 0$, $i = 1$, and $i = 7$. Note that $RAND_0$ corresponds to the 16th entry, and $RAND_8$ corresponds to the 24th entry in the 32-byte input array x , which is fed into the hashing function in the flowchart in Figure 2.5. As a result, after the eight iterations, eight pairs of RAND are obtained, each producing the same SRES.

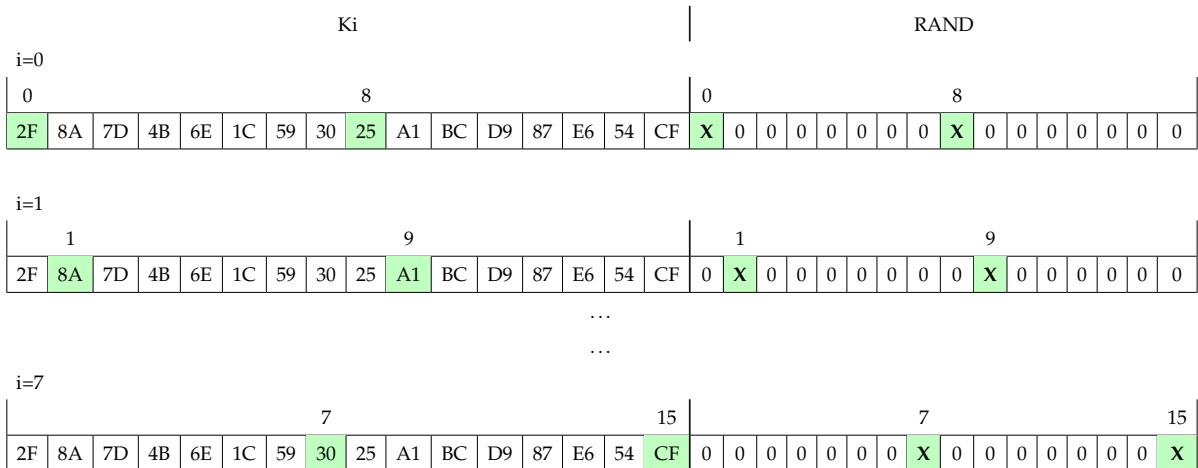


Figure 2.8.: Inputs of the hashing function for the collision attack; Ki is fix.

2. Background

The search for RAND pairs was performed on the SIM card, and K_i was fixed during this process. Now, with the RAND pairs as fixed input, the search for the key bytes can be performed on the computer without the SIM card. The positions K_i and K_{i+8} are now counted through, while the other positions of K_i are set to 0. If there is a collision of the RAND pairs $(\text{RAND}_i^1, \text{RAND}_{i+8}^1)$, $(\text{RAND}_i^2, \text{RAND}_{i+8}^2)$ for K_i and K_{i+8} , two new key bytes have been found. Figure 2.9 highlights the corresponding positions of the array for $i = 0$ and $i = 7$. After eight iterations, the entire secret key K_i can be recovered in this way.

Ki																RAND																
i=0																																
08																08																
X	0	0	0	0	0	0	0	0	X	0	0	0	0	0	0	F5	67	41	D9	32	A8	EB	C6	07	1C	89	A2	54	BF	D0	EA	
																C2	EA	F9	74	B1	53	D8	6A	01	DC	B6	09	8F	35	72	B4	
...																																
...																																
i=7																																
715																715																
0	0	0	0	0	0	0	0	0	X	0	0	0	0	0	0	X	3F	2A	8E	9B	4E	60	5A	7D	0F	72	4C	13	24	3E	F9	5C
																	A9	B0	C1	D2	E3	F4	05	06	81	B5	08	19	F3	6D	9F	C3

Figure 2.9.: Inputs of the hashing function for the collision attack; RAND is fix.

The collision attack leverages the birthday paradox from probability theory. This principle posits that in a set of randomly chosen items, the likelihood of two items being identical (a collision) increases faster than intuitively expected.

The original Briceno-Goldberg-Wagner (BGW) collision attack required around 150,000 challenges. Nevertheless, optimizations exist to decrease this number. The first two levels of the COMP128-1 hashing function are not uniformly random. Some RAND challenge pairs are more effective, potentially causing multiple collisions. By prioritizing these productive challenges, efficiency improves. Additionally, running the attack in parallel for all key pairs and stopping when seven of eight pairs are determined further speeds up the process, as the last pair can be brute-forced offline [11]. Overall, through optimizations, the number of requests can be reduced to approximately 20,000 [12].

After extensive discussions with many experienced GSM engineers, Briceno et al. have determined that over-the-air collision attacks on GSM systems are practically feasible for sophisticated attackers using a fake base station. Although Briceno et al. refrained from carrying out the attack themselves due to legal restrictions under U.S. law, the validation of these insights by GSM experts underscores the real risk of over-the-air collision attacks.

2.3. Iridium

The Iridium network, a significant player in the world of satellite communication, represents a remarkable blend of innovation and global connectivity. Launched in 1998, the network was a brainchild of Iridium SSC, a company originally founded by Motorola. Today, it is operated by Iridium Communications Inc., which took over after Iridium SSC filed for bankruptcy in 1999.

Named after the element Iridium, the network initially planned to have 77 satellites, reflecting the atomic number of Iridium. However, it currently operates with 66 active LEO satellites, circling the Earth at about 780 kilometers altitude in a north-south orientation, as illustrated in Figure 2.10. These satellites move at approximately 27,000 kilometers per hour, completing an orbit around the Earth in about 100 minutes. This constellation of satellites enables the Iridium network to provide global coverage, including over oceans and polar regions.

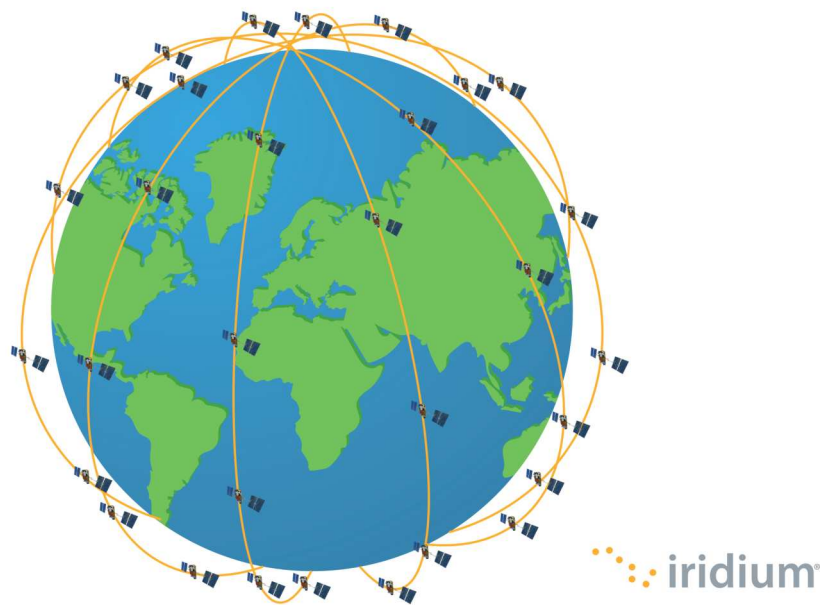


Figure 2.10.: Iridium satellite constellation [13].

Iridium offers a range of critical services, particularly in areas where traditional communications networks are unavailable, like maritime and polar areas. Its portfolio includes voice and data services, such as satellite phones, broadband, and IoT solutions, catering to 2.2 million global users [1]. These services are essential across various sectors, including maritime, aviation, military, emergency response, and remote area communication.

In terms of access protocols, Iridium utilizes a combination of Frequency Division Multiple Access (FDMA) and Time Division Multiple Access (TDMA), which efficiently manage the allocation of communication channels to numerous users. The

Iridium network operates in the L-band, specifically in the 1616-1626.5 MHz frequency range. This range is utilized for both uplink and downlink communication. Each satellite of the Iridium network is capable of producing 48 spot beams, with each spot beam covering a minimum footprint of 400 kilometers. Collectively, these spot beams form a satellite footprint, which can reach a diameter of up to 4,700 kilometers. [14]

Users can gain access to the Iridium network with an Iridium GO! device, which acts as a portable satellite hotspot. By connecting their smartphones or laptops to the Iridium GO! via WiFi, they can make calls, send texts, and access data services globally. Uplink signals sent by Iridium GO! can be received over a distance of more than 30 kilometers [5].

Iridium NEXT was the network's ambitious project to replace its original satellite fleet with new, more advanced satellites. Launched between 2017 and 2019, these next-generation satellites offer enhanced communication capabilities and higher data speeds, significantly upgrading the network's performance and services. [15]

The Iridium network is currently heavily reliant on the United States Department of Defense (DoD), which serves as both a major customer and a significant source of revenue. The DoD uses the network's global communication capabilities for various military operations. This partnership emphasizes Iridium's strategic value in defense communication and contributes to its financial sustainability, ensuring the network's ongoing availability and service enhancement.

Despite the emergence of new players such as SpaceX's Starlink, which aims to provide global broadband internet service, the Iridium network continues to hold a vital role in global communication. Iridium's strength lies in its ability to provide voice and data coverage in the most remote and inaccessible regions of the world, where traditional cellular and broadband networks are unavailable. This capability makes Iridium particularly crucial for maritime, aviation, military, and emergency response communication, ensuring its relevance in the modern satellite communication landscape. Although the outdated bandwidth of 2.4 kbps of an Iridium GO! device is too slow for smooth surfing of modern websites, users can still send and receive emails and text messages, as well as make phone calls, worldwide.

2.3.1. Authentication in Iridium

The authentication process is a fundamental security measure in the Iridium network ensuring that only authorized users can access the network's services. It is designed to prevent security threats such as unauthorized access, eavesdropping, or identity theft. However, the exact authentication process and the algorithms used by Iridium are kept under wraps.

In the document *AMS(R)S Manual Part II Version 4.0*, it is stated that "The Iridium authentication process is adapted without change directly from GSM specifications"

[14]. Further, Veeneman [3] claims that the A3 and A8 algorithms of the GSM standard are combined into a single SIM instruction called "RUN GSM ALGORITHM". Veeneman further claims that a random number RAND and the secret key K_i stored on the SIM card are used as inputs to calculate the signed response SRES and the ciphering key K_c . While Veeneman does not specify the exact algorithm used, he notes that the last ten bits of K_c are always zero, which is a unique characteristic to the COMP128-1 and COMP128-2 algorithms but not to COMP128-3. This suggests that Iridium might be using an older version of the COMP128 algorithm, since COMP128-3, an improvement in terms of security, does not exhibit this trait in the last ten bits. Veeneman's website was last updated in November 2021, indicating that Iridium may still be using outdated GSM algorithms. In addition, Klein et al. [6] were able to confirm the transmission of RAND and SRES during the Iridium authentication process in 2022.

The authentication process in the Iridium network is similar to that described for GSM in Section 2.2.1, except that RAND and SRES are transmitted over Iridium satellites rather than a terrestrial network. When a user attempts to register on the network, the request is relayed from the satellites to the ground station. The ground station then consults a database to find the appropriate secret key K_i for the requesting SIM card and generates a new RAND, which is sent back to the Iridium device via the satellites.

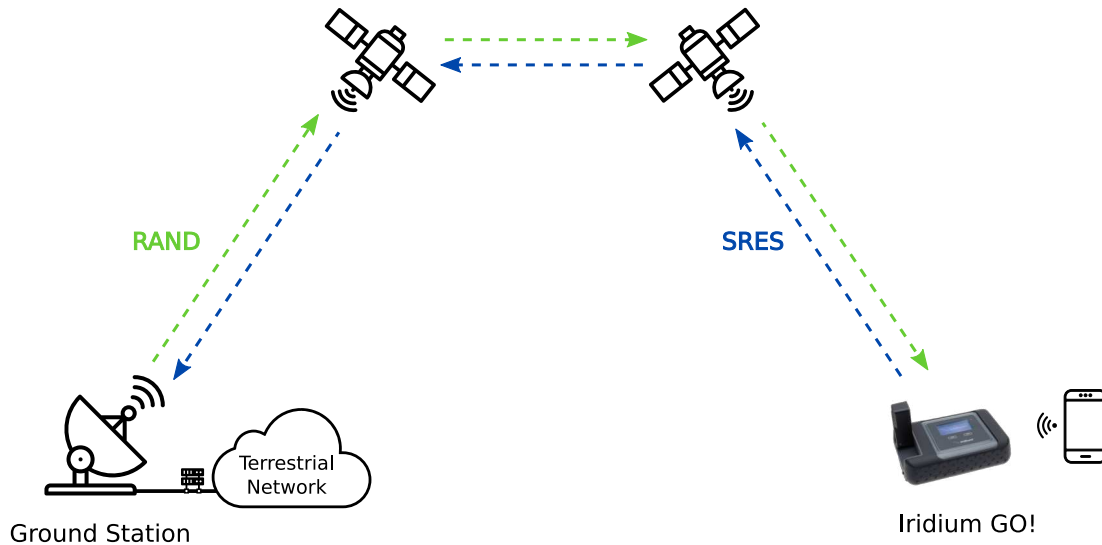


Figure 2.11.: Authentication process in the Iridium network.

The SIM card inside the Iridium device executes the A3/A8 algorithm, using the newly received RAND and the secret key K_i stored on the SIM card as inputs. The Iridium device then sends the SRES, calculated by the SIM card, back to the ground station, which has also performed the A3/A8 algorithm independently and now compares the two SRES values for a match. If the SRES from the user aligns with the SRES

from the ground station, it verifies that the user is in possession of the Iridium SIM card, and access to the Iridium network is granted. The exchange of RAND and SRES is visualized in Figure 2.11. The signals can be routed through one or more satellites, enabling authentication on the Iridium network worldwide.

Iridium also employs IMSI and TMSI from the GSM standard. Notably, the TMSI assigned to the subscriber is static and unencrypted. It can be intercepted over the downlink when located within the same spot beam. [5]

As an additional authentication mechanism, users must log into the Iridium GO! app with their Iridium account credentials to connect their smartphone to the WiFi network of an Iridium GO! device and access Iridium services. These credentials are sent unencrypted in the uplink and can therefore be intercepted if the attacker places an Iridium antenna near the victim's Iridium GO! [6].

2.4. gr-iridium and iridium-toolkit

gr-iridium and iridium-toolkit, developed by the Chaos Computer Club Munich, are essential resources for researchers and enthusiasts in the analysis and understanding of the technical aspects and security of the Iridium network.

gr-iridium is a software developed as part of the GNU Radio project, which is a free software development toolkit for implementing software-defined radios (SDR). gr-iridium leverages this platform to capture and decode signals from the Iridium satellite network. It enables users to record the RF (radio frequency) signals transmitted by Iridium satellites using an SDR receiver. This is an essential initial step in signal analysis and a valuable tool for researchers. [16]

iridium-toolkit is a collection of tools specifically designed for processing and analyzing data decoded by gr-iridium. After gr-iridium captures and converts Iridium signals into a comprehensible format, iridium-toolkit comes into play. It allows researchers to further analyze this data to gain a deeper understanding of the data structure, communication protocols, and potential security vulnerabilities in the Iridium network. The toolkit can be used to investigate encryption, signal structure, or to identify specific message types within the Iridium network. [17]

The following is an example of a single ring alert (IRA) frame.

```
IRA: p-1694003548 000000821.2502 1626299392 94% -39.22|-094.31|27.28 124 DL
sat:077 beam:20 xyz=(+1085,+0137,+1157) pos=(+46.61/+007.20) alt=013 RAI:48
?10 bc_sb:20 P01: PAGE(tmsi:0ca5b2e2 msc_id:01) {OK}
```

All parsed frames start with a header of the following structure [18].

2. Background

Field Number	Content
0	Frame type
1	UNIX timestamp in seconds when the recording started
2	Time in milliseconds inside the recording
3	Frequency in Hertz
4	Confidence in percent
5	Signal strength (signal level, noise level, signal-to-noise ratio)
6	Length in symbols
7	Direction (DL for downlink and UL for uplink)

Ring alerts also contain additional information, including details about the satellite, the spot beam, and TMSI pages. IRA frames exclusively contain the following fields.

Field Number	Content
8	Satellite number
9	Spot beam number
10	Satellite or spot beam position
11	Satellite or spot beam altitude
12	Ring alert interval
13	Unknown
14	Broadcast channel sub-band
15	TMSI pages

A Link Control Word (LCW) is characterized by a 3-bit frame ID that describes its functionality. Moreover, all LCWs have the capacity to carry information pertinent to adjustments in power and frequency as well as data regarding handoffs. LCW 2 data packets, contained in IDA frames, include the authentication packets. Besides the authentication request and response, these frames are also used for call setup, call teardown, SMS, and Iridium-specific parts. In addition to LCW 2 for data packets, there are also LCW 0 for voice frames, LCW 1 for packet-based data, LCW 3 with an unknown purpose, and LCW 7 for synchronization packets. [19]

The following are two IDA frames containing LCW 2 data packets. The first packet contains the authentication challenge, while the second packet carries the response.

```
IDA: p-1694003548 000003023.0647 1622880512 91% -37.84|-092.19|19.12 179 DL
LCW(2,T:hndof,C:handoff_cand,2ab,369,010101010111101101001)
001 cont=0 1 ctr=000 000 len=19 0:0000 [05.12.03.9b.33.63.23.a7.88.d6.34.97.
3e.04.d9.e1.19.aa.67] 050f/0000 CRC:OK 0000 SBD: ....3c#...4.>.....g.
```

```
IDA: p-1694003548 000003068.6760 1622832512 100% -23.17|-092.35|28.12 179 UL
LCW(2,T:maint,C:<silent>,000000000000000000000000)
001 cont=0 1 ctr=000 000 len=06 0:0000 [05.14.5c.cd.76.26]
241f/0000 CRC:OK 0000 SBD: ..\.v&.....
```

3. Related Work

3.1. Side-Channel Attacks on Comp-128-1 in GSM

In their paper “Partitioning attacks: or how to rapidly clone some GSM cards”, Rao et al. introduce a new class of side-channel attacks called partitioning attacks. These attacks demonstrate how the COMP128-1 algorithm on GSM SIM cards can be compromised, even in implementations that are resistant to some side-channel attacks. The idea of their work lies in exploiting the challenges of implementing the COMP128-1 algorithm on 8-bit SIM cards, specifically handling 9-bit compression tables. The paper theorizes that programmers probably split the 9-bit indexed table $T[0]$ into two 8-bit indexed sub-tables ($T[0]_0$ and $T[0]_1$) due to the 8-bit architecture of SIM cards. Thus, the internal structure on the SIM card does not match the one depicted in Table 2.1, with a single table of 512 entries at level 0, but instead consists of two sub-tables, each with 256 entries. This split creates different power signals when accessing each table, enabling the attacker to differentiate between accesses to $T[0]_0$ and $T[0]_1$, thereby exposing key information. This technical nuance in handling 9-bit quantities on an 8-bit platform forms the basis for the successful partitioning attack outlined in the paper. The partitioning attack is significantly more efficient than the collision attack from Briceno et al. While the original BGW collision attack requires about 150,000 chosen inputs for success, the partitioning attack can recover the entire 128-bit key of COMP128-1 with less than 1,000 invocations using random inputs, 255 chosen inputs, or even just 8 adaptively chosen inputs. Rao et al. highlight a fundamental key weakness in table lookups used by cryptographic algorithms, as their attack requires significantly fewer resources compared to previous methods. Furthermore, the authors propose a novel and resource-efficient table lookup mechanism to protect against attacks, particularly in devices with limited resources such as SIM cards. [20]

The paper “Combined side-channel attacks on COMP128” investigates the improvement of the partitioning attack against the COMP128-1 algorithm used in GSM authentication. Levina et al. combine the partitioning attack, which primarily measures power consumption, with other side-channel attacks such as timing, probing, and fault injection. The study concludes that timing attacks have limited effectiveness, while probing and fault injection significantly increase the attack’s speed, reducing the required queries by a factor of eight. This method, particularly with fault injection, balances invasiveness and efficiency, providing a faster way to compromise COMP128-1 while maintaining attack simplicity. [21]

3.2. Security in Satellite Systems

In his study on the Thuraya satellite network, Driessen effectively decrypted communications by reverse-engineering the GEO-Mobile Radio Interface (GMR-1), a standard sharing similarities with GSM. This method, capable of extracting session keys within an hour, exposes vulnerabilities not only in Thuraya but potentially in other systems employing GMR-1, like SkyTerra and TerreStar. [22]

Pavur et al. unveil significant security flaws in maritime Very Small Aperture Terminal (VSAT) networks, showing that unencrypted internet traffic can be easily accessed via satellites using readily available and inexpensive consumer devices. [23]

The document “Satellite-based communications security: A survey of threats, solutions, and research challenges” provides a comprehensive overview of various security threats, solutions, and emerging challenges in satellite communications. It categorizes security into two main branches: physical-layer security and cryptography schemes. The paper delves into physical-layer confidentiality, anti-jamming schemes, anti-spoofing strategies, and the implementation of quantum-based key distribution schemes. However, the authors focus primarily on defense mechanisms and solutions to improve the security of satellite communications, rather than providing detailed examinations of specific attacks. [24]

3.3. Security in the Iridium Network

Oligeri et al. present in their paper “GNSS spoofing detection via opportunistic IRIDIUM signals” a novel approach for GNSS spoofing detection, particularly useful in isolated areas such as the open ocean, using the Iridium satellite network. The research is characterized by the use of opportunistic Iridium Ring Alert (IRA) messages combined with detailed parameters of the Iridium constellation reverse-engineered from extensive data collection using Software Defined Radios (SDRs). This method provides an independent, albeit slightly less accurate, localization solution that is invaluable in environments where other GNSS spoofing detection techniques fall short. Significantly, the paper highlights the challenges of spoofing Iridium signals, mainly due to the higher signal strength required for low Earth orbit (LEO) satellites and the complexity of simulating a coherent array of beams. This aspect makes the Iridium network a robust alternative for navigation safety, especially for autonomous maritime vessels in remote areas. [25]

In the technical white paper “A wake-up call for satcom security” by Santamarta, IOActive focused on the security posture of various SATCOM systems, including Iridium products. Specifically, the paper identified critical security vulnerabilities in two Iridium products, Iridium OpenPort and Iridium Pilot. These products, which

are stationary antennas, were found to have hard-coded credentials and undocumented protocols that pose potential security risks. The research extended beyond Iridium, uncovering vulnerabilities in a further nine products from various satellite systems. What stands out, however, is Iridium's unique response to the findings. Unlike the other companies involved, Iridium took a proactive approach by addressing the issues. This response demonstrates Iridium's commitment to improving and updating its security features, and highlights its awareness and responsiveness to potential security threats. [26]

Jedermann et al. (2021), Oligeri et al. (2022), and Smailes et al. (2023) have significantly advanced research in the field of secure authentication in the Iridium network through their respective works. Their research addresses the challenges and vulnerabilities inherent in such systems, proposing novel solutions that could reshape the approach to authentication and data integrity in LEO satellite communications. We will briefly examine each of their ideas below. [27, 28, 29]

In the paper "Orbit-based Authentication Using TDOA Signatures in Satellite Networks" by Jedermann et al., the authors propose a novel method for authenticating satellite communications using Time Difference of Arrival (TDOA) signatures. This approach is based on the unique orbital trajectories of satellites, which can be matched against TDOA-based signatures to verify satellite communications. The paper demonstrates this concept through extensive simulations using real-world satellite data, particularly focusing on Iridium and Starlink systems. The results show that this orbit-based authentication method can achieve low false authentication rates, making it a promising lightweight security enhancement for existing satellite systems. The approach is particularly relevant for systems like Iridium, as it utilizes physical layer characteristics (orbital information and TDOA signatures) for authentication, offering a potential solution to enhance the security of these networks against spoofing and other attacks. [27]

Oligeri et al. also present in "PAST-AI: Physical-layer authentication of satellite transmitters via deep learning" a significant advancement in satellite communication authentication, specifically for the Iridium network. The utilization of AI methodologies, notably Convolutional neural networks (CNNs) and autoencoders, to analyze I-Q signal samples is a significant step forward in enabling the identification and authentication of satellite transmitters. This approach marks a crucial improvement in enhancing the security of satellite communications, addressing the limitations of traditional cryptographic methods. Despite the challenges posed by the low signal quality and high velocity of LEO satellites like Iridium, the research demonstrates the feasibility of accurate authentication, significantly mitigating the vulnerability of Iridium communications to spoofing and replay attacks. [28]

"Watch this space: Securing satellite communication through resilient transmitter fingerprinting" by Smailes et al. represents another approach for secure authentication in the Iridium satellite network. Building on the advancements of Oligeri et al.'s AI-based methods and Jedermann et al.'s TDOA signature authentication, this study

introduces SatIQ. The system leverages high sample rate radio transmitter fingerprinting to combat signal spoofing and replay attacks. Employing a combination of autoencoders and Siamese neural networks, SatIQ effectively analyzes the unique impairments caused by transmitter hardware on downlinked radio signals. Its robust performance, demonstrated on the Iridium constellation, and adaptability to new transmitters position it as a valuable asset in enhancing satellite communication security. [29]

The paper “RECORD: A RECEPTION-Only Region Determination Attack on LEO Satellite Users” by Jedermann et al. from 2024 introduces a novel passive attack method that poses a significant threat to location privacy in LEO satellite communications. This method, known as RECORD, is based solely on receiving messages from LEO satellite users on the ground. The key insight of the paper is that by simply observing the downlink communications from these satellites, attackers located on Earth can estimate the region in which the users are located. This is particularly relevant for security or military applications, where location privacy is paramount. The research involved building a distributed satellite reception platform to implement the RECORD attack and then analyzing its accuracy and limitations through real-world measurements using Iridium satellite communications. The findings revealed that with just 2.3 hours of traffic observation, an attacker could narrow down an Iridium user’s position to an area with a radius of less than 11 km, which is significantly smaller than the satellite beam size of 4,700 km in diameter. Further simulations suggested that with longer observation periods (over 16 hours), the location could potentially be narrowed down even further, to a radius of less than 5 km. [5]

4. Outdoor Field Experimentation

4.1. Experimental Setup

The objective of this Chapter was to intercept the authentication packets within the Iridium network. By analyzing intercepted messages, we aimed to identify both the RAND in the satellite's downlink and the corresponding SRES in the Iridium GO!'s uplink. Additionally, we aimed to determine the exact moment of authentication.

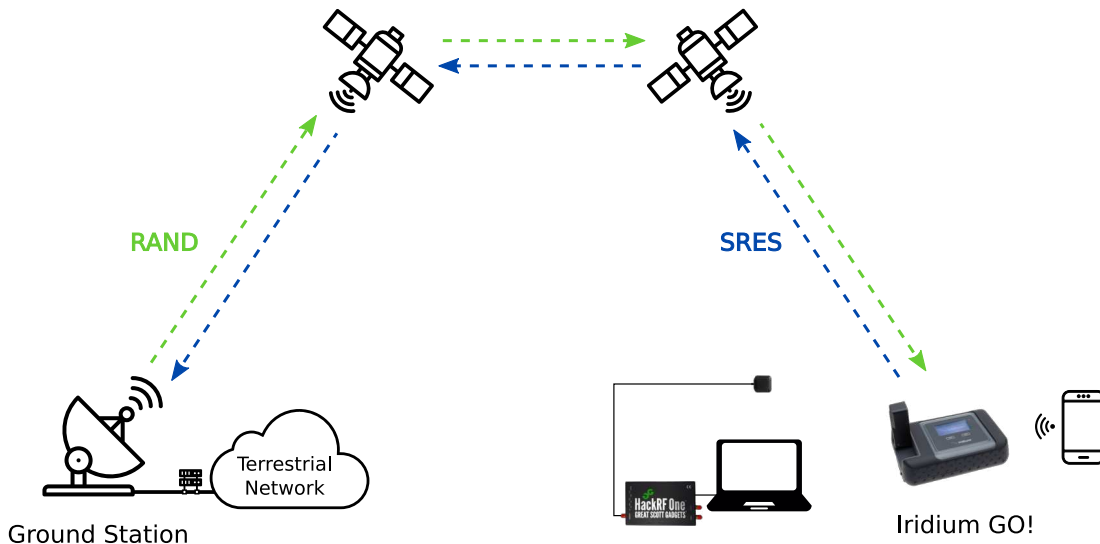


Figure 4.1.: Interception setup: Iridium antenna connected via a hackRF SDR to a laptop running gr-iridium to intercept RAND and SRES during an authentication process in the Iridium network.

We initiated our experimentation with field measurements of self-generated Iridium traffic. We conducted our first measurements near Weselberg in an open field without obstacles, except for a few wind turbines. The North-South direction in particular, along which the Iridium satellites are orbiting, was entirely unobstructed. The setup for intercepting the ongoing Iridium traffic consisted of an Iridium antenna connected via a hackRF SDR to a laptop. We recorded the signals on the laptop with the Iridium burst detector and demodulator gr-iridium [16]. The operating system on the recording laptop was Ubuntu 18.04 LTS. The general setup is illustrated in Figure 4.1, while the specific setup in Weselberg is shown in Figure 4.2.



(a) Traffic generation: Smartphone connected to WiFi network of Iridium GO!.



(b) Interception: Iridium antenna connected via a hackRF SDR to a laptop running gr-iridium.

Figure 4.2.: Experimental setup in Weselberg.

For generating Iridium traffic we used Iridium GO!. It is essentially a portable satellite hotspot device that acts as an Iridium to WiFi adapter. It establishes its own WiFi network, enabling smartphones and other devices to connect and access Iridium's satellite communications services, including e-mail and web browsing. We connected a standard commercial smartphone to the WiFi network of the Iridium GO! device and then sent and received e-mails via the Iridium network. Sending and receiving an e-mail consisted basically of the following steps:

1. Switching on Iridium GO! by raising the antenna
 - a) "Initializing" (displayed on Iridium GO!)
 - b) "Searching" (displayed on Iridium GO!)
 - c) "Registering" (displayed on Iridium GO!)
 - d) Iridium GO! registered and ready
2. Connecting phone to WiFi network of Iridium GO!
3. Login with Iridium account
Open the Iridium GO! App on the phone and login with username and password.
4. Send or receive e-mail via the e-mail/web client
 - a) Write e-mail (optional and offline)
Open the Iridium Mail & Web App on the phone and compose a new E-Mail offline.

- b) Tap on "SEND/RECEIVE MAIL"
 - i. "connecting to the internet" (displayed on the phone)
 - ii. "sending/receiving mail" (displayed on the phone)
 - iii. "connection closed" (displayed on the phone)
- c) Read e-mail (optional and offline)

At the beginning, the distance between Iridium GO! and the laptop with the antenna was approximately 0.7 km. The first challenge was to receive both the uplink and downlink signals. We wanted to record both the uplink and downlink signals in a single file. Furthermore, we aimed to determine the precise timing and nature of the authentication packets.

gr-iridium saves a .bit file of the record on the laptop. The iridium-toolkit [17] library can subsequently parse the .bit file with different modes, generating, for example, a .pcap or a .parsed file for further analysis.

4.2. Experimental Results

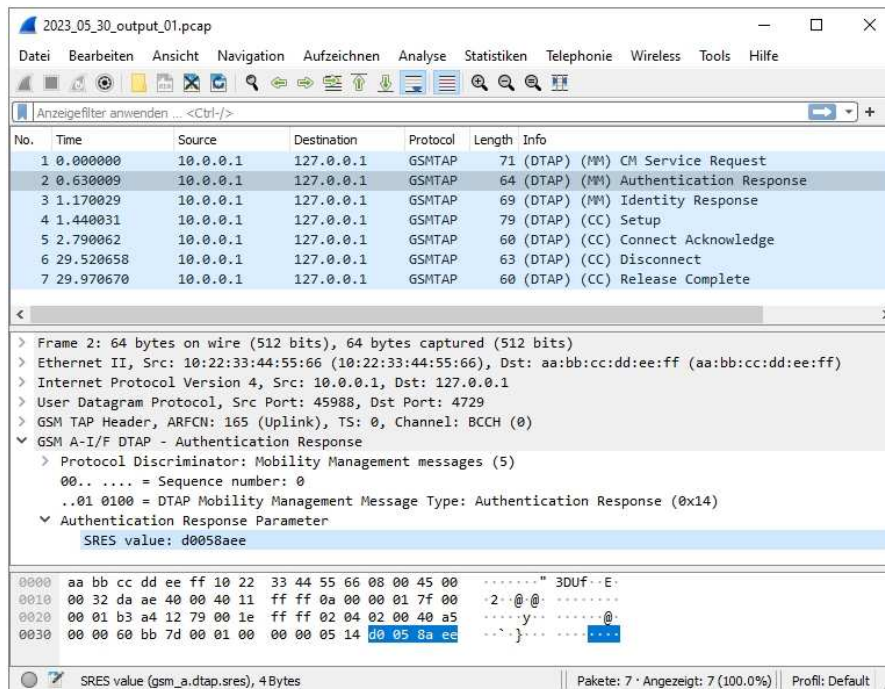


Figure 4.3.: .pcap file opened with Wireshark.

In the first recordings we received both uplink and downlink signals. Although we captured authentication packets in the .pcap files, they were exclusively uplink packets containing the SRES. We took the SRES and searched for its character string in the

4. Outdoor Field Experimentation

respective .parsed file. The initial discovery was that each occurrence of a SRES in the .parsed file is preceded by "[05.14.".

In Figure 4.3, we opened a .pcap file containing one of our recordings with Wireshark. While the uplink authentication response has been identified, the corresponding downlink authentication request is missing. The SRES value "d0.05.8a.ee" is preceded in the .parsed file by "[05.14.":

```
001 cont=0 1 ctr=000 000 len=06 0:0000 [05.14.d0.05.8a.ee]
```

We assumed that the uplink signal was drowning out the downlink signal. Therefore, we attempted to equalize the signal strength approximately to the same level by employing shadowing and increasing the distance. Shadowing was not effective, as it resulted in the loss of the uplink signals without line of sight.

In Weselberg, the maximum distance at which the line of sight was maintained was approximately 1.7 kilometers. However, the uplink signal strength remained stronger than the downlink.

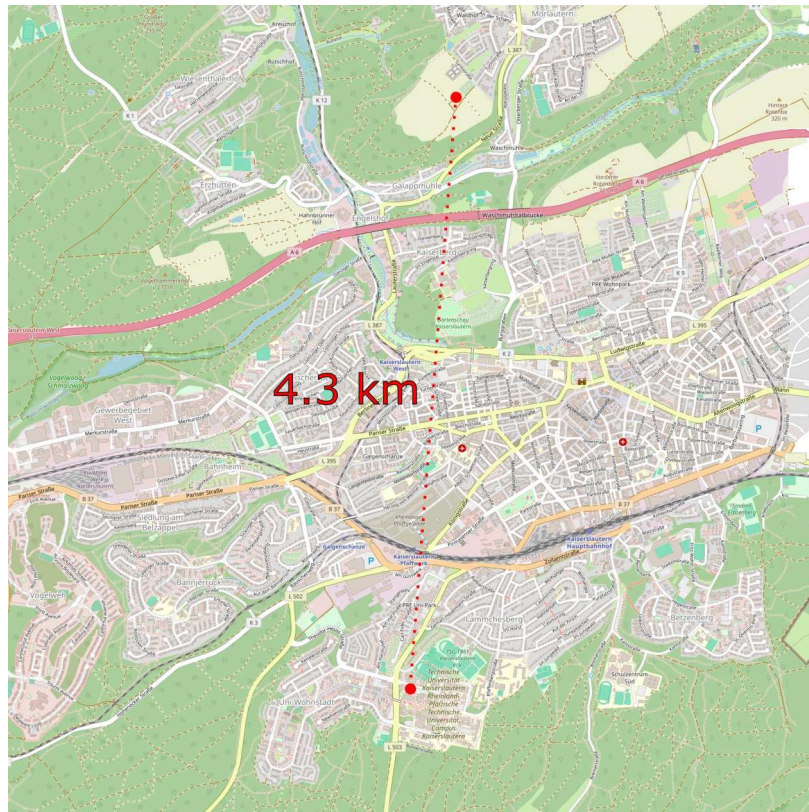


Figure 4.4.: Line of sight from Morlautern to the university parking garage roof [30].

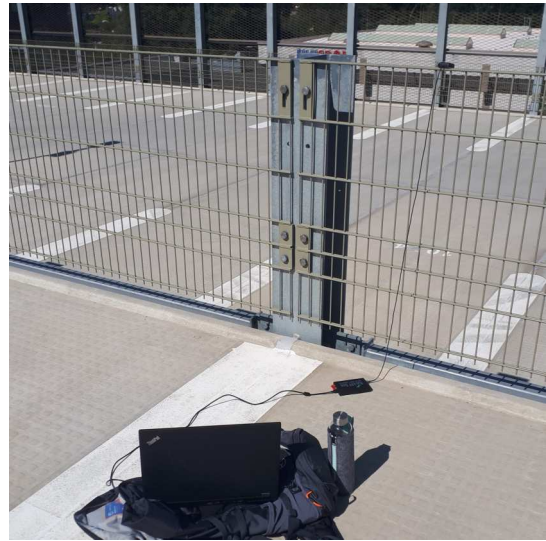
We then had two options. We could set up a second laptop with an antenna to record only the downlink. During our measurements in Weselberg, we would position the second laptop out of line of sight at the university. The second option was to find another location where we could extend the distance further. We opted for the second

option, as the use of a second laptop would generate large files and pose challenges in merging the signals later. As we wanted to associate each RAND in the downlink with the corresponding SRES in the uplink, the timestamps had to precisely match.

As a new location, we found an open field in Morlautern that had a direct line of sight to the university parking garage roof. The line of sight is depicted on the map in Figure 4.4. One person recorded the Iridium signals using an Iridium antenna connected via a hackRF SDR to a laptop on the parking garage roof, while the other person sent e-mails from the field in Morlautern using a smartphone connected to the Iridium network via Iridium GO!. Additionally, we mounted the Iridium GO! on a tripod for unobstructed transmission. Figure 4.5 shows the exact test setup. The distance between the antenna and Iridium GO! was approximately 4.3 km.



(a) Traffic generation: Smartphone connected to WiFi network of Iridium GO!.



(b) Interception: Iridium antenna connected via a hackRF SDR to a laptop running gr-iridium.

Figure 4.5.: Experimental setup in Morlautern.

This time, we received the authentication packets for both the uplink and the downlink. We were able to identify RAND-SRES pairs in the parsed file. The RAND in the downlink is preceded by "[05.12.". A short time later, the SRES in the uplink follows, initiated by "[05.14.". Before the actual RAND, there are consistently two hexadecimal digits that, unlike "[05.12.", are not always the same and do not belong to the RAND. The purpose of these two digits remains unclear to us. In this intercepted challenge, the actual RAND begins with "9b", and the digits of unknown purpose are "03".

[05.12.03.9b.33.63.23.a7.88.d6.34.97.3e.04.d9.e1.19.aa.67]

We wrote Python scripts to search for the strings "[05.12." and "[05.14." in multiple files, facilitating a faster identification of the authentication packets. Moreover, we

observed that the authentication process is carried out anew with each e-mail transmission when the "SEND/RECEIVE MAIL" button is pressed. Related to the status steps mentioned earlier, the authentication takes place during step 4(b)i, which is displayed as "connecting to the internet" on the smartphone. Hence, we don't need to restart Iridium GO! or login again in the Iridium user account each time but only tap on the "SEND/RECEIVE MAIL" button on the smartphone to generate new RAND-SRES pairs. A screenshot of a sample log from connecting to the mail server is provided in Figure 4.6.

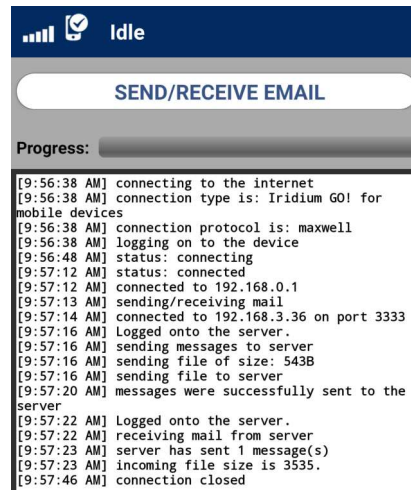


Figure 4.6.: Displayed log on the phone while sending an e-mail.

In addition, our measurements confirmed that the login credentials for Iridium user accounts can indeed be found unencrypted in the uplink. These unencrypted credentials are transmitted to the satellite for verification during every connection setup, specifically at step 4(b)i.

Below are two lines from our recorded data. The first line shows the user name and the second line shows the password in plain text. The structure of these lines is as described in Section 2.4.

```

IIP: p-1694003548 000005138.8826 1622834816 100% -25.28|-091.96|30.40 179 UL
LCW(1,T:maint,C:<silent>,00000000000000000000)
type:04 seq=021 ack=019 cs=211/OK len=031 [87.24.00.00.01.01.08.0a.00.07.
a8.46.5a.21.8c.c9.6c.6f.67.69.6e.20.65.2e.6a.65.64.65.72.6d.61]
FCS:OK/ed6d6e IP: .$. . . . . FZ!..login e.jederma
  
```

```

IIP: p-1694003548 000005150.2893 1622834816 100% -24.52|-092.27|29.72 179 UL
LCW(1,T:maint,C:<silent>,00000000000000000000)
type:04 seq=022 ack=019 cs=210/OK len=031 [6e.6e.20.6b.61.65.73.65.6b.75.
63.68.65.6e.2e.31.33.33.37.21.20.61.31.2e.30.2e.30.31.31.38.20]
FCS:OK/9bfa24 IP: nn kaesekuchen.1337! a1.0.0118
  
```

5. Key Extraction

5.1. Experimental Setup

As previously mentioned, the authentication process of Iridium is directly adapted without any modifications from the GSM specifications, and the last ten bits of K_c are always set to zero [3]. Both the GSM algorithms COMP128-1 and COMP128-2 share the common characteristic of consistently setting the last ten bits of the 64-bit ciphering key to zero. Hence, it can be presumed that Iridium implements either COMP128-1 or COMP128-2.

In our investigation, our approach included the execution of a collision attack, a method exclusively applicable to COMP128-1. A successful attack would prove the utilization of COMP128-1. Conversely, an unsuccessful attack would serve as an indication that either COMP128-2 or a modified version of COMP128-1 is likely implemented. The success of the attack is determined by the complete recovery of the secret key K_i .

We used Woron Scan v1.09 and Sim Scan v2.01 for the key recovery attack. Both programs are Windows executable (.exe) files that implement an optimized version of the collision attack described in Section 2.2.3. The scanning process, which lasts from 1 to 120 minutes per SIM card, is influenced by factors such as the SIM card reader, the scanning frequency (higher frequencies result in up to twice the scanning speed), and the scanning software employed. The outcome of the scanning process includes two codes, K_i and IMSI.

Some providers implement request limitations on the COMP128-1 algorithm to mitigate collision attacks. It is essential that this threshold exceeds the regular lifetime request count of a SIM card to ensure that legitimate users are not impacted. Many operators in the USA, for example, set it to 65,535 [31]. Exceeding this limit can lead to the blocking of the SIM card, requiring replacement with a new one from the operator. To prevent repeated reading of SIM cards, the K_i and IMSI codes found by the programs should be securely stored in a location inaccessible to others.

Woron Scan allows faster scanning than Sim Scan, but it occasionally encounters errors with certain SIM cards. It is advisable to initiate scanning with Woron Scan, and if any challenges arise, consider using Sim Scan as an alternative. If the SIM card is to be duplicated, we recommend to use SIM Scanner v5.15, which we will revisit later in Chapter 6.

Since GSM technology and its research are somewhat older, we had to use card readers and programs, in part, on Windows XP. In our experimentation, we ordered new Iridium SIM cards, and we utilized four different SIM card and smart card readers of various price ranges:

- Fdit USB SIM Card Reader (Figure 5.1a)
- Simore TS-120P SIM Card Reader (Figure 5.1b)
- HID OMNIKEY 6121 Mobile (Figure 5.1c)
- CSL - USB Smart Card Reader (Figure 5.1d)



Figure 5.1.: Card readers used in our experiments.

Furthermore, we used osmo-sim-auth [32] at various points, which is essentially a Python script designed for utilization with a PC-based smart card reader to retrieve GSM authentication parameters from a SIM card. osmo-sim-auth can be executed in the console, where the 16-byte RAND value is provided as a 32-hex-digit command-line argument. The "-s" flag is set to enable SIM mode.

```
$ ./osmo-sim-auth.py -r 00000000000000000000000000000000 -s
```

The script will then execute the authentication algorithm on the inserted SIM card and return the SRES.

```
Testing SIM card with IMSI 901033260418222
```

```
GSM Authentication
```

```
SRES:      b121340f
```

```
Kc:       b3fbed0360d83400
```

The script thus allows the generation of RAND-SRES pairs for a SIM card without needing to know the secret key Ki.

5.2. Experimental Results

Woron Scan was able to recover the secret key Ki in 20,711 steps as shown in Figure 5.2. We achieved the best results with the HID OMNIKEY card reader on Windows 10. Here, Woron Scan completed the task in only 6 minutes. However, the CSL card reader also took only just under 10 minutes on Windows 10. The program terminated with the recovered Ki and IMSI as output.

KI : CB 0C 65 0C 2F B1 51 AE FB 8D D2 1A 9C 33 0A BC
IMSI: 08 99 10 30 23 06 14 28 22

The complete protocol of the secret key Ki extraction can be saved as a .wrn file and later reopened with Woron Scan.

The successful extraction of the key proves that COMP128-1 is implemented on the Iridium SIM cards without any modifications, since the Ki search of Woron Scan is only applicable to COMP128-1. Further, we repeatedly conducted the collision attack with Woron Scan and Sim Scan on the same Iridium SIM card and could not determine any request limit, such as after 60,000 or 100,000 attempts, beyond which the card would become unusable.

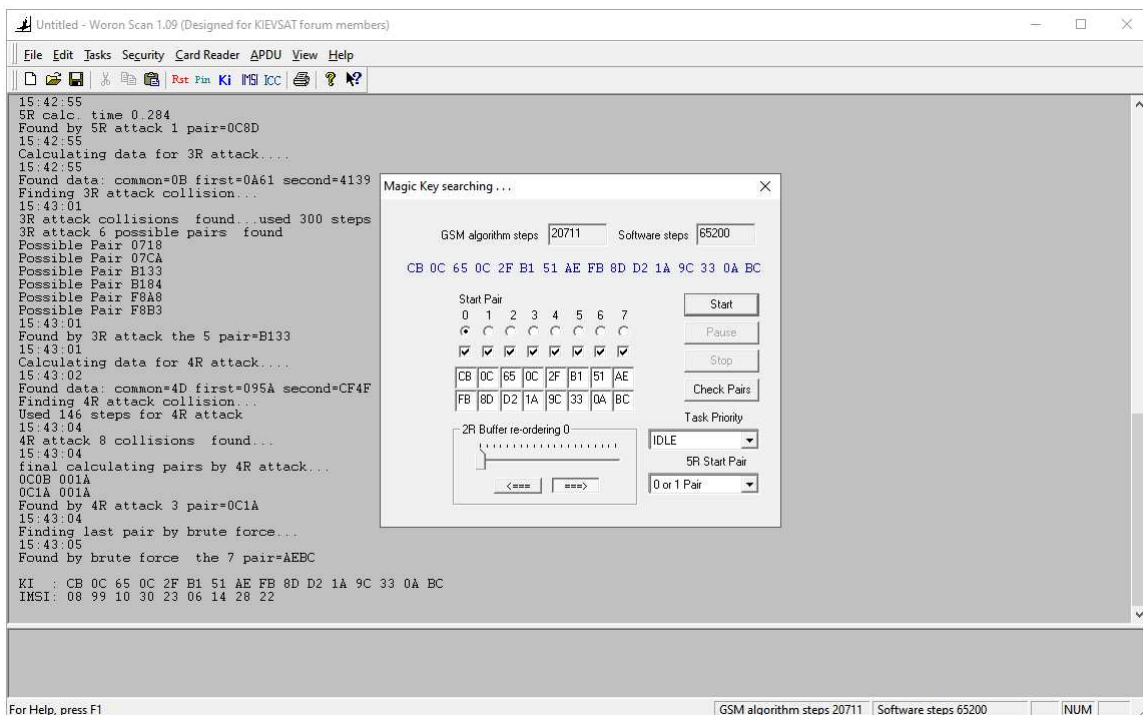


Figure 5.2.: Woron Scan executed on Windows 10 with HID OMNIKEY card reader.

Not every SIM card and smart card reader was recognized by every program on every operating system. We used the card readers on Windows XP, Windows 10, and Ubuntu. We were able to use Woron Scan on both Windows XP and Windows 10.

Sim Scan was utilized on Windows XP, and osmo-sim-auth on Ubuntu 22.04 LTS. A comprehensive overview of compatibility is provided here.

- Woron Scan v1.09
 - Windows XP
 - * Simore TS-120P SIM Card Reader
 - Windows 10
 - * HID OMNIKEY 6121 Mobile
 - * CSL - USB Smart Card Reader
- Sim Scan v2.01 and SIM Scanner v5.15
 - Windows XP
 - * Fdit USB SIM Card Reader
- osmo-sim-auth
 - Ubuntu 22.04 LTS
 - * HID OMNIKEY 6121 Mobile
 - * CSL - USB Smart Card Reader

It seems that Sim Scan cannot handle high-frequency card readers, as this error message appears with newer card readers: "CAUTION: If you use the high frequency reader. Please change to normal one instead.". However, the old card reader from Fdit was not recognized on Windows 10, which is why we could only use Sim Scan with the low frequency card reader from Fdit on Windows XP.

Sim Scan therefore took significantly longer (42 minutes), but ultimately required about the same number of requests (20,644). During the execution, we got the impression that Woron Scan also performed the search for the key bytes on the computer more quickly.

Woron Scan, on the other hand, was able to process higher frequencies and could be used with the respective card readers on both Windows XP and Windows 10.

Having established that COMP128-1 was implemented without any modifications, we proceeded to implement the open-source program `comp128.c` of the algorithm provided by Osmocom [33]. We set the reconstructed key K_i , obtained through Woron Scan, as a fixed input in the program and we modified the code to take the RAND input as a console argument. Our slightly modified version `comp128-1.c` can be found in Appendix A.1.

We used osmo-sim-auth to verify the SRES computed by `comp128-1.c`. We computed the SRES for the same RAND first entirely in software and then in hardware, specifically on the Iridium SIM card. We utilized the HID OMNIKEY 6121 Mobile and CSL


```

Terminal
File Edit View Search Terminal Help
~$ ./comp128-1 2F8A7D4B6E1C593025A1BCD987E654CF
SRES: 86648F0B
Kc: D290020FF8349800

~$ ./osmo-sim-auth.py -r 2F8A7D4B6E1C593025A1BCD987E654CF -s
Testing SIM card with IMSI 901033260418222

GSM Authentication
SRES: 86648F0B
Kc: D290020FF8349800

~$

```

Figure 5.3.: Execution of COMP128-1 in both software and hardware using the same challenge. The first command runs `comp128-1.c`, while the second command executes the algorithm directly on the SIM card.

- USB Smart Card Reader as card readers, as the remaining two were not recognized in Ubuntu.

Figure 5.3 shows the command-line output. In the first command, `comp128-1.c` is executed with an arbitrary RAND, and in the second command, the authentication algorithm on the Iridium SIM card is executed using the same RAND. With our software-implemented COMP128-1 algorithm, we obtained the same SRES and Kc as calculated by the Iridium SIM card. Thus, we completely replicated the authentication algorithm implemented on the Iridium SIM cards in software. Hence, Iridium has also adopted the compression tables listed in Appendix A.1.

In summary, it can be stated that the secret key K_i of an Iridium SIM card can be extracted using a collision attack, but this process requires physical access to the SIM card. In addition, the attack can be repeated as often as required, as we could not find any request limits.

Since Iridium adapted COMP128-1 from GSM without any modifications, the side channel attacks on COMP128-1 from Section 3.1 should also be applicable if one possesses the appropriate hardware. The side channel attacks also require physical access to the SIM card, as they involve measurements such as power consumption.

It should also be noted that a radio-based collision attack could potentially be carried out with a fake base station. Further exploration of potential radio-based attacks is discussed in Chapter 8.

6. SIM Card Cloning

6.1. Experimental Setup

Given that we were certain that COMP128-1 is implemented on Iridium SIM cards without any modifications, the question arose as to whether we could clone Iridium SIM cards. GSM SIM cards implementing COMP128-1 can indeed be cloned. The GSM specification for SIM cards is widely accessible, and the sole requirement for cloning a SIM card is the 128-bit COMP128 secret key Ki that is embedded in the card [34].

In our setup, we utilized SIM Scanner v5.15 on Windows XP and the Fdit USB SIM Card Reader. As the writable SIM card, we selected the SIM MAX 12 in 1 Card, which even provides the option of combining 12 frequently used mobile numbers from different regions, countries, or networks into a single SIM card.

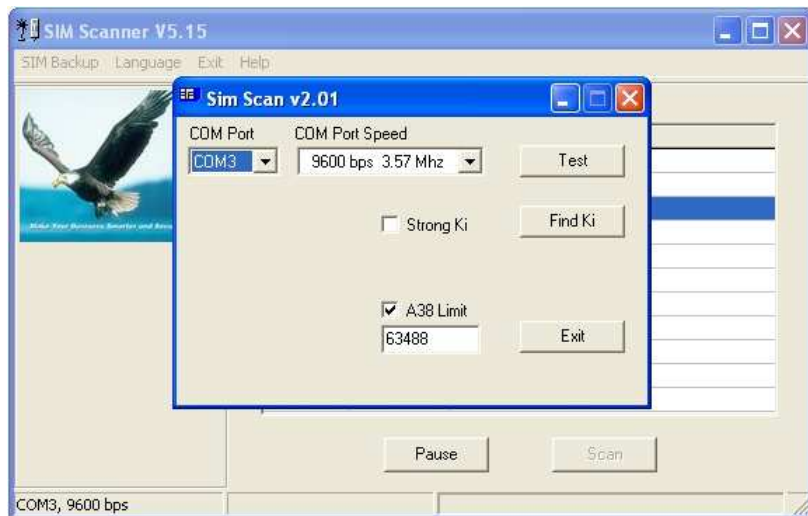


Figure 6.1.: SIM Scanner v5.15 and Sim Scan v2.01.

In Chapter 5, we successfully extracted the key Ki using Woron Scan and saved the output in a .wrn file. However, Woron Scan does not provide an option to write to SIM cards. On the other hand, SIM Scanner v5.15 is capable of writing to SIM cards but lacks the functionality to input the key Ki directly and cannot read .wrn files. Instead, the key extraction process must be executed again. As shown in Figure 6.1,

SIM Scanner v5.15 incorporates the program Sim Scan v2.01 for this purpose, which saves the key extraction results in a .dat file. SIM Scanner v5.15 can then read the .dat files and write their information to writable SIM cards. As already discussed in Chapter 5, SIM Scan can only handle low-frequency card readers, which is why we used the Fdit card reader on Windows XP.

To confirm the success of the SIM card cloning process, we once again utilized the osmo-sim-auth library and checked whether the original and cloned SIM cards, when provided with the same RAND, compute the same SRES. For this task, we used the CSL card reader with the Ubuntu 22.04 LTS operating system. Due to legal restrictions, we refrained from testing a successfully cloned SIM card by inserting it into an Iridium GO! device and thereby directly authenticating it within the Iridium network.

6.2. Experimental Results

We successfully reran the key extraction process using Sim Scan and exported a .dat file. Sim Scan required 20,664 steps and the task took 42 minutes to complete on Windows XP with the Fdit card reader.

Afterwards, we swapped the SIM cards and inserted the SIM MAX card into the Fdit card reader. In the "Write to SIM" screen, we proceeded to read the SIM MAX card. Subsequently, twelve slots are displayed, allowing us to write the data from the .dat file onto them and we wrote the data on the first slot, as shown in Figure 6.2.

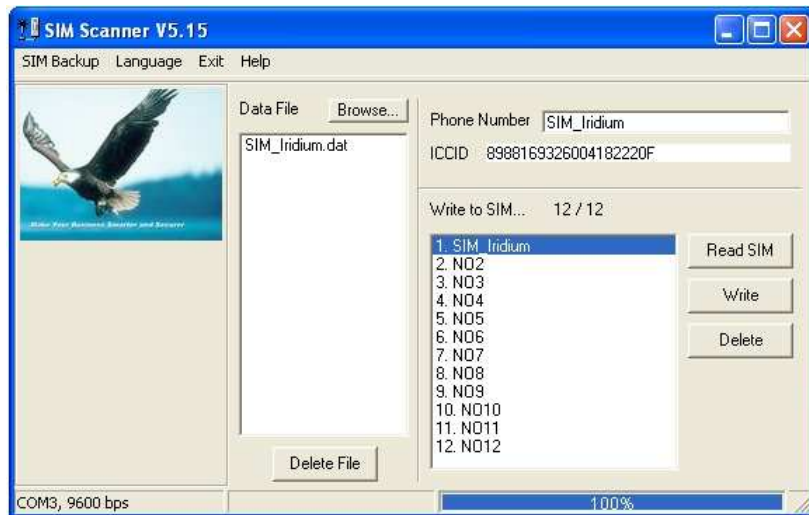
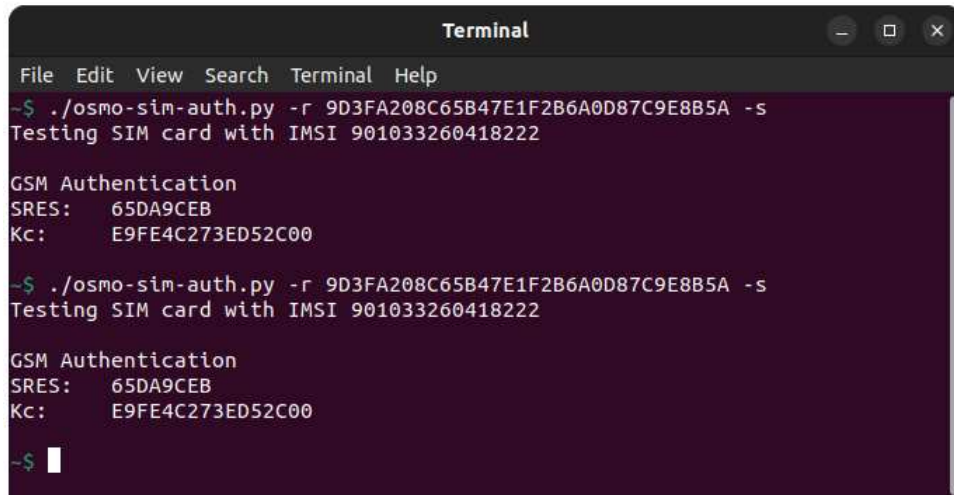


Figure 6.2.: Writing SIM MAX card with SIM Scanner v5.15.

We switched to Ubuntu and inserted the Iridium SIM card in the CSL card reader. We first executed the authentication algorithm COMP128-1 on the Iridium SIM card with

an arbitrary RAND. Then, we replaced the Iridium SIM card with the SIM MAX card in the card reader and initiated another authentication request with the same RAND.

The cloned SIM MAX card was recognized and produced the same SRES as the Iridium SIM card. The commands and outputs of both SIM cards are provided in Figure 6.3. Hence, it is established that Iridium SIM cards can be cloned and replicated arbitrarily. Due to the reusability of the .dat file, the original Iridium SIM card only needs to be read once and can then be cloned indefinitely.



```

Terminal
File Edit View Search Terminal Help
~$ ./osmo-sim-auth.py -r 9D3FA208C65B47E1F2B6A0D87C9E8B5A -s
Testing SIM card with IMSI 901033260418222

GSM Authentication
SRES: 65DA9CEB
Kc: E9FE4C273ED52C00

~$ ./osmo-sim-auth.py -r 9D3FA208C65B47E1F2B6A0D87C9E8B5A -s
Testing SIM card with IMSI 901033260418222

GSM Authentication
SRES: 65DA9CEB
Kc: E9FE4C273ED52C00

~$

```

Figure 6.3.: Execution of COMP128-1 on the Iridium SIM card (first command) and on the cloned SIM MAX card (second command).

When connecting to the web server, the Iridium device transmits the current user's login credentials, including the username and password, in an unencrypted format. Thus, during a permission check, it is possible to intercept the username and password of a targeted Iridium account with an Iridium antenna nearby.

SIM card cloning combined with intercepting unencrypted Iridium account data can result in a complete takeover of an Iridium account. When the RECORD attack by Jedermann et al. [5] is also added, there is a risk of systematic attacks for account takeovers.

For instance, an attacker might pose as an online retailer, distributing SIM cards from which the secret key Ki has been previously extracted. The Ki IMSI pairs could be stored in a database along with the corresponding .dat files. Furthermore, an attacker could intercept the static TMSI used by Iridium during the authentication attempt. This can be done by inserting a SIM card, not yet assigned to any account, into an Iridium device and intercepting the unencrypted TMSI in the downlink. The attacker could then wait for the TMSIs of the distributed SIM cards to go online and locate them using the RECORD attack. Afterwards, the attacker could intercept the uplink at the victim's location to discover the credentials. Finally, with a cloned SIM card from the corresponding .dat file, the attacker can take over the entire account.

Hence, the extraction and cloning of Iridium SIM cards pose a substantial real-world threat, granting potential attackers broad access to users' communication and identity. The following are some security threats:

- **Unauthorized Access:** Unauthorized individuals can gain access to the Iridium network using a cloned Iridium SIM card and intercepted Iridium account credentials, potentially leading to unauthorized use of communication services.
- **Identity Theft:** Cloning a SIM card enables an attacker to impersonate the legitimate user, engaging in fraudulent activities or even committing identity theft.
- **Eavesdropping:** A cloned SIM card grants attackers the ability to intercept calls and eavesdrop on conversations and messages intended for the legitimate user, posing a significant threat to the confidentiality of sensitive information.
- **Denial of Service:** Cloning can lead to service disruption for the legitimate user as both the original and cloned SIM cards may attempt to use the Iridium network simultaneously, causing conflicts and potential service interruptions.
- **Financial Losses:** The unauthorized use of Iridium services by attackers can lead to financial losses for the legitimate user.

The outlined security threats underscore importance of implementing strong protective measures and encryption protocols. The risks associated with unauthorized access, identity theft, eavesdropping, service disruptions, and financial losses highlight the multifaceted nature of the security threats and the necessity for a more robust authentication algorithm on Iridium SIM cards.

7. Code Analysis of COMP128-1

7.1. Experimental Motivation

In Chapter 5, we demonstrated that Iridium still uses the outdated COMP128-1 algorithm of the GSM standard for authentication, and in Chapter 6, we were even able to clone SIM cards. In both Chapters, we utilized Woron Scan and Sim Scan, which leverage a collision attack, capitalizing on the fact that the hash function of COMP128-1 is not considered secure.

The purpose of this Chapter is to perform an in-depth examination of the COMP128-1 algorithm and its hash function. The objective was to identify vulnerabilities through detailed code analysis and to determine whether current methods can extract K_i , the secret key of the SIM card, more efficiently. An implementation of COMP128-1 by osmocom is provided in Appendix A.1 for reference.

The hash function of COMP128-1, with its five levels, is executed a total of eight times, as we described in detail in Section 2.2.2. Our approach was to analyze the algorithm from the perspective of its output, aiming to determine the range of possible inputs that could lead to a given output. To accomplish this, we initially examined the entire round structure in more detail. Furthermore, we considered each level of the hashing function individually and then a complete execution of the hashing function, including all five levels.

The output of the COMP128-1 algorithm includes both SRES and K_c . However, as we demonstrated in Chapter 4, we were only able to intercept the RAND in the satellite's downlink and the SRES, which is then sent back in the uplink from the Iridium device to the satellite for authentication. This means that we could only know 32 bits of the total 128-bit output produced by COMP128-1. This limitation posed a significant challenge in our analysis, as we had access to only a fraction of the output.

Thus, the motivation for our research in this Chapter was to determine whether K_i could be reconstructed using only a few intercepted RAND-SRES pairs. The primary computations for this attack would take place on the attacker's computer. This would constitute a passive attack conducted solely through radio signals, eliminating the need for physical access to the SIM card. In Chapter 8, we further discuss the potential of conducting an active attack via radio signals.

7.2. Experimental Results

When analyzing the COMP128-1 algorithm from an output perspective, there are a certain number of possibilities leading to a certain output. As illustrated in the flowchart of Figure 2.5, K_i is prefixed to x'' in each round to restore the input for the hashing function to 256 bits. Hence, K_i is used as a static value in each of the eight rounds. Therefore, we initially considered calculating a certain number of possibilities for the last two rounds in reverse, we can discard all possibilities where the first 128 bits only appear in one round. This pattern applies to all rounds. Any possibilities where the first 128 bits of the input to the hashing function do not appear in all eight rounds can be discarded, regardless of the dimension of the number of possibilities.

The permutation block is located between the hashing functions of the individual rounds. During the permutation, each bit of the 128-bit value of x' is individually shifted, and each position k is moved to a new position k' with $k' = f(k) = (k \cdot 17) \bmod 128$. In addition, the newly generated 128-bit sequence is then divided into bytes. Within each byte, the order of the bits is reversed, resulting in x'' .

The function $f(k)$ is an invertible function, since 17 and 128 are coprime, meaning they have no common divisors other than 1. To specify the inverse function, we search for the multiplicative inverse of 17 modulo 128, which can be found by using the extended Euclidean algorithm. In this case, 113 is the multiplicative inverse since $(17 \cdot 113) \bmod 128 = 1$. Hence, the first part of the permutation can be reversed using the inverse function $f^{-1}(k') = (k' \cdot 113) \bmod 128 = k$. Reversing the order of the bits in a byte, essentially applies a deterministic transformation to the arrangement of the bits. This one-to-one mapping is bijective and can be inverted by applying the same transformation again. In mathematical terms, the function g that reverses the bits in a byte is its own inverse, denoted as $g = g^{-1}$. Therefore, both parts of the permutation block are invertible, making the complete permutation block invertible.

Algorithmically, a single lookup table can be used to store for each of the 128 positions of x'' , from which position of x' the value originated. To reverse the permutation, the positions can be swapped back according to the lookup table.

In the following, we analyze the computation of the output of COMP128-1 in the final step. For this purpose, we will examine the computation in the final stage, level 4 in the 8th round of COMP128-1. The red rectangle in Figure 7.1 highlights where the computation of the output in level 4 occurs. The 16 adjacent pairs in level 4 are interconnected using butterfly compression, resulting in the 16 adjacent pairs in the output. The output then consists of 32 hexadecimal (4-bit) positions. The butterfly compression for level 4 is as follows.

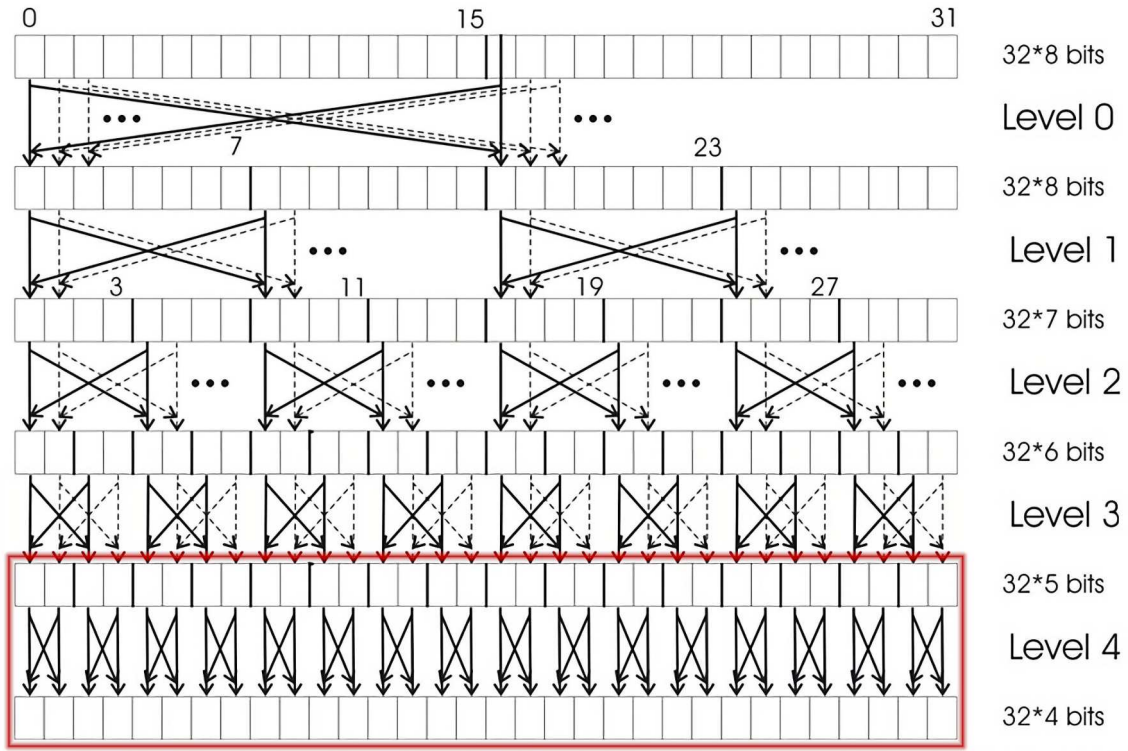


Figure 7.1.: Butterfly compression of the hashing function at level 4 [9].

$$\begin{aligned}
 y &= (X[i] + 2 \cdot X[i + 1]) \mod 32 \\
 z &= (2 \cdot X[i] + X[i + 1]) \mod 32 \\
 X[i] &= T[4][y] \\
 X[i + 1] &= T[4][z] \\
 i &= 0, 2, 4, \dots, 26, 28, 30
 \end{aligned}$$

Table 7.1 shows the compression table for level 4. Each number from 0 to 15 occurs twice in the table with 32 positions, each representing a hexadecimal digit. From the compression table, the entry at position y is written to position i of the output. The entry of the compression table at position z is written to position $i + 1$ of the output.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
15	12	10	4	1	14	11	7	5	0	14	7	1	2	13	8	10	3	4	9	6	0	3	2	5	6	8	9	11	13	15	12

Table 7.1.: Compression table of level 4 used in the hashing function.

In the following, we will consider a specific example. We assume that the output of COMP128-1, namely the calculated SRES, begins with the digits 6 and 8. Our goal is

to determine which entries in level 4 can compute the pair (6,8). For this purpose, it is necessary to consult the compression table to determine the positions at which 6 and 8 occur. Table 7.1 indicates that 6 appears at positions 20 and 25, and 8 appears at positions 15 and 26. Therefore, y could have been 20 or 25, and z could have been 15 or 26 in level 4. If we substitute the two possibilities for y and the two possibilities for z into the butterfly compression, we obtain four equations. For each of these four equations, we can calculate a solution set and then intersect the solution sets of y with those of z .

$$\begin{aligned} 20 &= (X[i] + 2 \cdot X[i + 1]) \mod 32 \\ 25 &= (X[i] + 2 \cdot X[i + 1]) \mod 32 \\ 15 &= (2 \cdot X[i] + X[i + 1]) \mod 32 \\ 26 &= (2 \cdot X[i] + X[i + 1]) \mod 32 \end{aligned}$$

$$\begin{aligned} X[i] &\in 0, \dots, 31 \\ X[i + 1] &\in 0, \dots, 31 \end{aligned}$$

$$Y_{20} := \{(24, 30), (4, 8), (16, 2), (10, 21), (0, 10), (28, 12), (22, 31), (18, 1), (30, 27), (12, 20), (6, 7), (2, 9), (26, 29), (20, 0), (24, 14), (4, 24), (8, 6), (16, 18), (14, 19), (22, 15), (18, 17), (6, 23), (26, 13), (10, 5), (0, 26), (8, 22), (28, 28), (30, 11), (12, 4), (2, 25), (14, 3), (20, 16)\}$$

$$Y_{25} := \{(13, 22), (25, 16), (19, 19), (27, 15), (9, 8), (21, 2), (3, 11), (17, 20), (7, 25), (11, 23), (31, 13), (5, 26), (15, 5), (23, 17), (25, 0), (19, 3), (1, 28), (9, 24), (3, 27), (31, 29), (29, 30), (13, 6), (23, 1), (27, 31), (21, 18), (17, 4), (7, 9), (11, 7), (5, 10), (1, 12), (15, 21), (29, 14)\}$$

$$Z_{15} := \{(4, 7), (9, 29), (20, 7), (13, 21), (24, 31), (5, 5), (0, 15), (28, 23), (27, 25), (6, 3), (10, 27), (21, 5), (16, 15), (30, 19), (22, 3), (25, 29), (7, 1), (14, 19), (1, 13), (29, 21), (31, 17), (3, 9), (23, 1), (17, 13), (15, 17), (19, 9), (26, 27), (2, 11), (8, 31), (12, 23), (11, 25), (18, 11)\}$$

$$Z_{26} := \{(18, 22), (14, 30), (7, 12), (4, 18), (9, 8), (10, 6), (29, 0), (2, 22), (21, 16), (23, 12), (25, 8), (26, 6), (6, 14), (12, 2), (3, 20), (8, 10), (22, 14), (28, 2), (17, 24), (20, 18), (24, 10), (1, 24), (11, 4), (27, 4), (16, 26), (19, 20), (31, 28), (0, 26), (5, 16), (15, 28), (30, 30), (13, 0)\}$$

For each of the four equations, the solution sets were determined by systematically testing all potential values within the defined range. Specifically, by substituting every possible number from 0 to 31 for $X[i]$ and $X[i + 1]$ into the equations, we can compute the respective outcomes. For calculating the solution sets, we utilized the Python

functions `calculate_solution_set_y` for the y -equation and `calculate_solution_set_z` for the z -equation, which are listed in Appendix A.2.

A solution set contains all pairs $(X[i], X[i + 1])$ that satisfy the respective equation. When we intersect a solution set for the y -equation with a solution set from the z -equation, we find that exactly one pair remains that appears in both sets.

$$Y_{20} \cap Z_{15} = \{(14, 19)\}$$

$$Y_{20} \cap Z_{26} = \{(0, 26)\}$$

$$Y_{25} \cap Z_{15} = \{(23, 1)\}$$

$$Y_{25} \cap Z_{26} = \{(9, 8)\}$$

Therefore, there are four possibilities for the positions i and $i + 1$ in level 4, from which the 6 at position i and 8 at position $i + 1$ in the output can be calculated. In fact, it generally is true for each level that exactly one possibility remains when intersecting two solution sets. Hence, for every pair of any given level, there are exactly four possibilities for the pair of the previous level that computed it. We demonstrated algorithmically through empirical testing that exactly four possible origin pairs were computed for every combination of pairs in each level. The code for this proof can be found in Appendix A.2.

In each level, each possibility generates four additional possibilities. Therefore, the back-calculation starting from the output results in $4^5 = 1024$ possibilities for a pair in level 0. Consequently, for all 16 pairs in level 0, there are a total of $1024^{16} = 2^{160}$ possibilities as input for the hashing function that could have produced a specific output.

The search space has already been narrowed down significantly, as there are 2^{256} possibilities for the input of the hashing function with complete brute force. Since in each round K_i is in the first half of the input to the hashing function, it is already included in the 2^{160} possibilities that emerge after the first back-calculation. As described above, one approach could be to consistently compare the first half of the input of the hashing function across rounds to further restrict the search space. To achieve this, however, the number of possibilities from the last round would need to be further reduced. Another approach would be to minimize the number of possibilities to such an extent that a brute force attack becomes feasible after back-calculating the last round. Either way, 2^{160} possibilities are still too many and must be reduced by analyzing further dependencies that occur in the butterfly compression.

To demonstrate that there are dependencies between the levels of the hashing function, we back-calculated from the output to level 3. We considered blocks of four, meaning two pairs, because these are interconnected independently of each other from level 3 to the output. For each pair, there are $4^2 = 16$ possibilities for its generation. Therefore, when considering two pairs, there are $16 \cdot 16 = 256$ possibilities.

We considered all possible values for two pairs, which form a block of four hex-digits in the output. This leads to $16^4 = 65,536$ potential combinations, for each of which

we back-calculated levels 4 and 3. This resulted in 256 possible inputs in level 3 for each of the 65,536 possibilities in the output. We then calculated forward from level 3 to determine how many of the 256 possibilities actually generate the respective output. Of approximately 97% of the 65,536 output possibilities, exactly 64 of the 256 possibilities led to the specific output. Only in 25 out of the 65,536 cases of the output did all 256 possibilities actually lead to this output. Table 7.2 contains the precise figures for this experiment, while Appendix A.3 includes the corresponding code.

Number of calculations generating the original output	Total number of occurrences	Total percentage share
64	63,561	96.99%
96	1,436	2.19%
128	514	0.78%
256	25	0.04%

Table 7.2.: Statistical evaluation of the 65,536 possibilities of a four-digit block in the output. It was conducted to determine, for each possible output, how many of the corresponding 256 possibilities in level 3 actually generate the respective output.

Hence, we demonstrated that when back-calculating from the output to level 3, in 97% of the cases, $\frac{3}{4}$ of the 256 possibilities can be discarded. This indicates a strong hidden dependency between the levels, which can further reduce the total number of 2^{160} possibilities. The code in Appendix A.3 can be similarly extended to also back-calculate the remaining levels.

Brute force attacks, currently conceivable on the scale of around 2^{60} , could be enabled through this process. However, as previously mentioned, we can only intercept the SRES, which constitutes 8 hex-digits of the 32-hex-digit output of COMP128-1. When we examine the butterfly compression in Figure 7.2 more closely, we see that the back-calculation of possibilities only reaches up to level 2. This is because the back-calculation up to level 2 occurs in independent blocks of eight. However, we only know the first eight-block of the output. For the back-calculation from level 2 to level 1, we must assume all possibilities for positions 8 to 31, which would result in brute force. Nonetheless, with each additional intercepted SRES, the search space could be further narrowed, eventually compensating for the fact that the latter $\frac{3}{4}$ of the output of COMP128-1 is unknown.

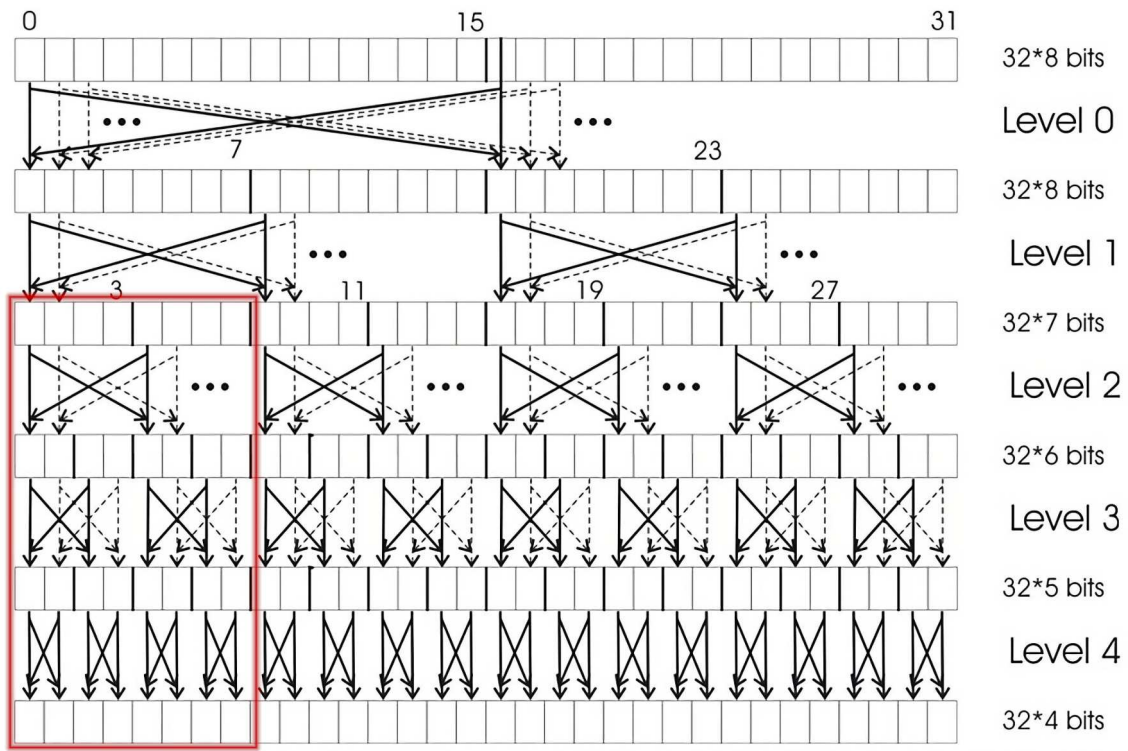


Figure 7.2.: Butterfly compression of the hashing function. Calculation of SRES starting from level 2 [9].

8. Indoor Radio Experimentation

8.1. Experimental Setup

Physical access to the SIM card is required to determine the secret key K_i through a collision attack, as described in Chapter 5. The attack is then executed directly on the SIM card, which is connected to a laptop via a card reader running a program like Woron Scan or Sim Scan. In this Chapter, we investigate whether such a collision attack can be carried out entirely by radio, without the need for physical access to the SIM card.

The initial idea was to capture the raw data of an Iridium radio transmission and then isolate the authentication request. For this purpose, we sent e-mails from a smartphone connected to an Iridium GO! device in the field in Morlautern and intercepted the radio transmission on the university parking garage roof with an Iridium antenna connected to a laptop via hackRF SDR. The experimental setup for generating the raw data therefore corresponds to that in Figures 4.4 and 4.5. The completely isolated authentication request would then be replayed to the Iridium GO!, suggesting a kind of replay attack. If the Iridium GO! responds to the authentication request with an authentication response, it would be an indication that the collision attack could potentially be executed by radio.

Due to legal regulations and protected frequency bands, our radio transmissions were conducted in shielded environments to ensure that no signals could leak outside and interfere with external communication systems.

The motivation for this Chapter is to assess the feasibility of creating a fake base station, similar to known IMSI-catcher attacks in GSM, in the form of a satellite simulator. The purpose of such a simulator, which could only imitate the basic functions, would be to trick the Iridium GO! device into believing it is communicating with a legitimate Iridium satellite. Instead of a single authentication request with one RAND, the fake satellite would transmit thousands of authentication requests, theoretically prompting Iridium GO! to respond with the corresponding SRES each time. This eliminates the need for physical access to the SIM card, requiring only proximity to the Iridium GO! device.

8.2. Experimental Results

In our recordings, we again captured both the RAND in the downlink and the SRES in the uplink. We initially split the .raw file in half using the Linux split command and then checked which half contained the authentication request. However, during the parsing of the halved files, we found that the authentication request could be entirely lost when splitting, especially if it was near the cutting edge of the file. This is likely due to the signals fading in and out, and the data around the signals is also needed for correct parsing. Therefore, we used the Linux head and tail commands to trim the file from the front and back until almost only the following authentication request remained.

```
IDA: p-1694003548 000000714.5535 1622880540 92% -37.84|-091.01|18.55 179 DL
LCW(2,T:hndof,C:handoff_cand,2ab,369,010101010111101101001)
001 cont=0 1 ctr=000 000 len=19 0:0000 [05.12.03.9b.33.63.23.a7.88.d6.34.97.
3e.04.d9.e1.19.aa.67] 050f/0000 CRC:OK 0000 SBD: ....3c#...4.>.....g.
```

We replayed the isolated authentication request to the Iridium GO! in a shielded environment. Although the status of the Iridium GO! changed to "Registering," we were unable to receive an Authentication Response.

Publicly available information about the Iridium network does not specify whether the network authenticates to the user. This lack of detail leaves open the theoretical possibility of a fake base station attack, where a rogue station could impersonate a legitimate network entity.

Since Iridium has adopted the authentication process directly from GSM, and the network or base station does not authenticate itself in GSM, it is highly likely that Iridium also lacks a mechanism for authenticating the network to the client.

As the Iridium GO! responded and attempted to register on the network, a radio-based attack may be feasible. If one were to program a satellite simulator with the basic functions to impersonate a satellite, it would likely be possible to initiate a collision attack and send thousands of authentication requests to the Iridium GO!.

After extensive discussions with many knowledgeable GSM engineers, Briceno et al. [10] have come to the conclusion that over-the-air collision attacks on GSM systems are practically feasible for sophisticated attackers using a fake base station.

Oligeri et al. [25] suggest in their research that spoofing Iridium signals is a challenging task, due to various factors such as the complexity involved and the high signal strength. However, while they point out the difficulties of imitating Iridium signals, they do not categorically state that it is impossible.

If an Iridium satellite could be impersonated in its basic functions, the attack could also be combined with the work "RECORD: A REception-Only Region Determination Attack on LEO Satellite Users" [5]. The combination of geotracking and a potential collision attack via radio signals presents a significant security threat. Initially,

a user is pinpointed using geotracking. Following this, the login credentials of the victim's Iridium account can be intercepted via radio signals as they are sent unencrypted during the login process. A potential radio collision attack would allow attackers to duplicate the SIM card and ultimately take over the account. This sequence of events would not only violate location privacy, but also compromise digital identity and control.

9. Conclusion and Future Work

In this thesis, we have successfully broken the authentication process in the Iridium network. Our research revealed that Iridium still relies on the GSM standard for authentication and uses the COMP128-1 algorithm, known for its security vulnerabilities, without any modifications. We performed collision attacks against COMP128-1 using Woron Scan and Sim Scan on an Iridium SIM card, extracting the secret key Ki in just 6 minutes. This allowed us to clone the Iridium SIM card. Cloning the Iridium SIM card enables an attacker to fully compromise an Iridium account by additionally intercepting the unencrypted credentials transmitted to the satellite. However, the attacker must have physical access to the SIM card for cloning and be in close proximity to the uplink to intercept the credentials.

To circumvent physical access to the SIM card, the feasibility of executing an attack using intercepted RAND-SRES pairs was analyzed. Additionally, the alternative of performing an attack similar to an IMSI catcher was considered. In our radio experiments, we intercepted authentication packets in both the downlink and uplink of our self-generated Iridium traffic. We discovered, using gr-iridium and the iridium-toolkit, that the RAND in the downlink is initiated with "[05.12.", and the subsequent SRES in the uplink with "[05.14.". Further, we also evaluated a code analysis of COMP128-1 to determine the feasibility of passive attacks based solely on radio-intercepted RAND-SRES pairs. We demonstrated that there are strong dependencies between individual levels in the butterfly compression, which reduce the effective search space for Ki. Our shielded radio experiments indicated that collision attacks conducted purely via radio signals using a satellite simulator are conceivable as a sophisticated active attack.

In future work, conducting further code analysis using modern methods could be beneficial. However, the main challenge is that an intercepted SRES only provides 8 out of the 32 hex digits of the COMP128-1 output. The construction of a satellite simulator is another promising area for future research. This simulator would aim to replicate the basic functions of an Iridium satellite and be capable of sending authentication requests to Iridium devices. This challenging task must, of course, adhere to legal frameworks, as the Iridium frequencies are regulated. These attacks would make it possible to take over Iridium accounts without ever having physical possession of the targeted SIM card.

We could not identify even basic countermeasures against the collision attack on COMP128-1, such as a mechanism that would trigger the destruction of the SIM

card after 60,000 authentication requests, as known from other providers. Consequently, we recommend the replacement of all SIM cards for the 2.2 million global users of the Iridium network to ensure the integrity and confidentiality of their communication. The GSM Association classifies COMP128-1 as deprecated, emphasizing the need for Iridium to update its authentication algorithm. While COMP128-2 is not recommended due to known vulnerabilities, the association suggests transitioning to more secure algorithms like COMP128-3 or the preferred GSM-MILENAGE (G-MILENAGE), a GSM variant of MILENAGE based on AES, to ensure robust 2G authentication. For broader compatibility and future-proofing, operators are encouraged to adopt 3G authentication standards and support algorithms like MILENAGE and TUAK, and consider developing their own algorithms for enhanced security and customization [8]. Furthermore, Iridium should definitely encrypt the login credentials of the Iridium account in the uplink in the future.

Bibliography

- [1] Iridium Communications Inc. *Iridium Unveils Project Stardust: Developing the Only Truly Global, Standards-Based IoT and Direct-to-Device Service*. [Online; accessed January 26, 2024]. Jan. 2024. URL: <https://investor.iridium.com/2024-01-10-Iridium-Unveils-Project-Stardust-Developing-the-Only-Truly-Global,-Standards-Based-IoT-and-Direct-to-Device-Service>.
- [2] GSM Association. *Security Guidelines for UICC Profiles Version 1.0*. [Online; accessed January 26, 2024]. 2020. URL: <https://www.gsma.com/security/wp-content/uploads/2020/12/FS.27-v1.0.pdf>.
- [3] Dan Veeneman. *Decode Systems: Iridium*. [Online; accessed January 26, 2024]. 2021. URL: <http://www.decodesystems.com/iridium.html>.
- [4] Ditel Technology. *Differences between LEO and GEO Satellites*. [Online; accessed January 26, 2024]. URL: <https://www.diteltech.com/info-detail/differences-between-leo-and-geo-satellites>.
- [5] Eric Jedermann, Martin Strohmeier, Vincent Lenders, and Jens Schmitt. "RECORD: A REception-Only Region Determination Attack on LEO Satellite Users". In: *33rd Usenix Security Symposium* (2024). URL: <https://disco.cs.uni-kl.de/discfiles/publicationsfiles/JSLs23.pdf>.
- [6] Michele Klein, Eric Jedermann, and Jens Schmitt. "Security Analysis of Device-Authentication in the Iridium Satellite Network". In: 2022. URL: https://disco.cs.uni-kl.de/discfiles/completed_theses/klein22.pdf.
- [7] Osmocom. *Implementation of COMP128-2 and COMP128-3*. [Online; accessed January 26, 2024]. 2013. URL: <https://github.com/osmocom/libosmocore/blob/master/src/gsm/comp128v23.c>.
- [8] GSM Association. *Security Algorithm Deployment Guidance Version 3.0*. [Online; accessed January 26, 2024]. 2022. URL: <https://www.gsma.com/security/wp-content/uploads/2022/09/FS.35-v3.0.pdf>.
- [9] Dingfelder. *COMP128 - Kollisionsattacke*. [Online; accessed January 26, 2024]. 2002. URL: <https://www.cits.ruhr-uni-bochum.de/imperia/md/content/leander/ausarbeitungcomp128final.pdf>.
- [10] Marc Briceno, Ian Goldberg, and David Wagner. *GSM cloning*. [Online; accessed January 26, 2024]. 1998. URL: <http://www.isaac.cs.berkeley.edu/isaac/gsm.html>.

- [11] Stuart Wray. "COMP128: A birthday surprise". In: *swray@ bournemouth. ac. uk, une description* (2003).
- [12] Marcin Olawski. "Security in the GSM network". In: *IPSec. pl. Stream ciphers* (2011).
- [13] Iridium Communications Inc. *Satellites & Network*. [Online; accessed January 26, 2024]. 2024. URL: <https://www.iridium.com/media-center/satellites-network/>.
- [14] International Civil Aviation Organization. *AMS(R)S Manual Part II Version 4.0*. [Online; accessed January 26, 2024]. 2007. URL: [https://www.icao.int/safety/acp/Inactive%20working%20groups%20library/ACP-WG-M-Iridium-8/IRD-SWG08-IP05%20-%20AMS\(R\)S%20Manual%20Part%20II%20v4.0.pdf](https://www.icao.int/safety/acp/Inactive%20working%20groups%20library/ACP-WG-M-Iridium-8/IRD-SWG08-IP05%20-%20AMS(R)S%20Manual%20Part%20II%20v4.0.pdf).
- [15] Iridium Communications Inc. *Iridium Declares Victory; \$3 Billion Satellite Constellation Upgrade Complete*. [Online; accessed January 26, 2024]. Feb. 2019. URL: <https://investor.iridium.com/2019-02-06-Iridium-Declares-Victory-3-Billion-Satellite-Constellation-Upgrade-Complete>.
- [16] Stefan Zehl and "schneider". *gr-iridium*. [Online; accessed January 26, 2024]. Chaos Computer Club München, 2016. URL: <https://github.com/muccc/gr-iridium>.
- [17] Stefan Zehl and "schneider". *iridium-toolkit*. [Online; accessed January 26, 2024]. Chaos Computer Club München, 2014. URL: <https://github.com/muccc/iridium-toolkit>.
- [18] Stefan Zehl and "schneider". *iridium-toolkit*. [Online; accessed January 26, 2024]. Chaos Computer Club München, 2014. URL: <https://wiki.muc.ccc.de/iridium:toolkit>.
- [19] Stefan Zehl and "schneider". *Iridium Satellite Hacking*. Presented at The Eleventh HOPE. [Online; accessed January 26, 2024]. July 2016. URL: <https://infocondb.org/con/hope/the-eleventh-hope/iridium-satellite-hacking>.
- [20] Josyula R Rao, Pankaj Rohatgi, Helmut Scherzer, and Stephane Tinguely. "Partitioning attacks: or how to rapidly clone some GSM cards". In: *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE. 2002, pp. 31–41.
- [21] Alla Levina, Mikhail Korovkin, and Daria Sleptsova. "Combined side-channel attacks on COMP128". In: *2014 IEEE 8th International Conference on Application of Information and Communication Technologies (AICT)*. IEEE. 2014, pp. 1–3.
- [22] Benedikt Driessen. "Eavesdropping on satellite telecommunication systems". In: *Cryptology EPrint Archive* (2012).
- [23] James Pavur, Daniel Moser, Martin Strohmeier, Vincent Lenders, and Ivan Martinovic. "A tale of sea and sky on the security of maritime VSAT communications". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1384–1400.

- [24] Pietro Tedeschi, Savio Sciancalepore, and Roberto Di Pietro. "Satellite-based communications security: A survey of threats, solutions, and research challenges". In: *Computer Networks* 216 (2022), p. 109246.
- [25] Gabriele Oligeri, Savio Sciancalepore, and Roberto Di Pietro. "GNSS spoofing detection via opportunistic IRIDIUM signals". In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2020, pp. 42–52.
- [26] Ruben Santamarta. "A wake-up call for satcom security". In: *Technical White Paper* (2014).
- [27] Eric Jedermann, Martin Strohmeier, Matthias Schäfer, Jens Schmitt, and Vincent Lenders. "Orbit-based Authentication Using TDOA Signatures in Satellite Networks". In: *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21)*. ACM. Abu Dhabi, United Arab Emirates, 2021. DOI: 10.1145/3448300.3469132. URL: <https://disco.cs.uni-kl.de/discfiles/publicationsfiles/JSSSL21.pdf>.
- [28] Gabriele Oligeri, Savio Sciancalepore, Simone Raponi, and Roberto Di Pietro. "PAST-AI: Physical-layer authentication of satellite transmitters via deep learning". In: *IEEE Transactions on Information Forensics and Security* 18 (2022), pp. 274–289.
- [29] Joshua Smailes, Sebastian Köhler, Simon Birnbach, Martin Strohmeier, and Ivan Martinovic. "Watch this space: Securing satellite communication through resilient transmitter fingerprinting". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 608–621.
- [30] OpenStreetMap Foundation. *OpenStreetMap*. [Online; accessed January 26, 2024]. 2024. URL: <https://www.openstreetmap.org>.
- [31] Yuanyuan Zhou, Yu Yu, François-Xavier Standaert, and Jean-Jacques Quisquater. "On the need of physical security for small embedded devices: a case study with COMP128-1 implementations in SIM cards". In: *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers* 17. Springer. 2013, pp. 230–238.
- [32] Osmocom. *osmo-sim-auth*. [Online; accessed January 26, 2024]. 2011. URL: <https://github.com/osmocom/osmo-sim-auth>.
- [33] Osmocom. *Implementation of COMP128-1*. [Online; accessed January 26, 2024]. 2010. URL: <https://github.com/osmocom/libosmocore/blob/master/src/gsm/comp128.c>.
- [34] Mohsen Toorani and A Beheshti. "Solutions to the GSM security weaknesses". In: *2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*. IEEE. 2008, pp. 576–581.

A. Source Code

A.1. C Implementation of COMP128-1

The code in Listing A.1 is an C implementation of the GSM authentication algorithm COMP128-1, also known as A3/A8 algorithm. This code is primarily inspired by the work of [10] but has been thoroughly rewritten based on various online resources that describe the algorithm. This C implementation serves as a realization of the algorithm as detailed in leaked documents from the GSM standards and verified through reverse-engineering of a working SIM card. The code is the effort of Sylvain Munaut and is distributed under the GNU General Public License v2.0+. [33]

Our modifications in the code start at the "Code adjustments start here" comment and primarily involve changes to the input and output mechanisms. Within the code, the secret key Ki is hardcoded, and the random challenge RAND is passed as a command-line argument. The SRES and Kc values are then computed by the function comp128v1 and printed in the console.

This implementation of COMP128-1 was used in Chapter 5 to prove that key extraction was successful. Consequently, it was demonstrated that the COMP128-1 authentication algorithm is indeed implemented on the Iridium SIM cards. It was also established that Iridium utilizes the compression tables exactly as they are specified in the code, indicating a direct and unchanged implementation of the COMP128-1 algorithm in the Iridium network.

Listing A.1: C Implementation of COMP128-1 [33].

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

/* The compression tables */
static const uint8_t table_0[512] = {
    102, 177, 186, 162, 2, 156, 112, 75, 55, 25, 8, 12, 251, 193, 246, 188,
    109, 213, 151, 53, 42, 79, 191, 115, 233, 242, 164, 223, 209, 148, 108, 161,
    252, 37, 244, 47, 64, 211, 6, 237, 185, 160, 139, 113, 76, 138, 59, 70,
    67, 26, 13, 157, 63, 179, 221, 30, 214, 36, 166, 69, 152, 124, 207, 116,
    247, 194, 41, 84, 71, 1, 49, 14, 95, 35, 169, 21, 96, 78, 215, 225,
    182, 243, 28, 92, 201, 118, 4, 74, 248, 128, 17, 11, 146, 132, 245, 48,
    149, 90, 120, 39, 87, 230, 106, 232, 175, 19, 126, 190, 202, 141, 137, 176,
    250, 27, 101, 40, 219, 227, 58, 20, 51, 178, 98, 216, 140, 22, 32, 121,
    61, 103, 203, 72, 29, 110, 85, 212, 180, 204, 150, 183, 15, 66, 172, 196,
    56, 197, 158, 0, 100, 45, 153, 7, 144, 222, 163, 167, 60, 135, 210, 231,
    174, 165, 38, 249, 224, 34, 220, 229, 217, 208, 241, 68, 206, 189, 125, 255,
```

A. Source Code

```

239, 54, 168, 89, 123, 122, 73, 145, 117, 234, 143, 99, 129, 200, 192, 82,
104, 170, 136, 235, 93, 81, 205, 173, 236, 94, 105, 52, 46, 228, 198, 5,
57, 254, 97, 155, 142, 133, 199, 171, 187, 50, 65, 181, 127, 107, 147, 226,
184, 218, 131, 33, 77, 86, 31, 44, 88, 62, 238, 18, 24, 43, 154, 23,
80, 159, 134, 111, 9, 114, 3, 91, 16, 130, 83, 10, 195, 240, 253, 119,
177, 102, 162, 186, 156, 2, 75, 112, 25, 55, 12, 8, 193, 251, 188, 246,
213, 109, 53, 151, 79, 42, 115, 191, 242, 233, 223, 164, 148, 209, 161, 108,
37, 252, 47, 244, 211, 64, 237, 6, 160, 185, 113, 139, 138, 76, 70, 59,
26, 67, 157, 13, 179, 63, 30, 221, 36, 214, 69, 166, 124, 152, 116, 207,
194, 247, 84, 41, 1, 71, 14, 49, 35, 95, 21, 169, 78, 96, 225, 215,
243, 182, 92, 28, 118, 201, 74, 4, 128, 248, 11, 17, 132, 146, 48, 245,
90, 149, 39, 120, 230, 87, 232, 106, 19, 175, 190, 126, 141, 202, 176, 137,
27, 250, 40, 101, 227, 219, 20, 58, 178, 51, 216, 98, 22, 140, 121, 32,
103, 61, 72, 203, 110, 29, 212, 85, 204, 180, 183, 150, 66, 15, 196, 172,
197, 56, 0, 158, 45, 100, 7, 153, 222, 144, 167, 163, 135, 60, 231, 210,
165, 174, 249, 38, 34, 224, 229, 220, 208, 217, 68, 241, 189, 206, 255, 125,
54, 239, 89, 168, 122, 123, 145, 73, 234, 117, 99, 143, 200, 129, 82, 192,
170, 104, 235, 136, 81, 93, 173, 205, 94, 236, 52, 105, 228, 46, 5, 198,
254, 57, 155, 97, 133, 142, 171, 199, 50, 187, 181, 65, 107, 127, 226, 147,
218, 184, 33, 131, 86, 77, 44, 31, 62, 88, 18, 238, 43, 24, 23, 154,
159, 80, 111, 134, 114, 9, 91, 3, 130, 16, 10, 83, 240, 195, 119, 253,
}, table_1[256] = {
19, 11, 80, 114, 43, 1, 69, 94, 39, 18, 127, 117, 97, 3, 85, 43,
27, 124, 70, 83, 47, 71, 63, 10, 47, 89, 79, 4, 14, 59, 11, 5,
35, 107, 103, 68, 21, 86, 36, 91, 85, 126, 32, 50, 109, 94, 120, 6,
53, 79, 28, 45, 99, 95, 41, 34, 88, 68, 93, 55, 110, 125, 105, 20,
90, 80, 76, 96, 23, 60, 89, 64, 121, 56, 14, 74, 101, 8, 19, 78,
76, 66, 104, 46, 111, 50, 32, 3, 39, 0, 58, 25, 92, 22, 18, 51,
57, 65, 119, 116, 22, 109, 7, 86, 59, 93, 62, 110, 78, 99, 77, 67,
12, 113, 87, 98, 102, 5, 88, 33, 38, 56, 23, 8, 75, 45, 13, 75,
95, 63, 28, 49, 123, 120, 20, 112, 44, 30, 15, 98, 106, 2, 103, 29,
82, 107, 42, 124, 24, 30, 41, 16, 108, 100, 117, 40, 73, 40, 7, 114,
82, 115, 36, 112, 12, 102, 100, 84, 92, 48, 72, 97, 9, 54, 55, 74,
113, 123, 17, 26, 53, 58, 4, 9, 69, 122, 21, 118, 42, 60, 27, 73,
118, 125, 34, 15, 65, 115, 84, 64, 62, 81, 70, 1, 24, 111, 121, 83,
104, 81, 49, 127, 48, 105, 31, 10, 6, 91, 87, 37, 16, 54, 116, 126,
31, 38, 13, 0, 72, 106, 77, 61, 26, 67, 46, 29, 96, 37, 61, 52,
101, 17, 44, 108, 71, 52, 66, 57, 33, 51, 25, 90, 2, 119, 122, 35,
}, table_2[128] = {
52, 50, 44, 6, 21, 49, 41, 59, 39, 51, 25, 32, 51, 47, 52, 43,
37, 4, 40, 34, 61, 12, 28, 4, 58, 23, 8, 15, 12, 22, 9, 18,
55, 10, 33, 35, 50, 1, 43, 3, 57, 13, 62, 14, 7, 42, 44, 59,
62, 57, 27, 6, 8, 31, 26, 54, 41, 22, 45, 20, 39, 3, 16, 56,
48, 2, 21, 28, 36, 42, 60, 33, 34, 18, 0, 11, 24, 10, 17, 61,
29, 14, 45, 26, 55, 46, 11, 17, 54, 46, 9, 24, 30, 60, 32, 0,
20, 38, 2, 30, 58, 35, 1, 16, 56, 40, 23, 48, 13, 19, 19, 27,
31, 53, 47, 38, 63, 15, 49, 5, 37, 53, 25, 36, 63, 29, 5, 7,
}, table_3[64] = {
1, 5, 29, 6, 25, 1, 18, 23, 17, 19, 0, 9, 24, 25, 6, 31,
28, 20, 24, 30, 4, 27, 3, 13, 15, 16, 14, 18, 4, 3, 8, 9,
20, 0, 12, 26, 21, 8, 28, 2, 29, 2, 15, 7, 11, 22, 14, 10,
17, 21, 12, 30, 26, 27, 16, 31, 11, 7, 13, 23, 10, 5, 22, 19,
}, table_4[32] = {
15, 12, 10, 4, 1, 14, 11, 7, 5, 0, 14, 7, 1, 2, 13, 8,
10, 3, 4, 9, 6, 0, 3, 2, 5, 6, 8, 9, 11, 13, 15, 12,
};

static const uint8_t *_comp128_table[5] = { table_0, table_1, table_2, table_3, table_4 };

static inline void
_comp128_compression_round(uint8_t *x, int n, const uint8_t *tbl)
{

```

```

int i, j, m, a, b, y, z;
m = 4 - n;
for (i=0; i<(1<<n); i++)
    for (j=0; j<(1<<m); j++) {
        a = j + i * (2<<m);
        b = a + (1<<m);
        y = (x[a] + (x[b]<<1)) & ((32<<m)-1);
        z = ((x[a]<<1) + x[b]) & ((32<<m)-1);
        x[a] = tbl[y];
        x[b] = tbl[z];
    }
}

static inline void
_comp128_compression(uint8_t *x)
{
    int n;
    for (n=0; n<5; n++)
        _comp128_compression_round(x, n, _comp128_table[n]);
}

static inline void
_comp128_bitsfrombytes(uint8_t *x, uint8_t *bits)
{
    int i;
    memset(bits, 0x00, 128);
    for (i=0; i<128; i++)
        if (x[i>>2] & (1<<(3-(i&3))))
            bits[i] = 1;
}

static inline void
_comp128_permutation(uint8_t *x, uint8_t *bits)
{
    int i;
    memset(&x[16], 0x00, 16);
    for (i=0; i<128; i++)
        x[(i>>3)+16] |= bits[(i*17) & 127] << (7-(i&7));
}

/*! Perform COMP128v1 algorithm
* \param[in] ki Secret Key K(i) of subscriber
* \param[in] rand Random Challenge
* \param[out] sres user-supplied buffer for storing computed SRES value
* \param[out] kc user-supplied buffer for storing computed Kc value */
void
comp128v1(const uint8_t *ki, const uint8_t *rand, uint8_t *sres, uint8_t *kc)
{
    int i;
    uint8_t x[32], bits[128];

    /* x[16-31] = RAND */
    memcpy(&x[16], rand, 16);

    /* Round 1-7 */
    for (i=0; i<7; i++) {
        /* x[0-15] = Ki */

```

```

memcpy(x, ki, 16);

/* Compression */
_comp128_compression(x);

/* FormBitFromBytes */
_comp128_bitsfrombytes(x, bits);

/* Permutation */
_comp128_permutation(x, bits);
}

/* Round 8 (final) */
/* x[0-15] = Ki */
memcpy(x, ki, 16);

/* Compression */
_comp128_compression(x);

/* Output stage */
for (i=0; i<8; i+=2)
    sres[i>>1] = x[i]<<4 | x[i+1];

for (i=0; i<12; i+=2)
    kc[i>>1] = (x[i + 18] << 6) |
               (x[i + 19] << 2) |
               (x[i + 20] >> 2);

kc[6] = (x[30]<<6) | (x[31]<<2);
kc[7] = 0;
}

/* Code adjustments start here. */
/* Convert a hexadecimal string to a byte array. */
void hexStringToBytes(const char *hexString, uint8_t *byteArray, size_t byteArraySize) {
    size_t length = strlen(hexString);
    /* Check if the input length is valid (even and within size limits). */
    if (length % 2 != 0 || length / 2 > byteArraySize) {
        fprintf(stderr, "Invalid input\n");
        return;
    }
    /* Convert each pair of hex characters to a byte. */
    for (size_t i = 0; i < length / 2; i++) {
        sscanf(hexString + 2 * i, "%2hhx", &byteArray[i]);
    }
}

int main(int argc, char *argv[]) {
    uint8_t ki[16], rand[16], sres[4], kc[8];
    /* Hardcoded hexadecimal string for 'ki'. */
    const char *kiHexString = "CB0C650C2FB151AEFB8DD21A9C330ABC";

    /* Check for the correct number of command-line arguments. */
    if (argc != 2) {
        fprintf(stderr, "Please enter the challenge as an argument.\n");
        return 1;
    }
}

```



```

/* Convert 'ki' and 'rand' from hex to bytes. */
hexStringToBytes(kiHexString, ki, sizeof(ki));
hexStringToBytes(argv[1], rand, sizeof(rand));

/* Call COMP128-1. */
comp128v1(ki, rand, sres, kc);

/* Print the results. */
printf("SRES:\t %02X%02X%02X%02X\n", sres[0], sres[1], sres[2], sres[3]);
printf("Kc:\t %02X%02X%02X%02X%02X%02X%02X%02X\n", kc[0], kc[1], kc[2], kc[3], kc[4], →
      kc[5], kc[6], kc[7]);

return 0;
}

```

A.2. Calculation of the Originating Pairs

For a given level of the hashing function, this Python code computes, for all possible combinations of a pair, all originating pairs in the previous level that can generate this pair. In the code, the level and its corresponding compression table can be adjusted via the variables *b* and *t*. The compression tables from Listing A.1 are employed in the code. The solution sets are computed by the functions `calculate_solution_set_y` for the *y*-equation and `calculate_solution_set_z` for the *z*-equation.

In the code analysis presented in Chapter 7, we utilized the insight that this code demonstrated algorithmically through empirical testing that exactly four possible origin pairs were computed for every combination of pairs in each level.

Listing A.2: Python program that computes for all possible combinations of a pair, all originating pairs in the previous level.

```

import sys

# Calculate the solution set for the y-equation with modular base b.
def calculate_solution_set_y(y, b):
    solution_set = set()
    for xn in range(b):
        xm = (y - 2 * xn) % b
        if 0 <= xm <= b - 1:
            solution_set.add((xm, xn))
    return solution_set

# Calculate the solution set for the z-equation with modular base b.
def calculate_solution_set_z(z, b):
    solution_set = set()
    for xm in range(b):
        xn = (z - 2 * xm) % b
        if 0 <= xn <= b - 1:
            solution_set.add((xm, xn))

```

```

return solution_set

def proof_four_possibilities():
    # The compression tables are defined as in Appendix A.1.
    table = [table_0, table_1, table_2, table_3, table_4]

    # Define the counter of pairs with exactly four possibilities .
    counter = 0

    # Define the modular basis b and the corresponding table index t.
    # The possible values for (b,t) are: (16,4), (32,3), (64,2), (128,1) (256,0)
    b = 64
    t = 2

    for xm in range(0, b):
        # Find positions in the table where the value matches xm.
        positions_y = [i for i, value in enumerate(table[t]) if value == xm]
        # Calculate the solution set for y for each found position .
        solution_dict_y = {i: calculate_solution_set_y(i, 2 * b) for i in positions_y}

    for xn in range(0, b):
        # Find positions in the table where the value matches xn.
        positions_z = [i for i, value in enumerate(table[t]) if value == xn]
        # Calculate the solution set for z for each found position .
        solution_dict_z = {i: calculate_solution_set_z(i, 2 * b) for i in positions_z}

    duplicates = {}
    # Iterate over all combinations of positions and solution sets .
    for q, solution_y in solution_dict_y.items():
        for solution_y_tuple in solution_y:
            for r, solution_z in solution_dict_z.items():
                for solution_z_tuple in solution_z:
                    # If a common solution is found, record it as a duplicate .
                    if solution_y_tuple == solution_z_tuple:
                        if (q, r) not in duplicates:
                            duplicates[(q, r)] = []
                            duplicates[(q, r)].append(solution_y_tuple)
    # Check if the number of duplicates is exactly 4.
    if len(duplicates) == 4:
        counter += 1
    else:
        sys.exit("Proof_failed")

    # Print the final count of pairs with exactly four possibilities .
    # For b=64 and t=2: "4096 from 4096 pairs have exactly four possibilities ."
    print(str(counter) + "_from_" + str(b * b) + "_pairs_have_exactly_four_possibilities.")

```

A.3. Statistical Evaluation of Back-Calculation

The Python code in Listing A.3 was used to evaluate the dependency of different levels in the hash function of COMP128-1. For each of the 65,536 possibilities of a four-digit block in the output, level 4 and level 3 are back-calculated, resulting in 256

possible inputs in level 3. It was conducted to determine, for each possible output, how many of the corresponding 256 possible inputs in level 3 actually generate the respective output. In the code, the compression tables from Listing A.1 and the functions `calculate_solution_set_y` and `calculate_solution_set_z` from Listing A.2 are utilized.

The results are detailed in Chapter 7 and reveal a significant hidden dependency between the levels. This code can be similarly extended to also back-calculate the level 2, level 1, and level 0.

Listing A.3: Python program for back-calculating for each possible four-digit block in the output the corresponding possible inputs at level 3 of the hash function.

" calculate_solution_set_y " and " calculate_solution_set_z " are defined as in Appendix A.2.

```
def quad_calculation():
    # The compression tables are defined as in Appendix A.1.
    table = [table_0, table_1, table_2, table_3, table_4]

    results = {}
    # Iterate through all possible digits of the quad.
    for quad_0 in range(0, 16):
        for quad_1 in range(0, 16):
            for quad_2 in range(0, 16):
                for quad_3 in range(0, 16):

                    # Define arrays for the back-calculation of level 4 and level 3.
                    level_4 = [[] for _ in range(32)]
                    level_3 = [[[ for _ in range(4)] for _ in range(32)]

                    # Define array of the original quad.
                    level_5 = [quad_0, quad_1, quad_2, quad_3]

                    # Back-calculate level 4.
                    for ar_idx in range(0, 4):
                        if (ar_idx % 2 == 0) and level_5[ar_idx] != -1:
                            xm = level_5[ar_idx]
                            # Find positions in the table where the value matches xm.
                            positions_y = [i for i, value in enumerate(table[4]) if value == xm]
                            # Calculate the solution set for y for each found position.
                            solution_dict_y = {i: calculate_solution_set_y(i, 32) for i in positions_y}

                            xn = level_5[ar_idx + 1]
                            # Find positions in the table where the value matches xn.
                            positions_z = [i for i, value in enumerate(table[4]) if value == xn]
                            # Calculate the solution set for z for each found position.
                            solution_dict_z = {i: calculate_solution_set_z(i, 32) for i in positions_z}

                            duplicates = {}
                            # Iterate over all combinations of positions and solution sets.
                            for q, solution_y in solution_dict_y.items():
                                for solution_y_tuple in solution_y:
                                    for r, solution_z in solution_dict_z.items():
                                        for solution_z_tuple in solution_z:
                                            # If a common solution is found, record it as a duplicate.
```

```

    if solution_y_tuple == solution_z_tuple:
        if (q, r) not in duplicates:
            duplicates[(q, r)] = []
            duplicates[(q, r)].append(solution_y_tuple)
            level_4[ar_idx].append(solution_y_tuple[0])
            level_4[ar_idx + 1].append(solution_y_tuple[1])

# Back-calculate level 3.
keys_4 = [0, 1]
for ar_idx in keys_4:
    if level_4[ar_idx] and level_4[ar_idx + 2]:
        for pos_idx in range(0, 4):
            xm = level_4[ar_idx][pos_idx]
            # Find positions in the table where the value matches xm.
            positions_y = [i for i, value in enumerate(table[3]) if value == xm]
            # Calculate the solution set for y for each found position.
            solution_dict_y = {i: calculate_solution_set_y(i, 64) for i in positions_y}

            xn = level_4[ar_idx + 2][pos_idx]
            # Find positions in the table where the value matches xn.
            positions_z = [i for i, value in enumerate(table[3]) if value == xn]
            # Calculate the solution set for z for each found position.
            solution_dict_z = {i: calculate_solution_set_z(i, 64) for i in positions_z}

            duplicates = {}
            # Iterate over all combinations of positions and solution sets.
            for q, solution_y in solution_dict_y.items():
                for solution_y_tuple in solution_y:
                    for r, solution_z in solution_dict_z.items():
                        for solution_z_tuple in solution_z:
                            # If a common solution is found, record it as a duplicate.
                            if solution_y_tuple == solution_z_tuple:
                                if (q, r) not in duplicates:
                                    duplicates[(q, r)] = []
                                    duplicates[(q, r)].append(solution_y_tuple)
                                    level_3[ar_idx][pos_idx].append(solution_y_tuple[0])
                                    level_3[ar_idx + 2][pos_idx].append(solution_y_tuple[1])

# Counter for the calculation of the original quad.
counter = 0

for i in range(4):
    for j in range(4):
        for k in range(4):
            for l in range(4):
                # Quad block of level 4.
                level_4_forward = [-1, -1, -1, -1]

                # Calculate pair (0,2) of level 4.
                xm = level_3[0][i][j]
                xn = level_3[2][i][j]
                y = (xm + 2 * xn) % 64
                z = (2 * xm + xn) % 64
                level_4_forward[0] = table_3[y]
                level_4_forward[2] = table_3[z]

                # Calculate pair (1,3) of level 4.

```

A. Source Code

```

xm = level_3[1][k][1]
xn = level_3[3][k][1]
y = (xm + 2 * xn) % 64
z = (2 * xm + xn) % 64
level_4_forward[1] = table_3[y]
level_4_forward[3] = table_3[z]

# Quad block of the output.
level_5_forward = [-1, -1, -1, -1]

# Calculate pair (0,1) of the output.
xm = level_4_forward[0]
xn = level_4_forward[1]
y = (xm + 2 * xn) % 32
z = (2 * xm + xn) % 32
level_5_forward[0] = table_4[y]
level_5_forward[1] = table_4[z]

# Calculate pair (0,1) of the output.
xm = level_4_forward[2]
xn = level_4_forward[3]
y = (xm + 2 * xn) % 32
z = (2 * xm + xn) % 32
level_5_forward[2] = table_4[y]
level_5_forward[3] = table_4[z]

# If the original quad is found, increase the counter by 1.
if level_5_forward == [quad_0, quad_1, quad_2, quad_3]:
    counter += 1

# Total count of calculations generating the original quad.
if counter in results:
    results[counter] += 1
else:
    results[counter] = 1
# Print the results .
for arg_1, arg_2 in results.items():
    print(f'Number_of_calculations_generating_the_original_output:_{arg_1},_Total_number_→
of_occurrences:_{arg_2}')

```