Time-Memory Trade-Off Attack on the A5/1 Stream Cipher in Golang

Arza Henrie
CS3110 001
Utah Valley University
Orem, UT
10914763@uvu.edu

***Abstract*—This paper provides a technical overview of the A5/1 encryption algorithm, its role in GSM networks, and known vulnerabilities. We delve into the details of a specific attack on A5/1, demonstrating its practical feasibility. By understanding the weaknesses of A5/1, we highlight the importance of strong cryptographic techniques in modern communication systems and the need for continuous security assessments.**

***Index Terms*—A5/1 Stream Cipher, Time-Memory Trade-Off Attack, GSM Network Security, Cryptographic Vulnerabilities, Encryption Algorithms, Go Programming Language, Cryptanalysis, Cellular Communication Security**

I. Introduction

In June 1982, the Netherlands proposed a unified European mobile system to prevent incompatible national systems from fragmenting the 900 MHz band. This proposal ignited a movement towards a standardized, pan-European mobile communication system. The Groupe Spécial Mobile (GSM) was subsequently formed to develop this new technology [1].

The GSM group's efforts led to the creation of the Global System for Mobile Communications (GSM), which revolutionized mobile telephony. From its inception in the early 1990s, GSM rapidly gained popularity and became the dominant mobile technology worldwide for several decades. Millions would adopt and use later mobile communication technologies (Figure 1)

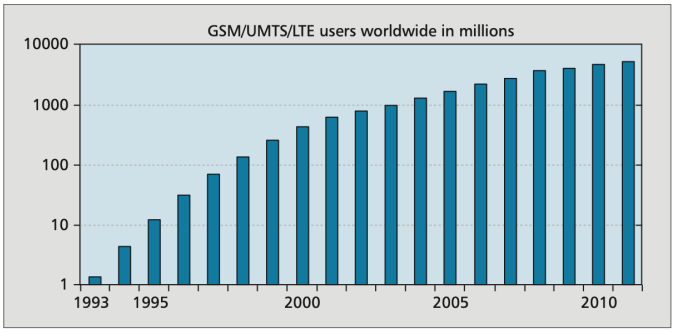

Fig. 1. GSM, UMTS and LTE users worldwide (Source: Hillebrand, Friedhelm

II. Technical Overview of A5/1

The GSM network would rely on the A/51 stream cipher for privacy and security. Although the cipher was kept secret, through reverse engineering in 1999, the internal workings were disclosed. [2] A5/1 accepts a 64-bit session key in GF(2)^{22}

$$K_s = (K_0, K_1, ..., K_63)$$

This indicated that the key is a sequence of 64 bits, each of which can be either 0 or 1. This is our finite field where the arithmetic operations are performed in modulo 2. An initial 22-bit vector IV in GF(2)^{22} is also accepted by the symmetric cipher. The hardware behind A5/1 consists of three linear feedback shift registers, R1, R2, and R3 with lengths 19, 22, and 23 bits. The LFSRs represent primitive feedback polynomials that generate a sequence of maximum lengths. This ensures that shift cycles repeat a minimum number of times, and the output of the LFSRs are long and unpredictable. For enhanced security, clocking is leveraged in the LFSRs determined by a majority logic function based on the least significant bits or the clocking bits (CB) of the three registers. The CBs are the 8th, 10th, and 10th bit of R1, R2, and R3.[3] (Figure 2)

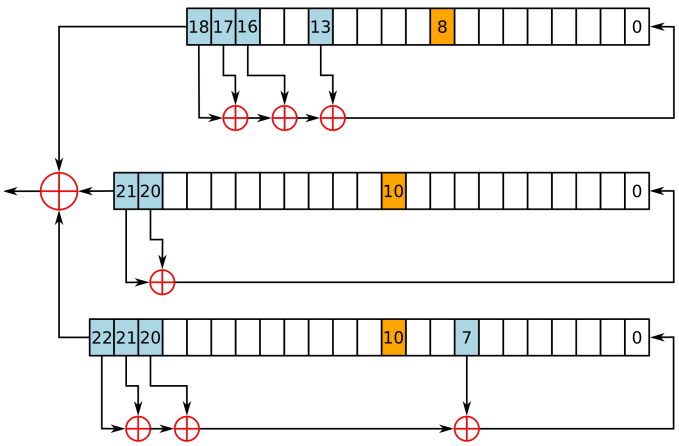

Fig. 2. 3 LFSRs of A5/1 (Source: Wikipedia) [4]

A keystream generator is needed before the A5/1 can be used. The initialization phase begins by setting all bits to 0 and initializing the bits with the 64-bit key. For each bit in K_s all three registers shift one bit to the left. The current bit of the key is XORed with each register's feedback polynomial

and then XORed into each register's least significant bit. This process ensures the key is mixed across the three registers through repeated shifting and modulus 2 arithmetic. After the 64 cycles for each of the bits in the 64-bit key, the same process is followed with the 22-bit initialization vector. After the 86 clock cycles, the registers undergo 100 cycles of irregular clocking using the majority function. No output is produced for these 100 cycles. The clocking step servers to randomize the internal state and make reverse engineering from the keystream more difficult.

The registers, now initialized, proceed to generate the keystream used for encryption. For each keystream bit, the registers are clocked according to the majority rule, which examines specific clocking bits in R1, R2, and R3. Only registers matching the majority value shift, and the output bits of R1, R2, and R3 are XORed together to produce one keystream bit. This bit is XORed with a corresponding plaintext or ciphertext bit to encrypt or decrypt the message. This irregular clocking mechanism complicates predictions of the internal state, making the cipher's output appear random and unpredictable.

III. Attacks on A5/1

There are a handful of attacks that perform cryptanalysis and exploit some of A5/1's weaknesses. The internal state of A5/1 consists of 64 bits which is inadequate in today's cryptographic standards. Modern encryption schemes typically run with larger states to prevent attacks from recovering the internal state. The small size allows attackers with sufficient compute to fully explore all possible states, leading to potential recovery of keys or even the states with minimal keystream data. The key setup routine or initialization phase is too straightforward. Patterns can be inferred in the initial state. The weak setup means that attacks can predict initial states based on portions of observed data, making reverse-engineering the state a viable option with today's computers.

The TMTO (Time-Memory Trade-Off) attack exploits the predictability of keystream generation. By precomputing a large table of possible keystream sequences, the table can be used to associate keystreams to probable internal states. If the keystream can be intercepted, an attacker could match the keystream to the precomputed values. Cipher Downgrade attack forces the device to use a less secure encryption algorithm like A5/2 with intentional weaknesses. Once the device operates on A5/2, the attacker can crack the weaker. Other attacks such as brute-force and rainbow tables can also be used. Although computationally intensive, brute-force, could be used to try all possible 64-bit keys. Rainbow tables can be used to store cryptographic hashes of key-state combinations, accelerating the key discovery process by reducing the need for repeated computations.

IV. TMTO Attack Implementation

A lookup table to store keys and keystreams may seem like a straightforward task at first. However, before we dive into generating all the keys, let's do some math to understand the scale of the problem. We are using 64-bit keys and 64-bit keystreams.

$$2^{64} = 18,446,744,073,709,551,616$$

To store a key and keystream in the table, we need to calculate how much storage is required. Let's assume each entry is 16 bytes: 8 bytes for the key and 8 bytes for the keystream.

$$\begin{aligned} &18,446,744,073,709,551,616 \text{ entries} \\ &\times 16 \text{ bytes} \\ \hline &295,147,905,179,352,825,856 \text{ bytes} \end{aligned}$$

This storage requirement comes out to around 295 exabytes, or approximately 268 million terabytes. Given the enormity of this number, I had to take these constraints into account when designing the implementation. Some potential solutions include reducing the key space to 8 bits or generating part of the table, then injecting the known key into the table before performing keystream computation and lookup for decryption. I went with the later.

A. Core Data Structures

1) LFSR Struct: The Linear Feedback Shift Registers (LFSRs) are the fundamental components of the A5/1 encryption mechanism. They are represented in Go as follows:

```
type LFSR struct {
  state    uint32
  mask     uint32
  size     int
  clockBit int
}
```

The fields of the `LFSR` struct are defined as:

- **`state`**: Represents the current state of the LFSR, where each bit corresponds to a bit in the shift register.
- **`mask`**: A binary mask defining the feedback taps for the LFSR. This mask determines which bits participate in the feedback calculation.
- **`size`**: The total number of bits in the LFSR. It specifies the length of the shift register.
- **`clockBit`**: Indicates the position of the bit used to control clocking decisions during majority voting.

This structure allows efficient modeling and manipulation of LFSRs for both encryption and keystream generation.

2) Rainbow Table: A rainbow table is a key-value data structure used to precompute and store relationships between keystreams and their corresponding encryption keys. Since it is designed for runtime efficiency, the table is implemented as a Go `map`:

```
func PrecomputeTable(keyspace uint64,
    keystreamLength int, knownKey uint64,
    insertionPoint uint64) map[string]uint64 {
  finalTable := make(map[string]uint64)

  for key := uint64(0); key < keyspace; key++ {
    lfsr1, lfsr2, lfsr3 := a5.InitializeA5_1(key, 0)
    keystream := a5.GenerateKeystream(lfsr1, lfsr2,
    lfsr3, keystreamLength)
```

```

    // Store the keystream as raw bytes (binary)
    keystreamBytes := string(keystream)
    finalTable[keystreamBytes] = key

        ...
}
```

a) Parameters of `PrecomputeTable`:

- **`keyspace`**: Defines the range of possible encryption keys. A larger keyspace increases the precomputation time but ensures coverage.
- **`keystreamLength`**: Specifies the length of the keystream (in bits) to be generated and stored for each key.
- **`knownKey`**: A user-provided key injected into the table to simulate precomputed values or validate the attack. (I am not computing all those values)
- **`insertionPoint`**: Specifies how many keystreams should be generated before injecting the `knownKey`.

3) Design and Implementation Notes:

- **Volatile Nature**: The rainbow table exists only during program execution and is discarded upon exit. This transient design ensures minimal memory usage beyond runtime.
- **Key Mapping**: The keystreams are stored as binary strings mapped to their respective keys.
- **Progress Tracking**: The function periodically logs progress to the console, allowing users to monitor the generation process.
- **Key Injection**: Injecting a known key mid-process enables quick validation and debugging of the precomputed table.

B. Key Functions

1) Clocking: This function clocks the LFSR with irregular clocking. The clocking is dependent on the boolean passed as `clockBit`.

```
func (l *LFSR) Clock(clockBit bool) {
    if clockBit {
        feedback := parity(l.state & l.mask)
        // Shift the state and insert the feedback bit at the LSB
        l.state = ((l.state << 1) | feedback) & ((1 << l.size) - 1)
    }
}
```

2) LFSR Initialization: This function initializes the three LFSRs given a 64-bit key and a 22-bit `frameNumber` that is typically computed at session start:

- All registers are instantiated with a 0 state, a mask that represents their predefined taps, and the clocking bits.
- Load the key into the LFSRs by XORing each bit of the key into the state of the LFSRs.
- Load the frame number into the LFSRs.
- Clock the LFSRs 100 times.
- Return three pointers to the initialized LFSRs.

```
func InitializeA5_1(key uint64, frameNumber uint32) (*LFSR, *LFSR, *LFSR) {
    lfsr1 := &LFSR{state: 0, mask: 0x072000, size: 19, clockBit: 8}
    lfsr2 := &LFSR{state: 0, mask: 0x300000, size: 22, clockBit: 10}
    lfsr3 := &LFSR{state: 0, mask: 0x700080, size: 23, clockBit: 10}

    // Load key into LFSRs
    for i := 0; i < 64; i++ {
        bit := (key >> (63 - i)) & 1
        lfsr1.state ^= uint32(bit) << (i % lfsr1.size)
        lfsr2.state ^= uint32(bit) << (i % lfsr2.size)
        lfsr3.state ^= uint32(bit) << (i % lfsr3.size)
    }

    // Load frame number into LFSRs
    for i := 0; i < 22; i++ {
        bit := (frameNumber >> (21 - i)) & 1
        lfsr1.state ^= uint32(bit) << (i % lfsr1.size)
        lfsr2.state ^= uint32(bit) << (i % lfsr2.size)
        lfsr3.state ^= uint32(bit) << (i % lfsr3.size)
    }

    // Clock LFSRs 100 times after loading key and frame number (standard A5/1 initialization step)
    for i := 0; i < 100; i++ {
        lfsr1.Clock(true)
        lfsr2.Clock(true)
        lfsr3.Clock(true)
    }

    return lfsr1, lfsr2, lfsr3
}
```

3) Keystream Generation: This function generates the keystream:

- Three LFSR pointers are passed along with a length for the number of bits to generate for the keystream.
- For each bit in the keystream, a majority vote of the clocking bit is computed. The LFSRs are clocked depending on their clocking bit.
- The keystream bit is computed as the XOR of the least significant bits of the three LFSR states.

```
func GenerateKeystream(lfsr1, lfsr2, lfsr3 *LFSR, length int) []uint8 {
    keystream := make([]uint8, length)
    for i := 0; i < length; i++ {
        m := majorityVote(lfsr1.ClockingBit(), lfsr2.ClockingBit(), lfsr3.ClockingBit())
        lfsr1.Clock(lfsr1.ClockingBit() == m)
        lfsr2.Clock(lfsr2.ClockingBit() == m)
        lfsr3.Clock(lfsr3.ClockingBit() == m)
        keystream[i] = uint8((lfsr1.state & 1) ^ (lfsr2.state & 1) ^ (lfsr3.state & 1))
    }
    return keystream
}
```

C. Data Flow

The data flow of this program and attack is straightforward:

1) The program prompts the user for their name, which is then concatenated with a predefined message to form the plaintext.
2) The user is asked to input a 64-bit hexadecimal key, which is validated for correctness.
3) The provided key, along with a constant frame number, is used to initialize the three LFSRs of the A5/1 cipher.
4) The encryption process is carried out on the plaintext using the initialized LFSRs, producing the ciphertext.
5) Rainbow tables are precomputed, with a specified number of entries and an insertion point, to optimize the TMTO attack by saving time and computational resources.
6) Decryption is attempted by searching the rainbow table for a matching keystream. If found, the corresponding (injected) key and the frame number are used to decrypt the ciphertext.

D. Program Output

The output of the program is as follows:

```
*********************************
* Welcome to the A5/1 TMTO *
*********************************
What is your name:
MyName
Hello, MyName. We will be encrypting..
*********************************
* Please enter a 64-bit hexadecimal key *
* (e.g. 1234567890abcdef) *
*********************************
1234567890abcdef
*********************************
* Encryption Complete! *
*********************************
Ciphertext: 21486921214c794e61....
*********************************
* Generating Table... *
*********************************
Progress: Generated 0 entries
Progress: Generated 100000 entries
Progress: Generated 200000 entries
Progress: Generated 300000 entries
Progress: Generated 400000 entries
Progress: Generated 500000 entries
Progress: Generated 600000 entries
Progress: Generated 700000 entries
Progress: Generated 800000 entries
Progress: Generated 900000 entries
This is taking too long...
Planted key at 999999: 1234567890abcdef
Generated table with 1000001..
*********************************
* Attempting to Decrypt... *
*********************************
*********************************
* Decryption Successful! *
*********************************
Found key: 1234567890abcdef
Found keystream: 00000100000100010101000...
Decrypted plaintext:  Hi! MyName...
```

V. Analysis and Conclusion

The Time-Memory Trade-Off (TMTO) attack is feasible, especially with the advent of quantum computing and other advanced computing systems. Given my limited resources, I could have reduced the keyspace to lower the security level; however, implementing the 64-bit version of A5/1 was a valuable learning experience.

The biggest challenge faced was handling the vast size of the keyspace. I considered using Go's goroutines to compute the keystreams and keys in parallel. While parallelization could speed up computations, the immense size of A5/1's keyspace presents a major limitation. Even with parallelization, it is not feasible to store or compute every possible keystream, especially with my available memory and storage.

1) Is It Really a Microsecond?: In the context of cryptographic algorithms like A5/1, it's often assumed that key generations or keystream iterations can be completed in around one microsecond. To validate this assumption, I decided to time each key generation for every iteration during an A5/1 keystream generation. My experiment reached about 70 million keystream generations before I had to terminate the process.

The results indicate that, on average, the A5/1 algorithm takes approximately 1 microsecond to complete a single iteration. This timing is consistent with the common assumption that certain cryptographic computations can be completed in microseconds.

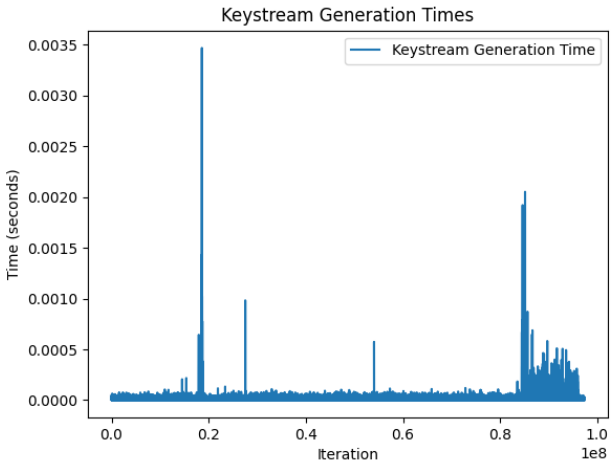

Fig. 3. Iteration Benchmarking

However, it's important to note that some outliers exist in the data. These anomalies may be attributed to various factors such as inefficiencies in my code, hardware limitations, or environmental factors influencing the benchmark. Despite

these outliers, the general trend supports the idea that the A5/1 algorithm performs around the 1-microsecond mark per iteration.

VI. License

The project is licensed under the MIT license.

References

[1] F. Hillebrand, "The creation of standards for global mobile communication: Gsm and umts standardization from 1982 to 2000," *IEEE Wireless Communications*, vol. 20, no. 5, pp. 24–33, Oct 2013, eBSCOhost. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=15092f86-ac06-3e8b-9ebe-80a3c2950839

[2] S. Robinson, "Researchers crack code in cell phones," *The New York Times*, Dec 1999. [Online]. Available: https://www.nytimes.com/1999/12/07/business/researchers-crack-code-in-cell-phones.html

[3] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Berlin Heidelberg, 2010.

[4] W. contributors, "A5/1," https://en.wikipedia.org/wiki/A5/1cite$_n ote - 1, 2024, accessed : 2024 - 08 - 09.[Online].Available : https : //en.wikipedia.org/wiki/A5/1cite_n ote - 1$