A Practical Meet-in-the-Middle Attack on SIGABA

George Lasry
The CrypTool Team
george.lasry@cryptool.org

Abstract

The SIGABA is an electromechanical encryption device used by the US during WWII and in the 1950s. Also known as ECM Mark II, Converter M-134, as well as CSP-888/889, the SIGABA was considered highly secure, and was employed for strategic communications, such as between Churchill and Roosevelt. The SIGABA encrypts and decrypts with a set of five rotors, and implements irregular stepping, with two additional sets of rotors generating a pseudo-random stepping sequence. Its full keyspace, as used during WWII, was in the order of $2^{95.6}$. It is believed that the German code-breaking services were not able to make any inroads into the cryptanalysis of SIGABA (Mucklow, 2015; Budiansky, 2000; Kelley, 2001).

The most efficient attack on SIGABA published so far is a known-plaintext attack that requires at least $2^{86.7}$ steps.[1] Although it is more efficient than an exhaustive search, it is not practical, even with modern computing (Stamp and Chan, 2007; Stamp and Low, 2007).

In this paper, the author presents a novel meet-in-the-middle (MITM) known-plaintext attack. This attack requires $2^{60.2}$ steps and less than 100 GB RAM, and it is feasible with modern technology. It takes advantage of a weakness in the design of SIGABA. With this attack, the author solved a MysteryTwister C3 (MCT3) Level III challenge (Stamp, 2010). The author also presents a series of new challenges, which will also appear in MTC3.

[1] To date, no ciphertext-only attack has been proposed, except for an attack that requires multiple messages in depth (Savard and Pekelney, 1999).

This paper is structured as follows: In Section 1, the SIGABA encryption machine is described, including a functional description and an analysis of its keyspace. In Section 2, prior attacks on SIGABA are surveyed, and a novel MITM known-plaintext attack is presented, including an analysis of its work-factor, and how it was used to solve MysteryTwister C3 (MCT3) challenges (Stamp, 2010). In Section 3 and in the Appendix, new challenges are presented, as well as the reference code for a SIGABA simulator used to create those challenges.[2]

1 The SIGABA Encryption Machine

In this section, a short functional description of the SIGABA is given, as well as an analysis of its keyspace size.

1.1 Functional Description of SIGABA

The functional description of SIGABA presented here focuses on the features essential to the understanding of the new attack presented in Section 2. A complete description of the machine and its history may be found in the references (Savard and Pekelney, 1999; Sullivan, 2002b; Stamp and Chan, 2007; Mucklow, 2015; Kelley, 2001; Pekelney, 1998; Sullivan, 2002a).

The SIGABA encryption and decryption mechanism consists of three banks of five rotors each, the *cipher* bank, the *control* bank, and the *index* bank, as depicted in Figure 1. Each rotor of the *cipher* bank has 26 inputs and 26 outputs (similar in concept to the Enigma rotors, but with different wirings). The *cipher* rotors implement encryption (from left to right), and decryption (from right to left). The rotors of the *cipher* bank step according to an irregular pseudo-random pattern generated by the *index* and the *control* rotor bank.

[2] This work has been supported by the Swedish Research Council, grant 2018-06074, DECRYPT - Decryption of historical manuscripts.

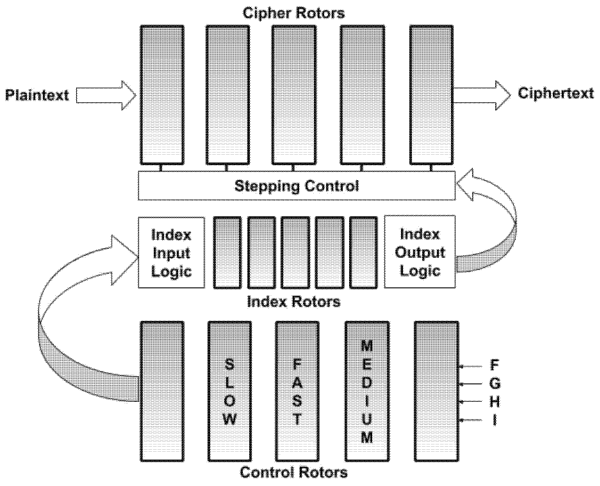

Figure 1: SIGABA – Functional Diagram

Input (to index rotor bank)	Logic (A-Z are the outputs of the control rotor bank, ∨ indicates logical OR)
Input 1	B
Input 2	C
Input 3	D ∨ E
Input 4	F ∨ G ∨ H
Input 5	I ∨ J ∨ K
Input 6	L ∨ M ∨ N ∨ O
Input 7	P ∨ Q ∨ R ∨ S ∨ T
Input 8	U ∨ V ∨ W ∨ X ∨ Y ∨ Z
Input 9	A
Input 10	(inactive)

Figure 2: Index Input Logic

Stepping Control (of cipher rotor)	Logic (I_1-I_{10} are the outputs of the index rotor bank, ∨ indicates logical OR)
Rotor 1	$I_1 \vee I_{10}$
Rotor 2	$I_8 \vee I_9$
Rotor 3	$I_6 \vee I_7$
Rotor 4	$I_4 \vee I_5$
Rotor 5	$I_2 \vee I_3$

Figure 3: Index Output Logic

The *control* bank consists of five rotors, each with 26 inputs and 26 outputs. The *cipher* rotors and the *control* rotors are interchangeable, and are selected from a set of ten rotors. Furthermore, those rotors can be installed in two possible orientations – forward or reverse (thus increasing the size of the keyspace by a factor of $2^{10} = 1,024$). Interestingly, the *cipher* rotors and the *control* rotors move through the alphabet in reverse order (e.g., from D to C, or from C to B) when installed in forward orientation, and in alphabetical order (e.g., from D to E, or from E to F) when installed in reversed orientation. The leftmost and rightmost *control* rotors are stationary and do not rotate. The *fast* rotor always steps (interestingly, this rotor is located between the *slow* and the *medium* rotors). If the *fast* rotor steps from O to N (while in forward orientation) or from O to P (while in reversed orientation), the *medium* rotor also steps (Pekelney, 1998; Sullivan, 2002a).[3] Similarly, if the *medium* rotor steps from O to N (while in forward orientation) or from O to P (while in reversed orientation), the *slow* rotor also steps. At each encryption step, the inputs F, G, H, and I of the rightmost (stationary) rotor are activated and fed with electrical current (and the 22 remaining input are always inactive). The 26 outputs of the leftmost *control* rotor enter the *index input logic*, described in Figure 2.

The *index* bank consists of a set of five stationary rotors, which do not rotate during encryption or decryption, and they each have 10 inputs and 10 outputs. Those rotors are not interchangeable with the *cipher* and *control* rotors, and they can only be installed in a forward orientation. The *index input logic*, described in Figure 2, maps its 26 inputs into 10 outputs, which enter the leftmost *index* rotor. The *index output logic*, described in Figure 3, maps the 10 outputs of the *index* rightmost rotor into five *stepping control* signals, controlling the stepping of the five *cipher* rotors. The design of the *control* and *index* rotor banks, in conjunction with the *index input logic* and the *index output logic*, ensures that at least one of the five *cipher* rotors will step, but no more than four *cipher* rotors ever step (Stamp and Chan, 2007, p. 203).

Encryption is performed as follows, assuming that the 15 rotors have been installed. The machine must be set to the encryption mode. The operator selects the starting positions of the rotors,

[3] (Stamp and Chan, 2007; Savard and Pekelney, 1999) describe different implementations for the stepping mechanism.

and types the plaintext on the SIGABA keyboard. The plaintext symbol is applied to the *cipher* rotors from left to right, producing the ciphertext symbol on a printing device. After encryption of a symbol, the *cipher* rotors step according to the state of the *stepping control* (see Figure 1). After the *cipher* rotors have stepped, some of the *control* rotors step, thus generating (via the *index* rotors) a new state for the *stepping control* of the *cipher* rotors. The process is repeated for the next plaintext symbols.

Decryption works similarly, except that the device must be set to the decryption mode, and the cipher symbols (typed on the keyboard) are applied to the *cipher* rotors from right to left, the resulting plaintext being printed.

1.2 Analysis of the Keyspace

Assuming that there is a set of ten rotors from which the *cipher* and *control* rotors are selected, there are 10! possible selections for those rotors. Each one of those rotors may be installed in either a forward or reverse orientation. The size of the keyspace for the settings of the ten rotors of the *cipher* and *control* banks is therefore $10! \cdot 2^{10} \cdot 26^{10} = 2^{78.8}$.

There are 5! possible ordering of the *index* rotors. The size of the keyspace for the settings of the rotors of the *index* bank is therefore $5! \cdot 10^5 = 2^{23.5}$. The combined size of the SIGABA keyspace is $2^{78.8+23.5} = 2^{102.3}$.

However, the size of the keyspace for the *index* bank is limited by the fact that the five rotors implement a (stationary) permutation of the ten inputs and the ten outputs are mapped by the *index output logic* into only five outputs. Therefore, the size of the practical keyspace for the *index* rotors including the *index output logic* is only $10!/2^5 = 113,400 = 2^{16.8}$, and the combined size of the practical keyspace of SIGABA is as follows:

$$2^{78.8+16.8} = 2^{95.6} \quad (1)$$

For comparison, the size of the keyspace the German Enigma I was in the order of 2^{77} (Stamp and Low, 2007, p. 31), and it is 2^{56} for the more recent DES.

2 Cryptanalysis of SIGABA

In this section, prior attacks on SIGABA are reviewed, and a novel MITM attack is presented.

2.1 Prior Attacks

(Savard and Pekelney, 1999) describe a ciphertext-only attack on SIGABA which requires a series of 10 to 15 messages in depth, that is, encrypted with the same key.[4] First, the plaintexts are recovered using Kirchoffs superimposition.[5] The alphabets which represent the effect of the five *cipher* rotors are reconstructed, for each position of the ciphertexts/plaintexts. Positions at which only the leftmost or the rightmost *cipher* rotor move are identified, and the wiring of those rotors are recovered, by comparing the alphabet at such a position with the alphabet at the following position. The authors describe how the wiring of the inner *cipher* rotors can also be recovered, and they suggest additional methods to recover the wiring of the *control* rotors.

In (Lee, 2003), attacks on simplified and weakened versions of the SIGABA are presented.

(Stamp and Chan, 2007; Stamp and Low, 2007) describe a known-plaintext attack in two phases. It assumes that the wiring of the rotors is known, while their order, orientation, and starting positions, are unknown. At the first phase, only the *cipher* rotors are considered, and all their possible settings (rotor selection, orientation, starting positions) are evaluated. For each such setting, the first ciphertext symbol is applied across the five *cipher* rotors, and the decrypted symbol is compared to the known-plaintext symbol. To check the second symbol, all options for the stepping of the *cipher* rotors are tested (there are 30 such options, as at least one rotor steps, and at most four), and the *cipher* rotors step accordingly. If after stepping, decrypting the second ciphertext symbol produced the expected known-plaintext symbol, the process is repeated for the next symbols. Only those *cipher* rotor settings that survive the test for all the known-plaintext symbols are retained. With 100 letters of known-plaintext, about $2^{34.5}$ *cipher* rotor settings are expected to survive, out of $2^{43.4}$ possible *cipher* rotor settings.

At the second phase, all the surviving *cipher* rotor settings are exhaustively tested against all possible *control* and *index* settings. The total workfactor of the attack is $2^{86.7}$, and it is therefore more

[4] An unlikely scenario, given the US Army and Navy's strict operational security procedures. The method, however, requires little processing time, unlike the other attacks presented in here.

[5] The authors do not provide any detail on how the plaintexts can be recovered using superimposition. Furthermore, the required length for the messages in depth is not specified.

efficient than a simple brute-force attack by a factor of $2^{95.6-86.7} = 2^{8.9}$. The authors also propose a method with a workfactor of $2^{84.5}$, but with only a 0.82 probability of success.

2.2 A New Meet-in-the-Middle Known-Plaintext Attack

This new attack assumes that the wiring of the rotors is known, but their order, orientation, and starting positions are unknown. It was inspired by the attack described in (Stamp and Chan, 2007), and also consists of two phases. While (Stamp and Chan, 2007) is essentially an (optimized) exhaustive search, the new attack is a divide-and-conquer MITM attack. It is significantly more efficient than a simple brute-force attack.[6] It only requires a minimum of 8 known-plaintext symbols.[7] The first phase (described in Section 2.3) generates a set of *cipher* rotor settings and stepping sequences so that the expected known plaintext is accurately reproduced when decrypting the ciphertext. The second phase (described in Section 2.4) generates feasible *cipher* stepping sequences, by testing all possible *control* and *index* settings, and matching the resulting *cipher* rotor stepping sequences against those gathered during the first phase. The whole process – phase 1 and phase 2 – is repeated for all possible partitions of the ten *cipher* and *control* rotors, into two sets of five rotors.

[6]MITM attacks are applicable to modern multi-stage encryption systems (Diffie and Hellman, 1977). The approach is illustrated here with a system with two sequential encryption stages, E_1 and E_2. For simplicity, we assume each stage has a separate key, K_1 and K_2, with N_1 and N_2 bits, respectively. A known-plaintext MITM attack for such a system could potentially be developed. The attack has two phases. In the first phase, the plaintext is encrypted with E_1 only, for each value of K_1. The resulting encryptions are stored is a hash table mapping each such partial encryption to the relevant K_1. The second phase checks for all possible values of K_2, decrypts the ciphertext using only E_2, and checks whether this partial decryption (via E_2) matches one of the partial encryptions (via E_1) stored in the hash table. The overall complexity of this attack is the maximum of 2^{N_1} and 2^{N_2}, compared to $2^{N_1+N_2}$ for a brute-force search. This comes at the expense of additional memory for the hash table, which the two phases use to "meet in the middle". Such an attack is also effective against 2-DES (seriated DES with two stages, each using a 56-bit key), and to achieve a level of security higher than with DES (or 1-DES), three stages (or 3-DES) are required.

[7]The attack also works, albeit less efficiently, with less than 8 known plaintext symbols. It may also utilize more than 8 symbols, but the author found that this number allows for a good trade-off between the size of the required memory and the overall processing time for the attack.

2.3 Phase 1 of Meet-in-the-Middle Attack

At the first phase, only the *cipher* rotors are considered. For each partition of the ten *cipher* and *control* rotors (divided into two sets of five rotors), the first phase produces a hash table mapping all *cipher* rotor stepping sequences, to their corresponding *cipher* rotor settings, that together produce a decryption matching the 8 known-plaintext letters. The structure and contents of the hash table is described later in this section. This is different from the first phase of (Stamp and Chan, 2007), which instead generates a simple list of matching *cipher* rotor settings, but other than that, the first phase of the new attack and of (Stamp and Chan, 2007) are similar.

All orders of the five rotors (allocated for the *cipher* bank in the partition), their orientations, and starting positions are tested. The first ciphertext symbol is applied through the five *cipher* rotors, and the output is compared to the expected known-plaintext symbol. If they match, all possible stepping options are tested, and the second symbol is processed and checked. The next symbols are (recursively) checked, and if there is a match for all the known-plaintext symbols, the corresponding *cipher* rotor stepping sequence and the *cipher* rotor settings are added to the hash table.

Stepping Sequence (Hash Key)	(Maps to) **Cipher Rotor Settings**						
		Rotor Selection					**Starting Positions**
		1	2	3	4	5	
01011 01000 11001 00111 00111 11110 00010	⇒	8R	0	4R	7	1	H Y J N H
	⇒	1	7R	0	8R	4	T U A L M
01111 01100 11100 11001 00010 10101 10001	⇒	1R	4	8R	7	0	K H J N M
11010 10011 00111 10101 00111 00110 10011	⇒	0	8R	7R	4	1R	E Q A M B

Figure 4: Meet-in-the-Middle Attack – Hash Table

The structure of the hash table is illustrated in Figure 4. This example is for a partition in which rotors 0, 1, 4, 7, and 8 are allocated to the *cipher* rotors. The hash key represents the stepping sequence of the *cipher* rotors, applied during the decryption of the 8 symbols for which there is known plaintext. Since stepping occurs after decryption, only the first 7 stepping patterns are relevant, as the eighth stepping pattern only affects the ninth symbol. Each stepping pattern consists of 5 Boolean values, one for each *cipher* rotor. The hash table key therefore consists of 7 groups of 5 bits.

The first group represents the stepping of the *ci-*

pher rotors after decrypting the first symbol. In the first entry in the hash table illustrated in Figure 4, 01011 indicates that *cipher* rotors in slots 2 (from the left), 4, and 5 (the rightmost rotor) step after decrypting the first symbol. Similarly, the next group, 01000, indicates that only the *cipher* rotor in slot 2 (from the left) steps after encrypting the second symbol.

A hash key (the stepping sequence) maps into one or more *cipher* rotor settings.[8] Each such setting includes the selection and order of the rotors in the 5 slots of the *cipher* rotor bank, their orientation, and their starting positions. In the first entry in Figure 4, the order of the *cipher* rotors is 8, 0, 4, 7, and 1 (installed in *cipher* bank slots 1 to 5, from left to right). Rotors 8 and 4 (first and third from the left) are in the reversed orientation (marked as 8R and 4R, respectively). The starting positions of the *cipher* rotors are H, Y, J, N, and H, respectively. The same stepping sequence (the first in Figure 4) also maps to a second *cipher* setting, with 1, 7, 0, 8, and 4 as the order of the rotors (7 and 8 are in the reversed orientation), and T, U, A, L, and M as their starting positions. This illustrates the fact that the hash table implements a one-to-many mapping, as there might be several distinct *cipher* rotor settings which reproduce the known plaintext, while the rotors step in an identical manner (as represented by the stepping sequence which is also the hash key).

Similarly, the second stepping sequence (hash key) in the hash table (Figure 4), which starts with 01111, maps to only one setting, with 1, 4, 8, 7, and 0 as the order of the rotors (1 and 8 are in reversed orientation), and set at starting positions K, H, J, N, and M.

Note that the presence of a combination of a stepping sequence and *cipher* rotor setting in the hash table, only indicates that under that same combination, the 8 symbols of the ciphertext can be decrypted to match the expected known plaintext. It does not indicate that such a stepping sequence is feasible and can be produced by the *control* and *index* rotor banks. Therefore, the need for a second phase, in order to generate all feasible *cipher* stepping sequences, and check whether they appear in the hash table created by the first phase. This second phase is described in Section 2.4.

[8]This is different from most MITM attacks, where the shared memory structure maps partial encryption or partial decryption results to partial keys. Instead, in this attack on SIGABA, the hash table maps stepping sequences of the *cipher* rotors to partial keys (the *cipher* rotor settings).

2.4 Phase 2 of Meet-in-the-Middle Attack

The second phase looks at all possible *control* and *index* settings, generates their resulting *cipher* rotor stepping sequences, and checks whether those stepping sequences exist in the hash table. If such a stepping sequence exists, then the combination of the *control* and *index* settings, together with the *cipher* setting associated with the stepping sequence, constitutes a candidate key. When applying such a candidate key to decrypt the 8 ciphertext symbols, the decryption is guaranteed to match the expected known plaintext.

Still, this is only a candidate key, as the fact that it properly decrypts the given 8 ciphertext symbol does not mean it will necessarily properly decrypt the rest of the message. To validate a candidate key, there are two options. Either more than 8 plaintext symbols are known, and the candidate key can be validated by checking whether it properly reproduces the remaining known-plaintext symbols. Alternatively a quality measure such as the Index of Coincidence can be applied to the decryption of the full message, together with a minimal threshold. Both methods may be combined, if the known plaintext is not long enough to safely rule out wrong candidate keys.[9]. Both validation methods also impact the workfactor. The choice of processing 8 letters of known-plaintext is a trade-off between the complexity of phase 1 (the longer the plaintext, the more steps in phase 1), the storage requirements, and the need to validate phase 2 matches (the longer the known-plaintext, the lower the probability for phase 2 false positives and the need for validation).

2.5 Workfactor Analysis

Phase 1 and phase 2 of the attack are applied repeatedly, on each partition of 10 rotors (those with 26 input and outputs) into a set of five *cipher* rotors and another set of five *control* rotors. The number of such partitions is equal to the number of ways to select 5 unordered rotors from a set of 10 rotors, that is, $10!/(5! \cdot 5!) = 252 = 2^{7.9}$.

Workfactor Analysis for Phase 1

For each partition, the number of possible *cipher* rotor settings is $5! \cdot 2^5 \cdot 26^5 = 2^{35.4}$. When evaluating a certain *cipher* setting, all options for the

[9]Based on experiments, 12 to 22 letters of known plaintext are necessary to rule out all 'false-positive' candidate keys.

stepping of the rotors are checked for each known-plaintext-ciphertext pair (the stepping after the 8th symbol is ignored). Since between one to four *cipher* rotors step after decryption, there are only $2^5 - 2 = 30$ possible stepping options, out of the theoretically possible $2^5 = 32$ stepping patterns of the 5 rotors (patterns 00000 – none of the rotors step, and 11111 – all rotors step, are not feasible). For each symbol tested, the probability or ruling out such a stepping option is $(26-1)/26$. So on average, $30 \cdot (1/26) = 1.154$ option for stepping of the *cipher* rotors survive after each decryption step, out of 30. The total number of operations for each *cipher* rotor setting is therefore $\sum_{i=0}^{7} 1.154^i = 13.9 = 2^{3.8}$. The total workfactor for phase 1 for a single partition is $2^{35.4+3.8} = 2^{39.2}$, and $2^{39.2+7.9} = 2^{47.1}$ for all partitions.

Workfactor Analysis for Phase 2

For a given partition, the number of possible *control* rotor settings is $5! \cdot 2^5 \cdot 26^5 = 2^{35.4}$, and the number of feasible options for the *index* rotors is $10!/2^{32} = 113,400 = 2^{16.8}$. Therefore, the workfactor for phase 2 is $2^{35.4+16.8} = 2^{52.2}$ for a single partition, and $2^{52.2+7.9} = 2^{60.2}$ for all partitions.

Overall Workfactor

The workfactor for phase 2 is the dominant one, and therefore, the overall workfactor for the attack is $2^{60.2}$. This attack is more efficient than a brute-force attack by a factor of $2^{95.6-60.2} = 2^{35.4}$, and it is feasible with modern technology. For comparison, a brute-force attack on a 56-bit DES key was successfully carried out already in 1998 (Gilmore, 1998).

Storage Requirements

Since the attack is a MITM attack, we still need to address the size of the hash table, which is generated in phase 1, separately for each partition of the *cipher* and *control* rotors. Based on simulations, the ratio between the number of stepping sequences generated by phase 1 and the number of possible *cipher* rotor settings is on average 0.107. There are therefore approximately $5! \cdot 2^5 \cdot 26^5 \cdot 0.107 = 2^{32.1}$ sequences generated for each partition of the ten rotors.

To represent a stepping sequence in the hash table (see Table 4), $7 \cdot 5 = 35$ bits are required. To represent the *cipher* rotor setting, 7 bits are required for the order ($5! = 120 \leq 2^7$ options), 5 bits for their orientation, and $5 \cdot 5$ bits for their starting positions, with a total of $7 + 5 + 5 \cdot 5 = 37$ bits. Since the vast majority of the entries in the hash table map to only one *cipher* rotor setting, the total size for an entry is about $35 + 37 = 72$ bits.[10] Implementing and storing this information in a practical hash table in RAM requires some additional overhead, and based on measurements using Java 10 Hashmap library, the overall amount of space required for each entry is approximately twice the amount of space for just the data of the entry, that is, $2 \cdot 72 = 144$ bits or about 18 bytes.[11] Therefore, the memory size required for the per-partition hash table can be estimated to be $2^{32.1} \cdot 18 = 4.6 \cdot 10^9 \cdot 18 = 80$ GB.

A possible optimization to reduce the size of the hash table consists of checking for additional matching known-plaintext-ciphertext symbols, if more than 8 plaintext symbols are known. For example, by checking an additional 7 symbols (total of $8 + 7 = 15$), about half of the stepping sequences may be ruled out. When checking the additional symbols (following the initial 8 symbols), we do not extend the size of the stepping sequence stored as the key in the hash table, and we still keep only the first 7 stepping patterns. We only check whether there is at least one possible continuation of this stepping sequence, following the 8 initial decryption steps. However, this optimization would also incur additional processing at phase 1, which would probably not affect the overall workfactor, as the workfactor of phase 2 is overwhelmingly dominant.

The results in this section are based on simulations performed on 200 hundreds of random keys and plaintexts. Further analysis is required to determine the optimal parameters (e.g. trade-off between processing time and storage space, derived from the length of the known plaintext processed), as well as more precise workfactor and storage analyses, for a practical attack on the full keyspace.

2.6 Solving the MysteryTwister C3 SIGABA Level III Challenge

In (Stamp, 2010), a known-plaintext SIGABA challenge is given. Due to the particular method the challenge was created, the effective size of keyspace is reduced, making the attack described above prac-

[10] Based on simulations.

[11] A more (or less) efficient implementation of a hash table may require different amounts of overhead.

tical on a consumer PC. With a probability of 31/32, it may be assumed that the starting position of four out of the five *cipher* rotors is A, and the same applies to the *control* rotors. If we assume this is indeed the case, the size of the keyspace of the *cipher* rotors and the *control* rotors are each both reduced by a factor of $26^4 = 2^{18.8}$. Under the same assumption, the workfactor of phase 1 for this challenge is therefore $2^{47.1-18.8} = 2^{28.3}$, and for phase 2, the workfactor is $2^{60.2-18.8} = 2^{41.3}$.

It took a few days on a 10-core Intel Core i7 6950X 3.0 GHz PC to complete the attack and solve the challenge. The assumption stated above was found to be true.[12]

3 New Challenges

A series of new SIGABA known-plaintext challenges is presented in Table 1 (see the Appendix), with various levels of difficulty.[13] For most challenges, the size of the keyspace is limited by setting several of the *cipher* and *control* rotors to a fixed position A. Challenge #2 is against a keyspace of a size similar to the keyspace for the challenge in (Stamp, 2010). The last challenge (#6) is against the full keyspace of SIGABA. Java source code, used to generate the challenges, is listed in the Appendix. It is compatible with the source code given in (Pekelney, 1998), and has been tested against another simulator (Sullivan, 2002a).[14]

4 Conclusion

The functional separation between the *cipher* rotor bank, and the *control* and *index* rotor banks, is a significant weakness, and it allows for a practical MITM attack. This attack is not feasible on systems like the Siemens and Halske T52d, the Russian Fialka, and the Hagelin CX-52, in which rotors have two functions – encryption/decryption, and controlling the stepping of other rotors.[15] Still, the attack on SIGABA proposed here would not have been feasible given WWII technology.

[12] *jerva* and *Integral* had previously found the solution to the challenge. *jerva* used methods described in (Stamp and Chan, 2007). *Integral*'s methods are unknown.

[13] All the plaintexts were extracted from Shakespeare writings. Each plaintext consists of the concatenation of two segments, extracted from different places. The first segment, with 100 letters, is given as a crib. The letter Z is used to represent a space.

[14] The SIGABA simulator source code given in (Stamp, 2010) and used to generate previous MysteryTwister C3 challenges has several incompatibility issues. The first one, described in the forum discussion, has to do with several rotors being unintentionally reset to position A. Another issue is with the stepping logic for the *medium* and *slow* rotors. In addition, the mappings for rotors in reversed orientation are incorrect.

[15] A MITM attack is also not feasible against Enigma. Although there are multiple components involved, there is no feasible 'meet-in-the-middle point' as the encryption path traverses the plugboard and the rotors back and forth via the reflector.

References

Stephen Budiansky. 2000. *Battle of wits: the complete story of codebreaking in World War II.* Simon and Schuster.

Whitfield Diffie and Martin E Hellman. 1977. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84.

John Gilmore. 1998. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* OReilly.

Stephen J. Kelley. 2001. Big Machines: Cipher Machines of World War II.

Michael Lee. 2003. *Cryptanalysis of the SIGABA, Master's Thesis.* University of California, Santa Barbara.

Timothy Jones Mucklow. 2015. *The SIGABA/ECM II Cipher Machine: "a Beautiful Idea".* National Security Agency, Center for Cryptologic History.

Richard S. Pekelney. 1998. ECMApp - Emulation of ECM Mark II. `https://maritime.org/tech/ecmapp.txt`, [Accessed: January, 18th, 2019].

John J. G. Savard and Richard S. Pekelney. 1999. The ECM Mark II: Design, History, and Cryptology. *Cryptologia*, 23(3):211–228.

Mark Stamp and Wing On Chan. 2007. SIGABA: Cryptanalysis of the Full Keyspace. *Cryptologia*, 31(3):201–222.

Mark Stamp and Richard M. Low. 2007. *Applied Cryptanalysis: Breaking Ciphers in the Real World.* John Wiley & Sons.

Mark Stamp. 2010. MysteryTwister C3 (MTC3), SIGABA Part 2 (Level III). `https://www.mysterytwisterc3.org/en/challenges/level-iii/sigaba-part-2`, [Accessed: December, 16th, 2018].

Geoff Sullivan. 2002a. CSG Sigaba (ECM Mark II) Simulator for Windows. `http://cryptocellar.org/simula/sigaba/index.html`, [Accessed: January, 18th, 2019].

Geoff Sullivan. 2002b. The ECM Mark II: some observations on the rotor stepping. *Cryptologia*, 26(2):97–100.

5 Appendix – Source Code and Challenges

Listing 1: SIGABA Simulator Source Code

```
package simulator;

class Sigaba {
    private Rotor cipherBank[] = new Rotor[5];
    private Rotor controlBank[] = new Rotor[5];
    private IndexRotor indexBank[] = new IndexRotor[5];
    Sigaba(String cph, String ctl, String idx,
           String cphP, String ctlP, String idxP) {
        for (int i = 0; i < 5; i++) {
            cipherBank[i] =
              new Rotor(cph.charAt(i * 2) - '0',
                        cph.charAt(i * 2 + 1) == 'R',
                        cphP.charAt(i) - 'A') ;
            controlBank[i] =
              new Rotor(ctl.charAt(i * 2) - '0',
                        ctl.charAt(i * 2 + 1) == 'R',
                        ctlP.charAt(i) - 'A');
            indexBank[i] =
              new IndexRotor(idx.charAt(i) - '0',
                             idxP.charAt(i) - '0');
        }
    }
    String encryptDecrypt(boolean decrypt, String in) {
        String outString = "";
        for (char c : in.toCharArray()) {
            outString +=
              (char)(cipherPath(decrypt, c - 'A') + 'A');
            cipherBankUpdate();
            controlBankUpdate();
        }
        return outString;
    }
    private void controlBankUpdate() {
        if (controlBank[2].pos == (int) 'O' - 'A') {
            // medium rotor moves
            if (controlBank[3].pos == (int) 'O' - 'A') {
                // slow rotor moves
                controlBank[1].advance();
            }
            controlBank[3].advance();
        }
        // fast rotor always moves
        controlBank[2].advance();
    }
    private static final int INDEX_IN[] =
                    {9, 1, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6,
                     6, 6, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8};
    // rotor stepping magnet
    private static final int INDEX_OUT[] =
            {1, 5, 5, 4, 4, 3, 3, 2, 2, 1};
    private void cipherBankUpdate() {
        boolean move[] = new boolean[5];
        for (int i = (int) 'F' - 'A';
             i <= (int) 'I' - 'A';
             i++) {
          int indexIn = INDEX_IN[controlPath(i)];
          move[INDEX_OUT[indexPath(indexIn)] - 1] = true;
        }
        for (int i = 0; i < 5; i++) {
          if (move[i]) cipherBank[i].advance();
        }
    }
    private int cipherPath(boolean decrypt, int c) {
        if (decrypt) {
          for (int r = 4; r >= 0; r--)
            c = cipherBank[r].rightToLeft(c);
        } else {
          for (int r = 0; r <= 4; r++)
            c = cipherBank[r].leftToRight(c);
        }
        return (c);
    }
    private int controlPath(int c) {
        for (int r = 4; r >= 0; r--)
          c = controlBank[r].rightToLeft(c);
        return (c);
    }
    private int indexPath(int c) {
        for (int r = 0; r <= 4; r++)
          c = indexBank[r].indexPath(c);
        return (c);
    }

    static class Rotor {
        private static final String[] WIRINGS = {
                "YCHLQSUGBDIXNZKERPVJTAWFOM",
                "INPXBWETGUYSAOCHVLDMQKZJFR",
                "WNDRIOZPTAXHFJYQBMSVEKUCGL",
                "TZGHOBKRVUXLQDMPNFWCJYEIAS",
                "YWTAHRQJVLCEXUNGBIPZMSDFOK",
                "QSLRBTEKOGAICFWYVMHINXZUDP",
                "CHJDQIGNBSAKVTUOXFWLEPRMZY",
                "CDFAJXTIMNBEQHSUGRYLWZKVPO",
                "XHFESZDNRBCGKQJILTVMUOYAPW",
                "EZJQXMOGYTCSFRIUPVNADLHWBK"};
        // Index for left to right.
        private static final int TO_RIGHT = 0;
        // Index for right to left.
        private static final int TO_LEFT = 1;
        private int wiring[][] = new int[2][26];
        int pos;
        private boolean reversed;
        Rotor(int wiringIndex, boolean reversed, int pos) {
            for (int i = 0; i < 26; i++) {
              wiring[TO_RIGHT][i] =
                WIRINGS[wiringIndex].charAt(i) - 'A';
              wiring[TO_LEFT][wiring[TO_RIGHT][i]] = i;
            }
            this.reversed = reversed;
            this.pos = pos;
        }
        void advance() {
            if (reversed) {
                pos = (pos + 1) % 26;
            } else {
                pos = (pos - 1 + 26) % 26;
            }
        }
        int leftToRight(int in) {
            if (!reversed) {
              return
                (wiring[TO_RIGHT][(in+pos)%26]-pos+26)%26;
            }
            return
              (pos-wiring[TO_LEFT][(pos-in+26)%26]+26)%26;
        }
        int rightToLeft(int in) {
            if (!reversed) {
              return
                (wiring[TO_LEFT][(in+pos)%26]-pos+26)%26;
            }
            return
              (pos-wiring[TO_RIGHT][(pos-in+26)%26]+26)%26;
        }
    }
    static class IndexRotor {
        private static final int WIRINGS[][] = {
                {7, 5, 9, 1, 4, 8, 2, 6, 3, 0},
                {3, 8, 1, 0, 5, 9, 2, 7, 6, 4},
                {4, 0, 8, 6, 1, 5, 3, 2, 9, 7},
                {3, 9, 8, 0, 5, 2, 6, 1, 7, 4},
                {6, 4, 9, 7, 1, 3, 5, 2, 8, 0}};
        private int wiring[] = new int[10];
        private int pos;
        IndexRotor(int wiringIndex, int pos) {
            System.arraycopy(WIRINGS[wiringIndex], 0,
              wiring, 0, 10);
            this.pos = pos;
        }
        int indexPath(int in) {
          return (wiring[(in + pos) % 10] - pos + 10) % 10;
        }
    }
    public static void main(String[] args) {
        Sigaba sigaba =
         new Sigaba("0R1N2N3N4R", "5N6N7R8N9N",
                    "01234", "ABCDE", "FGHIJ", "01234");
        String out = sigaba.encryptDecrypt(false,
          "AAAAAAAAAAAAAAAAAAAA");
        System.out.printf(
          "%s (expecting JISCALXDRWOQKRXHKMVD) \n", out);
        sigaba =
          new Sigaba("0R1N2N3N4R", "5N6N7R8N9N",
                     "01234", "ABCDE", "FGHIJ", "01234");
        String in = sigaba.encryptDecrypt(true, out);
        System.out.printf(
          "%s (expecting AAAAAAAAAAAAAAAAAAAA)\n", in);
    }
}
```

	Ciphertext	First 100 Plaintext Letters	Hint
#1	GSZQEMAGFULNFZHHRVUTCUEXU FBMPDGOROJRPMAUDOZMJWJCVH YCBZDELOWKVLYJLSZBQJXWXLR WOIMBVUTBAVRHPPPYQDTIURLV IQGIZSEVGXOYCMGESFOXDLPFT UQQCRDSRNFDTBDDULFJKQGXZB XKKIMSBSIUZSZNOOLCFRRVTOD XFQRRXLDEMSLORKXUCGDKCZKY ULDORUGEGDLTTROBUIVWJTBVH YWOKANYJCGQUYGPHSMWJRILZP SQJOXKKMEGMWQKXWVKF	AHZFOULZSHREWDZNEWSZBESHR EWZTHYZVERYZHEARTZIZDIDZN OTZTHINKZTOZBEZSOZSADZTON IGHTZASZTHISZHATHZMADEZME	All *cipher* and *control* rotors are at position A.
#2	ZMJHMLJTJSSHZBBMYXJRVZCUS PMETNBPZQCAHGYJDHJNQNMTHY EJAOOQYFSURONLTOGQVKOMABX QXGKRAVBZYWBRWYWGLBYFZNNA XIVJVOJYYBQGTWJIIZESYBRAN XEWYDRMYAINJWWDFWBVCTHRGL ZCTNHWWBRYSJSZSYMSSLUXBLZ STDBARVGCSMTJOWIRFXYIBZCF CCYRUXMUCISNUIFLCOJYZQTBY DWVFJDHZBJNSAPYAUYWQGFPYO ZJYWPCWVRSVCQTPHTFPGHCJAM CFZRHYNFXJVWWNNN	WOULDSTZTHOUZNOTZBEZGLADZ TOZHAVEZTHEZNIGGARDLYZRAS CALLYZSHEEPBITERZCOMEZBYZ SOMEZNOTABLEZSHAMEZFABIAN	The last 4 *cipher* rotors and the last 4 *control* rotors are at position A.
#3	HYYQUSBFHVDVKSLKSGUQIVZAR QKCQZBLLGCTCLQHZNBEQVUOJH BROKUKRYXWPGSPDJSWLLTDASB MTTPRPFHMSXPLBDENAYJWAQZD JDXGBJCWXNARABTTSEZBJDYHT NEIQCQRTFUAZDTTVBNHJGWQHF UHAPPBPYJAIXGELTILPULVSNC BJJIGFJNYDURTIVWYHTNKFSLS ALTHLBHYQBYXUK	TISZWONDERFULZWHATZMAYZBE ZWROUGHTZOUTZOFZTHEIRZDIS CONTENTZNOWZTHATZTHEIRZSO ULSZAREZTOPFULZOFZOFFENCE	The last 3 *cipher* rotors and the last 3 *control* rotors are at position A.
#4	CEXZZGZOYLDYPAGJQTFJSEYZP ORHMSTYLQVSJARJLCDBYXFPKB NREAEYVOPBQKYFYETXOUQNMAT CBWIIFKJWZJFWZHMJYQALVNXV UDUVEJGJNBWZRCVMIHDHLPOSD LSBPTFNEGWIAIRZZPIPPVEBWV VBGLNCGBKWFUUCVGTTGKGEHJQ XGEHVPLDDLALNWVNDOXTPPWCQ HNAWFTXVOWIZFVRWXBIIJDFAU TMCNWDHLSCHNOBQRURVLCXLVB YXDXKMPYIWPYOXPFXBNESBUCR WZECWXOUDTVVNRGGHPTE	IZWILLZBESPEAKZOURZDIETZW HILESZYOUZBEGUILEZTHEZTIM EZANDZFEEDZYOURZKNOWLEDGE ZWITHZVIEWINGZOFZTHEZTOWN	The last 2 *cipher* rotors and the last 2 *control* rotors are at position A.
#5	JJJWJZMPUKYDGRHSPIXTYPAPA IVGFOTXMFWRZLBRXQPNRYLCPF WNMZFHFSMVIEEDAHWZOMBIVPA RTAOWYOWRFACGAITUAFDFCTEV YZAQIQXVHZFCIBSVSQJAMYPTS YNWXBFBKDKVDOXZQQEVVGAAWI LRFYRGIPJCKVVPMQAEIAIMOPY XCSJFDAUHYZYVQJXGGZTMCAGW BEICRYROYCPNGEZQFVVQTSZBP SZYWCONNWMUBCNYQX	HOWZMIGHTZWEZSEEZFALSTAFF ZBESTOWZHIMSELFZTONIGHTZI NZHISZTRUEZCOLOURSZANDZNO TZOURSELVESZBEZSEENZPOINS	The last *cipher* rotor and the last *control* rotor are at position A.
#6	FWEYNOPSTLFMWXQITVTMRVHOL YDEIROBXPPVZVBLCSJPSYIXIY IJHJMCHAWSWAQBHSUVASAGYLR DJREKIFQUXBEJZUFVIJBJMWVT VSPHOQTRAECHEEJLBRCDTGXRP OVSJKDYYWNWIUTPXKVXSHDCBC WVYDGBVJLMCPZJROXKDPTDTMC PHXGCTHPDLVHYQHHFRTTKSOTE IWAXEDMUOVBLSLZUWFTYGNCQY YPHZRNJRBXYVVSNPYWAEMXOIV UQWAXAECBOODIPLWGCVQJVDCX GKCBXHCUK	TOZHAVEZNOZSCREENZBETWEEN ZTHISZPARTZHEZPLAYDZANDZH IMZHEZPLAYDZITZFORZHEZNEE DSZWILLZBEZABSOLUTEZMILAN	No hint given.

Table 1: New SIGABA Challenges