IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2025, No. 1, pp. 309–327. DOI:10.46586/tosc.v2025.i1.309-327

GPU Assisted Brute Force Cryptanalysis of GPRS, GSM, RFID, and TETRA

Brute Force Cryptanalysis of KASUMI, SPECK, and TEA3

Cihangir Tezcan[1,2] and Gregor Leander[3]

[1] Department of Cyber Security, Graduate School of Informatics, Middle East Technical University, 06800 Ankara, Turkey
firstname@metu.edu.tr
[2] Department of Computer Science and Engineering, Sabanci University, 34956 Istanbul, Turkey
[3] Ruhr University Bochum, Bochum, Germany
firstname.lastname@rub.de

Abstract. Key lengths in symmetric cryptography are determined with respect to the brute force attacks with current technology. While nowadays at least 128-bit keys are recommended, there are many standards and real-world applications that use shorter keys. In order to estimate the actual threat imposed by using those short keys, precise estimates for attacks are crucial.
In this work we provide optimized implementations of several widely used algorithms on GPUs, leading to interesting insights on the cost of brute force attacks on several real-word applications.
In particular, we optimize KASUMI (used in GPRS/GSM), SPECK (used in RFID communication), and TEA3 (used in TETRA). Our best optimizations allow us to try $2^{35.72}$, $2^{36.72}$, and $2^{34.71}$ keys per second on a single RTX 4090 GPU. Those results improve upon previous results significantly, e.g. our KASUMI implementation is more than 15 times faster than the optimizations given in the CRYPTO'24 paper [ACC+24] improving the main results of that paper by the same factor.
With these optimizations, in order to break GPRS/GSM, RFID, and TETRA communications in a year, one needs around 11, 22 billion, and 1.36 million RTX 4090 GPUs, respectively.
For KASUMI, the time-memory trade-off attacks of [ACC+24] can be performed with 142 RTX 4090 GPUs instead of 2400 RTX 3090 GPUs or, when the same amount of GPUs are used, their table creation time can be reduced to 20.6 days from 348 days, crucial improvements for real world cryptanalytic tasks.

Keywords: KASUMI · A5/3 · SPECK · TEA3 · cryptanalysis · GPU

1 Introduction

Cryptographic algorithms with short keys are susceptible to generic attacks like exhaustive key search and time-memory trade-off (TMTO) attacks in which an initial exhaustive key search like attack is performed to create tables in order to perform future exhaustive key searches in a short amount of time. For this reason, NIST recommends [FR24] at least 112-bit keys for symmetric key encryption algorithms and keys that will be used after 2030 should be at least 128-bit. In this respect, NIST removed SKIPJACK and 3DES from its standards which provided 80-bit and 112-bit security, respectively.

Although NIST took measures to remove algorithms with short keys from its standards, there are still many ISO/IEC standards with short keys. For instance, ISO/IEC 29192-3 lightweight stream cipher standard [ISO12] have two stream ciphers: Enocoro and Trivium.

Received: 2024-11-22 Accepted: 2025-01-23 Published: 2025-03-07

`Enocoro` supports both 80 and 128-bit keys. However, `Trivium` works only with 80-bit keys. Similarly, ISO/IEC 29192-2 [ISO19] lightweight block cipher standard `PRESENT` supports both 80 and 128-bit keys. And finally, the NSA designed block cipher `SPECK` supports short keys like 64, 72, and 96 bits. The 96-bit version became an ISO/IEC 29167-22 [ISO18] RFID air interface standard in 2018. Note that, although `SPECK` also supports longer keys, having shorter keys in a standard make them preferable for speed and low hardware footprint.

Another example for a short key is the `A5/1` stream cipher used in 2G GSM communications. `A5/1` stream cipher uses 64-bit key and it is very easy to eavesdrop GSM communications via exhaustive search or TMTO attacks [BBK03]. Due to these attacks, GSM Association adopted `KASUMI` block cipher as `A5/3` for 3G and GPRS. `KASUMI` block cipher supports 128-bit secret key, but for backward compatibility `A5/3` and GPRS use concatenation of the same 64-bit key twice as a 128-bit key, resulting in 64-bit security. Recently, practical TMTO attacks for these two protocols were provided in [ACC+24] which requires hundreds of GPUs to run for hundreds of days to create TMTO tables.

Sometimes algorithms that use short keys do not receive academic cryptanalysis due to security by obscurity practices. For instance, some cryptographic algorithms like the ones used in the European standard TETRA which is globally used by military, police, emergency services, prisons, and government agencies were kept secret for decades contrary to Kerckhoffs's principle. However, it was recently shown in [MBW23] that it is possible to reverse engineer the cryptographic algorithms used in this standard. It was shown that 80-bit secret keys are used in the four keystream generators used in TETRA and some of these algorithms were deliberately weakened to provide 32-bit security.

Moreover, many academic papers still propose new algorithms that use short keys like 64 or 80 bits. Thus, it is important to be able to measure how easy or hard it is to perform generic attacks on ciphers with short keys using general purpose computing devices.

Our Contribution

In this work, we optimized `KASUMI`, `SPECK`, and `TEA3` ciphers on GPUs to perform exhaustive key search attacks. Our best optimizations allowed us to try $2^{35.72}$, $2^{36.72}$, and $2^{34.71}$ keys per second on a single RTX 4090 GPU for `KASUMI`, `SPECK`-96-26, and `TEA3`, respectively. With these optimizations, in order to break GPRS/GSM, RFID, and TETRA communications in a year, one needs around 11, 22 billion, and 1.36 million RTX 4090 GPUs, respectively. Moreover, our `KASUMI` optimizations are 16.89x faster than the optimizations of recent CRYPTO'24 paper [ACC+24]. This result directly improves the main result of that paper by basically the same factor. More precisely, the time-memory trade-off attacks of [ACC+24] can be performed with 142 GPUs instead of 2400 GPUs or when the same amount of GPUs are used, their table creation time can be reduced to 20.6 days from 348 days.

Methodologically, those speed-ups are the result of many different GPU specific considerations. Although GPUs have thousands of cores, they are not as powerful as CPU cores. Moreover, GPU architecture imposes many limitations, making it a challenge to fully occupy the GPU in an implementation. In order to obtain the best performance and fully occupy the GPU, in our `KASUMI`, `SPECK`, and `TEA3` implementations we explored straightforward, table-based, and bitsliced implementation techniques. GPUs have different memory types namely registers, shared memory, and global memory. Registers are the fastest and global memory is the slowest when reading and writing. In our optimizations we tried to minimize the global memory usage, remove the shared memory bank conflicts and tried to minimize number of used registers. GPU kernels have two inputs: Number of blocks and number of threads in each block. To be able to do as much encryption as possible in a thread, we aimed to keep the number of threads to be as large as possible. To be able to run large number of threads in a kernel block, we tried to keep the used registers

to minimum. The number of used registers and some other properties actually depends on the used CUDA SDK and the target compute capability which actually represents a set of software and hardware features. Thus, we used different CUDA SDKs and compiled our codes for every possible compute capability that these SDKs allow to get the best performance.

Although our `KASUMI` optimizations are 16.89x faster than the optimizations of recent CRYPTO'24 paper [ACC+24], we do not know the main bottleneck causing this difference because the source codes of that work are not publicly available. Moreover, we have not observed any GPU optimizations of `SPECK` and `TEA3` ciphers. Thus, to the best of our knowledge, we are the first to provide GPU optimizations for these ciphers.

We made our CUDA codes for `KASUMI`, `SPECK`, and `TEA3` publicly available in order for the academic community to verify our results, better analyze these ciphers, verify theoretically obtained results, discover new properties, and compare future optimizations.

2 Preliminaries

Graphics processing units (GPUs) use single instruction multiple thread (SIMT) parallelization and provide especially superior speed compared to CPUs when the running algorithm is parallelizable. Modern GPUs have thousands of cores and since an exhaustive key search is embarrassingly parallelizable, each GPU core can try a different key without the need for communicating with other cores. However, GPU cores are not as powerful as CPU cores and due to architectural limitations, many optimizations need to be done to fully occupy the GPU.

In order to obtain the best optimizations, one needs to know the specifics of the used GPU. Some of the specifications of GPUs are provided by the manufacturers but some differences between different GPUs for some properties like the delays caused by shared memory bank conflicts can only be observed by performing experiments. And architectural changes between different generations of GPUs can significantly affect the performance of an implementation.

NVIDIA GPUs are categorized with respect to their compute capabilities (CC), which actually represents a set of software and hardware features. It should be noted that a CUDA device is backwards compatible. For example, an RTX 4090 GPU has a compute capability of 8.9 but it can run any code that is compiled with a lower compute capability. Codes compiled for different compute capabilities generally require different number of registers per thread and sometimes compiling for a lower compute capability provides better results. Currently the latest CUDA SDK has version 12.6 and the 12.x versions support compute capability 5.0 and higher. And the previous CUDA SDK 11.x versions support compute capability 3.5 and higher. In our benchmarks, we used many CUDA SDKs and compiled our codes for every compute capability that the SDK and the tested GPU supports to obtain the best results.

In this work, we used many different desktop and mobile GPUs with different architectures to show that our optimizations are valid for every GPU or architecture and not targeted for a specific GPU. The specifications of the GPUs that we used in this work are provided in Table 1.

Straightforward implementation, table-based implementation, and bitsliced implementation are the common strategies for implementing symmetric key encryption algorithms. But depending on the cipher design and GPU limitations, one technique may be superior to others. Thus, we tried these three strategies in our implementations.

Modern GPUs run many threads in blocks and they are grouped in warps that consist of 32 threads. On GPUs, data can be stored in registers, shared memory or global memory. Global memory is large but slow. Hence, we get the best optimizations when we can store everything in the fast registers. However, on modern GPUs we have at most 64K 32-bit

Table 1: The specifications of the GPUs that we used in this work. Clock rates are listed according to the maximum boost clock rates of the GPUs and they may differ depending on the manufacturer and the model. Table is sorted with respect to GPU compute capabilities (CC), which actually represents a set of software and hardware features.

GPU	Cores	Clock Rate	CC	Architecture
GTX 860M	640	1020 MHz	5.0	Maxwell
GTX 970	1664	1253 MHz	5.2	Maxwell
MX 250	384	1582 MHz	6.1	Pascal
RTX 2070 Super	2560	1770 MHz	7.5	Turing
RTX 4090	16384	2550 MHz	8.9	Ada Lovelace

registers for each block and a block can have at most 1024 threads. Generally if there are no bottlenecks, we can best occupy the GPU when we use 1024 threads which means we can spend at most 64 registers for each thread. Thus, generally it is not possible to use a bitsliced implementation for a symmetric encryption algorithm and still use blocks of 1024 threads because the implementation would require more than 64 registers to store internal values of the cipher.

Similarly, it is not possible to store large S-boxes in registers. And since global memory is slow, keeping these kinds of tables in shared memory provides the best speed. However, a warp can use 32 data lanes to reach the shared memory and if two threads try to use the same data lane, this causes a shared memory bank conflict and one thread needs to wait the other one. One way to avoid this problem is to store 32 copies of the S-box so that every thread in a warp can use its own data lane. This approach is used in [Tez21] for the table-based implementation of `AES` and the best performance of `AES` on GPUs was obtained via this technique.

There are many academic papers on GPU optimizations of symmetric encryption algorithms to obtain fast encryption. For instance, `AES` was optimized on GPUs many times in the past using the best commercially available GPUs of the time (e.g. [NAI17], [AS20b], [AFDM17], and [AS20a]). However, none of the source codes of those implementations were made publicly available. And it is not possible to make a fair comparison when two implementations are benchmarked on different GPUs. For this reason we made our codes publicly available and we compare our results in Table 2 with the best block cipher optimizations on GPUs which have publicly available source codes. It shows that our `KASUMI` and `SPECK` implementations provide more key trials per second than `AES`, `DES`, `KLEIN`, and `PRESENT`.

Table 2: Exhaustive key search attack performance on GPUs for various symmetric key encryption optimizations.

Cipher	Key	Block	Rounds	RTX 2070 Super	RTX 4090	Reference
PRESENT-80	80	64	31	$2^{29.73}$ keys/s	$2^{32.90}$ keys/s	[Tez22]
DES/3DES	56/168	64	16	$2^{30.78}$ keys/s	$2^{33.94}$ keys/s	[Tez22]
AES-128	128	128	10	$2^{32.43}$ keys/s	$2^{34.64}$ keys/s	[Tez21]
TEA3	80	-	-	$2^{32.54}$ keys/s	$2^{34.71}$ keys/s	This paper
KLEIN-64	64	64	12	$2^{33.19}$ keys/s	$2^{35.40}$ keys/s	[Tez24]
KASUMI	128	64	8	$2^{32.72}$ keys/s	$2^{35.72}$ keys/s	This paper
SPECK-96-26	96	64	26	$2^{34.49}$ keys/s	$2^{36.72}$ keys/s	This paper

Finally, we observed that all of the optimizations we performed in this work provided similar speed-ups for every GPU listed in the Table 1. Thus, our optimizations are not

designed for a specific GPU or architecture.

It is in general very hard to predict the performance of an algorithm on a specific GPU by looking at its performance on a different GPU because GPUs have different specifications and architectures. Moreover, increasing the number of cores or clock speeds may not linearly increase the performance due to bottlenecks in an implementation. Thus, making source codes publicly available is important for future research and comparison. Despite its shortcomings, we normalized the GPU performances with respect to the number of cores and the boost clock speeds in Table 3. This normalization might give a rough estimate for the performances of the GPUs we used with respect to RTX 3090 that is used in [ACC+24] and RTX 5090 that is recently announced.

Table 3: Performance normalization of GPUs with respect to their number of cores and boost clock speeds. Note that these numbers might be higher for some models depending on the manufacturer.

GPU	Cores	Clock Rate	Cores × Clock	Normalization
MX 250	384	1582 MHz	607488	$1.0000 = 2^{0.00}$
GTX 860M	640	1020 MHz	652800	$1.0746 \approx 2^{0.10}$
GTX 970	1664	1253 MHz	2084992	$3.4322 \approx 2^{1.78}$
RTX 2070 Super	2560	1770 MHz	4531200	$7.4590 \approx 2^{2.90}$
RTX 3090	10496	1695 MHz	17790720	$29.2857 \approx 2^{4.87}$
RTX 4090	16384	2550 MHz	41779200	$68.7737 \approx 2^{6.10}$
RTX 5090	21760	2407 MHz	52376320	$86.2179 \approx 2^{6.43}$

According to Table 3 we would expect RTX 4090 to be $9.22 \approx 2^{3.20}$ times faster than RTX 2070 Super. Comparing with Table 2 we see that this estimate fits well in some cases, while it is an overestimate of the difference in others. Since our normalization only focuses on processing speed and overlooks delays that may be introduced by memory read-and-write operations or architectural differences, it may be seen as a theoretical peak performance difference between GPUs.

2.1 KASUMI

KASUMI was designed by ETSI SAGE [3GP] and it is a modified version of the block cipher MISTY1 [Mat97]. It is a Feistel block cipher with 8 rounds and a block size of 64 bits. The key schedule of KASUMI simply consists of rotations on 16-bit values and XOR with constants. The round function of KASUMI contains FO and FL functions where FL function contains AND, OR, and rotation operations and FO function is also a 3-round Feistel structure which contains FI functions in each of its round. Moreover, FI functions is a four round Feistel structure which uses two S-boxes of sizes 7×7 and 9×9 consecutively in each round. KASUMI block cipher is pictured in Figure 1 where || symbol represents OR operation.

Although KASUMI supports 128-bit keys, A5/3 and GEA-3 both use a session key K_c of 64 bits as 128-bit $K_c||K_c$ KASUMI key for GSM and GPRS in order to have backward compatibility. We refer to this version as KASUMI-64 in this paper.

2.2 SPECK

SPECK [BSS+13] is a family of add-rotate-xor (ARX) lightweight block ciphers designed in 2013 by National Security Agency (NSA) of United States. SPECK supports seven key sizes: 64, 72, 96, 128, 144, 192, and 256 bits. Block size and number of rounds depend on

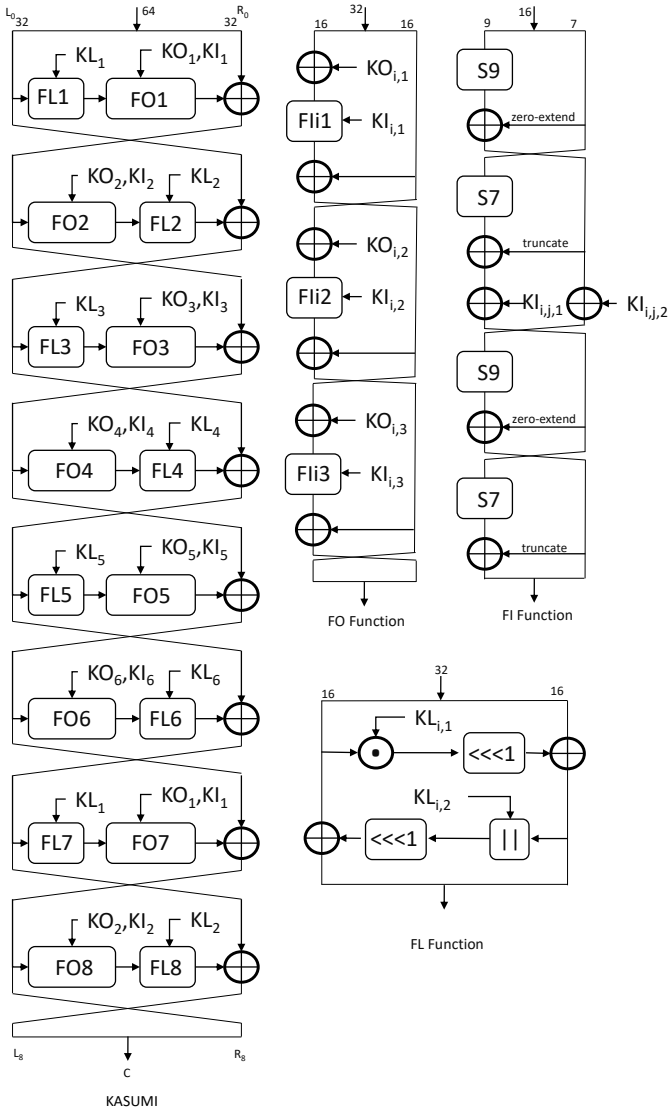

Figure 1: KASUMI block cipher.

the key size and possible variations are provided in Table 4. One round of SPECK is shown in Figure 2 and the key schedule of SPECK also uses this round function.

In Figure 2, for the block word size of 16 bits $\alpha = 7$ and $\beta = 2$. For every other block word size $\alpha = 8$ and $\beta = 3$.

We represent k-bit keyed SPECK with r rounds as SPECK-k-r. In this work we optimized SPECK-64-22, SPECK-72-22, SPECK-96-26, and SPECK-128-32 using the CUDA programming language and our optimizations can easily be modified for other variants of SPECK. SPECK became an ISO/IEC RFID air interface standard in 2018 (ISO/IEC 29167-22:2018). This standard was reviewed in 2024 and confirmed. Note that the ISO standard [ISO18] contains SPECK-96-26, SPECK-96-29, SPECK-128-32, and SPECK-256-34 versions and without a proper warning of the security implications of using a short key, users may prefer short keys for performance and low hardware footprint.

Table 4: Variations of SPECK block size in bits, key size in bits, and number of rounds.

Block size	Key size	Rounds
2x16 = 32	4×16 = 64	22
2x24 = 48	3×24 = 72	22
2x24 = 48	4×24 = 96	23
2x32 = 64	3×32 = 96	26
2x32 = 64	4×32 = 128	27
2x48 = 96	2×48 = 96	28
2x48 = 96	3×48 = 144	29
2x64 = 128	2×64 = 128	32
2x64 = 128	3×64 = 192	33
2x64 = 128	4×64 = 256	34

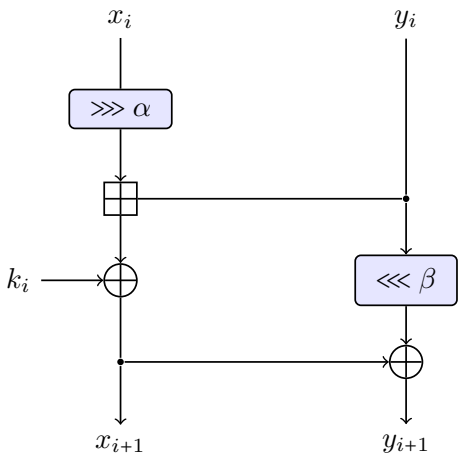

Figure 2: One round of SPECK family of ciphers.

2.3 TEA3

Terrestrial Trunked Radio (TETRA) is a European standard for trunked radio. It is globally used by military, police, emergency services, prisons, and government agencies. The cryptographic algorithms used in TETRA were kept secret for decades until it was shown how easy it is to reverse engineer in [MBW23]. They fully document their reverse engineering process on a Motorola MTM5400. Note that with millions of TETRA devices being deployed around the world, it was not hard for adversarial parties to obtain these cryptographic algorithms by reverse engineering or via stolen or leaked documents. But until the disclosure of these algorithms by [MBW23], academic community was not able to assess the security of these algorithms.

It was shown in [MBW23] that TETRA uses four very similar LFSR-based keystream generators called TEA1, TEA2, TEA3, and TEA4. They all consist of a key register and a state register which is initiated by an IV. Although all of these keystream generators use 80-bit secret keys, deliberately weakened TEA1 compresses the 80-bit key into 32 bits. Thus, it provides 32-bit security. The detailed structure of TEA4 is still unknown but since it was designed for commercial use and restricted export, it may have similar weaknesses of TEA1.

TEA2 and TEA3 are almost identical in design. They have a 64-bit state register and 80-bit key register. Both registers perform byte-wise shifting, they have two F functions for non-linear filtering and a single R function for bit reordering. Both of them use a different 8×8 S-box. One main difference is that in TEA3 the S-box output is XORed with a key register byte before feeding back. This difference must be further analyzed because TEA3's S-box is not a permutation as it was observed in [MBW23]. Both inputs $0x14$ and $0x9E$ provide the same output $0xC2$ and the output of the S-box can never be $0xD2$. As of now, it is not clear if this was a single bit error or a deliberate choice.

In our GPU optimizations we focused on implementing TEA3 and our codes can be slightly modified to obtain the same results for TEA2. Structure of TEA3 is provided in Figure 3. In order to produce the first keystream byte, TEA3 is clocked 51 times and it is clocked 19 times for the subsequent keystream bytes.

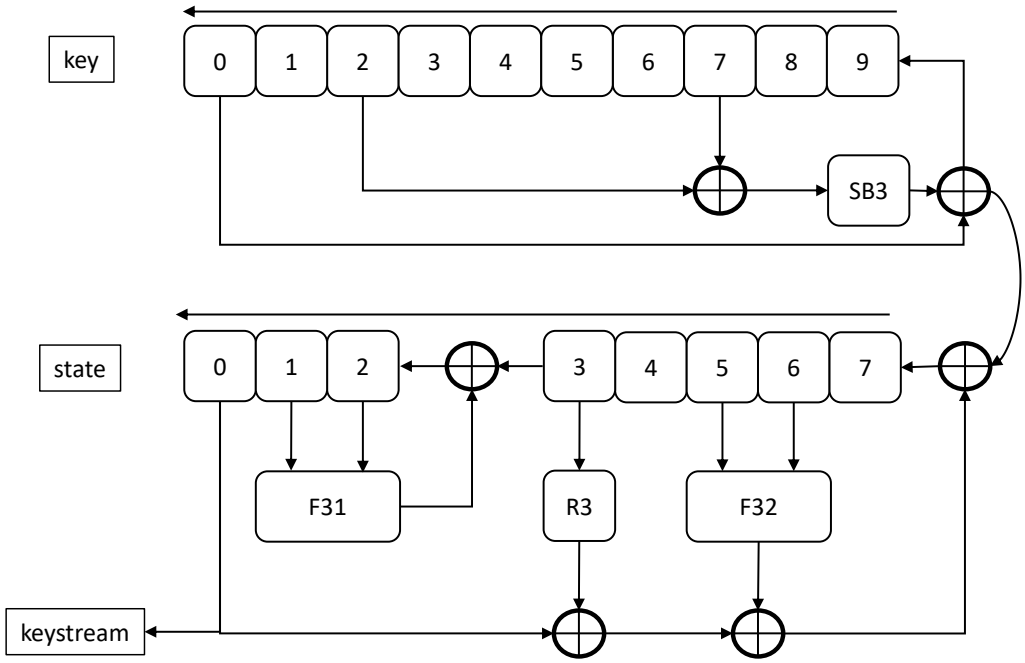

Figure 3: TEA3 keystream generator.

3 CUDA Optimizations of KASUMI, SPECK, and TEA3

There are generally three ways to implement symmetric key encryption algorithms on GPUs and depending on the design of the cipher, some methods provide better performance. These are naive/straightforward, table-based, and bitsliced implementations:

1. **Straightforward:** In straightforward implementations every operation of the encryption algorithm is implemented as they are. But it is generally not possible to parallelize an encryption algorithm to use all of the GPU cores since modern GPUs have thousands of cores. Thus, using each thread to perform a different encryption operation is preferred in this technique. In a brute force attack, each thread can try a different key in a subspace of the keyspace that is assigned to it. Similarly, for a block cipher mode of operation that offers parallelism like counter mode, each thread can encrypt a different block of the plaintext. Such an approach does not require communication between threads and thus avoids many memory operations. Since straightforward implementation requires every operation of the cipher to be performed, the main bottleneck comes from the number of instructions that are performed.

2. **Table-based:** Intermediate layers of some ciphers can be partitioned into disjoint subspaces so that every output for every possible input of these subspaces can be precomputed and stored in a table. This way performing those layers can be reduced to table look-up operations for the inputs of these subspaces and combining the values obtained from the tables. Such tables are generally too large to be stored in GPU registers and storing them in the global memory of the GPU causes a lot of delays since reading and writing to the global memory is slower. Thus, storing the tables in the shared memory provides the best performance. Shared memory bank conflicts are generally the main bottleneck in this approach. Moreover, storing very

large tables in the shared memory might reduce the occupancy of the GPU or might even not be possible due to the device limits.

3. **Bitsliced:** Programming languages and processors generally work on words that are multiples of 8 bits and bit operations can become more costly compared to hardware. Thus, hardware-oriented ciphers might use bit-level operations intensively so that the straightforward implementation might become taxing in software. Instead of working on bits of large words, we can store each bit in a different register so that we can avoid operations that try to access and use bits of a word. This bitslicing technique can be implemented in two different approaches on a GPU:

   (a) **Thread-level bitslicing:** In this approach, each bit of a word is assigned to a single thread in a GPU block. The optimizations in this approach focus on the choice of the memory types for transferring values between threads and the timing of memory operations. Since threads need to communicate to each other in this approach, the main bottleneck comes from the delays introduced by memory read and write operations.

   (b) **Variable-level bitslicing:** In this approach, every thread stores each bit of a word in a register. The data sizes cause the main bottleneck in this approach because we need to use a register for every bit of stored value. For example, we need to use n registers for an n-bit round key, n-bit block of a block cipher, or n-bit LFSR of a stream cipher. Thus, we can perform m encryptions in parallel at the thread-level when we use m-bit registers to store bits of m different inputs. However, we lose the full occupancy of a modern GPU when we use more than 64 32-bit registers.

3.1 GPU Optimization of `KASUMI`

We observed that the design of `KASUMI` is not suitable for an efficient table-based or bitsliced implementation on GPUs. Thus, we obtained the best performance when we used the straightforward implementation technique. We did the following optimizations and observations:

1. We stored the two S-boxes S7 and S9 of `KASUMI` in the shared memory. Keeping the S-boxes in the shared memory causes bank memory conflicts when different threads in a warp try to access different memory banks. When a shared memory bank conflict occurs, one thread has to wait the other. The amount of delay caused by this depends on the model of the GPU and the size of the data that is requested.

   The shared memory bank conflicts for S7 can be avoided by using the technique of [Tez21] where the S-box is duplicated 32 times in a special order so that every thread in a warp uses its own data lane. Although the S-boxes are of sizes 7 and 9 bits, in software implementations we use one and two bytes, respectively. Thus, duplicating the S-box S7 32 times requires 4 KBs. Although modern GPUs have 64KBs of shared memory, using 4KBs of shared memory reduced our occupancy of the GPU. This reduced occupancy causes a slowdown that is larger than the speed gain coming from removing the shared memory bank conflicts. Moreover, unlike the 32-bit values stored in the shared memory in [Tez21], we store 8-bit values in the shared memory for S7. This provides faster shared memory read speeds and the delays in the shared memory bank conflicts are shorter compared to the case of [Tez21].

   Since we need to spend two bytes for S9, duplicating the S-box S9 32 times requires 32 KBs of shared memory. Although modern GPUs have this much shared memory, such an implementation reduces the occupancy of the GPU and speed up coming from the

removed shared memory bank conflicts cannot compete with this slowdown. Thus, we kept a single copy of S7 and S9 in the shared memory in our best optimizations.

When a GPU kernel is called with n blocks of t threads, we initialize each thread to an integer `threadIndex.x` in $[0, n \times t - 1]$. In the following code, the first 128 threads of each block copy the S-box S7 and the first 512 threads of each block copy the S-box S9 from the global memory to the shared memory:

```
uint32_t threadIndex = blockIdx.x * blockDim.x + threadIdx.x;
if (threadIdx.x < 512) {
    if (threadIdx.x < 128) S7S[threadIdx.x] = S7G[threadIdx.x];
    S9S[threadIdx.x] = S9G[threadIdx.x];
}
__syncthreads();
```

The *__syncthreads*(); command in the last line above synchronizes the threads in the block to prevent threads to try to read before the values are written to the shared memory.

2. The round constants of `KASUMI` can be kept in the shared memory, or in registers, or they can be provided as they are inside the code. We observed that the best performance is obtained when independent registers are used for the round constants. In this case, we do not pass the constants to the GPU kernel. Instead, they are assigned to registers at the beginning of the kernel as follows:

```
uint16_t   c1 = 0x0123, c2 = 0x4567, c3 = 0x89AB, c4 = 0xCDEF, c5 = 0xFEDC,
           c6 = 0xBA98, c7 = 0x7654, c8 = 0x3210;
```

3. Our implementation of `KASUMI` for TMTO table creation and performing exhaustive key search are different because in an exhaustive search we rarely perform the encryption of the last round of a Feistel cipher. Because for `KASUMI` the left 32-bit of the seventh-round output is the right 32-bit of the ciphertext. Thus, the last round of the encryption is only performed when the target right 32-bit of the ciphertext is observed after the seventh round which happens with a probability of 2^{-32}. Thus, the last round is performed only 2^{32} times out of 2^{64} key searches.

4. In modern GPUs, best occupancy is obtained when the blocks consist of 1024 threads when there is no other significant bottleneck. However, this can be achieved when the kernel uses less than or equal to 64 registers per thread because a block can have at most 64K registers in modern GPUs.

   When the attacked plaintext left and right halves are represented with *plaintextl* and *plaintextr*, corresponding ciphertext is represented with *ciphertextl* and *ciphertextr*, and the S-boxes are represented as the arrays *S7d*[ ] and *S9d*[ ], the GPU kernel for our exhaustive key search attack is called with 1024 threads of 2048 blocks as follows:

```
KASUMI64ExhaustiveConstantsRegister <<< 2048, 1024 >>>
    (plaintextl, plaintextr, ciphertextl, ciphertextr, S7d, S9d);
```

   We call the *KASUMI64ExhaustiveConstantsRegisterTMTO* GPU kernel for TMTO table creation in a similar way. The only difference is that we pass the device arrays *ciphertextl_d*[ ] and *ciphertextr_d*[ ] to store TMTO tables in the GPU global memory:

```
KASUMI64ExhaustiveConstantsRegisterTMTO <<< 2048, 1024 >>>
    (plaintextl, plaintextr, ciphertextl, ciphertextr, S7d, S9d,
    ciphertextl_d,ciphertextr_d);
```

   When compiled with many different versions of CUDA SDK and choice of compute capabilities between 5.0 and 8.9, our TMTO codes always require less than 64 registers and our exhaustive search codes that conditionally performs the last round

require always more than 64 registers. Although the codes are very similar, CUDA compiler observes other optimizations for our exhaustive key search code when we conditionally perform the last round encryption. But these optimizations increase the register count. This prevents us to call the kernel blocks with 1024 threads and when we use 512 threads in blocks, conditionally performing the last round provides negligible speed up compared to the version where we perform all of the eight rounds with 1024 threads.

Since the increase in the number of required registers were due to the compiler's optimizations, we forced the CUDA SDK to use at most 64 registers when compiling our codes by using the command $-maxrregcount = 64$. This way, we achieved 10% speed up compared to the 8-round encryption of the TMTO table creation codes. When compiling our `KASUMI` implementations, not limiting the register count to 64 at the compile time would result in "too many resources requested for launch" error at the run time.

Our `KASUMI` benchmark results are provided in Table 5 and it can be seen that they are valid for many different GPUs and they are not optimized for a specific GPU architecture or model.

Table 5: Number of `KASUMI` encryptions per second when performing TMTO table creation and exhaustive key search on various GPUs.

GPU	TMTO Table Creation	Key Search
MX 250	$2^{29.54}$ encryptions/s	$2^{29.70}$ keys/s
GTX 860M	$2^{29.79}$ encryptions/s	$2^{29.88}$ keys/s
GTX 970	$2^{31.41}$ encryptions/s	$2^{31.54}$ keys/s
RTX 2070 Super	$2^{32.58}$ encryptions/s	$2^{32.72}$ keys/s
RTX 4090	$2^{35.56}$ encryptions/s	$2^{35.72}$ keys/s

We are not aware of any publicly available GPU optimizations of `KASUMI` so we were not able to benchmark other optimizations on our GPUs and compare them with our results. However, it is reported in [ACC+24] that it is possible to perform $2^{43.32}$ `KASUMI` encryption in 61 minutes on a single RTX 3090 GPU. This means that they achieve around $2^{31.48}$ `KASUMI` encryptions per second. Note that we achieved a similar performance with a GTX 970 GPU as it is shown in Table 5 and RTX 3090 is roughly 8.5 times faster than GTX 970 on average due to their differences in core numbers and clock speeds as shown in Table 3.

In [JRW11], three meet-in-the-middle key recovery attacks for full KASUMI-64 were provided. Their initial attack requires a single known plaintext/ciphertext pair and 2^{63} encryptions. Then the time complexity is reduced to $2^{62.75}$ encryptions when the attacker captures 1152 chosen plaintext/ciphertext pairs. And finally, the time complexity becomes $2^{62.63}$ encryptions when the data complexity is 2^{20} chosen plaintexts. With our GPU optimizations, the attack that requires $2^{62.63}$ encryptions can be performed in 4.47 years on a single RTX 4090.

We made our CUDA optimizations of `KASUMI` for creating TMTO tables and performing exhaustive key search publicly available[1] so that future optimizations can be easily compared with our optimizations.

[1] Our optimized `KASUMI` CUDA codes are publicly available at GitHub so that they can be used to verify our performance results, to analyze `KASUMI`, or to compare future optimizations: https://www.github.com/cihangirtezcan/CUDA_KASUMI

3.2 GPU Optimization of SPECK

As in the case of KASUMI, we observed that the design of SPECK is not suitable for an efficient table-based or bitsliced implementation on GPUs. Thus, we performed a straightforward implementation.

We optimized SPECK-64-22, SPECK-72-22, SPECK-96-26, and SPECK-128-32. Our SPECK benchmark results on various key sizes are provided in Table 6 and it can be seen that they are not optimized for a specific GPU architecture or model.

Table 6: Number of SPECK encryptions per second when performing exhaustive key search on various key sizes on various GPUs.

GPU	SPECK-64-22	SPECK-72-22	SPECK-96-26	SPECK-128-32
MX 250	$2^{30.56}$ keys/s	$2^{30.72}$ keys/s	$2^{31.43}$ keys/s	$2^{29.93}$ keys/s
GTX 860M	$2^{30.53}$ keys/s	$2^{30.72}$ keys/s	$2^{31.30}$ keys/s	$2^{29.86}$ keys/s
GTX 970	$2^{32.12}$ keys/s	$2^{32.43}$ keys/s	$2^{32.99}$ keys/s	$2^{31.38}$ keys/s
RTX 2070 Super	$2^{33.99}$ keys/s	$2^{34.27}$ keys/s	$2^{34.49}$ keys/s	$2^{32.97}$ keys/s
RTX 4090	$2^{36.20}$ keys/s	$2^{36.47}$ keys/s	$2^{36.72}$ keys/s	$2^{35.30}$ keys/s

In our implementations for various SPECK variants, we did the following optimizations and observations:

1. We observed that compiling our SPECK optimizations with compute capability 5.2 provides around $2^{0.06}$ better performance compared to a code compiled for compute capability 8.9. For example, trying 2^{43} keys for SPECK-96-26 using an RTX 4090 takes 80.63 seconds and 77.80 seconds when compiled for compute capabilities 8.9 and 5.2, respectively.

2. Current CUDA SDKs do not support compute capability less than 5.0 but using older CUDA SDKs allow us to use deprecated compute capabilities. Thus, we used CUDA SDK 11.1 and compiled our implementations for compute capability 3.5. However, using compute capability 3.5 did not provide any observable speed up compared to 5.2.

3. Although SPECK-64-22 and SPECK-72-22 have smaller number of rounds compared to SPECK-96-26, we obtained better speeds for SPECK-96-26. We observed that this is because the key and the data are stored in 32-bit registers in SPECK-96-26 due to its block size and on CUDA devices the rotation operation on 32-bit values is faster than 16 or 24-bit values. Thus, 64 and 72-bit versions might be broken faster in the future if new GPU instructions provide speed-ups for rotations on values smaller than 32 bits. We defined the rotation operations as macros in a straightforward and common way as follows:

```
#define ROTL16(x,r) (x<<r) | (x>>(16-r))
#define ROTR16(x,r) (x>>r) | (x<<(16-r))
#define ROTL24(x,r) (x<<r) | (x>>(24-r))
#define ROTR24(x,r) (x>>r) | (x<<(24-r))
#define ROTL32(x,r) (x<<r) | (x>>(32-r))
#define ROTR32(x,r) (x>>r) | (x<<(32-r))
```

   Note that using 32-bit unsigned integers for storing 16 or 24-bit values requires additional AND operations after the rotation with 0xFFFF or 0xFFFFFF, respectively.

4. In software implementations, it is a common practice to use a *for* loop to perform r rounds of encryption inside the loop. Using the #pragma unroll macro unrolls these loops and in the case of SPECK, this provided marginal speed-up in our implementations.

5. Our optimizations use small numbers of registers so that we can call the GPU kernels with 1024 threads as follows to try $2^{20+trial}$ keys:

```
speck64_exhaustive <<< 1024, 1024 >>> (ct_d, pt_d, K_d, trial);
speck72_exhaustive <<< 1024, 1024 >>> (ct_d, pt_d, K_d, trial);
speck96_exhaustive <<< 1024, 1024 >>> (ct_d, pt_d, K_d, trial);
speck128_exhaustive <<< 1024, 1024 >>> (ct_d, pt_d, K_d, trial);
```

Since a year has around $2^{24.91}$ seconds, one needs around 8 RTX 4090 GPUs to break SPECK-64-22 in a year. In order to break SPECK-72-22 in a year, one needs around 1575 RTX 4090 GPUs. And to break SPECK-96-26 in a year, one needs around 22 billion RTX 4090 GPUs. Note that SPECK-96-26 is included in the ISO/IEC 29167-22 [ISO18] RFID air interface standard. Although 22 billion GPUs are a lot, this number is going to reduce when new generation of GPUs like NVIDIA's 5000 series are announced and produced in 2025. According to our estimates in Table 3, we expect one would need around 17.5 billion RTX 5090 GPUs to break SPECK-96-26 in a year. Those numbers are by far exceeding today's practical capabilities. However, they show that devices built today with SPECK-96-26 may not be secure around 2050. Moreover, GPUs are general purpose computing devices and our results also show that if built, dedicated devices can break SPECK-96-26 faster than GPUs and would consume significantly less energy compared to GPUs.

We made our CUDA optimizations of SPECK for performing exhaustive key search publicly available[2] so that future optimizations can be easily compared with our optimizations.

3.3 GPU Optimization of TEA3

TEA3 is a keystream generator and it is not suitable for a table-based implementation due to its design. TEA3 can be seen as two parts in which one part applies the S-box on the key register and the other part applies F31, F32, and R3 functions, which consist of bit-level operations, on the state register. Initially we optimized TEA3 for GPUs using the straightforward implementation. However, this approach did not provide good results due to the bit level operations at the state registers. Our best optimization reached $2^{27.39}$ key trials per second on an RTX 4090 GPU.

TEA3 is also not suitable for a bitsliced implementation due to the 8-bit S-box operation on the key register. Thus, we combined the straightforward and bitsliced implementation techniques where we implemented the key register update part as a straightforward implementation and the state register update part as a bitsliced implementation.

For the bitsliced part, we tried using 8-bit, 16-bit, and 32-bit registers. In other words, we obtained single instruction multiple data (SIMD)-level parallelism at thread-level by trying 8, 16, and 32 keys in a single encryption, respectively. Note that although increasing the word size of registers increases the number of parallel encryptions, it also increases the number of registers used in each thread. Although our straightforward implementation requires less than 65 registers, using 8, 16, and 32-bit values in the bitsliced part increases the required number of registers to 127, 166, and 171, respectively. Note that these numbers change depending on the compiled compute capability and used SDK so we compiled our code using many SDKs and compute capabilities but none of them reduced these numbers to 64 or to 128 in the case of 16 and 32-bit word sizes. Thus, our kernel blocks have to decrease to 512 threads when the word size is 8 bits and to 256 threads when the word size is 16 or 32 bits.

In each thread, we use 32 different key registers and store the S-box output results at `uint32_t bSboxOut[32]`. Since the S-box has a size of 8 bits, the result can be seen as a 32×8 matrix. We need the transpose of this bit matrix to move from our straightforward implementation to bitslice implementation. The following code is used for this purpose:

[2]Our optimized SPECK CUDA codes are publicly available at GitHub so that they can be used to verify our performance results, to analyze SPECK, or to compare future optimizations: https://www.github.com/cihangirtezcan/CUDA_SPECK

```
unsigned m = 0x0000FFFF;
#pragma unroll
for (int l = 16; l != 0; l = l >> 1, m = m ^ (m << l)) {
#pragma unroll
    for (k = 0; k < 32; k = (k + l + 1) & ~l) {
        t = (bSboxOut[k] ^ (bSboxOut[k + l] >> l)) & m;
        bSboxOut[k] = bSboxOut[k] ^ t;
        bSboxOut[k + l] = bSboxOut[k + l] ^ (t << l);
    }
}
```

Note that any improvement to the above bit-level matrix transpose operation would increase the performance of our TEA3 implementation.

Note that limiting the maximum used register count to 64 (or to 128) as we did in our KASUMI implementation causes a performance loss in this case because the kernel really needs to keep more than 64 (or 128) registers. For instance, bitslicing the 80-bit state register already requires 80 registers per thread, exceeding the 64 limit. If we force CUDA SDK to use only 64 registers by using the command $-maxrregcount = 64$ as we did when implementing KASUMI, the required extra registers spill and are kept in the global memory. Reading and writing these values to and from the global memory or cache provides delays that significantly slow down our implementation.

Although decreasing the thread count in a block to 512 or 256 decreases the GPU occupancy and therefore the performance, the number of parallel executions in the bitsliced implementation still provide better results. We obtained the best results when we used 32-bit registers and achieved $2^{34.71}$ key trials per second on an RTX 4090. This is around 160 times faster than our straightforward implementation.

Our TEA3 exhaustive key search results are provided in Table 7 and it can be seen that they are not optimized for a specific GPU architecture or model.

Table 7: Number of TEA3 encryptions per second when performing exhaustive key search on various GPUs.

GPU	Key Search
MX 250	$2^{28.95}$ keys/s
GTX 860M	$2^{29.23}$ keys/s
GTX 970	$2^{30.53}$ keys/s
RTX 2070 Super	$2^{32.54}$ keys/s
RTX 4090	$2^{34.71}$ keys/s

Our best optimizations show that 80-bit key search for TEA3 would require 1.36 million RTX 4090 GPUs to break it in a year. And according to our estimates provided in Table 3, we expect 1.08 million RTX 5090 GPUs can break it in a year. Again, while clearly not practical today, it is likely to be so in the future and should therefore not be used in real life scenarios. The required time and the number of GPUs will be significantly reduced with every new generation of GPUs. Moreover, our results showing that GPUs can theoretically break TEA3 and the fact that this cipher is used by military, police, and government agencies, one might invest building ASICs to break TEA3 in a short time.

Our optimizations can also be used for performing TMTO attacks on TEA3 but note that table creation speed for TMTO can be at most 6 times slower than the speeds reported in Table 7. This is because we perform an early abort technique when performing the exhaustive key search attack. That is, if the first byte of the keystream does not provide the desired output, then there is no need to produce more keystream bytes. Since we try 32 different keys in our bitsliced implementation, the desired output is observed with probability 32/256 even though all of the 32 keys are wrong. Thus, we generate the second keystream byte with probability 1/8 but rarely produce the remaining keystream bytes. But in TMTO table creation, we have to produce the whole keystream in every encryption

and producing 10 bytes of keystream is enough for a TMTO attack. However, producing the next keystream bytes are easier since TEA3 is clocked 51 times when producing the first keystream byte and clocked 19 times for the rest of the keystream byte generation.

We made our CUDA optimizations of TEA3 for performing exhaustive key search publicly available[3] so that future optimizations can be easily compared with our optimizations.

4 Improved Time-Memory Trade-off Cryptanalysis of KASUMI-64

Although no source codes are provided, it is reported in [ACC+24] that it is possible to perform $2^{43.32}$ KASUMI encryption in 61 minutes on a single RTX 3090 GPU. This means that they achieve around $2^{31.48}$ KASUMI encryptions per second. Although an RTX 3090 is around 4 times faster than an RTX 2070 Super as it can be seen in Table 3, in this work we optimized KASUMI for GPUs so that we achieved $2^{32.72}$ KASUMI encryptions per second on an RTX 2070 Super. More importantly, we achieved $2^{35.56}$ KASUMI encryptions per second on an RTX 4090 GPU which is around 16.89 times faster compared to the results of [ACC+24] in which they used RTX 3090. Note that according to our normalizations provided in Table 3, we expect RTX 4090 to be at most 2.35 times faster than RTX 3090.

Our improved KASUMI implementation directly applies to the TMTO results of [ACC+24]. For instance, an attack in [ACC+24] that requires 289 days for precomputation when 600 GPUs are used can be performed in 17 days with the same amount of GPUs when our optimized codes are used. Similarly, the same precomputation can be done in 289 days with only 36 GPUs, instead of 600. Thus, our codes can be used to make a trade-off to perform precomputation using between 36 to 600 GPUs for 17 to 289 days. Similarly, the attack of [ACC+24] that requires 348 days for 2400 GPUs to do precomputations can be reduced to 20.6 days with our optimizations.

The precomputation phase of the TMTO attacks of [ACC+24] for GPRS and GSM are considered in three scenarios in which the attacker has access to 600, 1200, and 2400 RTX 3090 GPUs. Effect of our optimizations to these three scenarios are summarized in Table 8.

Table 8: Precomputation and attack time for performing TMTO attack on KASUMI-64.

Scenario	# GPUs	Time	Memory	Success	Time	Reference
Scen. 1	600 RTX 3090	289 days	100 TB	0.25	5 min	[ACC+24]
Scen. 1	36 RTX 4090	289 days	100 TB	0.25	5 min	Section 4
Scen. 1	600 RTX 4090	17 days	100 TB	0.25	5 min	Section 4
Scen. 2	1200 RTX 3090	348 days	125 TB	0.5	13 min	[ACC+24]
Scen. 2	71 RTX 4090	348 days	125 TB	0.5	13 min	Section 4
Scen. 2	1200 RTX 4090	20.6 days	125 TB	0.5	13 min	Section 4
Scen. 3	2400 RTX 3090	348 days	200 TB	0.75	14 min	[ACC+24]
Scen. 3	142 RTX 4090	348 days	200 TB	0.75	14 min	Section 4
Scen. 3	2400 RTX 4090	20.6 days	200 TB	0.75	14 min	Section 4

[3] Our optimized TEA3 CUDA codes are publicly available at GitHub so that they can be used to verify our performance results, to analyze TEA3, or to compare future optimizations: https://www.github.com/cihangirtezcan/CUDA_TEA3

5 Brute Force Cryptanalysis of KASUMI-64

The main idea behind the time-memory trade-off attacks is to perform precomputation of encryption operations slightly more than exhaustive key search once and record some of the results in a proper way so that in the future capturing a key can cost less than the exhaustive key search attack. However, one might prefer to perform exhaustive key search on GSM and GPRS for every attacked key instead of a TMTO attack due to the following reasons:

1. The passive TMTO attacks of [ACC+24] on GPRS assumes that the network is misconfigured so they affect a subset of GPRS networks.
2. The TMTO attack of [ACC+24] on well-configured GPRS networks requires an active attack in which the attacker has to inject messages.
3. The generic TMTO attack of [ACC+24] against GSM communications that builds a TMTO on a specific IV requires a known plaintext to be encrypted with that IV. This happens with 50% in a 1 hour and 44 minute GSM communication. Thus, the success probability of the attack increases when the attacked communication lasts longer. However, in many countries the communication is terminated after 1 hour and this limits the success probability of the attack to 21%.

Our implementation of KASUMI for TMTO table creation and performing exhaustive key search are different because in an exhaustive search we rarely perform the encryption of the last round. This is because the left 32-bit of the seventh-round output is the right 32-bit of the ciphertext. Thus, the last round of the encryption is only performed when the target right 32-bit of the ciphertext is observed after the seventh round which happens with a probability of 2^{-32}. Thus, the last round is performed only 2^{32} times out of 2^{64} key searches.

Since our optimizations allow $2^{35.72}$ keys per second on an RTX 4090, it takes 10.35 years for a single RTX 4090 to break KASUMI-64. Or to break KASUMI-64 in a year, 11 RTX 4090 GPUs are enough. If we use 2400 GPUs as suggested in [ACC+24] but this time for exhaustive key search attack instead of generating TMTO tables, it would take less than 38 hours to find the key in the worst case. Thus, instead of creating TMTO tables in 348 days as in [ACC+24] or 20.6 days by our optimized code, same amount of GPUs can be used for 1.5 days to capture the key via brute force attack. Moreover, switching to the exhaustive key search from TMTO attack increases the success probability from 21% to 100% and the requirement for the communication to be 1 hour is no longer needed. It can be as short as a few seconds.

Note that the GPU core numbers and therefore their performance increases in every generation of GPUs while their price and sometimes energy consumption remains the same. Thus, our exhaustive key search attacks are going to be more practical in the future. However, such technological improvements are not going to be beneficial for the one-time TMTO creation that is performed now.

Moreover, the three scenarios of [ACC+24] performs the precomputation on GPUs and record the results on SSDs and the attack is performed on 128-core servers. Thus, these scenarios require two, five, and ten 128-core servers and 100, 125, and 200 TB of SSDs, respectively. According to [ACC+24], these servers and SSDs would cost around 85 000, 206 250, and 410 000 USD, respectively. In our exhaustive search we do not need any CPU power or SSDs to record any tables. Thus, one can avoid these extra costs if they move from TMTO to exhaustive key search.

To summarize, using our optimized code for brute force cryptanalysis of a GPS communication instead of a TMTO attack of [ACC+24] has the following advantages when the scenario three with a one-hour call is considered:

1. Attack success probability increases from 21% to 100%.

2. Instead of requiring to capture a one-hour communication, communications of any lengths can be cryptanalysed.

3. The precomputation for 348 days is no longer required.

4. 200 TB SSD storage is no longer required.

5. Ten 128-core servers are no longer required.

The only disadvantage of our exhaustive key search attack against the TMTO attack of [ACC+24] is that it now requires 38 hours to break a communication, instead of 14 minutes. Note that 38 hours is the worst-case scenario and on average our brute force attack should take around 19 hours. Moreover, this duration will be shortened with every new generation of GPUs.

Instead of the exhaustive search, one can also perform the meet-in-the-middle attacks of [JRW11] on GPUs using our optimized codes. When we have 1 known plaintext, 1152 chosen plaintexts, or 2^{20} chosen plaintexts, the attacks of [JRW11] require $2^{63.03}$, $2^{62.75}$, and $2^{62.63}$ encryptions, respectively. Thus, if we use our CUDA codes to perform these attacks using 2400 RTX 4090 GPUs, the 38 hours required for the exhaustive search reduces to 19, 16, and 14.7 hours, respectively.

6 Conclusion

Symmetric key encryption algorithms that use short keys appear in standards and many real-world applications making them susceptible to exhaustive key search attacks. In this work we provided the best-known GPU optimizations of the `KASUMI`, `SPECK`, and `TEA3` ciphers to show that they can be broken by brute force attacks.

GPRS and GSM uses the 64-bit key version of `KASUMI` and we showed that it can be broken in a year just by using 11 RTX 4090 GPUs. And the attack reduces to 38 hours when 2400 GPUs are used. Our `KASUMI` implementation is more than 15 times faster than the optimizations given in the CRYPTO'24 paper [ACC+24] improving the main results of that paper by the same factor. Our optimizations can also be used to perform the meet-in-the-middle attacks of [JRW11] which have better time complexities than exhaustive search. With 2400 GPUs, the best attack in [JRW11] can be performed in 14.7 hours with our codes.

`SPECK` block cipher supports short keys like 64, 72, and 96 bits. Although 64 and 72-bit versions do not appear in the ISO/IEC standard for RFID air interface, 96-bit version does. Our best optimizations show that this version could be broken in a year when 22 billion RTX 4090 GPUs were used. Thus, currently 96-bit `SPECK` cannot be practically broken via brute force attacks.

European standard TETRA for trunked radio uses proprietary keystream generators which were kept secret for decades. Recently they were reverse engineered and it was shown that the best of these keystream generators uses 80-bit secret key. Our best optimizations show that it can be broken in a year by using 1.36 million RTX 4090 GPUs. This is an important threat in real-world because TETRA is used by military, police, emergency services, prisons, and government agencies.

All of these attacks will become more practical with every new generation of GPUs or if dedicated hardware is designed for breaking them. Thus, we strongly recommend avoiding keys shorter than 128 bits.

Acknowledgments

This work was supported by The Scientific and Technological Research Council of Türkiye (TÜBITAK) and German Academic Exchange Service (DAAD) Bilateral Research Cooperation Project (TÜBİTAK 2531 Project) under the grant number 123N546 and titled "Cryptanalysis of Symmetric Key Encryption Algorithms: Theory vs. Practice". The authors thank TÜBİTAK and DAAD for their support.

This work was also supported by the ERC project 101097056 (SYMTRUST) and the enCRYPTON project. The later has received funding from the European Union's Horizon Europe Research and innovation programme under grant agreement No: 101079319. Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union, European Commission or European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[3GP] 3rd generation partnership project, technical specification group services and system aspects, 3g security, specification of the 3gpp confidentiality and integrity algorithms; document 2: Kasumi specification, v.3.1.1 (2001).

[ACC+24] Gildas Avoine, Xavier Carpent, Tristan Claverie, Christophe Devine, and Diane Leblanc-Albarel. Time-memory trade-offs sound the death knell for GPRS and GSM. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part IV*, volume 14923 of *Lecture Notes in Computer Science*, pages 206–240. Springer, 2024.

[AFDM17] Ahmed A. Abdelrahman, Mohamed M. Fouad, Hisham Dahshan, and Ahmed M. Mousa. High performance cuda aes implementation: A quantitative performance analysis approach. In *2017 Computing Conference*, pages 1077–1085, 2017.

[AS20a] Sang Woo An and Seog Chung Seo. Study on optimizing block ciphers (aes, cham) on graphic processing units. In *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, pages 1–4, 2020.

[AS20b] SangWoo An and Seog Chung Seo. Highly efficient implementation of block ciphers on graphic processing units for massively large data. *Applied Sciences*, 10(11), 2020.

[BBK03] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 600–616. Springer, 2003.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Paper 2013/404, 2013.

[FR24] Hildegard Ferraiolo and Andrew Regenscheid. Cryptographic algorithms and key sizes for personal identity verification. *NIST SP 800-78-5*, 2024.

[ISO12] ISO/IEC 29192-3:2012. Information technology — security techniques — encryption algorithms — part 2: Stream ciphers. https://www.iso.org/standard/56426.html, 2012.

[ISO18] ISO/IEC 29167-22:2018. Information technology — automatic identification and data capture techniques. part 22: Crypto suite speck security services for air interface communications. https://www.iso.org/standard/70389.html, 2018.

[ISO19] ISO/IEC 29192-2:2019. Information technology — lightweight cryptography — part 2: Block ciphers. https://www.iso.org/standard/78477.html, 2019.

[JRW11] Keting Jia, Christian Rechberger, and Xiaoyun Wang. Green cryptanalysis: Meet-in-the-middle key-recovery for the full KASUMI cipher. Cryptology ePrint Archive, Paper 2011/466, 2011.

[Mat97] Mitsuru Matsui. New block encryption algorithm MISTY. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 1997.

[MBW23] Carlo Meijer, Wouter Bokslag, and Jos Wetzels. All cops are broadcasting: TETRA under scrutiny. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7463–7479, Anaheim, CA, August 2023. USENIX Association.

[NAI17] Naoki Nishikawa, Hideharu Amano, and Keisuke Iwai. Implementation of bitsliced AES encryption on cuda-enabled GPU. In Zheng Yan, Refik Molva, Wojciech Mazurczyk, and Raimo Kantola, editors, *Network and System Security - 11th International Conference, NSS 2017, Helsinki, Finland, August 21-23, 2017, Proceedings*, volume 10394 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2017.

[Tez21] Cihangir Tezcan. Optimization of advanced encryption standard on graphics processing units. *IEEE Access*, 9:67315–67326, 2021.

[Tez22] Cihangir Tezcan. Key lengths revisited: Gpu-based brute force cryptanalysis of DES, 3DES, and PRESENT. *J. Syst. Archit.*, 124:102402, 2022.

[Tez24] Cihangir Tezcan. Gpu-based brute force cryptanalysis of KLEIN. In Gabriele Lenzini, Paolo Mori, and Steven Furnell, editors, *Proceedings of the 10th International Conference on Information Systems Security and Privacy, ICISSP 2024, Rome, Italy, February 26-28, 2024*, pages 884–889. SCITEPRESS, 2024.