Optimized GPU Framework for Block Cipher Differential Search

Wei-Zhu Yeoh, Je Sen Teh*, Jiageng Chen,

***Abstract*—Differential cryptanalysis of block ciphers require the identification of differential characteristics (trails) that occur with high probabilities. The effort required to search for these characteristics increases exponentially as the number of rounds and block size increases. Differentials, which are clusters of differential characteristics sharing the same input and output differences, can be constructed to improve the overall distinguishing probability, thus improving the efficiency of a key recovery attack. Matsui's branch-and-bound algorithm that automates the seach for differential characteristics is commonly used to construct these differentials. However, the algorithm is still inefficient when enumerating a large number of characteristics, especially for block ciphers with large block sizes or number of rounds. In this paper, we improve upon the differential search by proposing a GPU-accelerated branch-and-bound framework that is more efficient and cost-effective as compared to its CPU counterpart. Efficiency is optimized by parallelizing all operations involved in a typical branch-and-bound algorithm by completing the entire search without transferring intermediate results back to the CPU. The meet-in-the-middle (MITM) approach is also adopted for further performance gains. We analyze the proposed framework in terms of both financial and computational costs based on simulations on the Google Cloud VM environment. When optimizing for performance, the proposed framework can achieve up to 90x speedup while saving up to 47% of the running cost as compared to a single CPU core. If cost-saving is the goal, the proposed framework can save up to 83% of the running cost while retaining a speedup of up to 40x as compared to single CPU core. The proposed framework is then applied to 128-bit TRIFLE-BC, 64-bit PRESENT and 64-bit GIFT, leading to the discovery of improved differentials. Notably, we identified the best differentials for PRESENT and 64-bit GIFT to date, with probabilities of $2^{-61.7964}$ and $2^{-60.66}$ for 16 and 13 rounds respectively. Although the differential probability for 43 rounds of TRIFLE-BC was not significantly improved, we were able to construct larger differentials with approximately 5.8x more characteristics than existing ones. Thus, the proposed GPU framework is currently the most efficient approach for enumerating 128-bit block cipher differentials, especially for a large number of rounds.** [1]

***Index Terms*—Automatic search, block cipher, branch-and-bound, cryptanalysis, differential, differential cryptanalysis, GPU**

I. Introduction

BLOCK ciphers are important cryptographic primitives that are used in various cryptographic applications such as Transport Layer Security (TLS) [1], OpenPGP [2], and SSH Transport Layer Protocol [3]. As such, there exists a multitude of cryptanalysis methods available for analyzing block cipher security, a notable one being differential cryptanalysis. Differential cryptanalysis [4] is a powerful tool that exploits the propagation of input (XOR) differences to construct statistical distinguishers that can be used to extract encryption keys. The resistance to differential cryptanalysis has become a de-facto requirement for any modern block cipher [5][6][7]. The success of differential cryptanalysis relies on the identification of differential characteristics (trails) that occur with sufficiently high probabilities.

A **differential characteristic** denoted by $\Delta X \xrightarrow{r} \Delta Y$ is constructed by propagating an input XOR difference, $\Delta X = M_1 \oplus M_2$ through r rounds of a block cipher to obtain the final output difference, $\Delta Y = C_1 \oplus C_2$, where (M_i, C_i) represent distinct plaintext-ciphertext pairs. If a characteristic has a high differential probability, $Pr(\Delta X \xrightarrow{r} \Delta Y)$, it can be used as a statistical distinguisher for key recovery attacks. These key recovery attacks can be made more efficient by taking into consideration the notion of differentials. **Differentials** are clusters of differential characteristics that share the same input and output differences but differ in their intermediary differences, δ_i. For example, $\Delta X \rightarrow \delta_1 \rightarrow \delta_2 \rightarrow \Delta Y$ and $\Delta X \rightarrow \delta_3 \rightarrow \delta_4 \rightarrow \Delta Y$ can be clustered to form a differential that occurs with a probability of $Pr(\Delta X \rightarrow \delta_1 \rightarrow \delta_2 \rightarrow \Delta Y) + Pr(\Delta X \rightarrow \delta_3 \rightarrow \delta_4 \rightarrow \Delta Y)$. Every differential characteristic added to a differential improves upon the differential's probability of occurrence. The search for these characteristics is a non-trivial process because the computational effort scales exponentially with respect to a cipher's block size and number of rounds.

GPUs is an integral part of the high-performance computing (HPC) cluster, as evidenced by their appearance in computing clusters that make it into the Top 500 Supercomputer rankings [8] based on computing power. As such, tapping into readily available GPU computational power for efficiency gains would greatly benefit cryptanalysts, especially for computationally intensive applications such as the identification of differentials. Although GPUs have already been utilized for other cryptography-related efforts such as hash collision attacks [9], accelerating asymmetric cryptography [10] and symmetric-key cryptography [11], it is yet to be utilized efficiently for the differential search problem. Even though a GPU approach has been attempted in [12], the differential search for a large number of rounds sees a dramatic drop in performance due to

*Corresponding Author

Wei-Zhu Yeoh and Je Sen Teh are both with the School of Computer Sciences, Universiti Sains Malaysia.

Jiageng Chen is with the School of Computer, Central China Normal University

memory limitations.

A. Related Work

Matsui first proposed a searching strategy based on the branch-and-bound method (B&B) for differential and linear trails which was applied on the Data Encryption Standard (DES) [13]. Various enhancements were later made to the Matsui B&B algorithm, adapting it to various block ciphers and using different pruning criteria. In [14], the B&B searching algorithm was adapted for ARX block ciphers by taking into account the differential property of addition operations. The differential search for ARX block ciphers was subsequently improved by [15] with the introduction of a sorted partial differential distribution table. The meet-in-the-middle (MITM) approach described in [16] trades off data complexity for improvements in terms of time complexity. It splits the search into two halves, storing intermediary round information for the first half which is later matched with the outputs of the second half of the search. However, this approach does not scale well beyond a general 32-bit block cipher or a 64-bit Feistel block cipher due to memory limitations. The searching strategy detailed in [17] primarily uses the number of active s-boxes to guide the pruning process of the B&B algorithm.

Alternatively, the mixed-integer linear programming (MILP) method proposed in [18] can be used to automate the search for differential characteristics. The proposed MILP method identifies the minimum number of active s-boxes for a particular block cipher by solving relevant linear inequalities using an MILP solver. The MILP method was later extended by [19] to enable the enumeration of differential or linear characteristics satisfying certain properties, which can then be used to form a differential or linear hull. However, the MILP method is inefficient when dealing with large block sizes or number of rounds. Similarly, SAT/SMT solvers have also been used to automate the construction of differentials. For now, this approach has been limited to 64-bit block ciphers [20].

General-purpose graphical processing unit (GPGPU) technology allows the use of GPU hardware to accelerate processes unrelated to graphic manipulation tasks. There exists various research work attempting to accelerate the B&B algorithm using GPUs [21], [22], [23]. In general, the GPU parallelization of the B&B algorithm is complex due to its irregular nature in terms of workload, control flow, and memory access [24], [25]. Previous work such as [21] and [22] only attempted to partially parallelize the operation of B&B. Parallelizing all four operations of the B&B algorithm as described in [25] and [26] alleviate branch irregularity using an integer-vector-matrix (IVM) structure. However, the IVM approach requires the intermediary solution to be coded in a square-like regular matrix structure for which the differential cluster search problem does not have an efficient representation for.

In [12], a GPU-based B&B search for differential clusters was proposed. However, the framework has several shortcomings which include generalizability issues due to the specific choices of fixed parameters rather than variables for work assignments and the parallelization of only three out of four operations of the B&B process. There also exists a bottleneck due to memory bandwidth, leading to a noticeable drop in performance when carrying out the search for a larger number of rounds. More recently, [27] used a GPU-based branch-and-bound search to compute the full differential distribution for block size-reduced variants of PRESENT that range from 8 to 28 bits. A straightforward implementation is used whereby each thread is assigned an input difference to compute. Each thread computes the difference propagation for one round and stores the result in an array whose size is equivalent to the number of differences (2^b for a b-bit block cipher). Thus, the proposed method is still infeasible for large block sizes due to memory limitations. In addition, the distribution of different branch sizes for different input differences reduces the GPU efficiency due to branch divergence.

B. Contribution

In this paper, we propose an optimized B&B GPU framework[2] for the automated search for block cipher differentials that overcomes prior performance bottlenecks. The GPU framework can be applied to various block ciphers without sacrificing efficiency. When configured for optimal performance, the proposed framework can achieve a speedup of up to 90x while saving up to 47% of the running cost as compared to a single CPU core. If cost-saving is the goal, the proposed framework can be configured to save up to 83% of the running cost while retaining a speedup of up to 40x as compared to single CPU core. When comparing the GPU-accelerated MITM approach to the traditional single-core CPU non-MITM approach, a speedup of 2292x is achieved, which significantly improves upon the results described in [12] where a speedup of only 58x the original throughput was achieved due to the reliance on the active s-boxes count for an input difference, ΔX.

Efficiency is optimized through new work distribution and acquisition strategies, implicit partitioning via numbered branches indexing, and minimal communication among threads. Moreover, the framework utilizes a kernel grid-level synchronization mechanism to achieve efficient communication among threads and efficient GPU-based recursion. The proposed GPU framework only requires the initial input differences, ΔX and ΔY, and can perform the search entirely in the GPU kernel. This leads to an additional advantage whereby the CPU and GPU can independently solve for different pairs of ΔX and ΔY with minimal communication apart from generating the required difference pairs. Therefore, proposed GPU framework also opens up the possibility of performing differential cluster search in heterogeneous CPU-GPU computing clusters which are quite prevalent.

As a proof-of-concept, the proposed framework is used to search for improved differentials for multiple block ciphers which include 128-bit TRIFLE-BC [28], 64-bit PRESENT [5], and 64-bit GIFT [7]. These block ciphers were selected to highlight the capability of the framework to generalize across multiple block ciphers with different block sizes and

[2]The source code of the framework is made publicly available for use at the following URL: https://github.com/yeohweizhu/gpu-differential-search-framework

designs. We improve upon the best-known differential probability for 16-round PRESENT from $2^{-62.13}$ to $2^{-61.7964}$. We also improved upon the best-known differential probability for 13-round GIFT from $2^{-61.3135}$ to $2^{-60.66}$ by constructing a differential with approximately one million characteristics. Although no significant improvement to the differential probability for TRIFLE-BC was obtained, we showed that the proposed framework can efficiently enumerate a large number of characteristics even for a block size of 128 bits. Differentials with over 3×10^6 characteristics can be constructed for 43-round TRIFLE-BC, which is approximately 5.8x more than in [12]. However, we wish to note that these cryptanalytic findings are not the main focus of the work, but serve to demonstrate the generalizability and efficiency of the proposed framework.

C. Outline

The remainder of this paper is structured as follows: Section II provides details on GPU architecture and NVIDIA GPGPU technology, CUDA. In Section III, a baseline serialized B&B algorithm for differential search is introduced. The GPU-accelerated framework and its performance evaluation are detailed in Section IV. In Section V, new differential results for selected block ciphers are provided. Section VI concludes the paper.

II. Preliminaries

A. GPU

Graphics processing units (GPUs) are computing hardware designed for efficient graphical or image data processing in a parallelized and multithreaded environment. It is based on the single instruction, multiple threads (SIMT) execution model where multiple threads that reside within each processing unit execute common instructions in lock-step. In 2006, NVIDIA introduced the Compute Unified Device Architecture (CUDA) programming interface which enables the use of GPUs for general-purpose computation unrelated to that of the graphical manipulation tasks. Over the years, GPU has gained traction as a hardware accelerator for specialized computation tasks such as machine learning [29], protein sequence alignment [30], and molecular dynamics simulation of thermal conductivities [31].

A **host** program residing in the CPU will assign tasks to CUDA threads that run on a separate physical **device** (GPU). We will be using the host and device terminology throughout the remainder of this paper. The relationship between the host and device code is illustrated in Fig.1. From the figure, we can see that the serial host code executes on the CPU whereas the parallel device code executes on the GPU. In CUDA, a kernel is a unit function that will be executed by different CUDA threads in parallel. A kernel launch is composed of a single grid that holds up to $2^{31} - 1$ blocks, whereas each block can contain up to 2^{10} threads.

Once a kernel is loaded into the GPU and launched, idle streaming multiprocessors (SM) will be assigned a block to execute. A group of 32 block threads known as a warp is then executed simultaneously. Warp threads can only execute one common instruction at a particular time. If threads within the warp were to diverge in their instruction code due to the nature of conditional branching, each branch will then be executed in different warp cycles because of the distinct instruction code. Therefore, we need to minimize the use of conditional branches to maximize GPU performance. Also, the number of threads per block should be a multiple of 32 due to the warp's group size.

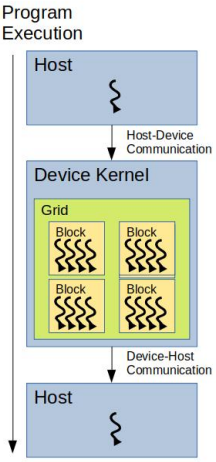


Fig. 1. Heterogeneous programming architecture of a typical GPU-accelerated algorithm.

Various types of memory are accessible by CUDA threads during kernel execution. The GPU memory hierarchy is as shown in Fig.2. Each thread has its own local memory that is inaccessible by other threads. Meanwhile, a shared memory space is available for each thread within the same block. A simple barrier synchronization primitive for shared memory, __syncthreads() is provided for achieving synchronization among thread blocks. Global memory, read-only constant memory, and read-only texture memory are accessible by all threads within the grid. Global memory has the slowest access speed and is accessed via 32-, 64-, or 128-byte memory transactions. Constant memory is best suited for broadcasting whereby all threads of the same warp need access to the same memory address. Texture memory is optimized for 2D spatial locality [32], yielding maximum throughput when threads of the same warp read or write to memory addresses that are adjacent to each other. The global, constant, and texture memory spaces are persistent across kernel launches by the same application. Therefore, the re-initialization of these memory spaces may need to be carried out as required by the program logic. As different memory types are better suited for different tasks, the memory access pattern of a CUDA program should be designed to take advantage of the different memory types in an effort to maximize memory throughput. This in turn improves upon the overall program efficiency.

Host and device memory spaces are physically distinct and are, by default, not synchronized by CUDA. As such, device memory allocation and transfer have to be managed explicitly by the users during runtime. However, there exists a unified

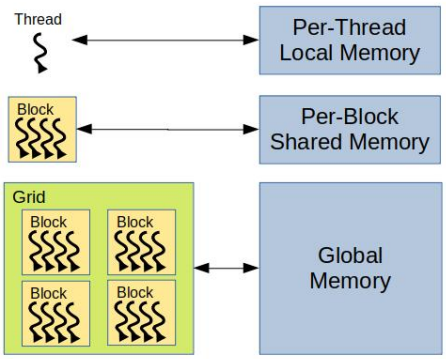


Fig. 2. GPU memory hierarchy.

managed memory that provides a single coherent memory space that alleviates the complexity of manual memory management. However, the proposed framework opts for manual memory management for greater flexibility in memory usage.

The CUDA dynamic parallelism feature enables a parent grid to launch its kernels (known as child grids) during its execution which allows a CUDA-specific recursive solution to be programmed. These nested kernel launches allows a CUDA program to complete a series of tasks without relying on the CPU to launch additional kernels. Through appropriate utilization of dynamic parallelism, it is possible to reduce the frequency and magnitude of memory transfers required by a CUDA program, potentially overcoming memory bottlenecks. There is however a hard limit on the recursion depth that may render some recursive solutions infeasible.

In parallel computation, threads communicate with one another either implicitly or explicitly to protect data integrity. This thread collaboration requires some form of synchronization. Historically, CUDA provides block-level thread synchronization to allow communication between threads within the same block. Other ways to achieve synchronization include implicit device synchronization by partitioning kernel launches or by utilizing built-in atomic functions to protect data integrity. In CUDA 9, NVIDIA introduced cooperative groups whereby partial or complete threads that reside within the same block or across multiple blocks could synchronize with one another to facilitate cooperation. Cooperative groups also have the ability to dictate grid or kernel level synchronization while a CUDA program is running. This feature allows a GPU kernel to compute B&B recursively, overcoming the recursion depth limitation imposed by dynamic parallelism. Hence, it is possible to model a slightly more complex data-parallel program to be executed by a GPU entirely but its performance will be difficult to optimize due to memory access patterns and thread divergence issues. The proposed framework opts for a grid-wise cooperative group to model the recursive nature of the B&B algorithm because of its flexibility to enforce grid-level, barrier-based synchronization.

III. Serialized Differential Search

This section introduces the base algorithm that will be parallelized by the proposed GPU framework. It is a sequential algorithm based on an enhanced version of Matsui's B&B algorithm. This algorithm will be used as the baseline algorithm to analyze performance gains and cost reduction of the proposed GPU-based framework. Matsui's algorithm uses the best differential characteristic probability found so far, $\overline{B_n}$ for a particular round n to prune branches and reduce the search space. $\overline{B_n}$ is updated throughout the search. As $\overline{B_n}$ approaches the best actual probability of the trail, B_n, the algorithm will approach its most efficient state.

The B&B algorithm consists of four operations: selection, branching, bounding, and pruning. Selection picks the next available node from a list of pending nodes to perform branching. Branching proceeds to decompose a parent node into child nodes whose costs are evaluated by the bounding operation. Pruning then eliminates nodes that fail the bounding operation, essentially filtering nodes that are not expected to produce desirable results. Thus, the search space can be reduced to a manageable size for large problem instances depending on how strict the bounding operation is.

A combination of the number of active s-boxes, AS_{BOUND} [16] and the differential probability threshold, P_{BOUND} [14] are used as the pruning rules in the proposed work. This specific combination facilitates greater pruning flexibility during the search while also effectively filters branches if configured correctly. In short, the serialized differential search algorithm is based on Matsui's B&B algorithm with the $\overline{B_n}$ pruning criteria replaced with AS_{BOUND} and P_{BOUND}. This algorithm is described in Algorithm 1.

To construct differentials, the serialized searching algorithm first identifies a set of individual differential characteristics. For each of these differential characteristics, the new GPU-based framework described in the following section is used to identify additional characteristics that correspond to the same input and output differences, thus forming differentials with improved probability. We refer to this process as the clustering process.

IV. GPU-Accelerated Framework for Differential Search

This section provides a general framework for the differential search of block ciphers that leverages upon the parallel processing power of GPUs. The MITM technique is incorporated into the framework to further enhance search efficiency. We also take other factors into consideration when designing the framework such as the s-box dependency of differential distribution tables (DDT) and CUDA hardware resources of different GPU architectures. These considerations ensure that the proposed framework is applicable to a wide range of block ciphers and GPUs. In the following descriptions, ΔU_n and ΔV_n denote the n^{th} nibble of ΔX and ΔY respectively. The size of each nibble is equivalent to the size of the s-box being used by the targeted cipher.

Algorithm 1 Serialized differential cluster searching algorithm with constraints on the probability and number of active s-boxes.

Input: Input difference ΔX and output difference ΔY.
Output: Probability $\Pr_c$ of $\Delta X \rightarrow \Delta Y$ cluster.
Adjustable Parameters:
1) AS_{BOUND} : Maximum of number of active sboxes for ΔY.
2) P_{BOUND} : Minimum probability of $\Delta X \rightarrow \Delta Y$.
3) P_{AS} : Estimated probability of a nibble $\Delta U \rightarrow \Delta V$.
procedure CLUSTER_SEARCH_ROUND_i $(1 \leq i < n)$
  for each candidate ΔY_i **do**
    $p_i \leftarrow \Pr(\Delta X_i, \Delta Y_i)$
    $AS_{i+1} \leftarrow W_{nibble}(\Delta Y_i)$
    if $AS_{i+1} \leq AS_{BOUND}$ **then**
      $p_{i+1} \leftarrow (P_{AS})^{AS_{i+1}}$
      $p_r \leftarrow (P_{AS})^{n-i-1}$
      if $[p_1, ..., p_i, p_{i+1}, p_r] \geq P_{BOUND}$ **then**
        call CLUSTER_SEARCH_ROUND_$(i+1)$
      end if
    end if
  end for
end procedure

procedure CLUSTER_SEARCH_ROUND_n
  for each candidate ΔY_n **do**
    if $\Delta Y_n == \Delta Y$ **then**
      $p_n \leftarrow \Pr(\Delta X_n, \Delta Y_n)$
      $P_c \leftarrow P_c + [p_1, ..., p_n]$
    end if
  end for
end procedure

A. Framework Description

1) Parallelization Model: All four operations (selection, branching, bounding, and pruning) involved in the B&B algorithm are fully parallelized in the proposed model. Parallelization of both bounding and pruning operations is achieved by processing the partial differential results obtained from the selection and branching operations within the same work thread. The parallelization of the selection operation and its subsequent branching operation can be modeled to span across multiple input differences rather than just one difference (which was previously performed in [12]). Let $NB_{\Delta X}$ be the number of differential trail branches for a particular input difference, ΔX. The bundling of multiple ΔX enables the parallelized selection and branching operations to leverage upon the aggregated problem space constituted by individual $NB_{\Delta X}$. This in turn maximizes the data-parallel processing capability of the GPU.

Let $B(\Delta X_k^r)$, where r is the round-number and k is the index position, denote the function that comprises of the branching, bounding, and pruning operations of B&B that produces the following round's branched partial differential characteristics,

$$\begin{aligned}\{&\Delta X^{r+1}_{(\sum_{l=1}^{k-1} NB_{\Delta X_l^r})}, \Delta X^{r+1}_{(\sum_{l=1}^{k-1} NB_{\Delta X_l^r})+1},\\ &\dots,\\ &\Delta X^{r+1}_{(\sum_{l=1}^{k} NB_{\Delta X_l^r})-1}, \Delta X^{r+1}_{(\sum_{l=1}^{k} NB_{\Delta X_l^r})}\}.\end{aligned} \tag{1}$$

Let D_n represent a set of differential characteristics after n rounds and $B(D_n)$ represents the branching operation performed on the set. The complete set of branched differential characteristics, $\mathbb{D}$ is defined as

$$\begin{aligned}\mathbb{D} &= \{D_0, D_1, D_2, \dots, D_n\},\\ D_0 &= \{\Delta X_0\},\\ D_1 &= B(D_0),\\ &= \bigcup_{\forall \Delta X_k^r \in D_0} B(\Delta X_k^r),\\ D_2 &= B(D_1),\\ &= \bigcup_{\forall \Delta X_k^r \in D_1} B(\Delta X_k^r),\\ &\vdots\\ D_n &= B(D_{n-1}).\end{aligned} \tag{2}$$

Both work acquisition and distribution strategies factor in the selection of appropriate differential characteristics, ΔX_k^r where k is the k^{th} differential characteristic in $\mathbb{D}$ after r rounds. Next,

$$\begin{aligned}T_i((I_1, I_2, \dots, I_{AS}), k, r) =&(\sum_{l=1}^{k-1} NB_{\Delta X_l^r})\\ &+ \sum_{j=1}^{AS}(I_j \times \prod_{n=0}^{j-1} NB_{\Delta U_n}[\Delta X_k^r]),\end{aligned} \tag{3}$$

$$I_n = \frac{T_i - (\sum_{l=1}^{k-1} NB_{\Delta X_l^r})}{\prod_{j=0}^{n-1} NB_{\Delta U_j}[\Delta X_k^r]} \pmod{NB_{\Delta U_n}[\Delta X_k^r]}, \tag{4}$$

where $NB_{\Delta U_0} = 1$ and the index sequence $(I_1, I_2, \dots, I_{AS})$, are used to ensure that a thread, T_i is working on the correct branches of ΔX_k^r, whereas $NB_{\Delta U_n}[\Delta X_k^r]$ denotes the number of possible partial branches, $\#\Delta V_n$ for ΔU_n of the n^{th} active s-box of ΔX_k^r.

Assuming that a GPU model has an infinite number of threads, parallelization of the B&B algorithm is achieved by first distributing tasks by computing Eq. 3. Meanwhile, threads acquire their tasks by computing the Eq. 4. Branching, bounding, and pruning are then executed sequentially, and the full parallelization of the algorithm is complete. The parallelization model described here resembles a typical breadth-first search. In practice however, these threads are essentially virtual, whereby GPU threads are mapped to one or multiple virtual worker threads. The mapping of these threads is discussed in the following subsection.

2) Meet-in-the-Middle Approach: The MITM approach described in [16] is an effective method for improving the efficiency of the differential search. As the search space grows exponentially when the number of rounds increases, the search can be made more efficient by dividing it into two connecting halves, each with α and β rounds. Searching α and β separately has a lower computational cost as compared to searching the entire $(\alpha + \beta)$ rounds. The MITM approach caches partial differential characteristics from one half, and matches them to partial differential characteristics obtained from the other (which is computed in reverse/decryption). Thus, the MITM approach trades memory space for performance gain by partially eliminating redundant computation.

For a block size of 32 bits (or equivalently, half the block of a 64-bit Feistel cipher), it is possible to store all 2^{32} possible differences that can be represented as a 32-bit data block. This amounts to $\approx 4GB$ worth of differential characteristic information in 32-bit floating-point format which takes up to $4 \times 8 = 32GB$ of memory space. For every additional bit of information that needs to be stored, the memory requirement is doubled. This memory requirement can exceed the capacity of RAM storage and require that the MITM intermediary results be written to secondary memory, i.e. hard disk drives. Nevertheless, storing 64 bits of information is infeasible for current memory storage solutions as a permutation of 64 bits requires approximately 18 exabytes. The latency for manipulating such a tremendous amount of memory further exacerbates the issue.

In order to practically implement the MITM approach beyond 32 bits, we must reduce the storage requirement for differential characteristics. To achieve this, intermediary characteristics can be encoded as

$$\begin{aligned}[Pos_{\Delta AV_i}, \Delta AV_i,\\ Pos_{\Delta AV_{i+1}}, \Delta AV_{i+1},\\ \dots,\\ Pos_{\Delta AV_{i+n}}, \Delta AV_{i+n}],\end{aligned} \tag{5}$$

where ΔAV_i is the i^{th} active non-zero (s-box) nibble, and $Pos_{\Delta AV_i}$ represents the position of the aforementioned nibble. The encoding method will capture all possible permutations of an intermediate difference, ΔY_α where $0 \leq AS_{\Delta Y_\alpha} \leq \nu$, where ν is the limit to the number of active s-boxes for differences being stored in memory. . The primary objective of this encoding method is to maximize ν with respect to memory size. A visual memory space reference is provided in Fig.3 which depicts the number of active s-boxes and their corresponding number of permutations. Table I summarizes the recommended number of active s-boxes with respect to memory feasibility.

MITM starts by diving the search process into two, namely a forward (encryption) α-round search and a backward (decryption) β-round search. The α-round search is basically a standard differential characteristic search. However, the original evaluation of ΔY_α during the α^{th} (final) round is replaced with the cache accumulation of ΔY_α and its corresponding probability. The cache is written to RAM using the encoding method described in Eq. 5 and stored for matching purposes.

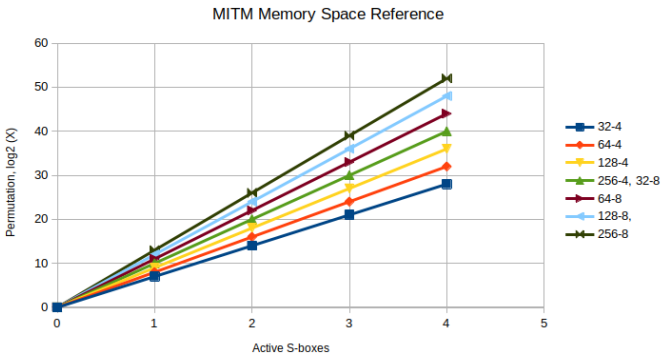


Fig. 3. MITM encoding reference based on Eq. 5.

TABLE I
RECOMMENDED $AS_{\Delta Y_\alpha}$ CONFIGURATION BASED ON EQ. 5.

Block Cipher Size (bit)	S-box Size (bit)	$AS_{\Delta Y_\alpha}$
32	4	FULL
64	4	3/4
128	4	3
256	4	3
32	8	3
64	8	3
128	8	2
256	8	2

Meanwhile, the β-round backward search starts from the output difference, ΔY and works its way to the middle (meeting point). In other words, if $\Delta X = \Delta X_0$ and $\Delta Y = \Delta X_n$, then $\Delta X_\alpha = \Delta X_0$, $\Delta X_\beta = \Delta X_n$, $\Delta X^1_\alpha = \Delta X_1$ and $\Delta X^1_\beta = \Delta X_{n-1}$. The reverse search phase requires the use of an inverted DDT and permutation based on the targeted block cipher's design. During the β^{th} (final) round, ΔY_β is encoded using the aforementioned encoding method, then matched with intermediary characteristics stored in the cache. All matched trails improves the overall differential (cluster) probability, P_c. As the search is divided into two halves, the P_{BOUND} specified for MITM approach represents both the forward search probability bound, P_{BOUND_α} and backward search probability bound, P_{BOUND_β}.

3) Proposed GPU Framework: .

The proposed parallelization model assumes that there is an infinite number of computing threads, T_s available to process $B(D_r)$ for a particular round of differential characteristics, where $|T_s| = |B(D_r)|$. In practice, GPU hardware can only accommodate a finite number of threads in a kernel grid. The number of kernel grid threads can be defined as $|GT_{grid}| = \#\{GT_i : GT_i \in \text{kernel grid}\}$, which can be computed as $|GT_{grid}| = thread\_block \times block\_num$. In the situation where $|GT_{grid}| < |T_s|$, GT_{grid} has to partition the round differential branching operation, $B(D_r)$ into equally divided sections known as μ-sections where $\mu = \{\mu_1, \mu_2, \dots, \mu_m\}$ such that $|GT_{grid}| \times |\mu| \geq |T_s|$ and $|\mu|$ is minimized. In the event where $|GT_{grid}| > |T_s|$, the unused threads will remain idle and wait for the rest of the grid to reach the same state. Only then will

the next set of operations continue. Let $\tau(GT_i)$ be a function that allocates a subset of virtual threads, T_r to each GT_i. The thread emulation of $GT_{grid} = \{GT_1, GT_2, \ldots, GT_n\}$ and its corresponding μ distribution can be defined as

$$\begin{aligned} \tau(GT_i) &= \{T_{(i-1)|\mu|+1}, T_{(i-1)|\mu|+2}, \ \ldots \ , T_{(i-1)|\mu|+|\mu|}\}, \\ \mu_j &= \bigcup_{i=1}^{n} \{T_{(i-1)|\mu|+j}\}, \end{aligned} \tag{6}$$

$$\begin{aligned} \tau(GT_i) &= \{T_i, T_{i+n}, T_{i+2n}, \ldots, T_{i+(|\mu|-1)n}\}, \\ \mu_j &= \bigcup_{i=1}^{n} \{T_{(j-1)n+i}\}. \end{aligned} \tag{7}$$

where $n = |GT_{grid}|$. Eq. 6 exploits the spatial locality of ΔX_k^r required by individual threads to drastically reduce the number of steps required to find ΔX_k^r for the remaining $|\mu|-1$ steps. Therefore, Eq. 6 is more preferable than Eq. 7 in this framework.

Storing all computational results of $B(D_r)$ for a large number of $|B(D_r)|$ in GPU memory is infeasible. To address this issue, we first partition the work units into μ subsections, $\{\mu_i, \mu_{i+1}, \ \ldots \ , \mu_{i+l}\}$ of $B(D_r)$. These subsections are executed in groups, M_k where $M_k = \{\mu_i, \mu_{i+1}, \ \ldots \ , \mu_{i+l}\}$ and $M = \{M_1, M_2, \ \ldots \ , M_q\}$, where q is the number of recursion cycle needed by GT_{grid} to completely emulate T_i in a particular round and l is obtained by computing $\frac{|T_s|}{|GT_{grid}|}$. This partitioning strategy allows the framework to execute a specified number of μ subsections as a group to reduce the overhead of recursion using the GPU's cooperative group feature. The search continues to operate recursively whereby the process of $B(D_r) \rightarrow (D_{r+1})$ is repeated until $B(D_s)$, where s is the target round. The search then moves on to process the next M group from the previous rounds. The relationship between μ and M is illustrated in Fig.4.

The aforementioned approach can be viewed as a hybridization of breadth-first and depth-first search. The algorithm starts off in a breadth-first search state which ends after processing an $|M_k|$ number of μ subsections. Then, the algorithm advances one level (depth-first state transition) and continues its breadth-first search strategy to process the first $|M_k|$ number of μ subsections of the current round. The process is repeated until the s^{th} round, where entire μ subsections are computed back-to-back before returning to the $(s-1)^{th}$-round. The entire process is repeated in a recursive manner. The choice of $|M_k|$ depends on GPU memory availability. In general, maximizing $|M_k|$ (and consequently minimizing $|M|$) will maximize efficiency. Fig. 5 illustrates the entire process.

Eq. 6 requires knowledge of $|\mu|$ in advance which can be calculated from $|B(D_{r+1})|$. For all ΔX_k^{r+1}, it is necessary to accumulate $NB_{\Delta X_k^{r+1}}$ during round r in order to calculate the relevant $|B(D_{r+1})|$ for round $r+1$. Since the recursive branching for the next round is carried out for the M_k^{r+1} group of subsections, only the $|B(M_k^{r+1})|$ group of subsections is required to be processed before advancing each level. For this purpose, all threads within the same block utilize block-level shared memory to accumulate the branching number of ΔX_k^{r+1} using the built-in block synchronization function

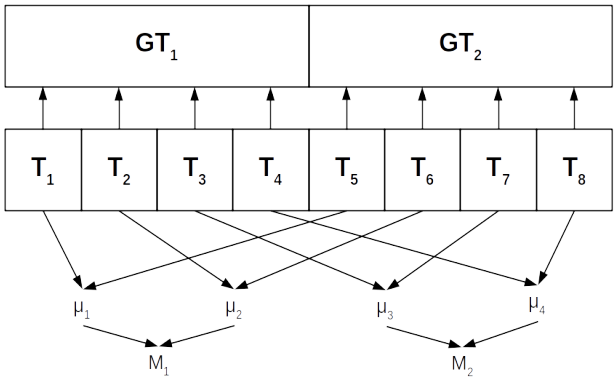


Fig. 4. Relationship between μ subsections and their corresponding groups M based on Eq. 6.

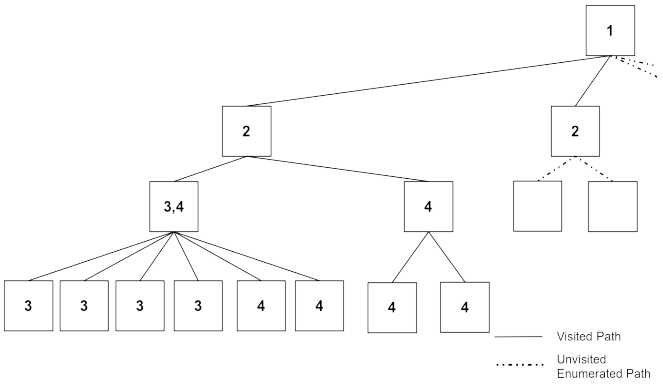


Fig. 5. Hybridized breadth and depth-first search example, where $|M_k| = 2$ and $GT_i = 2$.

$atomic\_add(target, value)$. Then, results accumulated from the block are assembled to form $|B(D_{r+1})|$.

Identification of ΔX_k^r and its branches $(I_1, I_2, \ \ldots \ , I_{AS})$ during $B(D_r)$ is performed using a linear search strategy. The search is further divided into three discrete levels, grid-level, block-level and thread level. The linear search is first executed at the grid-level to locate the targeted block, followed by the targeted thread within the identified block, and finally ΔX_k^r at the thread level. The GPU thread will omit pruned paths and keep a valid ΔX_k^{r+1} counter to facilitate the linear search process. We have also experimented with binary search as an alternative to linear search but did not obtain significant performance improvements.

The cooperative group feature is utilized for its ability to enforce grid-wise synchronization as required by the kernel. Specifically, a grid synchronization barrier is placed immediately after accumulating $|B(\Delta X_{block})|$ to ensure that it is ready to be referenced in the following round. The synchronization point also ensures the $global\_has\_operation$ is properly loaded with the correct value, which is used to determine whether to proceed to the next or return to the previous round.

When designing the proposed framework, the limitations of GPU resources in terms of its shared memory capacity, max register count, and max thread count have been taken into

Algorithm 2 Generalized GPU-accelerated B&B differential search (CPU)

Input: Input difference ΔX and output difference ΔY.
Output: Probability P_c of $\Delta X \rightarrow \Delta Y$ cluster.
procedure CLUSTER_SEARCH(ΔX)
    allocate device memory
    setup device memory for round 1
    call kernel CLUSTER_SEARCH_GPU_α
    reset device memory for round 1 and round 2
    setup device memory for round 1
    call kernel CLUSTER_SEARCH_GPU_β
    copy P_i from device to host
    $P_c \leftarrow (\sum_{i=1}^{T_{total}} P_i)$
end procedure

Algorithm 3 Simplified GPU-accelerated B&B differential search (kernel)

Input: Input Difference ΔX.
Output: Probabilities of ΔY that satisfy the searching constrained is accumulated in thread_num amount of P_i.
procedure CLUSTER_SEARCH_GPU_(α/β)
    while $r >= 0$ **do**
        for 1 to $|M_k|$ **do**
            //Selection, find the correct ΔX_k^r
            Select(thread_id, iter_count)
            Branch, Bound, Prune (thread_id, ΔX_k^r)
            if r == last_round **then**
                if forward **then**
                    Save to MITM cache array
                else
                    Match from MITM cache array
                    Save to final result array
                end if
            end if
        end for
        Update the state information
        Decide : $r \leftarrow r+1$, $r \leftarrow r-1$ or $r \leftarrow r$
    end while
end procedure

consideration. High GPU utilization requires efficient planning on resource utilization to maximize occupancy. Max register count is reduced by optimizing the computation pattern or by forcefully spilling register memory onto local memory using *__launch_bounds__* as provided by CUDA. Frequently accessed data is stored in shared memory or constant memory to improve the latency when accessing the data. However, this cannot be done for larger datasets and needs to be addressed on a case-by-case basis. The simplified version of the complete GPU framework algorithm along with the incorporation of the MITM technique is given in Algorithms 2 and 3. A more detailed algorithm for the GPU kernel can be found in Appendix A.

B. Performance evaluation

As a proof-of-concept, we apply the proposed framework described in Section IV-A on three block ciphers, the 128-bit TRIFLE-BC, 64-bit PRESENT and 64-bit GIFT. Efficiency and cost comparisons between the GPU framework and its CPU counterpart are performed based on the Google Cloud VM computing environment.

We configure the proposed GPU framework to utilize 1-dimensional blocks for the kernel. Since each block within a grid contains its own block threads, each thread is assigned a unique thread id based on its position in a given grid. This thread id assignment facilitates the process of work distribution and reduction. The number of threads per block (thread_block) is fixed at 128, SPACE_THREAD, $|M_k|$ is fixed at 64 and the number of blocks is maximized. This configuration allows the non-MITM variant of the GPU-accelerated algorithm to achieve 100% occupancy rate on the Tesla T4 GPU with 64 registers. Meanwhile, the MITM variant requires additional register spilling to achieve 100% occupancy. The permutation table is not loaded into shared memory as its size leads to occupancy reduction.

For the experiments to analyze the financial feasibility of the proposed GPU framework, we select the following parameters:

- $AS_{BOUND} = 4$
- $P_{BOUND_{offset}} = -21/-35$

For all characteristics that form a differential, the minimum probability bound used for the differential search can be calculated as

$$\min P_{char} = 2^{\log(P_{char_{best}}) + P_{BOUND_{offset}}}, \tag{8}$$

where $P_{char_{best}}$ represents the best probability of a differential characteristic found so far by the serialized B&B algorithm for a given $\Delta X \xrightarrow{r} \Delta Y$. The time taken for each device to finish computing r rounds is recorded. The cost percentage is then calculated as

$$\text{Cost} = \frac{\text{Cost}_{\text{device}}}{\text{Core Equivalence} \times \text{Cost}_{\text{ref}}} \times 100\%, \tag{9}$$

$$\text{Core Equivalence, CE} = \lceil \frac{\text{Time}_{\text{ref}}}{\text{Time}_{\text{device}}} \rceil \tag{10}$$

where Cost_{ref} refers to the cost of running the search using a reference (benchmark) device, a 3.1 GHz Intel Xeon CascadeLake processor core. The benchmark experiments involve constructing a differential for a given block cipher, for a specific iterative differential characteristic where $AS_{\Delta X} = 1$, $AS_{\Delta X} = 2$, and $AS_{\Delta X} = 2$. The same experiment is repeated for TRIFLE-BC, PRESENT and GIFT.

Note that the Google VM price structure is based on the us-central1 (Iowa) region's on-demand pricing excluding any sustained use discounts. The CascadeLake processors (provided by the C2 machine type) are only available in sets of 4 cores and 16-GB memory. Thus, the cost is divided by 4 to obtain the equivalent price of a single CascadeLake processor and 4-GB memory. Experimental results for the GPU-accelerated differential search without MITM are provided in Table II, specifically for TRIFLE-BC. Experimental results for the complete GPU-accelerated differential search with MITM are

provided in Tables III, IV, and V for TRIFLE-BC, PRESENT and GIFT respectively. Note that RX in the tables refers to X rounds of the block cipher.

TABLE II
COST COMPARISON OF THE NON-MITM SEARCH ON TRIFLE-BC CONSTRAINED BY $AS_{BOUND} = 4$ AND $P_{BOUND_{offset}} = -21$.

Device	Time(s)	Cost/Month	CE	Cost%
XEON CascadeLake 3.1GHz				
- R5	0.266	38.09	1	100
- R10	19.271	38.09	1	100
- R15	151.117	38.09	1	100
- R20	916.963	38.09	1	100
Tesla T4				
- R5	0.020	255.50	14	48
- R10	0.683	255.50	29	23
- R15	5.157	255.50	30	22
- R20	29.896	255.50	31	22
Tesla V100				
- R5	0.005	1810.40	54	88
- R10	0.265	1810.40	73	65
- R15	2.084	1810.40	73	65
- R20	12.706	1810.40	73	65

TABLE III
COST COMPARISON OF THE MITM SEARCH ON TRIFLE-BC CONSTRAINED BY $AS_{Bound} = 4$ AND $P_{BOUND_{offset}} = -21$.

Device	Time(s)	Cost/Month	CE	Cost%
XEON CascadeLake 3.1GHz				
- R10	1.61	38.09	1	100
- R20	30.588	38.09	1	100
- R30	197.938	38.09	1	100
- R40	1176.995	38.09	1	100
Tesla T4				
- R10	0.045	255.50	36	19
- R20	0.948	255.50	33	20
- R30	6.400	255.50	31	22
- R40	37.912	255.50	31	22
Tesla V100				
- R10	0.015	1810.40	107	44
- R20	0.400	1810.40	76	63
- R30	2.973	1810.40	67	71
- R40	18.295	1810.40	65	73

TABLE IV
COST COMPARISON OF THE MITM SEARCH ON PRESENT CONSTRAINED BY $AS_{Bound} = 4$ AND $P_{BOUND_{offset}} = -35$.

Device	Time(s)	Cost/Month	CE	Cost%
XEON CascadeLake 3.1GHz				
- R4	0.001	38.09	1	100
- R8	0.195	38.09	1	100
- R16	242.964	38.09	1	100
Tesla T4				
- R4	0.006	255.50	1	671
- R8	0.014	255.50	14	48
- R16	6.161	255.50	40	17
Tesla V100				
- R4	0.003	1810.40	1	4753
- R8	0.005	1810.40	39	122
- R16	3.140	1810.40	78	61

In terms of cost, we observe that both the MITM and non-MITM approaches have consistent results. A runtime cost reduction of up to 83% is observed for the GPU-accelerated B&B algorithm with MITM when using a Tesla T4 GPU unit. On the other hand, the cost reduction when using the more powerful (albeit less cost-effective) Tesla V100 GPU achieves a cost reduction of up to 47%. The cost analysis suggests that the GPU framework is more financially feasible for cloud-based implementations as compared to a regular CPU search. The costs saved from using the proposed GPU framework can be channeled towards more computing resources to conduct a larger scale differential search under a fixed budget.

TABLE V
COST COMPARISON OF THE MITM SEARCH ON GIFT CONSTRAINED BY $AS_{Bound} = 4$ AND $P_{BOUND_{offset}} = -35$.

Device	Time(s)	Cost/Month	CE	Cost%
XEON CascadeLake 3.1GHz				
- R4	0.003	38.09	1	100
- R8	0.349	38.09	1	100
- R16	5898.510	38.09	1	100
Tesla T4				
- R4	0.007	255.50	1	671
- R8	0.020	255.50	18	37
- R16	156.113	255.50	38	18
Tesla V100				
- R4	0.003	1810.40	1	4753
- R8	0.007	1810.40	50	95
- R16	65.994	1810.40	90	53

In terms of performance, a speedup of 2292 is achieved for 20 rounds of TRIFLE-BC using the MITM GPU-accelerated method as compared to the non-MITM CPU method. This is a significant improvement over the previously proposed GPU approach described in [12] which achieved a speedup of approximately 58 times under similar settings. When comparing the CPU and GPU implementations of the proposed framework with MITM, the GPU kernel can achieve a speedup of 90x on a high-performance Tesla V100 GPU while still delivering up to 40x on a lower-end Tesla T4 GPU. A similar performance boost can be observed in the non-MITM variant of the GPU framework as well. The performance boost obtained by using the proposed GPU framework leads to a more efficient construction of larger differentials, leading to improved differential probability.

The results indicate that the proposed framework can achieve high throughput while being cost-effective, making it useful for cryptanalysts who wish to construct large differentials for statistical attacks. However, the proposed framework is ineffective when used to analyze fewer rounds (such as 4 rounds) because there are too few branches to fully leverage upon the parallel processing power of the GPU. With that said, differential cryptanalysis is typically performed for a large number of rounds, for which the proposed GPU framework is useful. The GPU-accelerated MITM approach is also more computationally feasible for 128-bit or larger block ciphers with a large number of rounds as compared to existing approaches such as MILP or SAT solvers.

V. NEW DIFFERENTIAL RESULTS FOR EXISTING BLOCK CIPHERS

We use the proposed framework to search for improved differentials for TRIFLE-BC, PRESENT and GIFT, which are summarized in Tables VI, VII, and VIII respectively.

AS_{BOUND} is set to 4 for all experiments to ensure that the search can complete within a practical amount of time.

TABLE VI
DIFFERENTIAL FOR 43-ROUND TRIFLE-BC BOUNDED BY $AS_{BOUND} = 4$, $P_{BOUND_\alpha} = -150$ AND $P_{BOUND_\beta} = -150$.

ΔX	ΔY	P_c	# of Trails
000000000000b000	0000000000100000	$2^{-126.931}$	3.381×10^6
0000000000000000	0010000000000000		
0000000000000000	0000000200000002	$2^{-126.931}$	3.325×10^6
b000000000000000	0000000000000000		
0000000000000000	0020000000200000	$2^{-126.931}$	3.346×10^6
0007000000000000	0000000000000000		
0000000000000000	0000000000000000	$2^{-126.995}$	2.501×10^6
00000b0000000000	0000040000000400		

TABLE VII
DIFFERENTIAL FOR 16-ROUND PRESENT BOUNDED BY $AS_{BOUND} = 4$, $P_{BOUND_\alpha} = -62$ AND $P_{BOUND_\beta} = -62$.

ΔX	ΔY	P_c	# of Trails
000f000000000000f	0000050000000500	$2^{-61.7964}$	4.00×10^{10}
0000000000001001	0404040400000000	$2^{-62.1757}$	4.98×10^{10}
0007000000000007	0000050000000500	$2^{-62.2031}$	4.62×10^{10}
0f000000000000f00	0000050000000500	$2^{-62.5550}$	3.50×10^{10}

TABLE VIII
DIFFERENTIAL FOR 13-ROUND GIFT BOUNDED BY $AS_{BOUND} = 4$, $P_{BOUND_\alpha} = -75$ AND $P_{BOUND_\beta} = -75$.

ΔX	ΔY	P_c	# of Trails
0f0000000c000000	1010808040402020	$2^{-60.6600}$	1.26×10^5
0c000000e0000000	2020101080805050	$2^{-60.9556}$	2.31×10^5
0e0000000e000000	0202010108080404	$2^{-61.0341}$	2.32×10^5
0e00000060000000	4040202010108080	$2^{-61.2720}$	4.27×10^5

P_{BOUND_α} and P_{BOUND_β} is varied for different block ciphers to account for their distinct differential characteristic distributions. Values for $P_{BOUND_{offset}}$ fall in the range of $[2, 27]$.

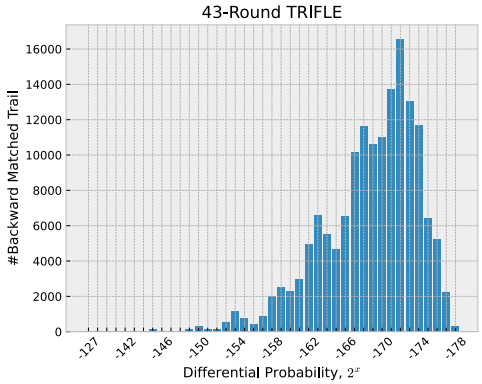


Fig. 6. Differential characteristics distribution of the best differential found for 43-Round TRIFLE.

A differential for 13-round GIFT with a probability of $2^{-60.66}$ has been identified, which is an improvement over the $2^{-61.3135}$ found in [33]. For 16-round PRESENT, the search has identified a differential with the probability of $2^{-61.7964}$ which is also an improvement over the $2^{-62.13}$ differential in [34] and the $2^{-62.27}$ differential in [17]. Thus, the proposed approach has identified the best differentials to date for 13-round GIFT and 16-round PRESENT. However, differentials for 43-round TRIFLE-BC could not be improved upon despite using more lenient searching bounds as compared to the $2^{-126.931}$ obtained in [12]. Note that the differentials constructed using the proposed framework consists of hundreds of thousands (GIFT) to billions (PRESENT) of individual characteristics.

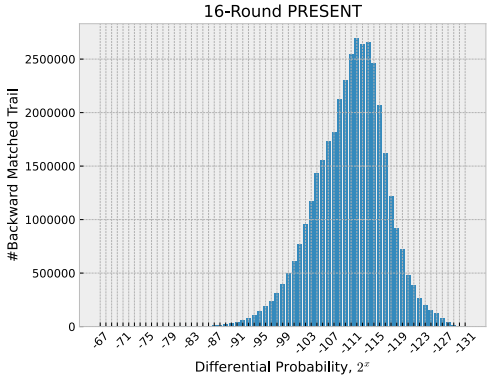


Fig. 7. Differential characteristics distribution of the best differential found for 16-Round PRESENT.

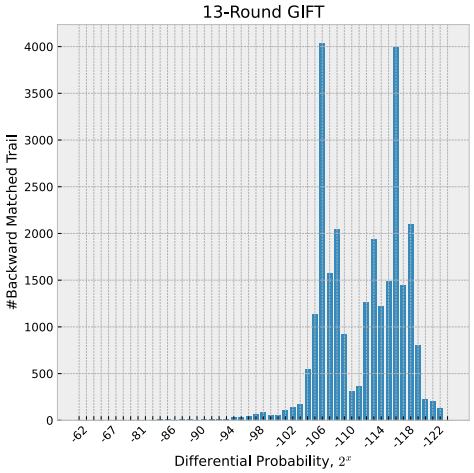


Fig. 8. Differential characteristics distribution of the best differential found for 13-Round GIFT.

Unfortunately, due to the inherent structure of MITM, it is infeasible to keep track of the exact partial characteristic probabilities and consequently the final characteristic probabilities which are required to assemble a complete differential characteristic distribution. In other words, the full differential distribution cannot be generated due to how the partial α characteristics are condensed into an intermediary array. On the other hand, collecting data about the differential distribution via the non-MITM method cannot be completed within a reasonable amount of time. Instead, a partial differential distribution can be constructed based solely on differential characteristics that have been matched during the reverse MITM matching phase. By adopting this approach, we generate the differential distributions of the best differentials for all three ciphers in Figs. 6, 7, and 8. Based on the Fig.6 and Fig.7, we can observe that the best differentials for both TRIFLE and PRESENT have an approximately normal

distribution of differential characteristics. However, the peak of the distribution for TRIFLE is skewed towards the right, implying that most of the individual differential characteristics have smaller probabilities. This explains why the proposed framework was unable to significantly improve upon existing differential probabilities. Meanwhile, the best differential for GIFT seems to follow a more erratic distribution. This implies that the differential characteristics for GIFT are not as evenly distributed as PRESENT, leading to a bigger improvement in terms of differential probability when a larger differential is constructed.

VI. Conclusion

In this paper, we proposed a new GPU-accelerated B&B framework for the differential search of block ciphers. The proposed framework was optimized for GPU parallel processing to achieve a substantial speedup when constructing large differentials (differentials that consist of large number of individual characteristics). Compared to an existing GPU approaches, the proposed framework is more practical adaptable to different GPUs and block ciphers. When compared to the original CPU-based non-MITM search, the proposed framework achieves a speedup of approximately 2292x. We demonstrate its practicality by applying the proposed framework on three different block ciphers, 128-bit TRIFLE-BC, 64-bit PRESENT and 64-bit GIFT. Experimental results indicate that the proposed GPU-accelerated algorithm can achieve up to a 90x speedup as compared to an equivalent single-core CPU algorithm. In terms of financial cost evaluated using Google Cloud VM, the proposed framework achieves savings of up to 83% when compared to a CPU setup with equivalent throughput. Therefore, the proposed GPU framework is both faster and cheaper than its CPU counterpart. The proposed framework can be used to effectively identify large differentials with higher differential probabilities, which can then be used in statistical-based attacks against existing block ciphers. In theory, the proposed framework also allows the utilization of existing CPU-GPU heterogeneous computing clusters as the entire search can be performed entirely on the GPU. Thus an separate differential search can be conducted on the CPU without interference. As additional contributions, we have also identified the best differentials to date for 16-round PRESENT and 13-round GIFT, with differential probabilities of $2^{-61.7964}$ and $2^{-60.66}$ respectively. We also show that the proposed GPU search is practical for 128-bit block ciphers with a large number of rounds by constructing large differentials for 43 rounds of 128-bit TRIFLE-BC.

Acknowledgment

This work is supported in part by the Ministry of Education Malaysia under the Fundamental Research Grant Scheme (FRGS), project number FRGS/1/2019/ICT05/USM/02/1, the National Natural Science Foundation of China under Grant No. 61702212 and the Fundamental Research Funds for the Central Universities under Grant No. CCNU19TS017.

References

[1] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC Editor, Tech. Rep. RFC8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[2] D. Shaw, "The Camellia Cipher in OpenPGP," RFC Editor, Tech. Rep. RFC5581, Jun. 2009. [Online]. Available: https://www.rfc-editor.org/info/rfc5581

[3] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol," RFC Editor, Tech. Rep. RFC4253, Jan. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4253

[4] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," *Journal of Cryptology*, vol. 4, no. 1, pp. 3–72, 1991. [Online]. Available: http://link.springer.com/10.1007/BF00630563

[5] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4727, pp. 450–466. [Online]. Available: http://link.springer.com/10.1007/978-3-540-74735-2_31

[6] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw, "The LED Block Cipher," *Cryptographic Hardware and Embedded Systems – CHES 2011*, pp. 326–341, 2011.

[7] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "GIFT: A Small Present," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, vol. 10529, pp. 321–345. [Online]. Available: http://link.springer.com/10.1007/978-3-319-66787-4_16

[8] S. Erich, D. Jack, S. Horst, and M. Martin, "November 19 | Top 500 Supercomputer," Nov. 2019. [Online]. Available: https://www.top500.org/lists/2019/11/

[9] M. Stevens, P. Karpman, and T. Peyrin, "Freestart Collision for Full SHA-1," in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, vol. 9665, pp. 459–483. [Online]. Available: http://link.springer.com/10.1007/978-3-662-49890-3_18

[10] R. Szerwinski and T. Güneysu, "Exploiting the Power of GPUs for Asymmetric Cryptography," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, E. Oswald and P. Rohatgi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5154, pp. 79–99. [Online]. Available: http://link.springer.com/10.1007/978-3-540-85053-3_6

[11] S. A. Manavski, "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography," in *2007 IEEE International Conference on Signal Processing and Communications*. Dubai, United Arab Emirates: IEEE, 2007, pp. 65–68. [Online]. Available: http://ieeexplore.ieee.org/document/4728256/

[12] W.-Z. Yeoh, J. S. Teh, and J. Chen, "Automated Search for Block Cipher Differentials: A GPU-Accelerated Branch-and-Bound Algorithm," in *Information Security and Privacy*, J. K. Liu and H. Cui, Eds. Cham: Springer International Publishing, 2020, vol. 12248, pp. 160–179. [Online]. Available: http://link.springer.com/10.1007/978-3-030-55304-3_9

[13] M. Matsui, "On correlation between the order of S-boxes and the strength of DES," in *Advances in Cryptology — EUROCRYPT'94*, G. Goos, J. Hartmanis, J. van Leeuwen, and A. De Santis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, vol. 950, pp. 366–375. [Online]. Available: http://link.springer.com/10.1007/BFb0053451

[14] A. Biryukov and V. Velichkov, "Automatic Search for Differential Trails in ARX Ciphers," in *Topics in Cryptology – CT-RSA 2014*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and J. Benaloh, Eds. Cham: Springer International Publishing, 2014, vol. 8366, pp. 227–250. [Online]. Available: http://link.springer.com/10.1007/978-3-319-04852-9_12

[15] K. Chen, X. Tang, P. Xu, M. Guo, W. Qiu, and Z. Gong, "An Improved Automatic Search Method for Diffierential Trails in TEA Cipher," *International Journal of Network Security*, vol. 18, no. 4, pp. 644–649, Jul. 2016.

[16] J. Chen, A. Miyaji, C. Su, and J. Teh, "Improved Differential Characteristic Searching Methods," in *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. New York, NY, USA: IEEE, Nov. 2015, pp. 500–508. [Online]. Available: http://ieeexplore.ieee.org/document/7371529/

[17] J. Chen, J. Teh, Z. Liu, C. Su, A. Samsudin, and Y. Xiang, "Towards Accurate Statistical Analysis of Security Margins: New Searching Strategies for Differential Attacks," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1763–1777, Oct. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7914659/

[18] N. Mouha, Q. Wang, D. Gu, and B. Preneel, "Differential and Linear Cryptanalysis Using Mixed-Integer Linear Programming," in *Information Security and Cryptology*, C.-K. Wu, M. Yung, and D. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7537, pp. 57–76. [Online]. Available: http://link.springer.com/10.1007/978-3-642-34704-7_5

[19] S. Siwei, H. Lei, W. Meiqin, W. Peng, Q. Kexin, M. Xiaoshuang, S. Danping, S. Ling, and F. Kai, "Towards Finding the Best Characteristics of Some Bit-oriented Block Ciphers and Automatic Enumeration of (Related-key) Differential and Linear Characteristics with Predefined Properties," 2014.

[20] R. Ankele and S. Kölbl, "Mind the gap - a closer look at the security of block ciphers against differential cryptanalysis," in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson Jr., Eds. Cham: Springer International Publishing, 2019, pp. 163–190.

[21] A. Borisenko, M. Haidl, and S. Gorlatch, "A GPU parallelization of branch-and-bound for multiproduct batch plants optimization," *The Journal of Supercomputing*, vol. 73, no. 2, pp. 639–651, Feb. 2017. [Online]. Available: http://link.springer.com/10.1007/s11227-016-1784-x

[22] N. Melab, I. Chakroun, M. Mezmaz, and D. Tuyttens, "A GPU-accelerated Branch-and-Bound Algorithm for the Flow-Shop Scheduling Problem," in *2012 IEEE International Conference on Cluster Computing*. Beijing, China: IEEE, Sep. 2012, pp. 10–17. [Online]. Available: http://ieeexplore.ieee.org/document/6337851/

[23] M. E. Lalami and D. El-Baz, "GPU Implementation of the Branch and Bound Method for Knapsack Problems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. Shanghai, China: IEEE, May 2012, pp. 1769–1777. [Online]. Available: http://ieeexplore.ieee.org/document/6270853/

[24] B. Gendron and T. G. Crainic, "Parallel Branch-and-Branch Algorithms: Survey and Synthesis," *Operations Research*, vol. 42, no. 6, pp. 1042–1066, Dec. 1994. [Online]. Available: http://pubsonline.informs.org/doi/abs/10.1287/opre.42.6.1042

[25] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens, "A GPU-based Branch-and-Bound algorithm using Integer–Vector–Matrix data structure," *Parallel Computing*, vol. 59, pp. 119–139, Nov. 2016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167819116000387

[26] ——, "IVM-Based Work Stealing for Parallel Branch-and-Bound on GPU," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski, and K. Wiatr, Eds. Cham: Springer International Publishing, 2016, vol. 9573, pp. 548–558. [Online]. Available: http://link.springer.com/10.1007/978-3-319-32149-3_51

[27] Z. Chen, J. Chen, W. Meng, J. S. Teh, P. Li, and B. Ren, "Analysis of differential distribution of lightweight block cipher based on parallel processing on GPU," *Journal of Information Security and Applications*, vol. 55, p. 102565, Dec. 2020.

[28] D. Nilanjan, G. Ashrujit, M. Debdeep, P. Sikhar, P. Stjepan, and S. Rajat, "TRIFLE," Mar. 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/trifle-spec.pdf

[29] D. Steinkraus, I. Buck, and P. Simard, "Using GPUs for machine learning algorithms," in *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*. Seoul, South Korea: IEEE, 2005, pp. 1115–1120 Vol. 2. [Online]. Available: http://ieeexplore.ieee.org/document/1575717/

[30] P. D. Vouzis and N. V. Sahinidis, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, Jan. 2011. [Online]. Available: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btq644

[31] J. Yang, Y. Wang, and Y. Chen, "GPU accelerated molecular dynamics simulation of thermal conductivities," *Journal of Computational Physics*, vol. 221, no. 2, pp. 799–804, Feb. 2007. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0021999106003172

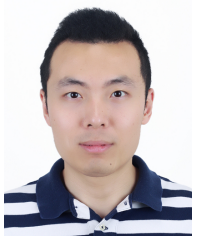
[32] D. Padua, Ed., *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011. [Online]. Available: http://link.springer.com/10.1007/978-0-387-09766-4

[33] H. Chen, R. Zong, and X. Dong, "Improved Differential Attacks on GIFT-64," in *Information and Communications Security*, J. Zhou, X. Luo, Q. Shen, and Z. Xu, Eds. Cham: Springer International Publishing, 2020, vol. 11999, pp. 447–462. [Online]. Available: http://link.springer.com/10.1007/978-3-030-41579-2_26

[34] M. Wang, Y. Sun, E. Tischhauser, and B. Preneel, "A Model for Structure Attacks, with Applications to PRESENT and Serpent," in *Fast Software Encryption*, A. Canteaut, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7549, pp. 49–68. [Online]. Available: http://link.springer.com/10.1007/978-3-642-34047-5_4

Wei-Zhu Yeoh received the B.Sc. (Hons) degree in Computer Science from Coventry University (a joint programme at INTI International College Penang). Upon completion of his B. Sc. degree, he is currently pursuing his M.Sc. in Computer Science from Universiti Sains Malaysia (USM) and is expected to graduate at the end of 2020. His research interests include symmetric cryptography, machine learning application in cryptography, parallel computing, and mobile computing.

Je Sen Teh received the B.Eng. degree (Hons.) majoring in Electronics from Multimedia University, Malaysia in 2011, then received his M.Sc. and Ph.D. in Computer Science from Universiti Sains Malaysia in 2013 and 2017 respectively. He is currently working as a Senior Lecturer in Universiti Sains Malaysia under the School of Computer Sciences. His research interests include symmetric cryptography, cryptanalysis and chaos theory.



Jiageng Chen received the B.S. degree from the School of Compuer Science and Technology, Huazhong University of Science and Technology (HUST) in 2004 and recieved his M.S. and PhD of computer science from the School of Information Science, Japan Advanced Institute of Science and Technology (JAIST) in 2007 and 2012, respectively. He was working as an Assistant Professor in School of Information Science, Japan Advanced Institute of Science and Technology from 2012 to 2015. And currently, he is an Associate Professor at the Computer School of Central China Normal University. His research areas include cryptography, especially in the areas of algorithms, cryptanalysis and secure designs.

APPENDIX A
GPU KERNEL ALGORITHM

Algorithm 4 GPU Kernel.

Input: Input Difference ΔX.
Output: Probabilities of ΔY that satisfy the searching constrained is accumulated in thread_num amount of P_i.
Adjustable Parameters:
1) AS_{BOUND} : Maximum of number of active s-boxes for ΔY.
2) P_{BOUND} : Maximum probability of $\Delta X \rightarrow \Delta Y$.
3) P_{AS} : Estimated probability of a nibble $\Delta U \rightarrow \Delta V$.
Assumption:
1) Non-active nibble (s-boxes) will have a difference value of zero. Thus, an attempt to differentially substitute it will yield $0 \rightarrow 0$ with a probability of 1.
procedure CLUSTER_SEARCH_GPU_(α/β)
    global memory: init α/β permutation table
    shared memory: init α/β DDT, branch size table
    thread_id_default $\leftarrow (GT_i - 1)$ ▷ $GT_{grid} - 1$
    $r \leftarrow 0$
    dx_ptr $\leftarrow 0$
    cur_iter $\leftarrow -1$
    MAX_PATH_ROUND $\leftarrow |GT_{grid}| \times |M_k|$
    while $r \geq 0$ **do**
        if cur_iter == -1 **then**
            cur_iter $\leftarrow \lceil \frac{|B(D_r)|}{|GT_{grid}|} \rceil$ ▷ $|\mu|$
            //T_i pointer
            cur_thread_id $\leftarrow$ thread_id_default $\times$ cur_iter
        end if
        has_operation $\leftarrow$ False
        //Valid ΔX^{r+1} is saved at the front
        thread_dx_num $\leftarrow 0$
        thread_ptr $\leftarrow \lfloor \frac{\text{dx\_ptr}}{|M_k|} \rfloor$
        block_ptr $\leftarrow \frac{grid\_thread\_ptr}{THREAD\_BLOCK}$
        loop_limit $\leftarrow$ (cur_iter $< |M_k|$? cur_iter : $|M_k|$)
        for cur_loop $\leftarrow 1$ to loop_limit **do**
            //Selection, find the correct $\Delta X^r_{\text{dx\_ptr}+1}$
            if dx_ptr $<$ MAX_PATH_ROUND **then**
                nb_temp $\leftarrow (\sum_{v=0}^{dx\_ptr} |B(\Delta X_v)|)$
                if cur_thread_id $<$ nb_temp **then**
                    //Skip if dx_ptr does not need to change
                    goto Branch_Bound_Pruning
                end if
                //Find the correct block
                bsum $\leftarrow \sum_{v=0}^{block\_ptr-1} |B(\Delta X_{block_v})|$
                init_block_ptr $\leftarrow$ block_ptr
                v $\leftarrow$ block_ptr
                for v $<$ BLOCK_NUM **do**
                    bsum += $|B(\Delta X_{block_v})|$
                    cond1 $\leftarrow$ cur_thread_id $<$ bsum
                    cond2 $\leftarrow |B(\Delta X_{block_v})| \neq 0$
                    if cond1 $\wedge$ cond2 **then**
                        bsum -= $|B(\Delta X_{block_v})|$
                        break
                    end if
                    v $\leftarrow$ v+1
                end for
                block_ptr $\leftarrow$ v
                bsum_t $\leftarrow \sum_{v=0}^{thread\_ptr-1} |B(\Delta X_{thread_v})|$
                t_temp $\leftarrow$ block_ptr $\times$ THREAD_BLOCK
                if block_ptr == BLOCK_NUM **then**
                    dx_ptr $\leftarrow$ MAX_PATH_ROUND
                else
                    //Find the correct thread
                    if init_block_ptr == block_ptr **then**
                        bsum $\leftarrow$ bsum_t
                    else
                        //bsum remained unchanged
                        thread_ptr $\leftarrow$ t_temp
                    end if
                    init_thread_ptr $\leftarrow$ thread_ptr
                    v $\leftarrow$ thread_ptr
                    while True **do** ▷ Guaranteed to find
                        bsum += $|B(\Delta X_{thread_v})|$
                        cond1 $\leftarrow$ cur_thread_id $<$ bsum)
                        cond2 $\leftarrow |B(\Delta X_{thread_v})| \neq 0$
                        if cond1 $\wedge$ cond2 **then**
                            bsum -= $|B(\Delta X_{block_v})|$
                            break
                        end if
                        v $\leftarrow$ v + 1
                    end while
                    thread_ptr $\leftarrow$ v
                    //Find ΔX^r_k
                    if init_thread_ptr == thread_ptr **then**
                        bsum $\leftarrow \sum_{v=0}^{dx\_ptr-1} |B(\Delta X_v)|$
                    else
                        //bsum remained unchanged
                        dx_ptr $\leftarrow$ thread_ptr $\times$ $|M_k|$
                    end if
                    v $\leftarrow$ dx_ptr
                    while True **do** ▷ Guaranteed to find
                        bsum += $|B(\Delta X_v)|$
                        if (cur_thread_id$<$ bsum) **then**
                            bsum -= $|B(\Delta X_{block_v})|$
                            break
                        end if
                        v $\leftarrow$ v + 1
                    end while
                    dx_ptr $\leftarrow$ v
                end if
            end if
        end if
        //Selection of B&B done

```
//Branch, Bound and Pruning
if dx_ptr < MAX_PATH_ROUND then
    has_operation ← True
    p_i ← Pr(ΔX^0 → ΔX^r_{dx_ptr})
    vl← cur_thread_id - Σ_{v=0}^{dx_ptr-1} |B(ΔX_v)|
    div_vl ← 1
    //Branch to predetermined child node
    for each ΔAU_i from ΔX^r_{dx_ptr} do
        I_i ← ⌊vl/div_vl⌋ mod NB_i
        ΔAV_i ← sorted DDT[ΔAU_i][I_i]
        update p_i
        div_vl ← div_vl × NB_i
    end for
    not_last_round ← r ≠ LAST_ROUND - 1
    if α_SEARCH ∨ not_last_round then
        Permutate ΔX^r_{dx_ptr} into ΔX^{r+1}
    end if
    if r ≠ LAST_ROUND - 1 then
        //Bounding and pruning
        AS_{ΔX^{r+1}} ← W_{nibble}(ΔX^{r+1})
        if AS_{ΔX^{r+1}} ≤ AS_{BOUND} then
            //Est. Pr(ΔX^{r+1})
            p_{ΔX^{r+1}} ← (P_{AS})^{AS_{ΔX^{r+1}}}
            //Est. Remaining Pr(ΔX)
            p_r ← (P_{AS})^{LAST_ROUND−r−2}
            p_est ← [p_{ΔX^r}, p_{ΔX^{r+1}}, p_r]
            if p_est ≥ P_{BOUND} then
                temp ← thread_ptr × |M_k|
                loc ← temp + thread_dx_num
                ΔX^{r+1}_{loc} ← X^{r+1}
                inc(thread_dx_num)
            end if
        end if
        //Invalid ΔX^{r+1} is ignored
    else
        if AS_{ΔX^{r+1}} ≤ AS_{MITM} then
            if α_SEARCH then
                atomic_add p_i to P_{MITM}
            else
                //BACKWARD_SEARCH
                P_i ← P_i + P_{MITM} × p_i
            end if
        end if
    end if
    cur_thread_id ← cur_thread_id + 1
end for
cur_thread_id ← cur_thread_id - 1
cur_iter ← cur_iter - loop_limit
//Prepare relevant information
if thread_id_default == 0 then
    global_has_operation ← has_operation
end if
if r ≠ LAST_ROUND -1 then
    sum up thread's branch as thread_bsum
    |B(ΔX_{thread^{blockIdx.x}_{theadIdx.x}})|_r ← thread_bsum
    //atomic_add the following
    |B(ΔX_{block_{blockIdx.x}})|_r ← thread_bsum
    __syncthreads()
    if threadIdx.x == 0 then
        //atomic_add the following
        |B(ΔX_{block})|_r ← |B(ΔX_{block_{blockIdx.x}})|
    end if
end if
//Prepare to advance,return, or terminate the round
is_last_r ← r == LAST_ROUND-1
cond1 ← is_last_r ∧ cur_iter == 0
cond2 ← ¬ global_has_operation
if ¬ is_last_r ∧ global_has_operation then
    //Advance a round
    iter_store_r ← cur_iter
    dx_ptr_store_r ← dx_ptr
    thread_id_store_r ← cur_thread_id
    r ← r + 1
    dx_ptr ← 0
    cur_iter ← −1   ▷ Indicate need initialization
    //Reset atomic_add value "Next" round
    |B(ΔX_{thread^{blockIdx.x}_{theadIdx.x}})|_r ← 0
    |B(ΔX_{block_{blockIdx.x}})|_r ← 0
    |B(ΔX_{block})|_r ← 0
else if cond1 ∨ cond2 then
    //Return to previous round
    while cur_iter == 0 do
        r ← r − 1
        if r < 0 then
            break
        end if
        cur_iter ← iter_store_r
        dx_ptr ← dx_ptr_store_r
        cur_thread_id ← thread_id_store_r
        //Reset atomic_add value "Next" round
        |B(ΔX_{thread^{blockIdx.x}_{theadIdx.x}})|_r ← 0
        |B(ΔX_{block_{blockIdx.x}})|_r ← 0
        |B(ΔX_{block})|_r ← 0
    end while
else
    //Repeat last round if cur_iter ≠ 0
    cur_thread_id ← cur_thread_id + 1
end if
end while
end procedure
```