Towards a Secure Integrated Heterogeneous Platform via Cooperative CPU/GPU Encryption

Zhendong Wang*, Rujia Wang§, Zihang Jiang†, Xulong Tang‡, Shouyi Yin† and Yang Hu*
* The University of Texas at Dallas, Email:{Zhendong.Wang, Yang.Hu4}@utdallas.edu
§ Illinois Institute of Technology, Email:rwang67@iit.edu
† Tsinghua University, Email: jiangzh15@tsinghua.org.cn, yinsy@tsinghua.edu.cn
‡ University of Pittsburgh, Email: tax6@pitt.edu

2021 IEEE 30th Asian Test Symposium (ATS) | 978-1-6654-4051-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/ATS52891.2021.00032

***Abstract*—Nowadays, emerging integrated heterogeneous platforms play major roles to host autonomous systems. However, the security issue that comes with such heterogeneous architectures has not been thoroughly explored and imposes great threats and vulnerabilities to these systems. We set out to explore the security issues for the heterogeneous architectures and the corresponding mitigation mechanisms. We investigate the side-channel timing attack in a modern integrated CPU/GPU platform and propose a CPU/GPU co-encryption mechanism CoENC to mitigate the timing attack to provide a secure platform for autonomous systems. Evaluations demonstrate CoENC can effectively enhance the security 29∼44 times compared to the baseline with an extra 14%∼31% latency overhead.**

***Index Terms*—Hardware Security, Integrated GPU, Timing Channel Attack, Cooperative CPU/GPU Encryption**

I. Introduction

Autonomous systems like automotive, robotics, drones depend on multiple sensors (e.g, cameras) to collect unprecedented amounts of data, and then perform intensive computations on the data (typically as image/video) to perceive the surroundings and act smartly. Typically, these systems are deployed in emerging integrated heterogeneous CPU/GPU platforms as the platforms possess powerful computation capabilities and perform excellently in terms of size, weight, and power (SWaP). Even so, building the autonomous systems on these iGPU platforms is further stymied by the unprecedented challenges that are specifically imposed by the *stringent security requirements* [1] of the applications, and the *intrinsic hardware restrictions* (e.g., limited memory, inherent latency [2]) of the embedded heterogeneous hardware. Security has always been a top priority for these autonomous systems, as the increasing sensing interfaces on the autonomous systems also introduce increasing threats and vulnerabilities. These threats and security vulnerabilities in the iGPU platforms provide opportunities for attackers to compromise the sensitive data and cause severe accidents and significant loss for the autonomous systems.

Nowadays, cryptography applications (e.g. Advanced Encryption Standard (AES) [3]) have been explored to be applied in GPU to provide security services thanks to GPU's high throughput. However, the implementations of cryptography algorithms on GPU can suffer from the security vulnerabilities such as the imperative side-channel timing attacks [4], [5]. Typically, these attacks utilize a correlation analysis to hack the encryption key, and some randomizing mechanisms are also proposed to thwart the attacks [6], [7].

In this paper, we explore the timing channel attack issues in a modern iGPU-based platform for the autonomous systems. Our investigation shows that the notoriously hard-to-tackle timing side-channel attack can be intuitively mitigated by leveraging *a cooperative CPU/GPU encryption mechanism, CoENC*. This mechanism is motivated by the fact that iGPU platform enables an efficient data interaction between CPU and GPU cores, providing an opportunity to effectively couple CPU and GPU as a whole to develop a secure execution environment for autonomous systems. CoENC utilizes CPU to proactively revise partial plain-text data to negatively affect the timing property of the collected samples in the baseline GPU encryption mechanism, and thus fundamentally mitigates the timing attack. Besides, CoENC utilizes the unified memory [8] and CUDA streams [9] techniques in the implementation to avoid additional data copy and support concurrent encryption on both CPU and GPU. Evaluations demonstrate that CoENC can enhance the security 29∼44 times compared to the baseline with an extra 14%∼ 31% latency incurred. In summary,

- We generalize a timing attack on AES implementation in a commercial iGPU platform targeting the security of autonomous systems like automobile, drone.
- We propose a CPU/GPU co-encryption mechanism CoENC to exploit the close interactions between CPU and GPU in the iGPU platform to effectively mitigate the timing attack. Evaluations show CoENC can significantly improve security with little latency performance degradation incurred.
- We define a metric $secure_score$ to intuitively indicate the defense ability of CoENC to holistically explore the trade-off behind the defense benefits and costs.

II. Background

A. Integrated CPU-GPU Platforms and Memory Management

NVIDIA Jetson Platform. NVIDIA issued a series of Jetson iGPU platforms targeting autonomous machines and applications. The platforms can deliver powerful computation capability, support multi-types of connections, and perform well in size, weight, and power. Besides, the iGPU platform typically share a global physical memory between CPU and GPU cores.

2377-5386/21/$31.00 ©2021 IEEE
DOI 10.1109/ATS52891.2021.00032

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:46:20 UTC from IEEE Xplore. Restrictions apply.

Thus, the limited memory resources can be bottlenecks of these autonomous applications [2], [10]. For example, AGX Xavier provide 16GB memory space while the autonomous solution Autoware officially recommends 32GB RAM [11].

Unified Memory Management. Conventionally, GPU memory management follows *copy-then-execute* process [8]. That is, the host CPU has to allocate memory region on both CPU and GPU side for the input data, leading to 2× data-size memory space consumption. As iGPU platform shares the physical memory between CPU and GPU cores, such a double-size memory allocation may cause great memory pressure. Then, the unified memory (UM) [8] is introduced to ease the memory management and reduce memory consumption. In CUDA programming, the API $cudaMallocManaged()$ corresponds to the UM allocation, which creates only one memory copy of the data and allows the data to be shared between CPU and GPU cores to reduce memory consumption.

B. AES Encryption and Implementation on GPU

AES encryption is a widely used symmetric-key algorithm and typically adopts 128, 192, and 256 bits as the standard key lengths [4]. Without losing generality, we discuss the 128-bits AES encryption, which utilizes a secret key of 16 bytes and operates on 16-bytes data blocks. AES-128 encryption is composed of 10 rounds of encryptions and each round is implemented by using a 16-bytes key of this round. In the T-tables AES-128 encryption, all of these operations are achieved by performing lookups on four T-tables. Each of the tables has 256 entries and each entry has 4 bytes [12]. Note our approach also applies to other key lengths of AES encryption.

AES on GPU divides the plaintext across multiple parallel threads to improve the encryption throughput [6]. Typically, each thread encrypts one block/line of the plaintext data. A warp composed of 32 threads can implement encryptions simultaneously on 32 blocks of different data. The data in one block is mapped to a thread sequentially. If the size of the plaintext exceeds 32 blocks, the data will be divided sequentially across several warps. In practice, to speedup the process, the T-tables used in encryption are pre-computed, and then stored in the global memory first, and finally pre-loaded into the D-cache of each SM for low-latency accesses [5].

III. Baseline Timing Attack

We adopt the approach in [4] as the baseline to implement the AES timing attack on GPU, which targets the last round of encryption. The critical observation is that each table lookup index t_i in the last round can be computed from a byte of the last round key (k_j) and the corresponding byte of ciphertext (c_j), which is independent of other ciphertext bytes. Thus, the attacker is able to observe the security leakage independently at per-byte-key level. The specific process is expressed in Eq. 1. T_4^{-1} indicates the last round reverse table lookup operation, t_i is the input status of the j_{th} byte data, k_j is the j_{th} byte of the key, and c_j is the ciphertext of the j_{th} byte data.

$$t_i = (T_4^{-1}[c_j]) \oplus k_j \qquad (1)$$

Algorithm 1 Timing GPU encryption in baseline model

1: **Part1**: **with real key**
2: **Input**: plaintext, key, T_4 table
3: **Output**: ciphertext, *time_real*
4: baseAddr = threadID*16
5: **for** `j = 0:16` **do**
6: state = plaintext[baseAddr + j]
7: start_time = clock()
8: ciphertext_tmp = (T_4[state][threadID] >> shift) $\oplus$ key[j]
9: *time_real*[j] = clock() - start_time
10: ciphertext[baseAddr + j] = ciphertext_tmp
11: **end for**

12: **Part2**: **with guessed keys**
13: **Input**: T_4^{-1} table, ciphertext
14: **Output**: *time_guess*
15: baseAddr = threadID*16
16: **for** `j = 0:16` **do**
17: **for** `k = 0:256` **do**
18: tmp_state = (ciphertext[baseAddr + j]) $\oplus$ k
19: start_time = clock();
20: c_j = (T_4^{-1}[tmp_state][threadID] >> shift) $\oplus$ k
21: *time_guess*[j] = clock() - start_time
22: **end for**
23: **end for**

Attack Model: The attacker mainly collects two sets of timing samples to recover each byte of the key (i.e., total 16 bytes) of the last round [5]. One set is collected with the real key while the other set is collected with the guessed keys. In each trial sample, we launch one warp of GPU threads to encrypt 32 blocks of plaintext, where each block consists of 16 bytes data. Specifically, *Step 1: Collect real-key time samples.* During each trial, the attacker can time the latency of every table lookup operation. Thus, each block can cause 16 table lookups in last round to generate 16-byte ciphertext. After N trials, the real key set will have $16 \times N$ timing points and N samples (Algorithm 1-part1). *Step 2: Collect guessed-key time samples.* A GPU profiling kernel with 32 threads is launched to collect the time samples with the guessed keys. Since each byte of the key can have 256 different values (i.e., from 0x00 to 0xFF), these 256 possible values will be utilized to repeat the last round encryption. As a result, the guessed key set will generate $16 \times 256 \times N$ time points and N samples (Algorithm 1-part2). *Step 3: Compute correlation.* The Pearson correlation coefficient [13] is calculated between the time samples of the real keys and 256 guessed keys. The key of one byte can be successfully recovered if the guessed key value with the highest correlation value exactly matches the real one.

Observations: We collect 100 samples trails for the two time sets on Xavier AGX platform [14], and then calculate the correlation coefficients between the encryption time with the real key and the encryption time with the guessed keys. Fig. 1 plots the correlation coefficients of all 256 possible values for the first 8 key bytes. For cases of the 1_{st}, 2_{nd}, 3_{rd}, 4_{th}, and 6_{th} key byte, the guessed key value indicated by the highest correlation value exactly matches the real value of the key byte. For the other three key bytes, the highest correlation value (indicated by the red cross) does not locate the real key

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:46:20 UTC from IEEE Xplore. Restrictions apply.

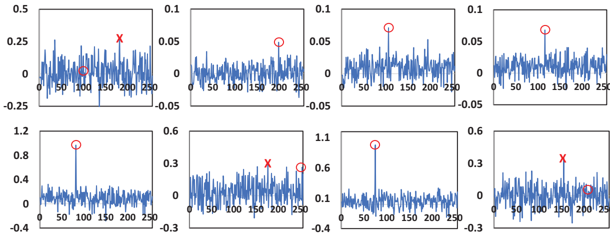

Fig. 1: Pearson's correlation coefficients to recover the first 8 bytes of the key. The x-axis indicates the 256 possible key values and the y-axis indicates the correlation value. The red circle locates the real key value and the red cross locates the guessed key value with the highest correlation coefficient.

value (indicated by the red circle) for this byte correctly. To sum up, there are 5 of 8 bytes recovered correctly. *Intuitively, this observation leads us to explore mitigating the correlation timing attack by effectively mitigating the correlation value lying in the right key value such that the attack is interfered and the correct key value can be thoroughly hidden.*

Meanwhile, we review the relationship between the success rate of recovered key bytes and the caused efforts of attacker, and then explore the implicit factors that can affect such relationship. Here, we use the required time samples to indicate the efforts, and observe that *the attacker can recover 11 bytes of the key and achieve a fair success rate of 68% by only collecting about 100 samples. With 5000 samples collected, around 13/16 can be recovered, and to recover all 16 bytes, more than 10000 samples are required*. In reality, it is an impractical job for the attacker to collect such large numbers of samples in reasonable time, as each sample requires exactly time-stamping each byte of the data within a GPU kernel. It is more common for the attacker to time-stamp the incoming and outgoing plaintext instead of within GPU kernels, and tens of thousands of sample trials may only contribute to a success rate reaching about 50% [4]. Moreover, the collected time information may also suffer from the GPU internal noise, such as the unpredictable GPU scheduling. Therefore, *if the attacker has to dedicate much more time cost to collect more time samples to reach as the same success rate as in the baseline attack model, the attacker may give up and the attack is effectively mitigated eventually.*

IV. CoENC: Cooperative CPU/GPU Encryption

In this section, we propose a cooperative CPU/GPU encryption mechanism CoENC in the integrated CPU/GPU platform to effectively mitigate the correlation timing side-channel attack. Simply put, CoENC utilizes CPU to encrypt partial plaintext data such that the entire plaintext message is encrypted by the CPU and GPU simultaneously. With CPU encryption involved, the timing property of the samples collected by the attacker will be negatively affected. As a result, the timing correlation can be reduced and the real key value may be hidden when the attacker tries to recover the key by identifying the highest correlation value. Meanwhile, CoENC adopts unified memory and CUDA streams to implement the cooperative CPU/GPU encryption smoothly without additional data copy and great performance degradation.

CoENC mechanism: essentially, the vulnerability of the table-based AES implementation on GPU is due to the indices of the table lookups operations being key-dependent, which can induce a strong correlation if the attacker exploits the encryption execution time. In baseline model, only GPU is involved in encryption. The attacker utilizes the real key and guessed keys (i.e., Algorithm 1) to collect two sets of time samples in the encryption. Then, the attacker is able to recover the key byte by byte by leveraging the Pearson correlation between the two sets of samples. Thus, if we can obfuscate the time properties of the samples collected during the GPU encryption with real keys, we can minimize the timing correlation though the attacker is still able to measure both time sets. Therefore, we propose to involve CPU in the encryption and leverage CPU to cooperate with GPU to compute the encryption together, that is CoENC mechanism. We show that CoENC can effectively mitigate and even eliminate the correlation timing attack without introducing high overheads or other vulnerabilities.

Fig. 2 shows the overall design of CoENC mechanism. Here, we use the process of attacking a plaintext message composed of 32 blocks to illustrate the mechanism. The mechanism works in a similar way for the longer messages. In Fig. 2, the rows indicate the 32 blocks/lines of the plaintext, and each column indicates one-byte data of a certain block. Differing from the baseline model where only the GPU encrypts the plaintext message with real key, in CoENC mechanism, the CPU can randomly select partial plaintext data (i.e., the 8_{th} byte in block 2, 10_{th} byte in block 3, etc.) and then encrypt this part of data by leveraging a CPU-version AES-128 function. Meanwhile, CPU pads dummy data (i.e., meaningless random values) to the address of the selected data (i.e., the 8_{th} byte, 10_{th} byte, etc.) such that the part of the pristine data is replaced and the plaintext message is revised. Then, the entire revised plaintext message, which contains some dummy data, is sent to the GPU for encryption by levering the GPU AES-128 function. In this process, the CPU and GPU encrypt their own data simultaneously (i.e., partial pristine data by CPU and the revised but complete plaintext message by GPU). After the encryption completes, the partial ciphertext data is returned from CPU (i.e., the encrypted result of the 8_{th} byte, 10_{th} byte, etc.) and the ciphertext message of the entire revised plaintext message is returned from GPU. Then, the CPU-returned ciphertext data is padded back to the ciphertext message returned by the GPU accordingly. Therefore, the attacker cannot know which bytes of the plaintext have been revised into dummy bytes. The only known public information is the number of bytes that CPU now is handling.

Meanwhile, we characterize the latency of GPU encrypting one-byte data and CPU encrypting one-byte data on NVIDIA Xavier AGX, respectively, and observe that GPU takes more than 10 cycles while CPU only takes a few cycles. Therefore, great chances are that GPU encryption can well overlap the

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:46:20 UTC from IEEE Xplore. Restrictions apply.

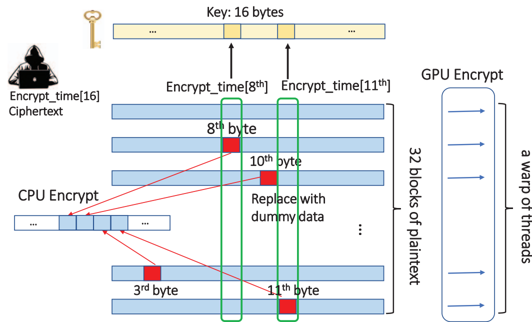

Fig. 2: CoENC, CPU-GPU co-encryption mechanism. The 32 blue strips indicate 32 lines of the plaintext data, and the red blocks indicate the bytes selected by CPU to perform the encryption. CPU replaces partial data with dummy data and then GPU encrypts the entire revised plaintext message. The attacker aims at collecting encryption_time[16] and ciphertext to recover the key.

CPU encryption without an extra performance loss. Differing from baseline model, CoENC replaces some data of the plaintext with dummy data, the encryption time on GPU part will thus vary from the case of encryption using the pristine plaintext. More importantly, CoENC can adjust the ratio of the selected data and allow CPU to randomly select partial data from the entire plaintext in each trial. Thus, the replaced data changes in each trial and multiple collected time samples can be negatively affected. Although the attacker can realize that the CPU encryption is involved in CoENC, the attacker cannot identify which part of data is encrypted on the CPU.

Besides, unified memory (UM) and CUDA streams [8], [9] are applied to avoid additional data copy and achieve CoENC in the commercial iGPU platform smoothly. Under UM, only one plaintext copy is stored in memory space and is shared between CPU and GPU. Thus, after the plaintext is sent to the memory space, CPU can directly reserve partial data, revise the data, and perform CPU encryption, while GPU can encrypt the entire revised plaintext in place (i.e., original data address) without necessarily operating on a second copy of the plaintext. Also, with CUDA streams support, CPU and GPU can perform its own encryption operation concurrently without the need of serializing their operations. Both facts can significantly benefit CoENC mechanism. Once encryptions are completed, both CPU and GPU can directly return the encrypted data back. All operations are implemented on one plaintext copy and there is no need to allocate a second plaintext copy and eliminate the data copy pressure in the memory-limited iGPU platform. Meanwhile, with CUDA streams supporting, the concurrent CPU-GPU co-encryption can be smoothly achieved. Algorithm 2 shows the process of CoENC mechanism. Line 3 allocates the UM space for the plaintext data and line 4 creates stream for the following GPU encryption. Line 5-7 indicate the CPU randomly reserve partial data and replace the data with dummy data. Line 8-9 indicate the concurrent CPU and GPU encryption streams. Line 11-12 indicate the partial ciphertext is padded back the entire ciphertext message to obtain the correct ciphertext and the encryption time of the 16 bytes of the key is returned. There is no need of allocating data copy by calling *cudaMalloc()* and copying data between host and device by calling *cudaMemcpy()* at all.

Algorithm 2 CoENC, CPU-GPU co-encryption with real key

```
1:  Input: plaintext
2:  Output: ciphertext, encryption_time[16]
3:  cudaMallocManaged(&plaintext, size)
4:  cudaStreamCreate(&s1)
5:  selected_index = rand()/512;
6:  partial_plaintext -> [CPU_data]
7:  dummy data -> partial_plaintext -> plaintext_revised
8:  AES_CPU([CPU_data], partial_ciphertext, ...)
9:  AES_GPU<<<Grid, Block, s1>>>(
    plaintext_revised, encryption_time[16], ciphertext_revised, ...)
10: cudaStreamSynchronize()
11: partial_ciphertext -> ciphertext_revised
12: return ciphertext, encryption_time[16]
```

We have shown that CoENC can minimize or eliminate the timing correlation attack on AES key in the iGPU platform. We consider the CPU side to be benign and it does not introduce additional secure vulnerabilities, as we can take the Jetson Trusty technology [15]. The iGPU platform is different from the conventional dGPU platform which connects the CPU DRAM and GPU DRAM via PCIe bus. This way, the iGPU platform avoids the typical PCIe bus snooping. Also, the data encrypted by CPU is randomly selected in each trial, and the attacker hardly identify what data is encrypted on CPU. Thus, the memory access of CPU encrypting this partial data is irregular and randomized. Meanwhile, the CPU encryption can be hidden by the GPU encryption part. It is non-trivial for the attacker to obtain the specific timing information of CPU encryption and to issue another timing attack on CPU side merely by following the irregular memory accesses. Besides, the concurrent CPU and GPU encryption may cause certain resources competitions and extra noises, further obfuscating the attacker, though some performance overhead may be incurred.

V. Evaluation

A. Experiment Settings and Metrics

We adopt NVIDIA Xavier AGX platform [14], which installs L4T 32.3.1 package. Xavier AGX has an 8-core ARM CPU and an 512-core Volta GPU, and 16GBs memory which is physically shared by CPU and GPU.

We utilize the attack *success rate, encryption execution latency, and the secure_score* to evaluate our design. The success rate is defined as how many bytes from the entire 16 key bytes can be successfully recovered by the attacker. On one hand, we test (1) the number of the recovered key bytes (i.e., from 0 to 16) for a given number of samples and (2) the number of samples required for the attack to recover a certain number of key bytes. As we stated in Sec. III, either the success rate is reduced or the attacker has to spend more time cost on collecting samples to reach a certain success rate, the

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:46:20 UTC from IEEE Xplore. Restrictions apply.

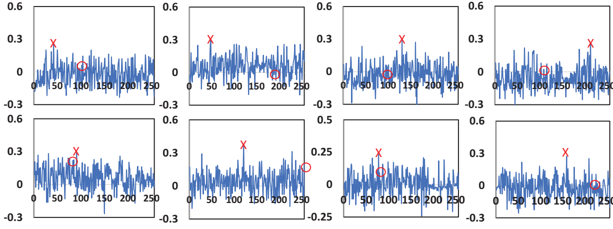

Fig. 3: Pearson's correlation coefficients to recover the first 8 bytes from CoENC mechanism. The x-axis indicates the 256 possible key values and the y-axis indicates the correlation value. The CPU ratio = 25% and sample number = 100.

attack is effectively mitigated. The *secure_score* is defined to intuitively measure the defense capability of CoENC as in Eq. 2. A higher score indicates a more secure model. Considering CPU encryption may affect the encryption performance, we holistically consider the execution latency and *secure_score* to explore the trade-off behind the defense benefits and costs.

$$Secure_Score = \frac{\text{Encryption Time} \times \text{Number of Samples}}{\text{Number of Recovered Keys} \times \text{Average Correlation}} \tag{2}$$

B. Evaluation Results

Attack success rate with various CPU encryption ratio. We adopt the regular timing attack in baseline model of Sec. III to analyze the attack success rate in CoENC mechanism. We set the ratio of CPU encryption 25% (i.e., CPU randomly select 25% of plaintext data to implement encryption) and collect 100 samples to test how many bytes of the key can be recovered by the attack. Fig. 3 plots the Pearson correlation coefficients of the first 8 bytes of the key. Compared to the correlation coefficients in Fig. 1, we observe that the correlation coefficient lying in the correct key value (indicated by the red circle) of the 5 bytes (i.e., the 1_{st}, 2_{nd}, 3_{rd}, 4_{th}, and the 6_{th}), which are successfully recovered in baseline model, is effectively reduced now. As a result, the attack fails to recover all the first 8 bytes merely by identifying the highest correlation coefficient (indicated by the red cross) in CoENC. The success rate for the 8 bytes is 0 now.

Then, we vary the ratio of CPU encryption from 0% to 100% and collect 100 samples to analyze how the ratio of CPU encryption impacts the attack success rate. Specifically, if the ratio equals 0 (i.e., the baseline), 11 of 16 key bytes are recovered and the success rate is 68%. With CPU ratio increasing, the success rate decreases significantly (ratio = 12.5% : 56%, ratio = 25% : 43%, ratio = 50% : 18% , ratio = 75% : 12% , ratio = 100% : 0%). Thus, CoENC mechanism can mitigate the attack well. As we stated in the Sec. IV, the CPU pads the dummy data to the pristine plaintext message, and then the GPU encrypts the revised plaintext data. Due to the dummy data being introduced, the time information collected by the attacker will be a variance from the time information of encryption with the correct data. CPU randomly selects the pristine data and replaces them with dummy data each time the attacker collects the time information using the real key for a key byte, thus, multiple samples are affected and

TABLE I: The number of recovered key bytes with different numbers of samples and different ratios of CPU encryption.

sample#	100	5000	∼10000
Ratio% = 0	11	13∼14	15∼16
Ratio% = 12.5	8∼9	11∼12	∼ 14
Ratio% = 25	6∼7	10∼11	12∼13
Ratio% = 50	3∼4	5∼7	8∼10
Ratio% = 75	∼2	∼3	∼5
Ratio% = 100	0	1∼2	2∼3

the calculated correlation coefficients are negatively impacted. If the attacker cannot construct a strong correlation between the collected time samples, the attack cannot recover a key byte correctly.

Attack success rate with increasing sample counts. With CoENC applied, we analyze the attack success rate when the attacker collects different numbers of samples, as shown in Table I. Generally, with more samples collected, more key bytes can be recovered. Meanwhile, we observe that under CoENC mechanism, the number of recovered key bytes is obviously reduced and the attack success rate decreases compared to the baseline. That is, to reach the competitive success rate to the baseline (i.e., 0% ratio), the attacker has to spend more time cost on collecting time samples. For example, in the baseline attack, the attacker collects 100 samples to recover 11 bytes. However, 50× more samples have to be collected in the case of CPU ratio equaling 25%, and 1000× more samples have to be collected in the case of CPU ratio equaling 50%. If the ratio is larger than 50%, only a few key bytes can be recovered though more than 10000 samples are collected, and the success rate is about 10%. In practice, the attacker may give up if thousands of hundreds of time cost has to be dedicated.

Performance and defense capability of CoENC mechanism. We evaluate execution latency and secure score of CoENC mechanism to holistically evaluate the defense benefits and costs, as shown in Fig. 4. The 0% ratio indicates the baseline model where only GPU encrypts the entire plaintext. Basically, the latency of the CPU encryption increases with its ratio increasing due to the fact that CPU encrypts an increasing amount of data. In comparison, the GPU always encrypts the entire (revised) plaintext message, the GPU encryption almost causes a constant latency though the different CPU ratio indicates a different ratio of pristine data is revised.

Meanwhile, we observe that if the CPU ratio is less than 50%, the CPU encryption causes much less latency than the GPU encryption. The GPU encryption dominates the co-encryption performance in CoENC mechanism and its latency can thoroughly hide the CPU encryption latency as well. CoENC possesses a negligible latency compared to the baseline. However, if the CPU ratio reaches between 50% and 75%, the CPU encryption causes a larger latency and CoENC mechanism also suffers from some performance degradation, e.g., an extra 14% latency for the ratio = 50% and an extra 31% latency for the ratio = 75% compared to the baseline. This may be caused by resources competition, data interaction, etc. If the CPU ratio reaches 100%, the CPU encryption causes a

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:46:20 UTC from IEEE Xplore. Restrictions apply.

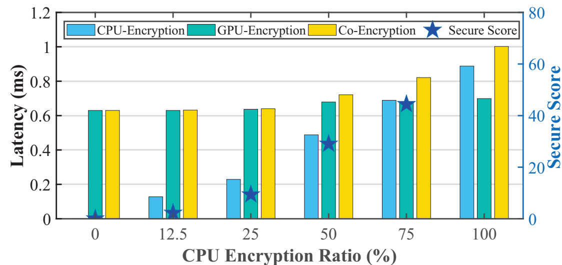

Fig. 4: Execution latency of CoENC mechanism per plaintext sample with CPU encryption ratio increasing. The x-axis indicates the CPU encryption ratio, the y-axis-left indicates the latency to encrypt the sample data, while the y-axis-right indicates the secure score. CPU-Encryption, GPU-Encryption and Co-Encryption indicate the latency of CPU encrypting the ratio of data, GPU encrypting the entire revised plaintext, and CoENC co-encrypting their own data simultaneously.

larger latency than the GPU encryption, and the latency caused by the CPU-GPU co-encryption is 59% larger than baseline. This is because if the CPU causes a competitive/larger latency, the GPU encryption cannot well overlap CPU encryption to hide the latency in CoENC mechanism.

Regarding the secure score, we observe that the score increases quickly with the increasing CPU encryption ratio, indicating that the defense capability of CoENC mechanism significantly increases. Specifically, if the ratio increases from 0% to 50%, the score increases about 29× compared to the baseline; if the ratio reaches 75%, the score increased about 44× compared to the baseline. With CPU ratio increasing, although the encryption time increases a little, the number of the recovered key bytes and average correlation coefficient both decrease significantly. Thus, CoENC provides a much more secure system when the encryption is cooperatively done by the CPU and GPU. Theoretically, if the ratio reaches 100%, the score is approaching ∞ (i.e., out of scope of the figure). That's an ideal case providing unbreakable protection to the key. In this case, all valid encryption is done by the CPU while the GPU only computes dummy data encryption, which is an unrealistic setting if we consider the defense costs. To summarize, in our experiments, we find that 50% or 75% CPU encryption ratio could result in significant defense capability enhancement (29-44×) while only introducing 14-31% performance overhead in CoENC mechanism.

VI. Related Work

Recently, several works demonstrate the AES implementation on GPUs is vulnerable to timing channel attacks due to the GPU's memory coalescing logic, and some mechanisms are developed to mitigate the attacks accordingly. [16] demonstrates that the memory leaks are possible at various levels of GPU memory hierarchy. Jiang et al. demonstrate a complete AES key recovery timing attack on discrete GPU platforms [4], and then exploit a fine-grained timing channel by leveraging bank conflicts in the GPU's shared memory [17]. Accordingly. Kadam et al. propose RCol and BCol to mitigate the GPU timing attack by randomizing the coalescing mechanisms [6], [7]. Lin et al. develops side-channel-resistant AES on GPU by reorganizing the T4 table in the cache/shared memory [5]. However, these works mainly utilize conventional discrete GPU platforms and do not consider the new settings of the emerging iGPU platforms for autonomous applications, which enable closer CPU/GPU interactions and provide opportunities to mitigate the timing attacks. Also, it is meaningful and challenging to deploy the CPU/GPU co-encryption in iGPU platforms.

VII. Conclusion

The autonomous systems, such as automobile systems, drones, are widely deployed in integrated CPU/GPU (iGPU) platforms and typically emphasize security as the priority. However, such cryptography service as AES implementation in the platforms suffers from the timing attack. We deep investigate the timing attack on the AES implementation and propose a CoENC mechanism to develop a secure execution environment in commercial iGPU platforms. Evaluations demonstrate that CoENC mechanism can effectively mitigate the timing attack without causing high performance degradation.

References

[1] Future of cyber security for connected and autonomous vehicles, 2020.
[2] Zhendong Wang, Zhen Wang, Cong Liu, and Yang Hu. Understanding and tackling the hidden memory latency for edge-based heterogeneous platform. In 3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20), 2020.
[3] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the advanced encryption standard (aes). Journal of Research of the National Institute of Standards and Technology, 106(3):511, 2001.
[4] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a gpu. In 2016 IEEE International symposium on high performance computer architecture (HPCA), pages 394–405. IEEE, 2016.
[5] Zhen Lin, Utkarsh Mathur, and Huiyang Zhou. Scatter-and-gather revisited: High-performance side-channel-resistant aes on gpus. In Proceedings of the 12th Workshop on General Purpose Processing Using GPUs, pages 2–11, 2019.
[6] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. Rcoal: Mitigating gpu timing attack via subwarp-based randomized coalescing techniques. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 156–167. IEEE, 2018.
[7] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. Bcoal: Bucketing-based memory coalescing for efficient and secure gpus. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 570–581. IEEE, 2020.
[8] Mark Harris. Unified memory for cuda beginners, 2017.
[9] Nvidia. Gpu pro tip: Cuda 7 streams simplify concurrency, 2015.
[10] Soroush Bateni, Zhendong Wang, Yuankun Zhu, Yang Hu, and Cong Liu. Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. In 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 310–323. IEEE, 2020.
[11] Autoware: Open-source software for urban autonomous driving.
[12] OpenSSL. https://www.openssl.org/.
[13] Karl Pearson. Note on regression and inheritance in the case of two parents. Proceedings of the Royal Society of London, 58:240–242, 1895.
[14] Nvidia. Jetson agx xavier developer kit.
[15] Trusty, a trusted execution environment.
[16] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. Cuda leaks: a detailed hack for cuda and a (partial) fix. ACM Transactions on Embedded Computing Systems (TECS), 15(1):1–25, 2016.
[17] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A novel side-channel timing attack on gpus. In Proceedings of the on Great Lakes Symposium on VLSI 2017, pages 167–172, 2017.

Authorized licensed use limited to: University of Pittsburgh. Downloaded on May 14,2026 at 05:46:20 UTC from IEEE Xplore. Restrictions apply.