2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

Owl: Differential-based Side-Channel Leakage Detection for CUDA Applications

Yu Zhao*, Wenjie Xue*, Weijie Chen*, Weizhong Qiang*‡, Deqing Zou*‡, Hai Jin†
* National Engineering Research Center for Big Data Technology and System
Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security
School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China
† National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology Wuhan, 430074, China
‡ Jinyinhu Laboratory, Wuhan, 430074, China
Email: {z_y, xuewenjie2021, weijie_chen, wzqiang, deqingzou, hjin}@hust.edu.cn

***Abstract*—Over the past decade, various methods for detecting side-channel leakage have been proposed and proven to be effective against CPU side-channel attacks. These methods are valuable in assisting developers to identify and patch side-channel vulnerabilities. Nevertheless, recent research has revealed the feasibility of exploiting side-channel vulnerabilities to steal sensitive information from GPU applications, which are beyond the reach of previous side-channel detection methods.**

Therefore, in this paper, we conduct an in-depth examination of various GPU features and present Owl, a novel side-channel detection tool targeting CUDA applications on NVIDIA GPUs. Owl is designed to detect and locate side-channel leakage in various types of CUDA applications. When tracking the execution of CUDA applications, we design a hierarchical tracing scheme and extend the A-DCFG (Attributed Dynamic Control Flow Graph) to address the massively parallel execution in CUDA, ensuring Owl's detection scalability. After completing the initial assessment and filtering, we conduct statistical tests on the differences in program traces to determine whether they are indeed caused by input variations, subsequently facilitating the positioning of side-channel leaks. We evaluate Owl's capability to detect side-channel leaks by testing it on Libgpucrypto, PyTorch, and nvJPEG. Meanwhile, we verify that our solution effectively handles a large number of threads. Owl has successfully identified hundreds of leaks within these applications. To the best of our knowledge, we are the first to implement side-channel leakage detection for general CUDA applications.

***Index Terms*—side-channel detection, GPU, CUDA applications**

I. Introduction

GPU has become an important part of modern computer architecture. The high parallel computing capability of GPUs has enabled an increasing number of compute-intensive applications to be deployed on them using computing platforms such as CUDA (Compute Unified Device Architecture) [1] and OpenCL [2]. Moreover, with the rise of AI applications such as ChatGPT [3] and Stable Diffusion [4], an increasing amount of private data is being processed on GPUs.

Corresponding Author: Weizhong Qiang. This work was supported in part by National Key Research and Development Program of China (Grant No. 2022YFB4501500 and 2022YFB4501502) and National Natural Science Foundation of China (Grant No. 62272181).

However, in recent years, researchers have discovered that GPUs are also vulnerable to side-channel attacks. Some studies have successfully applied existing side-channel attack methods of CPUs to GPUs, like Prime-Probe [5], while others have created new attack surfaces utilizing specific GPU mechanisms. For instance, an attacker can exploit the memory coalescing mechanism of NVIDIA GPUs to launch side-channel attacks and steal cryptographic keys from AES operations [6].

Several hardware-based strategies have been proposed to mitigate microarchitectural side-channel leakage [7]–[11]. However, their applicability is limited by the requirement for hardware modifications. More prevalent approaches are at the software level, primarily focused on eliminating microarchitectural traces associated with sensitive information in programs [12], [13].

Identifying and locating potential side-channel leakages in applications have sparked heated research in academia, which is a crucial step in patching vulnerabilities [14]–[17]. Unfortunately, existing side-channel leakage detection tools solely focus on CPU applications and have yet to cover GPU applications. Furthermore, we argue that the proposed methods cannot be effectively applied to detecting side-channel leakage in GPU applications.

On the one hand, current static-analysis-based methods typically model information flow [18]–[20], but GPU programs have different code patterns that pose unique challenges for such solutions. Symbolic-execution-based methods [21], [22] are also less effective in analyzing large-scale programs. On the other hand, dynamic analysis-based methods [14], [16], [17], [23] cannot capture the execution trace in GPUs and cannot cope with huge amount of GPU threads. For example, DATA [14], [24] claims to support multi-threading trace tracking. Specifically, it directly records the trace of each thread and performs differential analysis. However, the memory consumption increases proportionally with the number of threads, hindering its adoption towards thread-intensive CUDA applications.

Therefore, we present Owl, a side-channel leakage detection tool for CUDA applications. Overall, Owl is able to locate

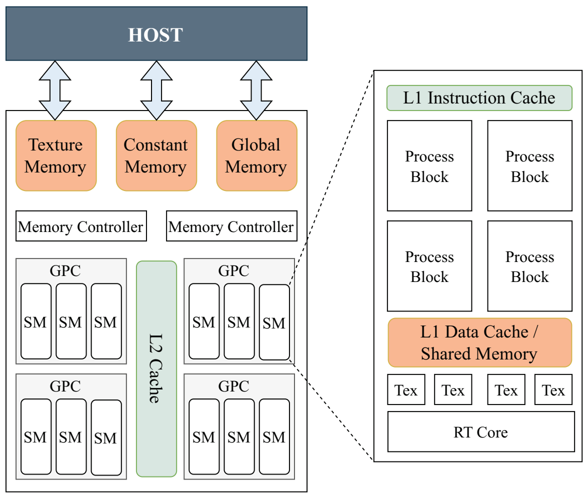


Fig. 1: NVIDIA Ampere architecture.

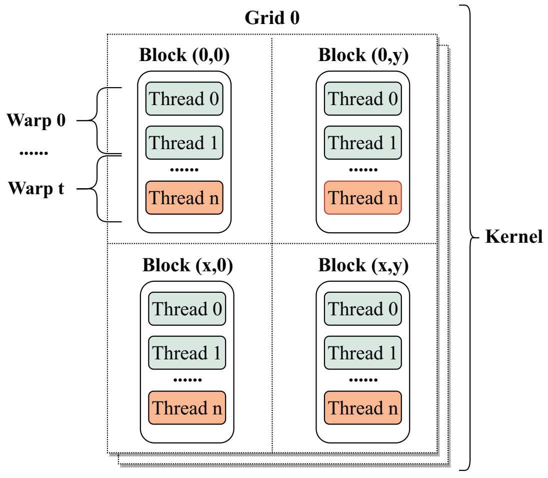


Fig. 2: Threads organization in a CUDA kernel.

GPU side-channel leakages on numerous types of CUDA applications, including cryptographic applications, deep learning, multimedia processing, etc. Owl records the basic blocks and memory addresses accessed from all threads of an executing GPU application. Then, Owl constructs A-DCFGs from the program traces and leverages the generated A-DCFGs for differential detection and leakage analysis.

In summary, the contributions of this paper are as follows:

- We present Owl [1], a side-channel leakage detection tool for CUDA applications that analyzes and locates side-channel leakage in various CUDA applications.
- We devise a method for GPU program trace collection, capturing the control flow information and memory accesses of each GPU thread within a CUDA application and integrating them into a single A-DCFG.
- We introduce a novel side-channel leakage testing method based on A-DCFG, which identifies basic blocks with control flow leaks and instructions with data flow leaks.

This paper is organized as follows: Section II explains necessary background information. Section III discusses related works on GPU side-channel attacks, as well as side-channel detection on both CPU and GPU. Section IV discusses side-channel leakages in GPUs, our threat model, and provides an overview of Owl. Section V,VI,VII present the detailed designs of Owl's key phases. Section VIII evaluates Owl and analyzes the experimental results. Section IX discusses possible side-channel leakage countermeasures. Section X concludes this paper.

II. Background

A. NVIDIA GPU

NVIDIA GPUs are widely recognized and extensively used, boasting a well-established ecosystem. In the Ampere architecture [25], as shown in Fig. 1, GPUs typically consist of three key memory components: global memory, constant memory, and texture memory. Global memory serves as a shared storage space accessible to all threads in a kernel. While sharing similar functionality, constant memory is limited to read-only access. Texture memory, on the other hand, is dedicated to storing and loading 2D and 3D image data, mainly used for texture mapping in graphics rendering.

The Ampere architecture incorporates numerous graphics processing clusters (GPCs), which encompass texture units, raster units, and streaming multiprocessors (SMs). The SMs are vital organizational units within GPUs, where four process blocks are responsible for performing core computational operations. The process blocks within an SM share an L1 instruction cache and an L1 data cache, while the L2 cache is shared among all SMs.

B. CUDA

NVIDIA GPU's universal architecture adopts the CUDA [1] programming model, which distributes tasks to multiple threads for parallel execution to make full use of the parallel processing capability of GPUs. For functions that utilize CUDA for accelerated parallel execution (i.e. kernel functions), additional thread organization is required to achieve optimal computation efficiency. A kernel function, simply called a kernel, is a GPU function invoked from the CPU code.

The organizing of threads in CUDA is depicted in Fig. 2. In the CUDA programming model, a thread serves as the smallest execution unit, and each thread has its own thread ID and execution context, and can access resources such as global memory, shared memory, and registers. Multiple threads collectively form a block with shared memory, enabling thread cooperation and synchronization. For larger-scale parallel computations, blocks can be further organized into a grid following a two-dimensional or three-dimensional pattern. To schedule the thread's execution, every 32 threads are grouped into a warp, where threads within a warp execute

[1]https://github.com/OwlCudaSCDetector/Owl.git

the same instructions but operate on different data (i.e. SIMD, Single Instruction, Multiple Data). A cooperative thread array (CTA) is an alias for a block, and the SM dynamically assigns warps to different CTAs to maximize hardware resource usage and improve parallel efficiency.

In CUDA programming, the device code executes on the GPU device, specifically designed for parallel computing tasks within CUDA kernels. The host code runs on the host CPU and is responsible for controlling the entire system and launching the device code. Both host and device code perform data transfer and synchronization operations to ensure the stability and efficiency of the entire system's operation.

Unlike in the host, when executing conditional branches in the device, CUDA does not use jump operations but instead employs a method known as *predicated execution* to control thread branch execution. Specifically, each thread in a warp utilizes a bit to indicate whether the thread is in an active state. Only threads that satisfy the condition execute the corresponding basic block of the conditional branch.

III. Related Works

A. Side-Channel Attacks on GPU

Modern web browsers usually require hardware acceleration for certain computing tasks, such as leveraging GPUs to accelerate webpage rendering. Existing studies have demonstrated recovering viewed webpages by monitoring memory usage and access patterns [26]–[28]. Timing side-channel attacks have also been proven feasible on cryptography applications like AES and can be leveraged to recover cryptographic keys [6], [29]–[33]. Similar to CPU counterparts, unsafe implementations of RSA's modular exponentiation algorithm can lead to the leakage of private keys [34], [35]. Additionally, GPU side-channel attacks have expanded to deep learning models, stealing their network structure and hyperparameters. Side-channel leakages of these models are then utilized for model extraction attacks (MEA). Prior side-channel attack techniques typically involve contention of GPU resources [26], [36], [37] such as PCIe detection [38]–[41] and PCIe congestion [42], [43]. Other works have targeted GPUs in smartphones [44], AR/VR systems [45] to infer user's inputs, gestures, and other sensitive information.

B. Side-Channel Leakage Detection

To our knowledge, a qualified side-channel detection tool for CUDA applications should meet the following four requirements: ❶ Binary analysis; ❷ Diverse targets; ❸ Accurate leakage positioning; and ❹ Scalability. The rest of this section explains these requirements in detail and whether existing works have addressed them.

❶ **Binary Analysis**. Given that side-channel vulnerabilities in programs are often intricately linked to microarchitecture, binary analysis can offer certain advantages to their detection. On the one hand, binary analysis is more accurate for analyzing assembly code and can be applied to closed-source code. On the other hand, the compilation and optimization process can introduce side-channel leakages that do not initially exist in the source code [50], [51]. Therefore, a majority of works choose binary analysis [14]–[16], [52]–[55], while only a few focus on detecting vulnerabilities in scripting languages [48]. Some studies examine the intermediate representation (IR) of programs that facilitates cache analysis [22], [47], [56], but similar to source code analysis, they also face challenges in program transformation resulting from compilation and optimization.

❷ **Diverse Targets**. The scope of sensitive information that needs protection extends beyond cryptographic keys. For instance, image processing applications that handle sensitive user data [49], including medical image processing, as well as the layered architecture and hyperparameters of deep learning models are also critical assets that require protection. Quite a few studies have primarily focused on detecting potential key leakage in cryptography applications [15], [57]–[63]. Some works have analyzed deep learning models and utilized side-channel detection tools to identify adversarial inputs for classification networks [21]. Additionally, a recent study Manifold-SCA [49] has highlighted the significance of media data, such as text, images, and videos in side-channel analysis.

❸ **Accurate Leakage Positioning**. Leakage positioning is essential for fixing side-channel vulnerabilities. Unfortunately, most static detection tools rely on cache models to calculate information leakage and cannot achieve precise positioning [20], [22], [52], [54]. In contrast, dynamic detection tools are relatively more suitable for vulnerability positioning. It is worth noting that excluding differences caused by non-deterministic factors is crucial for accurate side-channel analysis [14], [17], [49]. Specifically, relying solely on deterministic observations can generate false positives attributed to non-deterministic factors (e.g. oblivious RAM [64]) in the secret inputs.

❹ **Scalability**. Side-channel detection tools should be highly scalable, making them available to practical applications of various sizes and areas. Methods with high technical limitations, such as abstract interpretation [19], [20], and symbolic execution [15], [21], [52], [55], [59], face great challenges in analyzing huge programs. Others are restricted to regular single-threaded programs and do not consider complex multi-threaded programs [14], [16], [17]. Although DATA [14], [24] supports multi-threaded programs, it only records and conducts differential analysis on the trace of individual threads, thus limited to programs with few threads.

In addition, we assess the feasibility of existing works on CUDA applications. As for ❶, binary analysis strategies for CPU applications cannot be directly adopted for CUDA applications due to architectural differences, and the accuracy of source-code-based and IR-based solutions declines evidently [22], [46], [48]. As for ❷, cryptographic algorithms are not the primary concerns on GPUs; instead, targets such as multimedia processing and deep learning models require more security attention. A majority of previous approaches [14]–[16], [20], [22], [46]–[48] are confined to certain application, hindering their wider adoption. As for ❸, although dynamic detection approaches [16], [21], [47]–[49] perform much better at locating program leaks than static ones [20], [22], [46], they

TABLE I: Existing side-channel leakage detection works*.

	Blazer [46]	**CaSym** [22]	**CacheD** [15]	**DATA** [14]	**CANAL** [47]	**HyDiff** [21]	**MicroWalk** [16]	**Microwalk-CI** [48]	**Manifold-SCA** [49]	**CacheQL** [17]	**Owl**
❶	○	◑	●	●	◑	●	●	○	●	●	●
❷	○	○	○	○	○	◑	○	○	◑	◑	●
❸	○	○	◑	●	◑	◑	◑	◑	◑	●	●
❹	◑	○	○	◑	○	○	○	○	○	○	●

*○, ◑, ● denote unable to support, partially support, and fully support.

often fail to consider the impact of non-deterministic factors on the detection results. As for ❹, almost all existing methods lack scalability for parallel programs. Although a subset of static detection works do not consider program execution and may be available to CUDA applications, varying CUDA code patterns can hamper their effectiveness. Some dynamic detection tools [14], [24] consider multi-threaded programs yet with only a few threads. As a result, we claim that existing works do not satisfy the aforementioned requirements and are inadequate for CUDA applications, thus novel side-channel leakage detection tools for this scenario are urgently needed.

IV. Overview

A. *Side-Channel Leakage on CUDA*

CUDA applications are typically designed to execute in parallel in order to fully leverage the capabilities of GPUs. These applications often utilize a significantly larger number of threads compared to CPUs. However, the increasing number of threads may bring significant volatility to side-channel analysis. More specifically, due to the parallel execution in GPUs, the secret input may be divided into segments and processed in multiple threads. As a result, if attackers only obtain leakages from part of the threads, they cannot fully reconstruct the complete secret input. Additionally, attackers cannot determine whether the differences in control/data flow on a particular thread are brought by the inputs or the observations from different threads, as each segment may be different.

Another difference comes from the software architecture of CUDA application. The entire CUDA application consists of host code and device code, and they run on CPU and GPU respectively. The device code is wrapped into a kernel and called in host code as a function with arguments. During execution on the GPU, calling different kernel functions or passing different arguments leads to variations in the control/data flow of the host code. This implies that the leakage in CUDA programs is intricate, not only bound to the device code but also potentially extends to the related host code.

Control and data flow leakage are two main types of side-channel leakage. However, we believe that the two types should be reconsidered for CUDA applications, as the impact of the host code and device code on the GPU differs. Next, we further categorize control flow leakage and data flow leakage.

Host control flow leakage and **host data flow leakage** occur on the host code. The presence of input-dependent control or data flow in the program introduces variations in program behavior, resulting in different microarchitectural states on the CPU and forming a side-channel leakage. For example, execution of different branches in the *if-else* statement may lead to host control flow leakage, and accessing arrays may lead to host data flow leakage. The above two types of leakage have been extensively studied and discussed in academia, targeting which a plethora of detection methods have been proposed.

Kernel leakage also occurs on host code, but its impact extends to GPUs. This type of leakage often arises when the host code, during GPU-related operations, exhibits input dependencies, such as invoking different types or quantities of kernels under certain conditions. For instance, it may invoke a kernel within an *if* statement while invoking another kernel (or not invoking any) within an *else* statement. Compared to host leakage, kernel leakage propagates differences in the control and data flow of the host code to the microarchitecture of GPUs, which can be observed by GPU attackers. Generally, differences between kernels are relatively distinguishable to the attacker. Therefore, an attacker can easily obtain information about which and how many kernels are being executed in the program. In most cases, kernel leakage is coarse-grained so an attacker can only obtain a rough understanding of the victim's control flow. However, under certain settings, some sensitive information such as hyperparameters of DNN models is still susceptible to leakage.

In contrast, **device control flow leakage** and **device data flow leakage** occur on device code and do not influence CPU states. Similar to the host leakage, device leakage is derived from input-dependent control and data flow in the device code. The use of *if-else* statements, *for* loops, and accessing arrays may also introduce device leakage. However, due to several special mechanisms (e.g., CUDA predicated execution), there are differences between CPUs and GPUs in their leakage formation.

In this paper, we focus on kernel leakage, device control flow leakage, and device data flow leakage, due to their direct relation to GPUs. Nevertheless, the hardware architecture of GPUs and the software architecture of CUDA bring several new challenges to detecting the three types of leakage.

An initial challenge is the large number of threads in the device code that significantly increases the difficulty in tracking and recording execution traces. For individual threads executing in the GPU, we need to track and record their execution contents. However, when the number of threads increases, so does the amount of data that needs to be recorded. In extreme cases, if a CUDA application fully occupies the GPU, it can spawn millions of threads simultaneously, exploding the amount of information that needs to be recorded.

Another challenge caused by massive threads is trace analysis. As previously mentioned, the differences observed in the execution of individual threads do not necessarily indicate the presence of side-channel leakage. However, current side-channel leakage analysis mainly addresses single-threaded programs, and can not work effectively on CUDA applications. Take DATA for instance, it makes differential analysis on all traces in multi-threaded programs, implying that, to ensure accuracy, it must correctly identify thread traces with the same context from two sets of traces. However, as CUDA applications are often featured with massive threads, considerably high analysis overhead (n times differential analysis for n threads) makes it a daunting task for solutions like DATA.

The software architecture of CUDA applications also poses challenges to side-channel leakage detection. Although the device code alone executes in the GPU, its invocation and arguments are controlled by the host code. Therefore, execution details of the host code should also be covered, particularly when it comes to kernel leakage, which is directly caused by variances in control and data flow within the host code. Considering the above requirements, we need to implement cross-hardware trace tracking for CUDA applications to accurately record the variations in program control flow across CPU and GPU.

B. Threat Model

In order to cover as many side-channel leaks as possible, we consider a powerful side-channel attacker [14]. We assume that the attacker can observe accurate, fine-grained, and noise-free information targeting microarchitectural components within the GPU (e.g., network-on-chip [65]) as well as the memory hierarchy (e.g., caches [5]). Consequently, the attacker can construct complete runtime traces of the program (i.e., sequences of instructions, basic blocks, accessed memory addresses) based on the observations. We assume that the attacker can conduct offline analysis to establish the mapping between the secret inputs and the corresponding execution traces, thus the former can be recovered by observing the latter. Furthermore, we consider the attacker can disable or bypass the address space layout randomization (ASLR) of NVIDIA GPUs.

Out of Scope. We only concentrate on side-channel leaks on the GPU, thus we do not consider side-channel attacks targeting data transmission channels (e.g., PCIe).

C. The Design of Owl

Owl is a differential-based side-channel leakage detection tool targeting CUDA applications. Owl's insight lies in its ability to overcome the previously mentioned challenges, enabling side-channel leakage detection for CUDA programs.

To obtain the correct trace of a CUDA application, we employ two different instrumentation tools to track the CUDA application separately, Pin [66] for the host code and NVBit [67] for the device code. Both of them are dynamic binary instrumentation tools so that the program's original behavior remains unaffected.

To address the challenges posed by multi-threading, we adopt a special A-DCFG. Similar to regular DCFGs, we use nodes to represent basic blocks, and edges to represent transitions from one basic block to another. Additionally, we enhance the nodes to record all memory accesses in every thread, capturing information such as when and where the accesses occurred. Similarly, the edges contain information about the transitions in every thread. Hence, we can eliminate redundant information within threads, effectively addressing the issue of data explosion caused by multi-threaded tracking. Furthermore, by performing differential analysis on the A-DCFG instead of individual thread traces, we can analyze the execution information of the entire program rather than focusing on a specific thread. This approach greatly reduces the computational overhead of the analysis process while improving its accuracy.

The whole process of Owl consists of three phases, namely the trace recording phase, the duplicates removing phase, and the leakage analysis phase, as shown in Fig. 3.

Trace Recording Phase. During this phase, we refer to user-provided inputs to collect the program's execution traces, including the kernels called, the basic blocks executed, and the memory accessed by the program. We reconstruct the trace of each kernel into an A-DCFG, then multiple A-DCFGs form the program trace from each user-provided input; afterward, these traces are initially handled and assessed.

Duplicates Removing Phase. In this phase, we eliminate the inputs that generate duplicate traces. In CUDA applications, different inputs may also produce identical observations, thus removing duplicate traces is vital for more efficient leakage analysis. At this point, we decide whether the program exhibits side-channel leakage by examining whether all inputs generate identical traces.

Leakage Analysis Phase. During this phase, we expect to confirm that the differences discovered in the previous phase are genuinely derived from the inputs rather than the random factors during program execution, which are non-deterministic observations mentioned in Section III; later we identify exact locations of leakage in the program. Specifically, we repeatedly execute the program with fixed and random inputs and then record the traces, on which we subsequently perform leakage tests. By repeatedly executing fixed inputs, we are able to distinguish differences resulting from random factors. Then, we compare traces from fixed and random inputs in order to identify their respective side-channel leakage. Ultimately, we label the kernels and basic blocks that fail the leakage tests as containing actual side-channel leakage.

Overall, Owl provides a new solution for detecting GPU side-channel leakage in CUDA applications. Firstly, Owl implements GPU side-channel leakage detection, including both control flow leakage and data flow leakage, in order to pinpoint leakage locations inside the program. Secondly, Owl diminishes the interference of random factors during program execution, thereby effectively reducing the false positive rate. Thirdly, Owl supports various types of CUDA applications, including cryptographic applications, deep learning frameworks, multimedia applications, etc.

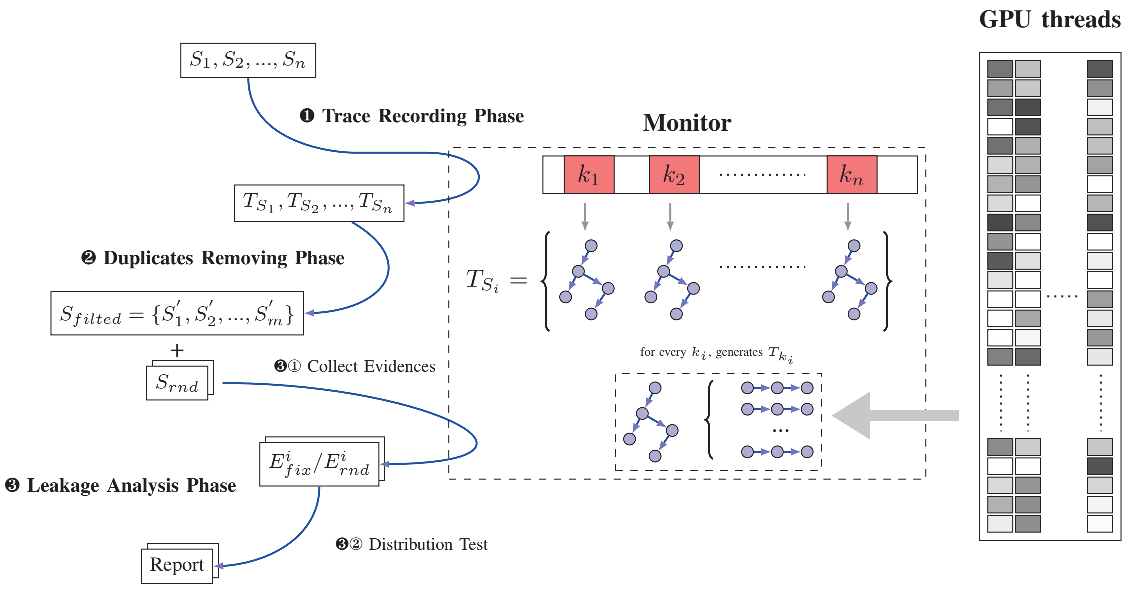


Fig. 3: Overview of Owl.

In the following sections, we describe Owl's key phases in detail.

V. Recording Traces

A. Tracing

As mentioned earlier, CUDA programs exhibit a more complex software architecture. To obtain accurate and detailed trace information, we employ different tracing strategies at various levels of program execution. We divide the whole execution process of CUDA applications into three levels. The top-level (i.e., program level) involves the program calling CUDA kernels during host code execution. The second level (i.e., kernel level) involves each kernel dispatched to GPUs, which is executed in warps concurrently. As warps are the smallest scheduling units in CUDA programming, we can simplify the CUDA software architecture by abstracting the execution process into multiple warps while ignoring other CUDA concepts such as blocks. Consequently, the third level (i.e., warp level) involves each warp accessing basic blocks and memory addresses. We track three levels of execution accordingly:

Program Level. At the program level, since we are not concerned with the trace information from the CPU side, we only track CUDA-related information in the host code, focusing on memory allocation and kernel invocations. Here we define S as the secret input, k as a kernel function, and P as the tested CUDA program. We write the device code sequence during P execution as $P = (k_1, k_2, \ldots, k_n)$, and its execution trace can be denoted as $T_P = (T_{k_1}, T_{k_2}, \ldots, T_{k_n})$, where each T_{k_i} corresponds to each invocation of kernel k_i. Although we consider parallel execution in GPUs, we do not consider parallel execution in CPUs, thus T_P is in chronological order.

Kernel Level. Every kernel generates multiple warps and executes in parallel on GPUs. In practice, due to hardware resource limitations (i.e. core number) and user customization, some warps may queue up and wait for others to complete, potentially resulting in another type of side-channel leakage. However, we only consider code-related leakage and exclude this type of leakage because of its volatility due to different hardware scales and user settings. Henceforth, we consider all warps under different blocks in a kernel as executing simultaneously. We define W_k as the set containing all warps generated by kernel k, which can be presented as $W_k = \{w_1, w_2, \ldots, w_m\}$. Therefore, the corresponding trace set of W_k will be $T_{W_k} = \{T_{w_1}, T_{w_2}, \ldots, T_{w_m}\}$.

Warp Level. A CUDA warp consists of multiple threads executing in parallel. However, due to predicated execution, there is no need to record them separately. If a thread within a warp does not need to execute a specific instruction, it will wait for other threads to complete the execution of that instruction instead of executing other instructions. This approach is employed to avoid warp divergence. Hence, whether a particular instruction is executed by one thread within a warp does not affect the control flow of the entire warp.

However, one active thread should be taken into account if it executes memory-accessing instructions that incur side-channel leakage. Therefore, we utilize a mapping to record the actual memory accesses performed by all threads within that warp, which holds pairs of addresses and the number of accesses.

Here, $T_w = (T_{b_1}, T_{b_2}, \ldots)$ records order of basic block b_i

accesses in the current warp where each T_{b_i} corresponds to a basic block within the warp's associated kernel. Each T_{b_i} consists of T_m, the memory record of each memory access instruction in the b_i, while T_m contains information about the memory address m and the total counts c of each accessed memory.

B. Reconstruct

After obtaining information about each kernel invocation and thousands of threads, Owl integrates the thread traces within each kernel into a single A-DCFG. A-DCFG is extended from DCFG [68] with extra information on nodes and edges, as described below:

- Similar to DCFG, each node N in our A-DCFG corresponds to a basic block b_i of the kernel, yet we extend the node with memory access information. N records memory access information at multiple instructions, where multiple memory records m_j for each instruction. m_j is a compilation of memory records generated by different warps during the j-th access to the basic block.
- Also similar to DCFG, we use edges to capture the transitions between basic blocks. Each edge is unidirectional and maintains information about the starting basic block, ending basic block, and the number of times the edge has been traversed. Additionally, we also record information about the previous edge in each edge, which will be used in leakage analysis. Similarly, we overlay all warps' transitions at the same basic blocks into a single edge.
- Our A-DCFG may have multiple start nodes and end nodes, as different warps may execute different code regions in the same kernel. Also, we do not record edges or nodes that are not executed.

By reconstructing traces into A-DCFGs, we can remove duplicate information from the traces, such as identical control flow and memory accesses, significantly reducing our memory assumption while facilitating Owl in analyzing programs with massive threads. Fig. 4 shows an example of A-DCFG formation: Different warps share the identical control flow transitions and their memory accesses with the same target address position are aggregated accordingly. Note that multi-threading related information such as the count of each control flow transition (numbers beside the arrows) is not dropped after the aggregation, but still preserved in the A-DCFG.

C. Implementation

We leverage NVBit [67], a dynamic binary instrument framework designed for GPU applications, to record traces of CUDA applications. During kernel startup, NVBit modifies the target binary by replacing the original instruction with a jump instruction that towards a trampoline code. In the trampoline code, NVBit first saves the thread context, then executes the user-defined instrumentation function, restores the program context, executes the original instruction, and finally jumps back. As the kernel code is modified, each launched thread executes our instrumentation function, which allows Owl to capture information from all threads. To avoid unnecessary noise, we also disable the ASLR of both host and device.

When tracing kernel function information at the program level, we find that using the initial address of the kernel function is not reliable for distinguishing kernel functions. Specifically, the kernel function address provided by NVBit during runtime mismatches the address invoked in the user code due to the compiler wrapping of the source kernel function, whose entry is replaced with the wrapping function. After completing a series of operations, it then jumps to the actual kernel function startup code. This leads to the following issues: 1) We are unable to distinguish kernel functions based on their addresses, as they are all launched by `cuLaunchKernel`[2] at the same address. Nevertheless, this problem can be resolved by obtaining the parameters of `cuLaunchKernel`. 2) Furthermore, we are also unable to distinguish different invocations of the same kernel function at different positions, as their parameters are identical. This can lead to erroneous results in our subsequent analysis of kernel internals, such as analyzing different kernels or kernels in different contexts. To address this, we use the call stack during the invocation of `cuLaunchKernel` as an identifier for the kernel function. This approach resolves the two aforementioned issues since the differences between different kernel functions and their invocation positions are reflected in their call stacks.

Additionally, similar to programs running on a CPU, CUDA applications also need to request memory allocation to store data, which is typically accomplished by invoking `cudaMalloc`[3] in the host code, and the memory address returned from `cudaMalloc` is affected by the memory layout, which can result in changes to accessed memory addresses. Therefore, we instrument the host code through Pin [66] to obtain memory addresses and sizes to be allocated at `cudaMalloc` call sites, and convert memory addresses to offsets during tracing.

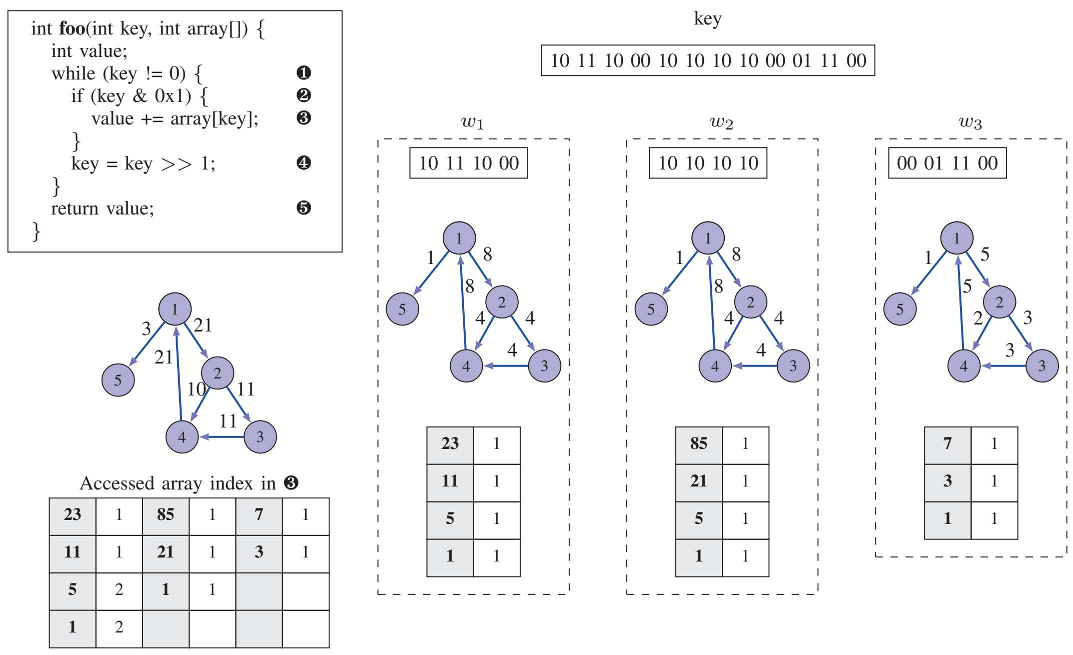
When tracing at the kernel level, NVBit sends trace information for all warps to the monitor. In the monitor, we identify different warps using both warp IDs as well as block IDs (warp IDs are unique in different blocks), and maintain the context of their traces.

At the warp level, we instrument two conditions in the CUDA kernel: basic block accesses and memory accesses. Upon entering a new basic block, the kernel sends the basic block ID which is the offset of the basic block inside the kernel, as well as the warp ID and block ID. When memory accesses occur, the kernel sends the target memory addresses and memory types[4], allowing for precise comparisons in our subsequent analysis.

[2]This denotes a family of kernel launch functions: `cuLaunchKernel`, `cuLaunchKernel_ptsz`, etc.

[3]This represents a family of memory allocation functions: `cudaMalloc`, `cudaHostAlloc`, `cudaMallocHost`, `cudaMallocManaged`, `cudaMallocAsync`, and `cudaMallocFromPoolAsync`, etc.

[4]According to NVBit, we categorize memory types into the following groups: `None`, `Local`, `Generic`, `Global`, `Shared`, `Constant`, `Global_to_Shared`, `Surface`, `Texture`.



```
int foo(int key, int array[]) {
    int value;
    while (key != 0) {                ❶
        if (key & 0x1) {              ❷
            value += array[key];      ❸
        }
        key = key >> 1;               ❹
    }
    return value;                     ❺
}
```

Accessed array index in ❸

23	1	**85**	1	**7**	1
11	1	**21**	1	**3**	1
5	2	**1**	1		
1	2				

23	1
11	1
5	1
1	1

85	1
21	1
5	1
1	1

7	1
3	1
1	1

Fig. 4: An example of A-DCFG formation.

VI. Filtering Traces

We expect to achieve two goals in this phase. Firstly, we expect to determine whether the program has potential side-channel leakage; secondly, we wish to extract inputs with particular traces and filter redundancy inputs in order to reduce the time cost in the leakage analysis phase.

In general, we achieve the targets by comparing each pair of traces to examine their consistency. We consider inputs that generate identical traces as the same class and believe that inputs of the same class have equal side-channel characteristics and do not leak side-channel information. Hence, we randomly select one input from each class for leakage analysis to avoid redundancy. On the one hand, we consider the program as side-channel leakage-free when there is only one class which implies that all traces are identical; on the other, we consider two traces as different (i.e., showing potential side-channel leakage) as long as they involve distinct kernel calls or if the A-DCFGs within the same kernel are different.

However, one possible case is that the initial user-provided inputs may not exhibit the side-channel leakages in the program. To mitigate this issue, the user can add more initial inputs. As extra initial inputs mean more execution path coverage, Owl is more likely to discover potential side-channel leakages in the program. Also, the overhead brought by additional inputs is controlled because inputs of the same class are filtered and not forwarded to the leakage analysis phase.

VII. Leakage Analysis

In the leakage analysis phase, we determine whether the differences in the execution traces are statistically input-dependent and identify their types and locations. More specifically, we repeatedly execute the program and record its traces with fixed inputs filtered in the previous phase and random inputs. Afterward, we compile the traces into various types of histograms, which are subsequently used for leakage testing. The leakage testing statistically checks whether histograms follow the same distribution.

A. Evidence

Before conducting distributed testing, we merge multiple traces obtained from repeated executions into a single piece of evidence. Generally, we generate two types of evidence, E_{fix} and E_{rnd}. E_{fix} is derived from fixed inputs, which are traces produced by filtered user inputs and then merged. E_{rnd}, on the other hand, is obtained by merging traces from random inputs. The merging process is shown as follows:

1) We utilize the `Myers` algorithm to compare two trace sequences from E_{fix} and E_{rnd}, then we align the sequences referring to kernel invocations.
2) For identical kernel invocations, we increment their invocation count and merge their DCFG as well. The algorithm for merging DCFGs is similar to the one used in the trace recording phase for merging warp traces. We aggregate information from each node and edge in the two DCFGs.

3) For distinct kernel invocations, we directly add them to the evidence.

E_{fix} and E_{rnd} contain the statistical features of the program's traces under fixed and random inputs, respectively. The input dependency may yield differences in the features, which is in line with our purpose. For instance, input-dependent memory accesses can lead to variations in the distribution of memory addresses and access frequencies between fixed and random inputs. However, the differences can also stem from non-deterministic factors within the program (e.g., random numbers). Therefore, we employ distribution testing to exclude differences in the evidence that are not genuinely input-dependent.

B. Distribution Test

Previous works [69], [70] have widely used Welch's t-test for the distribution test, but we use Kolmogorov-Smirnov (KS) test for a more generic assumption that does not require the traces to satisfy a normal distribution.

We assume $X_f = \{x_1, x_2, \dots, x_n\}$ and $Y_f = \{y_1, y_2, \dots, y_m\}$ are two independent random samples of feature f generated from E_{fix} and E_{rnd}, and the empirical distribution functions are denoted as

$$F_Y(t) = \frac{1}{m}\sum_{i=1}^{m} I_{y_i \leq t} \quad F_X(t) = \frac{1}{n}\sum_{i=1}^{n} I_{x_i \leq t} \tag{1}$$

where I is the indicator function. The null hypothesis of the KS statistics is that X and Y belong to the same probability distribution and the KS statistic $D_{X,Y}$ is calculated as

$$D_{X,Y} = \sup |F_X(t) - F_Y(t)| \tag{2}$$

We reject the null hypothesis if the KS statistic $D_{X,Y}$ exceeds the significance threshold $D_{n,m}$, which means the deviation of both samples is significant. The significance threshold is calculated as

$$D_{n,m} = \sqrt{-\ln(\frac{\alpha}{2}) \cdot \frac{1}{2}} \cdot \sqrt{\frac{n+m}{n \cdot m}} \tag{3}$$

where α is the confidence interval from 0 to 1.

Usually, we use p-value and compare it with the confidence level α, and the p-value of KS test is calculated as

$$p_{X,Y} = 2e^{-2(D_{X,Y})^2 \frac{n \cdot m}{n+m}} \tag{4}$$

We consider X to have a significant deviation from Y if $p_{X,Y} < (1 - \alpha)$ and consider it as a failed test. The success of the test indicates that the feature probability distribution of fixed inputs resembles that of random inputs, in other words, it is random factors rather than inputs that introduce the differences.

C. Leakage Test

During the leakage test, we employ feature extraction methods tailored to various types of leaks.

For **kernel leakage**, after obtaining two pieces of evidence merged from fixed-input traces and random-input traces respectively, we determine the presence of kernel leakage by comparing the differences in their kernel invocations. The differences include: 1) unaligned kernel invocations (i.e., the invocation is present in one while absent in another). 2) the aligned kernel invocations with different invocation counts. The differences reveal whether the program has vulnerabilities in the host code that may cause GPU side-channel information leakage.

For aligned kernels, we further analyze their inner execution. To verify the existence of **device control flow leakage** in the kernel, we firstly extract the control flow information of each node in the A-DCFG. Considering node N is executed n times, and each execution generates a pair of 2-tuples (src, dst), indicating the transition from one basic block to another (We consider the src of the first basic block and the dst of the last basic block as a special type of basic block.). Thus, we can use two vectors to represent the src and dst of N, shown as:

$$I = (x_1, x_2, x_3, \dots, x_k), \sum_{i=1}^{k} x_i = n \tag{5}$$

$$O = (y_1, y_2, y_3, \dots, y_p), \sum_{j=1}^{p} y_j = n \tag{6}$$

where x_i and y_j represent the respective counts of each type of src and dst. Thus, for N, there exists a control flow transition matrix $A_{k\times p}$ that satisfies:

$$I_{1\times k} \cdot A_{k\times p} = O_{1\times p} \tag{7}$$

According to the calculation formula of the matrix, $y_i = a_{1i}x_1 + a_{2i}x_2 + \cdots + a_{ki}x_k$, representing the probability distribution of the source of a specific output control flow. As I does not necessarily satisfy full column rank, A would have infinite solutions. However, we can construct a feasible solution by counting the number of each type of (src, dst). Referring to the control flow transition matrix, we can model the changes in program control flow under multi-threading scenarios and deploy differential analysis towards them.

To identify device control flow leakage, we generate a control flow transition matrix for every node in the A-DCFG and compile the matrices into histogram H_{cf}. In every H_{cf}, the x-axis represents the control flow and the y-axis represents the corresponding element in the control flow transition matrix $A_{k\times p}$.

$$H = (a_{11}, a_{12}, \dots, a_{1p}, a_{21}, \dots, a_{kp}) \tag{8}$$

If a pair of control flow transition matrices from the fixed and random inputs fail the distribution test, we conclude that there is a device control flow leakage in the basic block, and consider the current basic block as the leakage location.

Regarding **device data flow leakage**, we focus on memory access differences at the same instruction. The histogram H^i_{addr} records the address offsets of accessed memory (x-axis), as well as the number of times each address is accessed (y-axis). For every instruction's memory records $(m_1, m_2, \dots, m_n)$ in N, we compile them into

$(H_{addr}^1, H_{addr}^2, \ldots, H_{addr}^n)$, and we compare every pair of instructions from fixed and random inputs following the memory access order. However, for different inputs, the access count for each instruction may vary, so memory accesses in the extra instruction accesses do not have counterparts to form comparison pairs. Note that such differences are essentially due to control flow rather than data accesses, so we refer to them as control flow leakage. Besides, these extra basic block accesses are reflected in their control flow transition matrix where we can simply exclude them during the previous control flow leakage tests. For the remaining usual cases (i.e., memory access differences come from a pair of instruction accesses), we claim the existence of device data flow leakage if any instruction's H_{addr} fails the distribution test.

VIII. Evaluation

As described in Section III, we believe that a side-channel detection tool for CUDA applications should have sufficient scalability to detect various types of real-world programs and accurately locate the leakage positions to help developers identify vulnerabilities. Therefore, we evaluate Owl to answer the following research questions: **RQ1:** Can Owl identify side-channel leakage in CUDA applications? How accurate does Owl's detection work? **RQ2:** What is the memory overhead of Owl, and can it effectively scale to analyze GPU applications with a large number of threads? Although existing tools are not specifically designed for CUDA applications, we would like to explore their effectiveness and ask the **RQ3:** Are existing tools applicable to CUDA applications?

A. Evaluation Setup

For the following experiments, we execute 100 times for both fixed and random inputs in the leakage analysis phase, and set the confidence level to 0.95. The details of our experiment platform are listed in Table II.

We evaluate Owl in various types of CUDA programs, including a cryptographic application, Libgpucrypto, a popular deep learning framework, PyTorch, and a closed-source image processing tool, nvJPEG.

TABLE II: Parameters of the experiment platform.

Description	Value
CPU	Intel i9-12900 @ 2.40GHz
GPU	NVIDIA RTX A4000
OS	Ubuntu
Kernel	6.1.22
NVIDIA driver	525.105.17
CUDA	12.0

B. Answer to RQ1

TABLE III: Leaks detected by Owl.

Programs	Kernel leaks	D.F. leaks	C.F. leaks
Libgpucrypto	0/0	66/69	7/7
PyTorch	8/8	8/11	6/8
nvJPEG	0	45	98

To answer RQ1, we conduct detection on Libgpucrypto[5]'s AES and RSA encryption, several functions[6] in PyTorch, as well as the encoding and decoding processes of the nvJPEG, followed by manual analysis of the detected leaks to determine whether they are related to secret inputs, to assess Owl's accuracy. Table III shows the results.

In evaluating Libgpucrypto's AES and RSA encryption, Owl initially reports all the 18 control flow leaks, 173 data flow leaks, and zero kernel leaks. However, we find that some leaks at different basic blocks point to the same code location, which we attribute to loop unrolling performed by the CUDA compiler. Therefore, we further examine and filter these locations and screen the results into 7 control flow leaks and 69 data flow leaks. Through manually analyzing the 76 detected leaks, we confirm that 7 control flow leaks and 66 data flow leaks indeed depend on secret inputs, such as array accesses during table lookups in AES, as well as `if-else` branches in RSA.

As for PyTorch, we have identified 8 control flow leaks, 11 control flow leaks and 8 kernel leaks, among which 6 control flow leaks, 8 data flow leaks, and 8 kernel leaks have input dependence. We find that the optimizations for the special tensor in PyTorch cause these leaks. For example, one kernel leakage lies in the tensor serialization process, where PyTorch calls kernels based on whether the tensor is zero: Non-zero tensors trigger additional kernel calls. Besides, we find that the leaks are locally attached to a few functions, while most functions are leakage-free. We believe there are two primary reasons: Firstly, many functions in PyTorch are purely numerical computation functions, characterized by constant execution, thus do not exhibit side-channel leaks. Secondly, the predicate execution can mitigate side-channel leakages. In Libgpucrypto, as the keys are consistent in every thread, all threads execute the same control flow even if control flow dependency on the key exists. In contrast, secret inputs (e.g., tensors) in PyTorch are partitioned so the threads receive different secret inputs. However, according to predicated execution, even if there are different control flows within a warp, each thread in the warp will go through all basic blocks, thus revealing no information of the control flow. Take PyTorch's `max_pool2d` for example, a recent study [71] has reported side-channel leakage in the CPU implementation of `max_pool2d`. Although the implementation of `max_pool2d` in CUDA is nearly identical to its CPU counterpart, it does not exhibit any control flow leakage in our findings.

In evaluating nvJPEG, we process image data from the COCO-2014 dataset [72] to a fixed size, from which we randomly select as secret inputs. For the encoding process in nvJPEG, we have identified 98 control flow leaks, 45 data flow leaks, and zero kernel leaks, but none is found in the decoding process. Unfortunately, due to the absence of nvJPEG's source code, we are unable to conduct more detailed

[5]https://github.com/lwakefield/libgpucrypto

[6]Including `Tensor.__repr__`, `avgpool2d`, `maxpool2d`, `tanh`, `relu`, `sigmoid`, `softmax`, `conv2d`, `linear`, `crossentropy`, `mseloss`, `nllloss`.

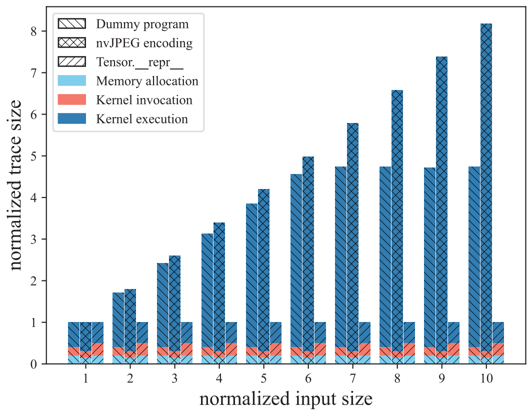


Fig. 5: The growth of Owl's trace size by input size.

analysis. Nevertheless, we have already disclosed our findings to NVIDIA.

Overall, Owl has identified hundreds of side-channel leaks within Libgpucrypto, PyTorch, and nvJPEG. In simple applications like Libgpucrypto, Owl achieves a 96% accuracy rate, while in a large-scale projects like PyTorch, Owl achieves 81%. Furthermore, the case of `max_pool2d` in PyTorch demonstrates Owl's sufficient ability to distinguish side-channel leakage from GPU to CPU, thereby reducing the false-positive rate. We also analyze the false positives generated by Owl and find that they usually reside in trivial parts of the code, such as thread synchronization (i.e., `_syncthreads`). The primary factor of false positives is that these parts may be indirectly affected by the inputs and Owl's distribution testing would fail to eliminate non-determinism in them. Notably, the evaluation of nvJPEG validates Owl's applicability towards side-channel leakage detection for closed-source software. This is particularly crucial in the CUDA ecosystem, where a majority of influential tools are closed-source, such as cuDNN and cuBLAS. Users of these closed-source programs are often left desperate for side-channel protection, as program providers usually overlook side-channel issues.

C. Answer to RQ2

For this part of evaluation, we use PyTorch, nvJPEG, and a dummy program as the test subjects. The reason we do not test Libgpucrypto is that the number of threads of it is not modifiable. Therefore, we design a dummy program that performs random array accesses to simulate the S-box lookup operation in the AES algorithm. We gradually raise the number of running threads by increasing the input size and record the corresponding trace size, the results of which are shown in Fig. 5.

When tracing the dummy program, the trace size initially increases with the thread number, but the growth becomes plateau later. The previous growth is primarily due to the proliferation of new memory accesses. When fewer threads running, different threads mostly access non-overlapping memory locations, thus the increase of threads generally means more new accesses and greater trace sizes. However, as the thread number continues to grow, memory addresses accessed by threads become repetitive and the number of distinct memory accesses gradually saturates. At this point, thanks to Owl's ability to merge duplicate memory accesses and reduce data redundancy, the trace size will end up stabilized.

However, we receive a different pattern on nvJPEG, where nvJPEG's trace size persistently grows in proportion with the increase of the input size. As we add the thread number by increasing the input size to where each thread may only process a single pixel of the image, different threads would still access non-overlapping memory addresses. Arguably, the effectiveness of Owl's merging strategy may diminish when newly added threads contribute significant new memory accesses. Nevertheless, Owl still successfully manages to record the trace of the nvJPEG program, including 128,000 threads (i.e., 100x1000 resolution), using only approximately 200MB of memory.

When testing PyTorch, we find that most functions in PyTorch behave similarly to nvJPEG, i.e., their trace sizes increase linearly with input size. However, we find that the `Tensor.__repr__` function does not increase its thread count with input growth. The function only uses a fixed number of threads to access individual data in the matrix so its trace size remains constant.

According to the evaluation results shown in Fig. 5, we observe that size of traces for memory allocations and kernel invocations does not changed with increasing input size. This outcome is primarily because both events occur within the host code. Specifically, the former is typically used to allocate memory space on the GPU for storing input-related data. In Owl, we record this using the starting address and size, so the record of memory space allocation remains constant regardless of the input size. The latter is usually initiated through the `cuLaunchKernel`. Generally, regardless of changes in input size, the kernel is only called once rather than multiple times. Therefore, for Owl, the main scalability challenges arise from tracing the executions within the kernels.

The evaluation result demonstrates three distinct patterns of trace size growth: ❶ fixed threads, where thread number is independent of the input, such as `Tensor.__repr__`; ❷ volatile threads with limited memory accesses, similar to the dummy program; ❸ volatile threads with unlimited memory access, like nvJPEG. From the evaluation results, Owl copes well with ❶ and ❷, meaning that regardless of the input, the trace size is reasonable and manageable. As for ❸, Owl's approach indeed has limitations, as the proliferation of new memory accesses makes it challenging to control the trace size. Fortunately, since Owl can eliminate control flow redundancy and repeated memory accesses among multiple threads, it still remains highly effective in analyzing thread-intensive CUDA programs.

Table IV shows Owl's performance. Notably, the analysis time and memory consumption for PyTorch are significantly

TABLE IV: Performance of Owl during the analysis of Libgpucrypto, PyTorch, and nvJPEG. Sizes and time in *Trace Collection* are per trace. RAMs in *Total* are the maximum memory used during the analysis.

Function		Trace Collection		Evidence Collection		Distribute Test		Total	
		Size(MB)	Time(s)	Traces	Time(s)	Traces	Time(ms)	RAM(GB)	Time(min.)
Libgpucrypto	AES	19.64	5.02	200	0.932	200	132.84	1.16	19.0
	RSA	250.42	15.60	200	4.602	200	39.65	2.32	52.8
PyTorch	Tensor.__repr__	0.21	49.30	200	0.009	200	0.39	9.55	164.4
	avgpool2d	0.02	46.60	200	0.002	200	0.05	9.49	155.3
	maxpool2d	0.02	46.46	200	0.002	200	0.05	9.50	154.9
	tanh	0.02	46.45	200	0.002	200	0.04	9.49	154.9
	relu	0.02	46.43	200	0.002	200	0.04	9.49	154.8
	sigmoid	0.02	46.34	200	0.002	200	0.04	9.49	154.5
	softmax	0.03	46.37	200	0.003	200	0.07	9.48	154.6
	conv2d	2.81	61.61	200	0.052	200	0.46	12.74	205.4
	linear	2.22	48.91	200	0.045	200	0.31	10.93	163.1
	crossentropy	0.09	47.09	200	0.004	200	0.12	9.52	157.0
	mseloss	0.04	46.99	200	0.003	200	0.10	9.52	156.6
	nlloss	0.07	46.54	200	0.003	200	0.05	9.48	155.2
nvJPEG	encoding	10.78	6.91	200	0.130	200	1.91	0.96	23.1
	decoding	5.85	4.48	200	0.137	200	0.84	0.92	14.9

higher than those for Libgpucrypto and nvJPEG, yet the trace size is much smaller for PyTorch. This abnormal increase in time and memory consumption is primarily due to Owl's tracing mechanism, which entails the tracing of the Python virtual machine during the analysis of PyTorch. Additionally, since Owl's tracing of kernels operates in parallel, it demonstrates commendable performance when tracking applications like nvJPEG, which utilize a large number of threads.

D. Answer to RQ3

We answer this research question by evaluating two typical tools: DATA [14], a dynamic analysis tool, and haybale-pitchfork [73], a tool based on LLVM IR analysis.

DATA utilizes Pin to monitor execution traces, but it cannot effectively detect side-channel leaks within CUDA kernels because it fails to observe traces inside the GPU. However, our evaluation showcases DATA's potential in identifying kernel leaks, as they are essentially originated from control-flow leaks of the host code.

For haybale-pitchfork, given that CUDA source code can be converted into LLVM IR, we attempt to cover its detection to CUDA kernels. However, this results in a substantial number of false positives, where we discover that it erroneously flags array accesses determined by thread IDs (a common practice in CUDA programming) as potential side-channel leaks. Furthermore, it misidentifies control flow leaks as it fails to account for predicate execution.

According to our evaluation of DATA and haybale-pitchfork, we argue that existing tools are generally ineffective in identifying side-channel leaks of CUDA programs, particularly those related to the device code. To summarize, tools relying on dynamic and binary analysis are insufficient in monitoring kernel operations; and tools based on source code or LLVM IR analysis have yet to consider CUDA-specific characteristics such as multi-threading, leading to considerable false positives.

IX. Countermeasures

Side-channel leakages have been widely used to launch model extraction attacks [5], [37], [38], [41], [74]. Existing work has proposed some relevant defense measures, including modifying hardware to eliminate memory access leakages [11], [38] and implementing DNN model obfuscation [13], [75], [76]. However, not all obfuscation strategies can guarantee sufficient security [75]. More general defense methods have also been proposed, such as hardware isolation [8], clearing microarchitectural information [7], and hiding secret-data-dependent memory access patterns [12]. The scatter-gather scheme has initially been used for CPU side-channel protection, and some work has applied it to encryption algorithm implementations on GPUs [77].

X. Conclusion

In this paper, we propose Owl, a side-channel leakage detection tool specifically designed to identify side-channel leakages in CUDA applications, addressing the current absence of GPU side-channel leakage analysis solutions. Owl captures the kernels launched within CUDA applications and integrates the traces of multiple threads for each kernel into a single A-DCFG, enhancing the accuracy and scalability of the detection process. We propose a statistical testing method based on A-DCFG to locate potential side-channel leaks in a program. Lastly, we evaluate Owl on Libgpucrypto, PyTorch, and nvJPEG, the results of which demonstrate that Owl can effectively identify and locate real side-channel leaks in the programs. Moreover, we believe that our DCFG-based dynamic detection approach can also be applied to other similar SIMT architectures, as is not coupled with any specific features of CUDA or NVIDIA device.

XI. Acknowledgements

We thank the anonymous reviewers for their constructive comments and valuable feedback that helped improve this work.

References

[1] NVIDIA, “CUDA.” https://developer.nvidia.com/cuda-zone, Accessed: June 2023.

[2] Khronos Group, “OpenCL.” https://www.khronos.org/opencl, Accessed: June 2023.

[3] OpenAI, “ChatGPT.” https://chat.openai.com/chat, Accessed: June 2023.

[4] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, “High-resolution image synthesis with latent diffusion models,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695, 2022.

[5] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. B. AbuGhazaleh, A. Marquez, and K. J. Barker, “Spy in the GPU-box: Covert and side channel attacks on multi-gpu systems,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023* (Y. Solihin and M. A. Heinrich, eds.), pp. 45:1–45:13, ACM, 2023.

[6] Z. Jiang and Y. Fei and David R. Kaeli, “A complete key recovery timing attack on a GPU,” in *Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, pp. 394–405, IEEE Computer Society, 2016.

[7] Y. Yadlapalli, H. Zhou, Y. Zhang, and C. Liu, “gguard: Enabling leakage-resilient memory isolation in gpu-accelerated autonomous embedded systems,” in *Proceedings of the 58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pp. 817–822, IEEE, 2021.

[8] B. Di, D. Hu, Z. Xie, J. Sun, H. Chen, J. Ren, and D. Li, “Tlb-pilot: Mitigating TLB contention attack on gpus with microarchitecture-aware scheduling,” *ACM Trans. Archit. Code Optim.*, vol. 19, no. 1, pp. 9:1–9:23, 2022.

[9] G. Kadam, D. Zhang, and A. Jog, “Rcoal: Mitigating GPU timing attack via subwarp-based randomized coalescing techniques,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pp. 156–167, IEEE Computer Society, 2018.

[10] G. Kadam, D. Zhang, and A. Jog, “Bcoal: Bucketing-based memory coalescing for efficient and secure gpus,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, pp. 570–581, IEEE, 2020.

[11] E. Karimi, Y. Fei, and D. R. Kaeli, “Hardware/software obfuscation against timing side-channel attack on a GPU,” in *Proceedings of the 2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pp. 122–131, IEEE, 2020.

[12] Z. Jiang, Y. Fei, A. A. Ding, and T. Wahl, “Mempoline: Mitigating memory-based side-channel attacks through memory access obfuscation,” *IACR Cryptol. ePrint Arch.*, p. 653, 2020.

[13] J. Li, Z. He, A. S. Rakin, D. Fan, and C. Chakrabarti, “Neurobfuscator: A full-stack obfuscation tool to mitigate neural architecture stealing,” in *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*, pp. 248–258, IEEE, 2021.

[14] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, “DATA - differential address trace analysis: Finding address-based side-channels in binaries,” in *Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018* (W. Enck and A. P. Felt, eds.), pp. 603–620, USENIX Association, 2018.

[15] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, “Cached: Identifying cache-based timing channels in production software,” in *Proceedings of the 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017* (E. Kirda and T. Ristenpart, eds.), pp. 235–252, USENIX Association, 2017.

[16] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, “Microwalk: A framework for finding side channels in binaries,” in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pp. 161–173, ACM, 2018.

[17] Y. Yuan, Z. Liu, and S. Wang, “Cacheql: Quantifying and localizing cache side-channel vulnerabilities in production software,” *CoRR*, vol. abs/2209.14952, 2022.

[18] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, “The program counter security model: Automatic detection and removal of control-flow side channel attacks,” in *Proceedings of the 8th International Conference on Information Security and Cryptology, ICISC 2005, Seoul, Korea, December 1-2, 2005, Revised Selected Papers* (D. Won and S. Kim, eds.), vol. 3935 of *Lecture Notes in Computer Science*, pp. 156–168, Springer, 2005.

[19] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, “Cacheaudit: A tool for the static analysis of cache side channels,” in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013* (S. T. King, ed.), pp. 431–446, USENIX Association, 2013.

[20] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, “Identifying cache-based side channels through secret-augmented abstract interpretation,” in *Proceedings of the 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019* (N. Heninger and P. Traynor, eds.), pp. 657–674, USENIX Association, 2019.

[21] Y. Noller, C. S. Pasareanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, “Hydiff: hybrid differential software analysis,” in *Proceedings of the 42nd International Conference on Software Engineering, ICSE '20, Seoul, South Korea, 27 June - 19 July, 2020* (G. Rothermel and D. Bae, eds.), pp. 1273–1285, ACM, 2020.

[22] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. T. Kandemir, “Casym: Cache aware symbolic execution for side channel detection and mitigation,” in *Proceedings of the 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pp. 505–521, IEEE, 2019.

[23] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, “Diffuzz: differential fuzzing for side-channel analysis,” in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019* (J. M. Atlee, T. Bultan, and J. Whittle, eds.), pp. 176–187, IEEE / ACM, 2019.

[24] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer, “Big numbers - big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations,” in *Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (S. Capkun and F. Roesner, eds.), pp. 1767–1784, USENIX Association, 2020.

[25] “NVIDIA Ampere Architecture.” https://www.nvidia.com/en-us/data-center/ampere-architecture/, Accessed: June 2023.

[26] H. Naghibijouybari, A. Neupane, Z. Qian, and N. B. Abu-Ghazaleh, “Rendered insecure: GPU side channel attacks are practical,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (D. Lie, M. Mannan, M. Backes, and X. Wang, eds.), pp. 2139–2153, ACM, 2018.

[27] S. Lee, Y. Kim, J. Kim, and J. Kim, “Stealing webpages rendered on your browser by exploiting GPU vulnerabilities,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pp. 19–33, IEEE Computer Society, 2014.

[28] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu, “Vulnerable GPU memory management: Towards recovering raw data from GPU,” *Proc. Priv. Enhancing Technol.*, vol. 2017, no. 2, pp. 57–73, 2017.

[29] Z. Jiang, Y. Fei, and D. R. Kaeli, “A novel side-channel timing attack on gpus,” in *Proceedings of the on Great Lakes Symposium on VLSI 2017, Banff, AB, Canada, May 10-12, 2017* (L. Behjat, J. Han, M. N. Velev, and D. Chen, eds.), pp. 167–172, ACM, 2017.

[30] Z. H. Jiang, Y. Fei, and D. Kaeli, “Exploiting bank conflict-based side-channel timing leakage of gpus,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–24, 2019.

[31] W. He, W. Zhang, S. Sinha, and S. Das, “igpu leak: An information leakage vulnerability on intel integrated GPU,” in *Proceedings of the 25th Asia and South Pacific Design Automation Conference, ASP-DAC 2020, Beijing, China, January 13-16, 2020*, pp. 56–61, IEEE, 2020.

[32] E. Karimi, Z. H. Jiang, Y. Fei, and D. R. Kaeli, “A timing side-channel attack on a mobile GPU,” in *Proceedings of the 36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*, pp. 67–74, IEEE Computer Society, 2018.

[33] J. Ahn, C. Jin, J. Kim, M. Rhu, Y. Fei, D. R. Kaeli, and J. Kim, “Trident: A hybrid correlation-collision GPU cache timing attack for AES key recovery,” in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*, pp. 332–344, IEEE, 2021.

[34] C. Luo, Y. Fei, and D. R. Kaeli, “GPU acceleration of RSA is vulnerable to side-channel timing attacks,” in *Proceedings of the International*

Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018 (I. Bahar, ed.), p. 113, ACM, 2018.

[35] C. Luo, Y. Fei, and D. R. Kaeli, "Side-channel timing attack of RSA on a GPU," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 32:1–32:18, 2019.

[36] H. Naghibijouybari, A. Neupane, Z. Qian, and N. B. AbuGhazaleh, "Side channel attacks on gpus," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 4, pp. 1950–1961, 2021.

[37] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. A. Faruque, "Leaky DNN: stealing deep-learning model secret with GPU context-switching side-channel," in *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pp. 125–137, IEEE, 2020.

[38] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, and Y. Xie, "Deepsniffer: A DNN model extraction framework based on learning architectural hints," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, Lausanne, Switzerland, March 16-20, 2020* (J. R. Larus, L. Ceze, and K. Strauss, eds.), pp. 385–399, ACM, 2020.

[39] X. Hu, L. Liang, X. Chen, L. Deng, Y. Ji, Y. Ding, Z. Du, Q. Guo, T. Sherwood, and Y. Xie, "A systematic view of model leakage risks in deep neural network systems," *IEEE Trans. Computers*, vol. 71, no. 12, pp. 3254–3267, 2022.

[40] D. Yang, P. J. Nair, and M. Lis, "Huffduff: Stealing pruned dnns from sparse accelerators," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023* (T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, eds.), pp. 385–399, ACM, 2023.

[41] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal DNN models with lossless inference accuracy," in *Proceedings of the 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (M. Bailey and R. Greenstadt, eds.), pp. 1973–1988, USENIX Association, 2021.

[42] M. Tan, J. Wan, Z. Zhou, and Z. Li, "Invisible probe: Timing attacks with pcie congestion side-channel," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pp. 322–338, IEEE, 2021.

[43] M. Side, F. Yao, and Z. Zhang, "Lockeddown: Exploiting contention on host-gpu pcie bus for fun and profit," in *Proceedings of the 7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*, pp. 270–285, IEEE, 2022.

[44] B. Yang, R. Chen, K. Huang, J. Yang, and W. Gao, "Eavesdropping user credentials via GPU side channels on smartphones," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, eds.), pp. 285–299, ACM, 2022.

[45] Y. Zhang, C. Slocum, J. Chen, and N. B. Abu-Ghazaleh, "It's all in your head(set): Side-channel attacks on AR/VR systems," in *Proceedings of the 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023* (J. A. Calandrino and C. Troncoso, eds.), pp. 3979–3996, USENIX Association, 2023.

[46] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (A. Cohen and M. T. Vechev, eds.), pp. 362–375, ACM, 2017.

[47] C. Sung, B. Paulsen, and C. Wang, "CANAL: a cache timing analysis framework via LLVM transformation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018* (M. Huchard, C. Kästner, and G. Fraser, eds.), pp. 904–907, ACM, 2018.

[48] J. Wichelmann, F. Sieck, A. Pätschke, and T. Eisenbarth, "Microwalk-ci: Practical side-channel analysis for javascript applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022* (H. Yin, A. Stavrou, C. Cremers, and E. Shi, eds.), pp. 2915–2929, ACM, 2022.

[49] Y. Yuan, Q. Pang, and S. Wang, "Automated side channel analysis of media software with manifold learning," in *Proceedings of the 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022* (K. R. B. Butler and K. Thomas, eds.), pp. 4419–4436, USENIX Association, 2022.

[50] L. Simon, D. Chisnall, and R. J. Anderson, "What you get is what you C: controlling side effects in mainstream C compilers," in *Proceedings of the 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pp. 1–15, IEEE, 2018.

[51] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (A. Cohen and M. T. Vechev, eds.), pp. 406–421, ACM, 2017.

[52] L. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Symbolic binary analyzer for security with applications to constant-time and secret-erasure," *ACM Trans. Priv. Secur.*, vol. 26, no. 2, pp. 11:1–11:42, 2023.

[53] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 875–890, ACM, 2017.

[54] B. Rodrigues, F. M. Q. Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016* (A. Zaks and M. V. Hermenegildo, eds.), pp. 110–120, ACM, 2016.

[55] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu, "Abacus: Precise side-channel analysis," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pp. 797–809, IEEE, 2021.

[56] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proceedings of the 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016* (T. Holz and S. Savage, eds.), pp. 53–70, USENIX Association, 2016.

[57] S. ul Hassan, I. Gridin, I. M. Delgado-Lozano, C. P. García, J. Chi-Domínguez, A. C. Aldaya, and B. B. Brumley, "Déjà vu: Side-channel analysis of mozilla's NSS," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20, Virtual Event, USA, November 9-13, 2020* (J. Ligatti, X. Ou, J. Katz, and G. Vigna, eds.), pp. 1887–1902, ACM, 2020.

[58] S. He, M. Emmi, and G. F. Ciocarlie, "ct-fuzz: Fuzzing for timing leaks," in *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pp. 466–471, IEEE, 2020.

[59] W. Wang, Y. Zhang, and Z. Lin, "Time and order: Towards automatically identifying side-channel vulnerabilities in enclave binaries," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pp. 443–457, USENIX Association, 2019.

[60] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "STACCO: differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 859–874, ACM, 2017.

[61] J. Wichelmann, C. Peredy, F. Sieck, A. Pätschke, and T. Eisenbarth, "MAMBO-V: dynamic side-channel leakage analysis on RISC-V," *CoRR*, vol. abs/2305.00584, 2023.

[62] S. Deng, M. Li, Y. Tang, S. Wang, S. Yan, and Y. Zhang, "Cipherh: Automated detection of ciphertext side-channel vulnerabilities in cryptographic implementations," in *Proceedings of the 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023* (J. A. Calandrino and C. Troncoso, eds.), pp. 6843–6860, USENIX Association, 2023.

[63] T. Yavuz, F. Fowze, G. Hernandez, K. Y. Bai, K. R. B. Butler, and D. J. Tian, "ENCIDER: detecting timing and cache side channels in SGX enclaves and cryptographic apis," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, pp. 1577–1595, 2023.

[64] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[65] J. Ahn, J. Kim, H. Kasan, L. Delshadtehrani, W. Song, A. Joshi, and J. Kim, "Network-on-chip microarchitecture-based covert channel in

gpus," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2021*, pp. 565–577, 2021.

[66] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), p. 190–200, Association for Computing Machinery, 2005.

[67] O. Villa, M. Stephenson, D. W. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for NVIDIA gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pp. 372–383, ACM, 2019.

[68] C. Yount, H. Patil, M. S. Islam, and A. Srikanth, "Graph-matching-based simulation-region selection for multiple binaries," in *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pp. 52–61, IEEE Computer Society, 2015.

[69] B. J. Gilbert Goodwill, J. Jaffe, and P. Rohatgi, "A testing methodology for side-channel resistance validation," in *NIST non-invasive attack testing workshop*, vol. 7, pp. 115–136, 2011.

[70] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017* (D. Atienza and G. D. Natale, eds.), pp. 1697–1702, IEEE, 2017.

[71] S. Shukla, M. Alam, S. Bhattacharya, P. Mitra, and D. Mukhopadhyay, "Whispering mlaas exploiting timing channels to compromise user privacy in deep neural networks," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2023, no. 2, pp. 587–613, 2023.

[72] T. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," in *Proceedings of the 13th European Conference on Computer Vision, ECCV 2014, Zurich, Switzerland, 2014, Part V* (D. J. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), vol. 8693 of *Lecture Notes in Computer Science*, pp. 740–755, Springer, 2014.

[73] C. Disselkoen, S. Cauligi, D. Tullsen, and D. Stefan, "Finding and eliminating timing side-channels in crypto code with pitchfork," in *TECHCON*, 2020.

[74] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pp. 1157–1174, IEEE, 2022.

[75] M. M. Ahmadi, L. Alrahis, A. Colucci, O. Sinanoglu, and M. Shafique, "Neurounlock: Unlocking the architecture of obfuscated deep neural networks," in *Proceedings of the International Joint Conference on Neural Networks, IJCNN 2022, Padua, Italy, July 18-23, 2022*, pp. 1–10, IEEE, 2022.

[76] M. M. Ahmadi, L. Alrahis, O. Sinanoglu, and M. Shafique, "Dnn-alias: Deep neural network protection against side-channel attacks via layer balancing," *CoRR*, vol. abs/2303.06746, 2023.

[77] Z. Lin, U. Mathur, and H. 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, GPGPU@ASPLOS 2019, Providence, RI, USA, April 13, 2019* (A. Jog and O. Kayiran, eds.), pp. 2–11, ACM, 2019.