iGPU Leak: An Information Leakage Vulnerability on Intel Integrated GPU

Wenjian HE* Wei Zhang† Sharad Sinha‡ Sanjeev Das§
*†Hong Kong University of Science and Technology, China
‡Indian Institute of Technology Goa, India §University of North Carolina at Chapel Hill, USA
*wheac@connect.ust.hk †wei.zhang@ust.hk ‡sharad@iitgoa.ac.in §sdas@cs.unc.edu

***Abstract*—Hardware accelerators such as integrated graphics processing units (iGPUs) are increasingly prevalent in modern systems. They typically provide multiplexing support where several user applications can share the iGPU acceleration resources. However, security in this setting has not received sufficient consideration. In this work, we disclose a critical information leakage vulnerability due to defective GPU context management. In essence, residual register values and shared local memory in the iGPU are not cleared during a context switch. As a result, adversaries can recover the secret key of a cryptographic algorithm running on an iGPU from a single snapshot of the leaking channel. User privacy is also under threat due to browser activity eavesdropping through website-fingerprinting attack with high accuracy and resolution. Moreover, this vulnerability can constitute a covert channel with a bandwidth of up to 8 Gbps.**

I. Introduction

Due to the ever-evolving performance demands, the central processing unit (CPU) is no longer the only computation powerhouse in many systems with the rise of hardware accelerators. The graphics processing unit (GPU), which features a massively parallel architecture, has been the most popular co-processor. It can accelerate a broad spectrum of data processing tasks, including, but not limited to, cryptographic computation [1], browser rendering [2] and machine learning [3].

As opposed to decades of development in CPU software protection, vendors have failed to devote sufficient efforts on the security of GPUs. Several attack vectors have been reported on NVIDIA discrete GPUs (dGPUs). For instance, researchers discovered that the DRAM contents of a dGPU can be leaked through the memory management interface of the driver software [2]. Later, GPU-side programs are found capable of dGPU memory stealing [4]. Furthermore, attackers can probe the statistics of memory utilization and performance counters on a dGPU, constituting a side-channel to track user activities including keystroke events and browser websites [5].

Although dGPUs attract more attention, the security of integrated GPUs (iGPUs) is more crucial in consideration of their ubiquity and capability. Every CPU is equipped with an on-chip iGPU on most Intel's desktop and mobile product series. Moreover, up to 71% of personal computers use iGPUs since they are not shipped with a dGPU [6]. In addition, modern iGPUs support complex programming models including general-purpose programmability. The intricate software and hardware stack of iGPUs results in a large attack surface for adversaries. Fortunately, we verify that they are immune to the aforementioned dGPU attacks thanks to the vastly different architecture of iGPUs. However, the reliability of the iGPU infrastructure has not received much scrutiny.

In this work, we focus on vulnerability analysis of iGPUs whose security has been underestimated for years. Our investigation discovers a serious data breach flaw in Intel iGPUs. The problem is caused by incomplete enforcement of the GPU context switch. When multiple applications offload tasks to an iGPU, the graphics driver is in charge of job scheduling by interleaving the occupation of the iGPU. However, data remnants of a previous GPU client are left uncleared in the micro-architecture of the iGPU, visible to the next occupant. As a consequence, an attacker can launch GPU malware to spy on sensitive information of a victim iGPU user. The malware can be written in accordance with standard specifications of the iGPU, therefore it looks like a regular GPU program to the system.

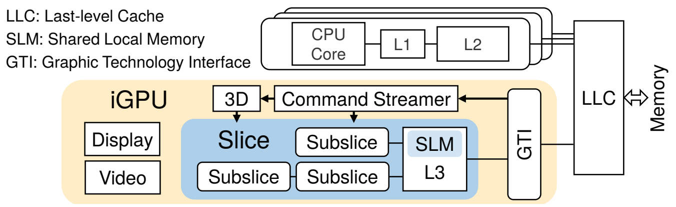

Fig. 1. Intel on-chip integrated GPU architecture

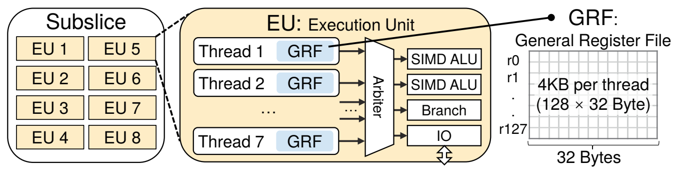

Fig. 2. Subslice micro-architecture of Intel integrated GPU

We illustrate how the vulnerability undermines security and privacy through three proof-of-concept attacks. Given the details about the GPU program, an adversary can initiate attacks to obtain secrets such as cryptographic keys. Even in the absence of understanding the GPU program, we can launch a powerful black-box attack to track browser activities with high precision. Besides these, the vulnerability can be exploited for data transmission between two sandboxed programs that ought to be prohibited from communicating with each other.

In summary, the main contributions of this work are as follows:

- We discover a critical information leakage vulnerability in integrated GPUs. To the best of our knowledge, this is the first work that reports an architectural vulnerability in integrated GPUs.
- We develop practical attacks in which adversaries can steal cryptographic keys, and snoop user activities on a popular browser.
- We demonstrate an end-to-end covert channel that is 3 orders of magnitude faster than other methods.

II. Background

A. Intel Integrated GPU Architecture

Many Intel CPUs follow a system-on-chip design where a GPU is tightly integrated with the processor cores [7]. Fig. 1 depicts a simplified diagram of the Intel CPU-iGPU platform. In contrast to discrete GPUs that maintain an isolated address space with a dedicated physical memory, the Intel iGPU shares the cache-coherent memory subsystem with CPU cores. In this architecture, the iGPU can use the virtual memory space of a CPU-side application. This architecture brings two advantages. First, the overhead of data transfers can be eliminated by virtue of the unified physical connection. Indeed, data copy between CPU and GPU is often a bottleneck for discrete GPUs. Second, CPU and iGPU can concurrently operate on pointer-rich data structures like trees and graphs.

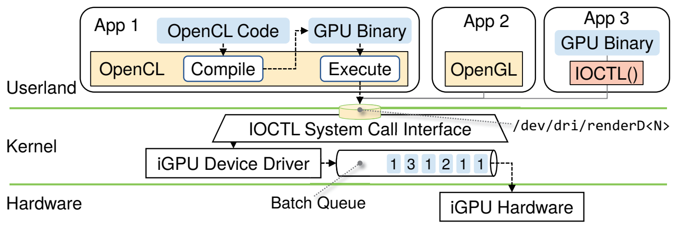

Fig. 3. Intel iGPU driver stack in Linux

We brief the micro-architecture of Intel iGPUs on the basis of Intel Gen 9 graphics [8]. An iGPU has one or more slices as the top-level hierarchy of its computation engine. Each slice contains a cache and three subslices. One of the components under inspection in this work is the shared local memory (SLM), which is a special portion of the cache. It serves as the internal storage of GPU subslices so that GPU threads can efficiently exchange data. Each subslice has exclusive access to 64 KB of the SLM space.

Within a slice, 24 execution units (EUs) are grouped into 3 subslices, so each subslice takes charge of 8 EUs. As illustrated in Fig. 2, an EU is similar to a 7-way hyper-threading processor core. These 7 threads share 4 functional units at the backend of the EU pipeline to execute instructions. Every individual thread maintains its own architectural state composed of an instruction pointer, an architectural register file and a general register file (GRF). The GRF is also a component we examine in this work. Each thread of the Intel iGPU is facilitated with a GRF as large as 4 KB. As depicted in Fig. 2, the GRF is logically divided into 128 registers, each of which is 32 bytes wide.

B. Linux GPU Driver Stack

For ease of development, most user applications harness the GPU with the help of high-level programming abstractions like OpenCL [9], OpenGL [10] and CUDA [11]. For example, OpenCL allows developers to create GPU programs in a C-flavor language.

When high-level code is ready, the GPU driver stack can compile and execute it, as shown in Fig. 3. This is accomplished by the cooperation of drivers in the user space and in the kernel space. As an agent between the user and the kernel, the userland driver provides a friendly interface to hide hardware details. This is usually implemented as a software shared library that applications can link with. In OpenCL, a compiler is shipped with the driver to compile user code to the GPU instruction set architecture (ISA) [12]. Before execution, the driver also helps to manage low-level data structures for the GPU program.

The userland driver submits jobs by means of system calls. On Linux, the kernel exposes an IOCTL interface to control the Intel iGPU, as shown in Fig. 3. Owing to the prevalence of graphical desktop interfaces, a program can invoke this system call without the superuser privilege. In a real machine, it is common that multiple users or programs request GPU computation at the same time. The device driver distinguishes them by associating a GPU context to each of them. When a job is submitted, the driver encapsulates the job and resources in the corresponding context into a batch, which is then queued in a buffer for the iGPU hardware to fetch and execute. Therefore, two neighboring batches in the queue can originate from different contexts.

The aforementioned description introduces the typical usage of iGPUs. However, an application is free to use the native IOCTL system call without the help of conventional userland drivers, as exemplified by App 3 in Fig 3. In this case, an application can launch a GPU program as long as it properly organizes the low-level metadata by itself.

TABLE I
SYSTEM CONFIGURATION

Machine Model	Dell OptiPlex 7040
CPU Model	Intel Core i7 6700 @ 3.4Ghz
iGPU Model	Intel HD 530 (Gen 9)
OS	1. Ubuntu Desktop 16.04 LTS with 4.15.0 kernel 2. Ubuntu Desktop 18.04 LTS with 4.18.0 kernel
OpenCL Driver	Intel Graphics Compute Runtime for OpenCL [9] version 18.42.11702 and version 19.26.13286

C. Threat Model and Testbed Setup

In this work, we strive to scrutinize whether an attacker can compromise security boundaries as an ordinary GPU client. To this end, we assume the adversary only has the permission to run a program at the non-privileged level, i.e., the root privilege is not available to the adversary. The adversary and the victim co-locate in the same physical machine but they are isolated such that the attacker cannot learn information about the victim. This can be a multi-user situation where the adversary holds a non-privileged account and tries to steal the secrets of another user on the same system. Given that current anti-virus software does not scan GPU binaries, the attacker can even be native spyware in the victim's account.

Our investigation has been performed on a commodity off-the-shelf machine detailed in Table I. The iGPU is an on-chip component in the CPU as mentioned, and its device driver is embedded in the kernel by default. The vulnerability can be reproduced in two versions of the Linux kernel in the Ubuntu distribution. Regarding the userland driver, we performed experiments on two versions of the Intel OpenCL library [9].

III. iGPU Leak Vulnerability

In this paper, we present a new information leaking attack vector on iGPUs. The problem is caused by the lack of data clearance during GPU context switches. In our study, we identify two sources of the leakage, in particular, the general register files (GRFs) and the shared local memory (SLM), which are detailed as follows.

A. Register Leakage

When a GPU job finishes, the device driver immediately hands over control to the next job without overwriting the registers of the GPU hardware threads. As a result, secret data left by a predecessor can be obtained by a subsequent GPU program. We verify this leakage by constructing two GPU programs: one is the victim writing specific values to GRFs, and the other is the spy which reads the GRFs. They are launched by different users on the system, and the experiment shows that the spy program can observe the values of the other.

This is a critical vulnerability by reason of the amount of private data exposed to attackers. As mentioned in Section II-A, every GPU hardware thread has 128 registers of size 32 bytes each, and the whole iGPU has 168 threads, resulting in 672 KB in total. These registers can always leak computation results of a victim, together with intermediate values and input data in usual cases. This is because the iGPU ISA does not support memory operand addressing [13], so the results must employ the GRF as the intermediate station before the store instruction can send them to the CPU. Similarly, the input has to be read into the GRF before computation.

The register leakage would not exist when GPU executables are generated by trusted compilers. However, as explained in Section II-B, it is allowed for users to supply their own binaries. Next, we describe how to craft and launch a malicious iGPU binary to read register remnants of other GPU clients.

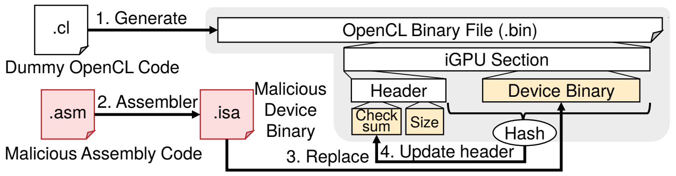

Fig. 4. Embedding malicious code into an OpenCL binary

TABLE II
OPENCL CODE FOR SLM LEAKAGE

	Victim's code	Attacker's code
1	void write_slm(__global uint *in){	void read_slm(__global uint *out){
2	__local uint slm[N];	__local uint slm[N];
3	size_t base = N*get_group_id(0);	size_t base = N*get_group_id(0);
4	for(size_t i=0; i<N; ++i)	for(size_t i=0; i<N; ++i)
5	slm[i] = in[base+i]; }	out[base+i] = slm[i]; }

Spyware Programming A rational compiler does not emit read instructions on undefined registers, therefore we cannot rely on high-level code to generate the spyware. Instead, we turn to assembly programming. The iGPU assembly syntax is well documented in [13]. An iGPU assembler `iga64` is found inside the repository of the Intel open-source graphics compiler [12]. We use it to translate assembly code to device binary.

Spyware Execution The next challenge is to orchestrate the execution of the GPU spyware. Although direct interaction with the device driver is feasible, for simplicity, we deceive the OpenCL runtime into accepting our spyware to avoid manual handling of low-level metadata. The procedure for building a malicious OpenCL binary is shown in Fig. 4. First, we generate a dummy OpenCL binary, which contains an empty GPU kernel. The compiled device binary of the empty kernel is inside the iGPU section of the OpenCL binary complex, and we replace it with our program. The checksum and size values are also updated to pass the integrity check of the OpenCL driver. Then, we can mislead the OpenCL runtime to load our binary and schedule it to run on the GPU. We clarify that our hacking of OpenCL binary is not a vulnerability but is an informal way to enable assembly programming in Intel OpenCL. In CPU software, assembly programming is a common optimization in performance-critical scenarios. We believe this should be the same for GPU programs, and security must not be compromised by assembly programming.

GPU Thread Scheduling To collect all the register residues, the spyware needs to be spawned to every GPU hardware thread. As the scheduling scheme of the driver stack is not explicit, it is necessary to validate the scheduling distribution to avoid incomplete coverage. In this regard, we add instructions in the spyware to read the unique identifiers of the hardware threads. In iGPU hardware. each thread holds a pre-defined index as the identifier in an architectural register dubbed sr0. We instruct each spawned instance of the spyware to read and report the index back to the CPU so the CPU-side program can verify the number of unique identifiers. According to our tests, the driver assigns one instance of the spyware to each hardware thread when we request to execute 168 independent instances in one GPU invocation.

B. Shared Local Memory Leakage

In addition to registers, the SLM is also not cleared at GPU context switches between tightly queued jobs. The SLM component in the micro-architecture of an iGPU has a close relationship with the high-level programming concept of *local memory* in OpenCL. As a result, we can verify this leakage channel with the OpenCL code in Table II. The *__local* qualifier at Line 2 is used to describe a variable in the OpenCL local memory scope, which is realized by the SLM in the iGPU hardware. Note that a subslice only has access to its own SLM space; therefore, we employ the *get_group_id()* OpenCL function to attain hardware allocation knowledge. Specifically, the function returns a unique index to GPU threads in the same subslice. Based on this, the calculated *base* offset ensures different threads do not read or write on overlapping memory space. To harvest SLM residues associated with the three subslices, three independent instances (i.e., threads) of our SLM spyware are spawned in every GPU invocation.

A snapshot of SLM contents can leak up to 192 KB of data. Fortunately, we monitored the SLM during the execution of various programs, but none of these programs seemed to leverage the SLM. By comparison, we can capture abundant traces of activity from the GPU registers. Though the SLM may be a low-risk leakage source for common users at present, it is unacceptable for mission-critical applications.

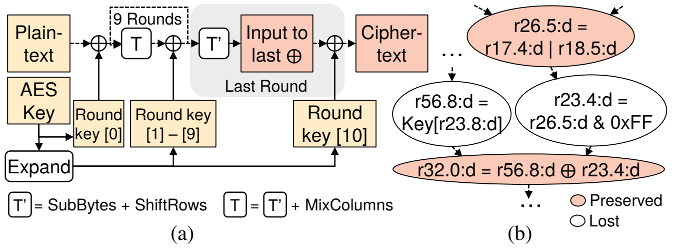

Fig. 5. (a) AES-128 encryption flowchart; (b) Register flow graph

IV. Attack Case Studies

The iGPU leak flaw can invite serious violation of security and privacy. In this section, we implement three different attacks to highlight the severity of the problem. Our exploitation includes a key-recovery attack against the Advanced Encryption Standard (AES), a website-fingerprinting attack against the Chrome browser, and a covert channel attack for unauthorized data transmission.

A. Attack I: AES Key Recovery

The AES is one of the most popular cryptographic algorithms. We use the AES implementation from engine-opencl [14], an extension of the OpenSSL toolkit [15]. As the name suggests, engine-opencl leverages OpenCL to accelerate cryptographic algorithms on a GPU. We first brief basic knowledge of the AES and then detail our attack to recover the complete AES keys.

AES encryption transforms a fixed length chunk of plaintext to ciphertext according to the AES key. Fig. 5(a) illustrates the diagram of AES-128 encryption. Before the transformation, a stream of bytes is derived from the initial AES key, which forms 11 round keys in AES-128. The key expansion is a reversible algorithm such that we can recover the AES key given any consecutive portion of the generated stream. The length of the portion needs to be at least the length of the AES key. As shown in Fig. 5(a), the plaintext input is XOR'ed with the first round key at the beginning of the AES. Subsequently, the AES algorithm applies several rounds of processing steps. In rounds 1-9, the data block goes through SubBytes, ShiftRows and MixColumns steps followed by an XOR operation with one of the round keys. In the last round, the MixColumns step is omitted, and the block is XOR'ed with the last round key at the end. Compared to AES-128, AES-196 has a longer AES key and more rounds of operations.

Next, we explicate our key-recovery attack against the AES algorithm. By principle, our white-box attack methodology can be generalized to compromise other iGPU programs, and consists of the following 4 steps.

1. Variable Identification: In the first step, an attacker tries to identify a set of sensitive variables of the victim program. In the

TABLE III
AES ATTACK RESULTS

Algorithm	AES key length	Leaked bytes	Full key recovery
AES-128	16 bytes	13	0.15 s
AES-192	24 bytes	20	2 min

case of AES, the goal of the attacker is to get the AES key. Given the aforementioned reversibility of the key expansion, the AES key can be recovered from round keys. Therefore, we classify round keys as sensitive variables. If round keys are not erased in GRFs, an adversary can steal their values via the register leakage vulnerability. Even if these registers get overwritten by other instructions in the AES program, we still have a chance to obtain a round key from the last-round XOR operation by the inverse of XOR: given one of the operands and the result of an XOR calculation, we can retrieve the other operand by taking the XOR of the known operand and the result. In AES, the XOR result, i.e., the ciphertext, has a high probability of remaining intact in registers since the iGPU needs to place outputs in the registers before sending them to the CPU, as mentioned in Section II-A. To conclude, an attacker can acquire the AES key from either the round keys or the data operand of the last XOR operation.

2. Register Flow Graph Generation: With the register leakage vulnerability, an attacker can capture the register state left by a victim GPU program. However, if the register mapping of the victim program is unknown, the attacker cannot determine the location of the sensitive variables. Worse still, variables are composed of long bytes that may irregularly scatter over many registers. Given the large size of the GRF, it is challenging to match registers with the variables in a GPU program. To solve the analysis difficulty, we develop a symbolic execution engine to produce a register flow graph of the GPU program. The graph contains information to reveal the register allocation as well as the data flow between registers. An example of the graph is depicted in Fig. 5(b). Each node in the graph represents a register write. A highlighted node means the written value is preserved until the program exits; in other words, these values are observable to adversaries due to the register leakage. By contrast, the data in white nodes are lost due to later writes to the same register. We record how the value is calculated in a node, while the edges between nodes help us to backtrace the source operands.

3. Register Mapping Analysis: With the help of the register flow graph, an attacker can enumerate every preserved register to find data of interest. In the case of AES, we can reduce the search space to the nodes with an XOR operator because they have a close relationship with our targets—the round keys and the input to the last XOR. In the example in Fig. 5(b), the attacker can learn one byte of the last round key. Specifically, we find that the round key in the r56 register is lost, but the r32 register divulges the XOR output. Hence, if the other input to the XOR operation exists, we can recover the key by the inverse of XOR. Though the required value in the direct parent r23 is missing, we notice it also resides in the ancestral node r26, resulting in the leakage of the key byte. After the analysis, the attacker can gain sufficient understanding of the register remnants to steal information about the victim.

4. Remnant Source Recognition: With the iGPU register spyware developed in Section III-A, an adversary can capture the register residues of other GPU programs. However, since the adversary is a non-privileged user who has no control over the GPU scheduling, the captured snapshots may be from many GPU clients. To precisely strike on a specific program, we need to filter out unrelated snapshots. With the help of the register flow graph, we can construct a rough fingerprint of the victim based on its invariable remnants of two types. The first type arises from constants loaded by the program. Alternatively, all-zero bits or all-one bits in registers, which are typically produced by mask instructions, could be helpful. For instance, in Fig. 5(b), the AND computation of the node r23 sets some bits of r23 to zero, which can be exploited in fingerprint construction if the value is not overwritten.

Based on the procedure above, we finally launch the key-recovery attack against the AES algorithm. In the experiment, while the attacker repeatedly executes the spyware to collect iGPU register residues, the victim launches AES tasks on the iGPU simultaneously. Once the captured data matches the fingerprint of the AES program, the attacker reads and reports the leaked bytes of round keys from the register dump. As summarized in Table III, our attack can steal most bytes of the secret key in AES-128 as well as in AES-196. With a pair of authentic plaintext and ciphertext blocks, it is computationally feasible to recover the entire AES key by brute force. On our machine, the brute-force attack can succeed within minutes, as shown in the last column of Table III.

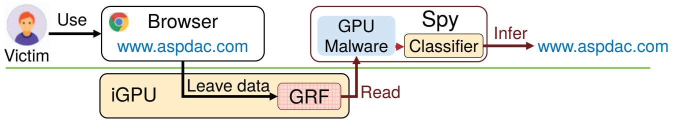

Fig. 6. Web-fingerprinting attack through iGPU

B. Attack II: Website Fingerprinting

The previous attack exhibits the power of an adversary when the implementation details of the program are available. In this subsection, we explore whether attacks are possible if the program is unknown. To this end, we demonstrate a web-fingerprinting attack against the prevalent Chrome browser. We take a stable Chrome release of version 73.0.3683.103 as the testbed, and we launch the browser with the default settings.

The attack overview is illustrated in Fig. 6. Modern browsers leverage GPUs to accelerate webpage rendering. As a consequence, when a user visits a website, rendering traces remain in the registers of the iGPU. Due to the iGPU leak vulnerability, these traces are exposed to adversaries. An attacker can train a machine learning (ML) model to monitor the browser activity, undermining the privacy of the victim. With the aim of a black-box attack, during the training of the classifiers, we do not adopt any prior knowledge of the rendering algorithms.

As a proof of concept, our attack targets the top 40 websites ranked in the world according to Alexa Top Sites [16]. Next, we elucidate our attack approach in the classic order of ML model development.

Training Set We collect a set of register dumps when visiting the frontpage of each website. During the visit, we imitate common viewing behaviors by hovering the mouse cursor randomly over the page, and we refresh the webpage several times. In consideration of the size of the GRFs, we cluster the register residues from one GPU thread as one training sample, meaning every snapshot of the iGPU yields up to 168 samples. The garnered samples are converted into feature vectors, as explained next in the feature construction step.

During data collection, we face two problems that could introduce noise if we use the naive register spyware developed in Section III-A. First, when the registers wake up from a power saving event, they provide our spyware with random numbers rather than leftovers of the browser. Second, due to the lack of register clearance, two consecutive snapshots may have strong correlation because the footprint of a previous rendering program may not be fully covered by an successor. In this regard, we instruct the spyware to reset the registers to zero before exit. Besides the benefit of sample correlation reduction, the clearance also helps to filter out samples from power saving events: when the spyware cannot detect any zeroed register, it

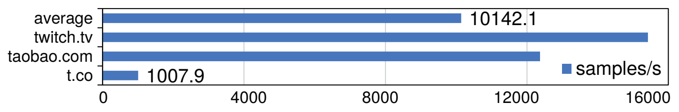

Fig. 7. iGPU utilization in terms of average sample harvest per second

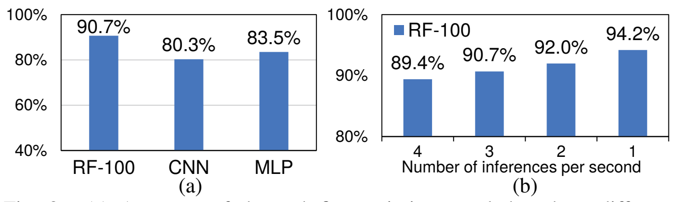

Fig. 8. (a) Accuracy of the web-fingerprinting attack based on different classifiers; (b) Inference accuracy at different time resolutions

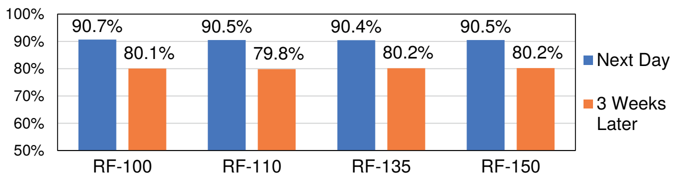

Fig. 9. Durability of the models against test sets collected at difference time

rejects the sample inasmuch as the power-on values of registers are most likely non-zero.

Feature Construction As said, we regard register residues of one thread from one GPU snapshot as one sample. However, due to the intentional blindness to prior knowledge, we do not know the best way to arrange the feature vector. Instead, we directly decompose the data of a sample into individual bytes, i.e. one byte constitutes one dimension of the feature vector. Lastly, we trim some of the dimensions with a low variance.

Pre-processing We observe that the browser produces identical register residues in some of the GPU threads despite the fact that the user views different websites. This imposes confusion in model training because identical samples are given different labels. We mitigate this issue by introducing a dummy class, and we move all misleading samples from their original classes to the dummy class. In this way, we intend to make the classifier focus on the unique characteristics of the websites, as their shared samples are categorized into a separate class. After this step, the dataset of the dummy class accounts for 7% of the whole training set.

Training Two popular machine learning frameworks are employed to develop classifiers. For the random forest (RF) model, we use sklearn [17]. For the convolutional neural network (CNN) and the multi-layer perceptron (MLP) model, we adopt keras [18]. The input shape of the RF and MLP classifiers is the same as the feature vector, while the input to the CNN is a matrix converted from the feature vector by reshaping. The classifiers learn to guess a website based on an input sample, or they report uncertainty by an output of the dummy class.

Inference We devise a two-step inference scheme to boost the accuracy of our attack. First, the classifier makes a guess for each sample. Then, we group a batch of guesses to vote for the final decision. In case the top voted guess is the dummy class, we select the guess in the second place to yield an informative result. We assume 3 inferences per second is a reasonable frequency for this attack, so we accumulate guesses every 0.33 second to form a voting pool.

Our scheme takes advantage of the extraordinary level of activity in iGPU threads. On average, an attacker can gather around 10142 samples per second when a victim browses different websites, as illustrated in Fig. 7. We also list the average sample production of three representative websites. Two of them require high GPU utilization owing to the complexity of their webpages, whereas the low-profile website has a simple page.

Test Set and Results The test set is built using the same process as the training set, followed by the feature construction step to match with the classifiers. For practical considerations, the test set is not collected on the same day as the training set but the next day. As shown in Fig. 8(a), our approach can achieve up to 90.7% accuracy at the resolution of 3 inferences per second, which implies the feasibility of the black-box attack. Among the three classification models, the RF model with 100 base estimators (RF-100) demonstrates the best performance. In addition, we test different inference frequencies on RF-100 in Fig. 8(b). As expected, we can reach even higher accuracy if a lower resolution is acceptable. Lastly, we evaluate the long-term performance of multiple RF models with a new test set gathered 21 days after the training of the models. Though the general layouts of the websites may remain unchanged, the content can have significant differences after 3 weeks. As illustrated in Fig. 9, the accuracy of our attack drops by around merely 10%, which implies that the attacker can reuse a trained model for a considerable amount of time.

C. Attack III: Covert Channel

The iGPU leak vulnerability constitutes a covert channel to exchange data. When the system prohibits direct communication between two entities, for instance, programs in isolated containers or virtual machines, they can abuse the iGPU to communicate. This is a violation of the access control of the system, and we evaluate this channel in this section.

In this attack, two cooperative iGPU clients repeatedly launch a GPU program to communicate. We design the program such that both sides can send and receive data simultaneously; therefore, the program first reads leftovers of the iGPU and then writes payload. For reliability, the payload is fragmented into packets. We refer to the design of the transmission control protocol (TCP) for the structure of the packet. Specifically, the header of our packets contains a sender identifier, a sequence number, an acknowledgment number, and a checksum. In case two identical iGPU jobs of the receiver are executed in succession, the sender identifier helps the receiver ignore the packets sent by itself. The sequence number assists the receiver to reassemble the payload from the sender. The sender decides the subsequent batch of packets to deliver according to the acknowledgment number received. The receiver verifies the integrity of the packet by the checksum, and corrupted packets are discarded.

We consider bothleakage channels, namely the GRF and the SLM. To transmit data via GRFs, we implement an iGPU program using assembly programming according to the procedure described in Section III-A. Each iGPU thread is designed to hold one packet of 3 KB, and the remaining 1 KB registers are reserved for program execution. On the other hand, we use OpenCL to program the SLM transmitter following the code in Table II. In both transmission programs, the checksum calculation and verification is accomplished on the iGPU for better throughput.

To validate our implementation, we realize a simple application layer protocol for file transfers on top of our iGPU packet protocol. We first transmit a 2 MB image, a small file of 128 B, and a 5 GB large file. Besides visual confirmation on the image, we verify that no bit error has occurred during the transmission because the sizes and hash values of the received files match those of the original files. Next, we quantify the bandwidths of the two channels by dumping the received data to the sink file `/dev/null`. We further consider two modes in the experiments: the simplex mode, in which only one end sends data while the other only receives; and the duplex mode,

TABLE IV
COVERT CHANNEL COMPARISON

	Device	Mechanism	Speed
This work	iGPU	Register (duplex)	8 Gbps
		Register (simplex)	4 Gbps
		SLM (duplex)	2.5 Gbps
		SLM (simplex)	1.3 Gbps
[19]	dGPU	Cache side-channel	4.3 Mbps
[20]	x86	Cache side-channel	4.0 Mbps
[21]	ARM	Cache side-channel	1.1 Mbps

in which both sides send and receive simultaneously. The measured bandwidths are presented in Table IV. In fact, the transmission rates are so high that if we do not write data to /dev/null, the bottleneck lies in the hard disk, which only has a continuous write capability of 1.8 Gbps. Comparing with other covert channels based on miro-architectures, our attack is faster by orders of magnitude.

V. COUNTERMEASURES

The vulnerability discovered in this paper reflects the concerning reality that security is often underestimated when new technology is introduced in the computer system. Memory isolation is a fundamental security principle in a multi-user setting; however, it is not properly enforced in iGPUs. We discuss the possible countermeasures to address the leakage issue, as below.

Kernel Patch By patching the graphics driver in the kernel, we can implement a clearance operation to wipe the traces of iGPU jobs. This eliminates the root cause of the leakage. However, according to our preliminary tests from user space, a GRF clearance costs around 9 μs, while an SLM clearance takes 15 μs. In view of the nontrivial overheads, we should not issue reset operations for every iGPU job. When the driver receives an iGPU job request, it could check whether the most recently queued job originates from the same context. Only when they differ should the driver emit a clearance command to the iGPU queue prior to the request.

Code Hardening Alternatively, the security fix can be implemented in the user space. For instance, the compiler of a userland driver can add cleaning instructions at the exit of a GPU program. Because the compiler possesses complete knowledge of the GRF and SLM allocation of a particular program, the cleaning code can safely bypass the unused registers and SLM space. However, we do not recommend this approach because the root cause still inhabits the system. Consequently, legacy programs with a customized userland driver remain vulnerable to attacks.

VI. RELATED WORKS

In the past several years, researchers have reported some studies about the security of dGPUs. Firstly, a memory leakage flaw was identified in [2] and [4]. Based on this vulnerability, an approach proposed by [22] could recover images displayed on the user's screen. Then, side-channel attacks were developed on dGPUs. For instance, the secret key of AES could be leaked from a cache-based timing side-channel [23]. In addition, the performance counter interface provided by the dGPU was demonstrated to be dangerous because it could be exploited as a side channel to track user activities [5]. The work [19] investigated the covert channels between concurrent dGPU kernels in depth. Lastly, the security threats of dGPUs were thoroughly discussed in [24].

The security implications of the iGPU have not attracted much scrutiny. In [25], the authors demonstrated that malicious iGPU software could accelerate the Rowhammer attack that inflicts bit errors on the DRAM. However, it is different from our work because their vulnerability roots in the DRAM rather than in the iGPU. Other research on iGPUs mostly focuses on functionality and performance. For example, an accurate simulator for the CPU-iGPU architecture was developed in [26]. DymGPU [27] optimized the performance of iGPU virtualization. Besides these, the internals of the iGPU compiler were introduced in [12] and [28].

VII. ACKNOWLEDGEMENTS

Intel confirmed the vulnerability and assigned CVE-2019-14615 as the public identifier. At the time of writing, Intel informs us that they are ready to release their mitigation. We appreciate Intel's professional handling of our report. We would also like to thank Intel for the feedback on our draft.

This work was supported by the Hong Kong University of Science and Technology Start-up Grant R9336.

VIII. CONCLUSION

In this paper, we disclose a crucial information leakage vulnerability on Intel integrated GPUs. We demonstrate that it can be exploited by adversaries to infringe the security and privacy of benign GPU users. A series of end-to-end attacks are developed to highlight the threat. The attack vector results from the flawed context management of GPU workloads.

REFERENCES

[1] V. Eduardo *et al.*, "Speculative Encryption on GPU Applied to Cryptographic File Systems," in *Proc. of USENIX FAST*, 2019.
[2] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities," in *IEEE S&P*, 2014.
[3] M. Abadi *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *Proc. of USENIX OSDI*, 2016.
[4] R. D. Pietro *et al.*, "CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix," *ACM TECS*, vol. 15, no. 1, pp. 15:1–15:25, 2016.
[5] H. Naghibijouybari, A. Neupane, Z. Qian *et al.*, "Rendered Insecure: GPU Side Channel Attacks are Practical," in *Proc. of ACM CCS*, 2018.
[6] J. Peddie and R. Dow, "Global GPU shipments mixed in Q1'19 reports," Jun 2019, https://www.jonpeddie.com/store/market-watch-quarterly.
[7] Intel, "Intel® Open Source HD Graphics, Intel Iris™ Graphics, and Intel Iris™Pro Graphics Programmer's Reference Manual," May 2016.
[8] "The Compute Architecture of Intel® Processor Graphics Gen9," 2015.
[9] "Intel(R) Graphics Compute Runtime for OpenCL(TM)," https://github.com/intel/compute-runtime.
[10] "OpenGL Overview," https://www.khronos.org/opengl/.
[11] "CUDA Toolkit," https://developer.nvidia.com/cuda-toolkit.
[12] A. Chandrasekhar, G. Chen, P. Chen, W. Chen, J. Gu *et al.*, "IGC: The Open Source Intel Graphics Compiler," in *Proc. of CGO*, 2019.
[13] Intel, "Intel® OpenSource HD Graphics Programmer's Reference Manual - Execution Unit ISA (Ivy Bridge)," May 2012.
[14] G. Johannes and P. Margara, "engine-cuda / engine-opencl," https://github.com/heipei/engine-cuda, 2012.
[15] "OpenSSL," https://www.openssl.org/.
[16] "Top Sites - Alexa," May 22th 2019, https://www.alexa.com/topsites.
[17] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[18] F. Chollet *et al.*, "Keras," https://keras.io.
[19] H. Naghibijouybari, K. N. Khasawneh *et al.*, "Constructing and Characterizing Covert Channels on GPGPUs," in *Proc. of MICRO*, 2017.
[20] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *Proc. of DIMVA*, 2016.
[21] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice *et al.*, "ARMageddon: Cache Attacks on Mobile Devices," in *Proc. of USENIX Security*, 2016.
[22] Z. Zhou, W. Diao, X. Liu *et al.*, "Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU," *Proc. of PETS*, 2017.
[23] Z. H. Jiang, Y. Fei, and D. Kaeli, "A Complete Key Recovery Timing Attack on a GPU," in *Proc. of HPCA*, 2016.
[24] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel *et al.*, "Understanding The Security of Discrete GPUs," in *Proc. of GPGPU*, 2017.
[25] P. Frigo, C. Giuffrida, H. Bos *et al.*, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *Proc. of IEEE S&P*, 2018.
[26] P. Gera, H. Kim *et al.*, "Performance Characterisation and Simulation of Intel's Integrated GPU Architecture," in *Proc. of ISPASS*, 2018.
[27] Y. Park, M. Gu *et al.*, "DymGPU: Dynamic Memory Management for Sharing GPUs in Virtualized Clouds," in *Proc. of FAS*W*, 2018.
[28] W.-Y. Chen, G.-Y. Lueh, P. Ashar, K. Chen, and B. Cheng, "Register Allocation for Intel Processor Graphics," in *Proc. of CGO*, 2018.