Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang*, and Rui Liu

Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU

Abstract: According to previous reports, information could be leaked from GPU memory; however, the security implications of such a threat were mostly overlooked, because only limited information could be indirectly extracted through side-channel attacks. In this paper, we propose a novel algorithm for recovering raw data directly from the GPU memory residues of many popular applications such as Google Chrome and Adobe PDF reader. Our algorithm enables harvesting highly sensitive information including credit card numbers and email contents from GPU memory residues. Evaluation results also indicate that nearly all GPU-accelerated applications are vulnerable to such attacks, and adversaries can launch attacks without requiring any special privileges both on traditional multi-user operating systems, and emerging cloud computing scenarios.

Keywords: GPU, Memory Management

DOI 10.1515/popets-2017-0016
Received 2016-08-31; revised 2016-11-30; accepted 2016-12-01.

1 Introduction

Modern graphics processing units (GPUs) have already been utilized in a broad spectrum of application domains, from graphic processing to bioinformatics and from matrix manipulation to neural networks, which however has inevitably introduced new security problems. Specifically, to circumvent memory bandwidth bottlenecks, discrete GPUs are normally equipped with dedicated high-speed memory systems that are managed and used exclusively by those GPUs; those memory systems are independent of the main memory systems controlled by CPUs. However, this heterogeneous memory architecture introduces a memory isolation issue: Memory isolation policies enforced by a CPU cannot be applied to GPU memory automatically, and any discrepancy between these two can lead to unwanted information leakage.

This issue has already been reported by previous works. Security implications and potential information leakages from GPU memory were first discussed in 2012 [10]. Some recent studies have revealed several concrete attacks against GPU memory [14, 18, 20, 21, 24]. In particular, Lee et al. demonstrated that adversaries could infer the websites visited by victims on a multiuser system by using statistical analysis of pixel colors [20].

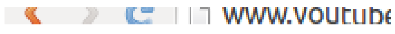
Threats from GPU memories were mostly overlooked. As far as we know, no mainstream GPU vendor released any patch to fix the problems discovered in previous studies. We believe that the fundamental reason is that the existing attacks to GPU memories have very limited impacts and consequences and thus are considered to be low-risk. For example, how data are exactly organized in GPU memories is still unknown because very little documentation is available, and existing attacks rely on side-channel attacks to infer noncritical information indirectly.

After a careful investigation to the problem, we argue that **potential security risks caused by contemporary GPU memory architectures are underestimated, and serious attacks may recover highly sensitive information directly from GPU memory residues by exploiting the GPU memory management vulnerability**. To this end, we studied how data are stored in GPU memories, propose a novel algorithm for recovering fragments of original images rendered by GPUs but left behind as memory residues, and demonstrated direct sensitive information recovery with several real-world applications including Google Chrome and Adobe PDF Reader. Evaluation results showed that adversaries are indeed able to obtain highly sensitive data , including images displayed on screens, text samples from documents, and matrix data.

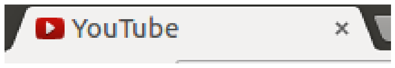
Zhe Zhou: Chinese University of Hong Kong, E-mail: zz113@ie.cuhk.edu.hk
Wenrui Diao: Chinese University of Hong Kong, E-mail: dw013@ie.cuhk.edu.hk
Xiangyu Liu: Chinese University of Hong Kong, E-mail: lx012@ie.cuhk.edu.hk
Zhou Li: ACM Member, E-mail: lzcarl@gmail.com
***Corresponding Author: Kehuan Zhang:** Chinese University of Hong Kong, E-mail: khzhang@ie.cuhk.edu.hk
Rui Liu: Chinese University of Hong Kong, E-mail: ruiliu@ie.cuhk.edu.hk

Several critical challenges must be addressed to extract raw data from GPU memory residues directly. First, identifying small image-like data chunks, ranging from several kilobytes to several megabytes; such chunks are usually taken from a memory space of several gigabytes. Second, inferring images' layouts (i.e., widths and heights) is vital, otherwise recovered images may be meaningless. Nevertheless, all types of meta information about images are missing from GPU memory, which mostly contains quantities of raw data for all pixels mingled with other nonimage data. Our strategy to address the first challenge entailed leveraging some unique patterns of adjacent bytes for image pixels. For the second challenge, we designed a novel algorithm based on a key insight regarding image data: Adjacent rows within an image are similar to each other, and when examined in the frequency domain, such similarities exhibit some cyclical patterns, with cycles usually equal to widths (i.e., the length of each row). With our algorithm, the memory layout of an image can be inferred and recovered without any quality loss.

The techniques proposed in this paper proved that GPU memory inference attacks are much more serious than researchers had previously thought. **First**, currently, adversaries are able to recover raw images from GPU memory residues and then extract much more sensitive information directly from such raw data (e.g., the email contents displayed in a browser that requested a GPU to render web pages). By contrast, previously, such adversaries can only perform side-channel attacks over nondecoded data and infer less critical information indirectly [20]. **Second**, adversaries can now attack a large group of vulnerable applications with the GPU-acceleration feature, because they are able to recover raw images rendered by GPUs (because of acceleration), whereas previously, their attacks could only infer insensitive information indirectly through statistical analysis of memory residues. In Section 5, four popular applications from different categories, namely Google Chrome (browser), Adobe Reader (document processor), GIMP (image processing) and Matlab (scientific computing), are evaluated. The results show that all of those applications are vulnerable in the end, but previously published attacks were only able to work partially on browsers (by inferring the web sites visited by victims). Considering that GPU acceleration is becoming increasingly popular in mainstream applications, including even production software like Microsoft Office and Libre Office, one can expect that sensitive documents and data could be increasingly shifted to GPUs for processing, resulting in widespread accidents of information leakage. **Finally**,



(a) Address Bar



(b) Tab Caption

Fig. 1. Recovered image From GPU memory indicating the URL of the displayed web page

with the techniques proposed in this paper, adversaries are able to launch attacks not only against traditional multiuser systems, but also against new and emerging platforms such as cloud computing (discussed in Section 5.3), which increases the victim population and makes the attack more serious than before.

Contributions. We summarize our contributions as follows:

- *A study that reveals a critical vulnerability in the GPU memory management mechanism.* We found that the GPU memory management strategy can be exploited by malicious programs that cross the memory isolation boundary to obtain raw memory data belonging to other processes; this is highly risky and leads to much more serious consequences than researchers expected before.
- *A novel methodology for recovering original images from GPU memory residues.* We propose a new approach that can automatically identify and recover images from the GPU memory residues of legitimate applications. Based on our insights on image data and signal processing techniques, we designed an algorithm that can determine image layouts and extract images effectively. Unlike previous works on GPU security problems, our approach is the first to show that original images and sensitive information can be recovered from GPU memory residues.
- *In-depth evaluation.* We evaluated our attacks against popular applications with typical uses of GPUs, including web browsing, document processing, photo editing, and scientific computing. The results show that all of those applications are vulnerable to such attacks. In addition to ordinary multiuser systems, we further tested our approach on a virtualized platform and noted successful results.

2 Background

In this section, we first provide a brief overview of a GPU computing model, and then present a summary of reported vulnerabilities in GPU memory management.

2.1 GPU Computing Model

Functions of modern GPUs have already exceeded displaying images. GPUs can now be used for diverse tasks, such as video encoding and decoding, page rendering, and other general-purpose computations. To effectively accomplish these new tasks, GPUs' internal architectures and computing models have evolved rapidly. Currently, it is typical for a single GPU chip to have tens or even hundreds of processing units (or cores) that can work in parallel and provide a staggering level of processing power. To concurrently satisfy the memory demands of numerous cores, a discrete GPU normally includes a dedicated memory space that is used and managed exclusively by the GPU and is independent from the main system memory (controlled by the CPU). Therefore, before the GPU can actually start to work, relevant data must be copied from the main memory to the GPU memory, after which computations can be executed in GPU memory; finally, results can be copied back from the GPU memory to the main memory.

Different sets of APIs exist to facilitate the use of GPUs and attached memory resources, including OpenGL, CUDA and OpenCL. OpenGL encapsulates low-level GPU operations so that applications can render graphics without directly touching GPU memories. By contrast, OpenCL (an open framework supported by most GPUs including Nvidia and AMD.) and CUDA (exclusively supported by Nvidia) provide low-level interfaces that enable users to directly manipulate GPU memories in almost the same manner as system memories (e.g., allocating and freeing memory objects). All of the exploits presented in this paper were engineered and evaluated with OpenCL and CUDA.

2.2 GPU in Virtualized Environment

Previously, GPUs were not virtualized and could not be used by virtual machines (VMs), but new techniques now have enabled VMs to obtain full access to GPU resources. Specifically, with the help of new I/O virtualization schemes, applications running on cloud computing platforms are now able to use powerful GPUs on the physical machine [6]. The GPU virtualization technique is called *GPU Passthrough*; in this technique, one GPU is assigned to a single VM instance that uses it in the same manner as a directly attached GPU card [21]. Although this scheme differs slightly from the sharing of other resources like network and hard disks, it has the least overhead and can reach bare-metal GPU performance. However, as discussed in Section 5.3, it also engenders potential problems of GPU memory residues, because the data left in the GPU memory by one VM could be accessed and analyzed by another VM.

2.3 Vulnerabilities in GPUs

GPUs were originally designed to provide high-speed computation power, but the designers gave insufficient consideration to data security, and as computing models and application scenarios evolve, the underlying security implications change accordingly, leading to new vulnerabilities. One prominent example is GPU memory management. Being independent of the main system memory, the GPU memory is managed by the GPU itself and thus may violate some security policies normally enforced by the OS and the CPU. For example, it is guaranteed that any newly allocated space in the main memory must be cleared to zero to avoid information leakage, but such policies are not enforced by the GPU, because a previous study demonstrated that newly allocated GPU memory without application-level initialization returns nonzero values[10].

Some studies have been published on the security implications of the architectures and features of modern GPUs. For example, Lee et al. [20] found that: although they could not decode or recover original images from GPU memories, they were able to infer the sites visited by victims, by linking a targeted memory trunk with the most similar one in a labeled training set. Nevertheless, this attack is limited by some restrictions. First, it cannot recover raw images from GPU memories and must rely on side-channel information. Second, it requires attackers to build profiles beforehand and thus is only applicable to limited scenarios of matching a target with an entry from a known dataset.

In this work, we report a major advance in this field. By analyzing and leveraging the common memory layout of GPU objects (i.e., images), we developed new techniques for recovering raw images from GPU memory residues. With our techniques, highly sensitive information can be directly extracted and harvested by adversaries. The present paper explains how to obtain data that previously published techniques were not able to obtain, including usernames, credentials, credit card numbers, and other sensitive information shown on web. Because an increasing number of mainstream applications are using GPU acceleration on workstations and cloud servers, hitherto unforeseen attacks on GPU memory could become widespread and dangerous.

3 Adversary Model

We assume that an adversary has successfully acquired the permission to access the GPU and run programs under a **nonprivileged** account on the target machine, and that his/her goal is to recover meaningful information from the GPU memory residues of other victim users (e.g., who share the same hardware but with different accounts). Several common scenarios satisfy the preceding assumptions. For example, in a school computing lab, each user might have a nonprivileged account on every computer. Suppose a victim had just used one lab computer to view some personal PDF documents (with GPU acceleration); another malicious user can dump the GPU memory and recover whatever information had been retained in the GPU memory. Shared computers are also common in companies. Consider a dedicated computer only used to view and process classified documents; a user with a low level of authorization might be able to recover sensitive information left in the GPU memory by a user with a high level of authorization who had just viewed secret documents on that shared computer.

For the scenario of cloud computing, we assume that adversaries have rented a VM with GPU support, and that their goal is to obtain private information from other cloud users (i.e., owners of other VMs on the same machine) through the GPU memory. Such settings are also common currently. For example, companies may use cloud servers with GPU acceleration to tag people in photos uploaded by their customers, convert document formats, or even perform scientific computations. Moreover, adversaries can obtain a VM on the same host machine, dump the GPU memory, and recover raw data such as photos, documents, and experimental data.

4 Attack: Image Recovery

In this section, we propose a method for recovering graphical data from GPU memory residues. We first present techniques regarding how to identify image-like tiles from GPU memories. Subsequently, we describe how to reconstruct images from tiles with imprecise boundaries. Finally, we demonstrate how to rearrange recovered images in the correct order.

4.1 Tile Extraction

The first step of the attack is to identify possible data blocks that are very likely to be parts of meaningful images, but this turns out to be a non-trivial task for two reasons. First, images' meta data are often stripped away when the images are loaded into GPU memories; thus, many relevant details about image objects are missing, and there is no simple means of locating the metadata or determining the dimensions of the images. Second GPU memories also hold considerable amounts of nonimage data, because modern GPUs are frequently used for nongraphical computations such as encryption and matrix calculation, which makes the problem even more complex and difficult. To address these challenges, we leverage several distinctive features of image data and have improved the prime-probe method proposed in the previous study [20]. More details are elaborated below.

Memory initialization. In this step, a differential analysis is executed over a piece of GPU memory to recognize regions that have been changed by victim applications. When started, the malicious application attempts to overwrite every byte of the GPU memory it can access with a predefined constant value, say 0xff; subsequently it focuses on all regions with non-0xff values, because only those regions may contain changes made by victim applications.

Data block extraction. After the memory is initialized, the malicious application then runs in the background and queries the amount of free GPU memory space periodically. If the size of the available GPU memory sharply increases, a victim application may have just released a chunk of GPU memory, and the malicious application makes a copy of the GPU memory using GPGPU APIs (e.g., `clEnqueueReadBuffer` in OpenCL). The probability of adversaries spotting sensitive information from the captured GPU memory is high because the GPU itself does not erase memory data before released memory space gets reallocated.

One may think that image regions can be extracted easily from GPU memory dumps by removing all the 0xff bytes. However, such a simple approach does not work effectively, because 0xff is also used as a valid pixel value as well as an alpha component. Removing all 0xff values naively would destroy original image data structures and break images into small pieces.

The observation that eliminating 0xff bytes would destroy data motivated us to develop a technique for stitching small image memory blocks together based on their relationships. Our strategy is to divide memory dumps into blocks of a fixed size, and then merge the ones that are consecutive in memory space and are likely to be pixels. The block size must be determined first. The size should be smaller than an image; otherwise, two or more images would appear in the same block and be regarded as one image. The block size must not be too small; otherwise, any big white chunk (in which all bytes were $0xff$ trunk) in an image would break the image down. After numerous experiments, we chose 4 K as the block size that works well in most cases. Therefore, we split the memory into 4-K blocks and filter out the blocks that are filled with all $0xff$ because they are probably not used after initialization. We also remove the blocks that are all $0x00$ because they are clean blocks zeroed out by developers or the OS. The remaining blocks are **concatenated into a bigger block** if they are consecutive in memory space, and we call it a tile. After this step, data blocks left behind by victim processes are extracted.

This method may still accidentally damage some valid pixels, since 0xff may also be a valid pixel, but this does not cause considerable problems, because it is equivalent to removing large chunks of white space in an image; the cropped image is still meaningful, just like removing white margins of a PDF file retains the meaningful text. By contrast, removing nonwhite pixels may destroy excessive amounts of information and make images difficult to recognize. This is exactly the reason why we chose 0xff (i.e., white color) as the canary.

The structure of the graphical data inside the tile is unknown at this point. As mentioned, GPU chip vendors do not provide any documentation about how they map a logical address to its physical address space. However, it seems reasonable to assume that application developers do not arbitrarily distort image layouts in logical address spaces under their control. Furthermore, although a GPU may conduct extensive optimization of its underlying data storage architecture (e.g., memory compression [3]), we found that such hardware optimization techniques are all transparent to upper layers, suggesting that regardless of how data are re-ordered, compressed, and encoded by GPU hardware, data seen by GPU applications retain their original formats.

Data blocks pruning. The blocks obtained in the previous step still require further pruning - any blocks used to hold nongraphical data are not considered in the present procedure. The distinctive structure of image data facilitates the identification of graphical blocks. For an image, each pixel is represented by a 4-byte word, or four 8-bit *channels*. The four channels correspond to the colors Red (R), Green (G), Blue (B), and Alpha component (A) value of the pixel separately. The alpha component value indicates the level of transparency of the pixel. By surveying numerous images, we found this value to be either $0x00$ or $0xff$, and if either of those values can be detected, we can infer that the pixel is either nontransparent or the channel is unused. Hence, we can judge if the data block is graphical by examining its alpha channel values.

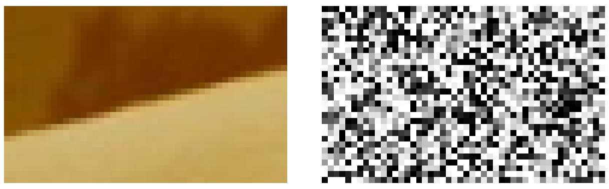
For a graphical data block, we also must determine the order of channels to guide the subsequent reconstruction steps. In theory, developers can choose any order, but they usually use the first or last byte of the 4-byte word as the alpha channel and sequentially align RGB values, because the pixel format is usually either RGBA or ARGB. To determine which format is used, we compute the percentage of $0xff$ or $0x00$ stored in each byte position of each 4-byte word (p), and compare it against a threshold (th). If $p > th$ for the last byte, the image format is considered as RGBA. If $p > th$ for the first byte, the image format is considered as ARGB. Otherwise, the block is discarded. In the evaluation, th is set to 20%.

Occasionally, developers may not use a standard format to represent a pixel, and we did encounter one such case, which is introduced in Section 5.

The final task of data pruning is to remove the heading and trailing elements filled with values of $0x00$ or $0xff$ and to retain the values in the middle (This does not mean that the boundary is now precisely determined). Ideally, a tile contains one and only one image; this was proved to the dominant case by our pilot experiments. However, we did observe a small portion of tiles containing parts from two or more images, because the locations of the included images in the memory space are too close.

4.2 Image Layout Inference: Problem

After a tile has been extracted, the next step is to infer the layout information associated with the embedded image. Assume that the tile occupies N 4-byte words in memory and the image occupies a sequence of W 4-byte words ($W \leq N$). The image could reside in any subarea of the tile. We denote the number of 4-byte words ahead of the image as s and the number after as e and $N = s + W + e$. We must identify s and e to retrieve the subarea. Since an image is represented with a two-dimensional matrix, we must also identify the number of rows (n) and columns (m, which equals W/n) to recover the original image.



(a) Normal Image **(b)** Signal-less TV

Fig. 2. Normal Image and Random Image

In general, the size of an image ranges from several kilobytes to several megabytes. It is infeasible to enumerate all the combinations of s, e and n and then let the attacker decide which combination can recover the original images. However, an image (especially a sensitive one) is quite different from other artificially generated data: **Strong similarities exist between consecutive rows and consecutive columns** (Figure 2a). Another favorable condition is that although an image could be compressed when stored on a hard disk or transmitted through the network, it is decompressed and usually loaded into a matrix structure into the GPU memory and the similarities are preserved. Transparent memory acceleration techniques may not break the similarities because the hardware guarantees that the upper layer application does not observe any distortion caused by performance optimization. We leverage this key insight to infer n or m and henceforth s and e (the details are described in Section.4.3). Our approach, however, is not designed to recover randomly generated images that resemble television screens with no signals (Figure 2b) or an image filled with identical pixels. These types of images usually do not contain sensitive information and are disregarded by the proposed exploit.

4.3 Image Layout Inference: Approach

When processed by a GPU, an image is usually stored in a two-dimensional matrix (denoted by a), and the value of a pixel can be read from $a[i, j]$, where i and j denote the i_{th} row and j_{th} column in the image matrix, respectively. However, a tile is just a sequential data block represented by a one-dimensional vector (denoted f) where $a[i, j] = f[s + i \times m + j]$. Our goal is to infer the correct s and m values that satisfy this equation. As stated in Section 4.2, the consecutive rows of an image are similar ($a[i, :]$ is similar to $a[i+1, :]$, for $0 \leq i \leq n-2$), and we leverage this constraint to determin the correct s and m.

An appropriate metric is still required for quantifying the similarities between rows. By examining different metrics, we found that the most useful metric is the amplitude spectrum in the frequency domain of the image matrix. If m is correctly inferred, the distribution of element values in each row should be similar, leading to strong periodicity of row values. In the subsequent paragraphs, we introduce an algorithm that first infers m and then derives s based on m. Our approach is demonstrated on four types of tiles, arranged in order of increasing processing difficulty. For each type, we remove one constraint from the previous type and the final type reflects the tiles extracted from genuine GPU memory dumps. Finally, we solve the number of redundant 4-byte words before and behind the image (s and e). Throughout this section, we use the tiles shown in Figure 3 as examples to demonstrate our approach.

Tile type I. We start from an easy case where the similarity can be trivially quantified. Assume that there are no trailing and leading redundant pixels and that all rows are identical ($s = 0$, $e = 0$ and $\forall i_1, i_2 \in [0, n-1], a[i_1, :] = a[i_2, :]$). Figure 3f illustrates this type of tile, in which one row filled with distinct pixels (Figure 3e) is duplicated three times. In this case, f is turned into a periodic function where $f[x] = f[x + m], \forall x \in [0, (n-1) \times m - 1]$ and m is the interval. Here, we leverage the spectrum produced by an **FFT (fast fourier transform)** algorithm to capture this interval. Fourier transforms can decompose a signal from the time domain into the frequency domain and are widely used in various applications including signal processing and image processing[4, 5].

Let the *amplitude spectrum* of f produced by the FFT be F (pixels are gray-scaled before FFT). We studied F and found that the interval between two nonzero components equals n, the number of rows. Therefore, in this case, the image can be easily recovered, and the proof for this observation is demonstrated as follows:

F for this tile is illustrated in Figure 3b, and Figure 3e shows the spectrum of only a row $F_0(k)$. $F(kn) = F_0(k)$ for $k = 0, 1, 2, ..., (m-1)$ and $F(kn)$s are nonzero (we call them main components), whereas the the amplitudes for other components are zero, according to the properties of periodical signals. Hence, the interval between two neighboring main components equals the number of rows (n) of the image matrix. The width m can be computed through N/n, and the image is then recovered by reshaping the tile with those parameters.

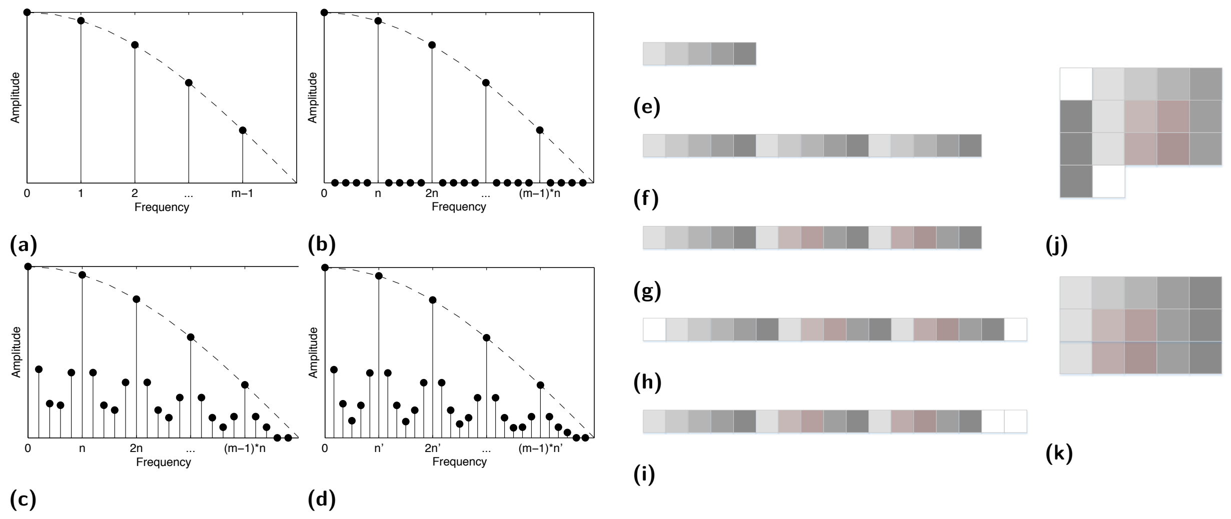


Fig. 3. Images used to illustrate three types of images. (a) Spectrum of a row picked from an image (illustrated in (e)). (b) Spectrum of an image with identical rows (f). (c) Spectrum of an image (g) that complies with our assumption that consecutive rows are similar. (d) Spectrum of an image with trailing or leading white pixels (h, i). (j) Image recovered without considering the leading pixels. (k) Perfectly reshaped image.

Tile type II. A general property of images encountered in this study is that all rows are not always identical. Therefore, we assume that neighboring rows are similar but not always identical. We assume tiles of this type have no redundant leading pixels and no redundant trailing pixels. We determined that again the FFT can be used to infer the number of rows and columns of the image matrix.

As an example, we assume that the tile looks like that in Figure 3g, and the amplitude spectrum F of the tile vector f is illustrated in Figure 3c. The main components of $F(k)$ occur when $k = 0, n, 2n, ..., (m-1) \times n$, which is the same as the spectrum of tile type I. However, because of the differences between two neighboring rows, the main components disperse and the amplitudes of the nonmain components are greater than zero now. Nevertheless, they are much lower than those of the main components; thus, the main components can be easily identified. We explain the scenario as follows:

We introduce n virtual images $v_1, ..., v_n$, all of which have the same layout as the original image. In particular, v_i is constructed by replicating the i_{th} row of a for n times; thus, $\forall i, x \in [0, n-1], v_i(x, :) = a[i, :]$. Let the amplitude of v_i for the k_{th} element be $V_i(k)$; obviously, it equals $F_i(\frac{k}{n})$, according to the previous analysis, if k is a multiple of n. For a sample tile shown in Figure 3g, the value of an element can be represented by $f(i \times n + j)$ and also by $F_i(k)$ through inverse FFT, as shown in Equation 1.

$$f(i \times n + j) = v_i(x, j) = \frac{1}{N} \sum_{k=0}^{N-1} W_N^{-k(xm+j)} V_i(k)$$
$$= \frac{1}{N} \sum_{k=0}^{N-1} W_N^{-k(xm+j)} F_i(\frac{k}{n}) = \frac{1}{N} \sum_{k=0}^{m-1} W_N^{-knj} F_i(k) \tag{1}$$

where W_N is the twiddle factor.

Equation 1 suggests that f consists of subcomponents, and the frequency of each sub-component is a multiple of n. Based on our observation that the consecutive rows vary slightly, the differences between $F_i(k)$ and $F_{i+1}(k)$ should also be small, and the combined $F(k)$ should be high when k is a multiple of n. For other values of k, the combined $F(k)$ is still low, which makes the main components stand out at $0, n, 2n, ..., (m-1) \times n$. Similarly, the interval between two main components is calculated to derive the values of n and then m.

Tile type III. Next, we consider a tile with a block of pixels ahead of and another block of pixels trailing the original image object; the value of each element in the blocks is zero. As stated in the theorem of DFT [4], the amplitude spectrum does not change when the original signal is circularly shifted. Accordingly the leading block ahead of the image, if any, can be shifted to its end without affecting the spectrum. Figure 3h illustrates an image with both leading and trailing blocks, and Figure 3i illustrates an image with only a trailing block; moreover, their spectra are the same (Figure 3d). We also assume that the trailing block is filled with zeros to avoid the effect of the disturbance of nonzero padding on the original spectrum under this simplified condition.

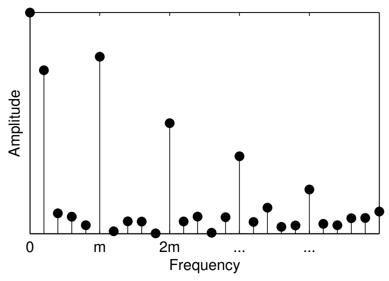


Fig. 4. Spectrum of $F(k)$ generated from tile type III

In signal processing, padding zero values to the end of the signal can enhance the resolution of the spectrum. Specifically, the number of points used to observe the spectrum increases. In this case, after the FFT, the interval between two consecutive main components is no longer equal to the height (n) of the original image. For instance, the main components are located at kn' in Figure 3d instead of kn in Figure 3c, although the sizes of these two images are the same. However, theoretically, the number of the main components is not changed, which can be used to infer the width (m) of the image. Nevertheless, this approach is not robust when the main components are not prominent (e.g., the spectrum of the component at $(m-1)*n'$ is close to the neighboring components in Figure 3d). **We solve this problem through another round of FFT over $F(k)$ and select the main component with the highest spectrum from the result, denoted as $F_{F(k)}$.** This approach works because the main components occur every n' points, which suggests that the frequency occurrence is m. Figure 4 illustrates the spectrum of Figure 3d. Clearly, the component with the maximum amplitude is located at m after low frequency components have been filtered out by an HPF (high pass filter). Even though the main components at km ($k > 1$) also show much higher amplitudes than the neighboring components do, their amplitudes are lower than the amplitude of the component at m. This can be explained from the nature of images: the similarity between a pair of interleaved rows should be relatively high but still lower than that between a pair of consecutive rows. After m is computed, n can be derived by dividing N by m. The derived n might be inaccurate since the tile in this case contains more elements than the image object, which must be adjusted in the subsequent padding removal step.

Prior to picking out the main components from $F_{F(k)}$, we remove the low-frequency components first (achieved using HPF). The low-frequency components could have high spectrum amplitudes (e.g., when the frequency is 0, their values are higher than the value at m), because neighboring pixels are all similar, which results in numerous low-frequency components. We use a threshold here to prune the low-frequency components. An excessively high threshold (more than m) would directly filter out the right answer, whereas an excessively low threshold would result in the selection of a wrong main component. We set it to the first frequency point with an amplitude below the mean value, because experience has shown low-frequency components decrease very rapidly. Their values tend to decrease below the mean value within some points, whereas the value of the first main component is much higher than the mean.

Nonzero Paddings. We assume that the padding digits before and after the image are all 0 in the previous simplified case, but that does not hold for all tiles extracted from real-world GPU dumps. Each digit could be filled by any value. The spectrum changes when the digits exhibit different periodicity lengthes and when the tile is sufficiently long. The chances of a spectrum change, however, are low, given that the quantity of meaningful pixels is usually considerably higher than that of the paddings. Therefore, the approach for tile type III can be applied to calculate m for this case (both leading and trailing paddings are nonzero) without any modification.

Padding removal. Finally, we propose strategies for computing s and removing the leading block. If the leading block is not removed, the recovered image cannot be correctly aligned; an example is shown in Figure 3j. We elaborate how to derive s for reshaping an image (Figure 3k) in the following paragraphs.

The present computation is derived from the premise that consecutive columns can be expected to be similar. Assume that the elements of a tile are placed sequentially into a matrix after m is correctly inferred (Figure 3j). To transform this matrix into the original image matrix, each row of the matrix should be shifted left for $s \ mod \ m$ elements if s elements are posited ahead. We aim to calculate $s \ mod \ m$ and remove those paddings.

The first and the last columns of the tile matrix should be quite similar, because they are in fact the $(m-s)_{th}$ and the $(m-s-1)_{th}$ columns in the original image. On the other hand, the s_{th} and $(s-1)_{th}$ column of the tile matrix should be quite different, because they are in fact the first and last columns (or boundaries) in the original image, respectively.

We utilize the preceding findings to design the following algorithm for inferring s. First, a distance array

Algorithm 1 ImageRecover

```
Input: $f$;                                             ▷ $f$ is the tile
$F_1 \leftarrow abs(fft(f))$;
$F_2 \leftarrow abs(fft(F_1))$;
$F_m \leftarrow mean(F_2)$;
$F_2 \leftarrow F_2/F_m$                                ▷ Normalization
$cutFreq \leftarrow locateFirstSmallerThanOne(F_2)$;
$F_2(0 : cutFreq - 1) \leftarrow 0$;                    ▷ High Pass Filter
$m \leftarrow locateMax(F_2)$;
if $F_2(m) < \theta_0$ then
    $Throw(notEnoughLength)$
end if
$N \leftarrow length(f)$;
$n \leftarrow \lfloor N/m \rfloor$;
$a \leftarrow reshape(f(0 : n * m - 1), n, m)$;

for $i \in [0 : m - 1]$ do
    $dist[i] \leftarrow distance(a(:, i), a(:, (i + m - 1) mod\ m))$
end for
$dist \leftarrow dist/mean(dist)$;                      ▷ Normalization
$s \leftarrow locateMax(dist)$;
$s' \leftarrow locateSecond(dist)$;
if $dist(0) < \theta_1$ && $dist(s)/dist(s') > \theta_2$ then
    $n \leftarrow \lfloor (N - s)/m \rfloor$;
    $a \leftarrow reshape(f(s : s + n * m - 1), n, m)$;
end if
Output: $a$;
```

dist is established, where the ith element stores the distance between the i_{th} and the $(i-1)_{th}$ columns ($\forall i \in [1, m-1]$) of tile matrix, and $dist[0]$ stores the distance between the last and the first columns. The distance between two columns is calculated by counting the number of element pairs of which the differences are greater than a predefined threshold θ_3. If $dist[0]/mean(dist) < \theta_1$ and $max(dist)/second(dist) > \theta_2$ (θ_1 and θ_2 are two thresholds), there exist s leading elements, and s is set to be the index of the maximum element in *dist*. If the first check using θ_1 is satisfied, the first and last columns are reasonably similar and should be located in the middle of the original image. If the check with θ_2 is satisfied, a column pair in the middle of the matrix somewhere has a distinctively low similarity and should be the real image boundary. Subsequently, we remove the first s elements from the tile and reshape the tile to a matrix with width m and height $(N-s)/m$. Figure 3k illustrates the final image.

Notably, we do not attempt to infer the positions of trailing blocks and remove them, because the trailing blocks only introduce additional lines below the image when displayed. Similarly, when $s > m$, our algorithm removes $s\ mod\ m$ elements and leaves additional lines above the image. These additional lines would not prohibit an adversary from recognizing the texts and objects.

Algorithm and parameters. The whole algorithm, including the tasks of preprocessing, identifying the numbers of rows and columns, and removing the leading block, is presented in Algorithm 1. We determined that if the input tile f is not long enough, the main components may be overwhelmed by other components. Therefore, we define a parameter θ_0 and set it to 1.5 in our evaluation. This can be used to warn attackers when the main component is not high enough. When the first main component is less than the threshold, the tile inferred is likely to be incorrect and is marked with the label "potential false-positive" before it is sent out to the attacker. In this study, because preliminary tests provided empirical evidence regarding acceptable parameter values, the other three thresholds namely, θ_1, θ_2 and θ_3 were set to 2, 1.2, and 5, respectively.

5 Evaluation

We evaluated our image recovery attack against four popular desktop applications that use GPUs to accelerate image or text rendering. Remarkably, our attack succeeded against various applications. We could also recover sensitive and private information such as users' profile images and email contents. We elaborate the settings and results as follows.

5.1 Evaluation Environment and Performance

We conducted the evaluation on AMD and Nvidia platforms. The specifications of testing environments are described in Table 1. The malicious application we developed applies OpenCL APIs to operate GPU memories, and Matlab was to recover images. We demonstrated the effectiveness of our attack against four popular applications on Ubuntu: Google Chrome, Adobe PDF Reader, GIMP and Matlab. All applications other than MATLAB were run on an AMD platform, because OpenCL is natively supported. Matlab was evaluated on the Nvidia platform because it requires CUDA support to operate the GPU, which is only available on Nvidia platforms. A third-party toolkit named opencl-tool-box that enables

	Virtualized Platform	AMD Platform	Nvidia Platform
GPU	Sapphire R7 250X	HD 6350 (CEDAR)	GTX 750 (Maxwell GM107)
Video Memory	1GB	512MB	1GB
GPU Driver Version	fglrx 15.200	fglrx 15.200	340.29
OS Version	Ubuntu 14.04 LTS (Both guest and host)	Ubuntu 14.04 LTS	Ubuntu 14.04 LTS
CPU	Intel Xeon E3-1225 v2	Intel Xeon E3-1225 v2	Intel Core 2 Duo E8400
Main Memory	24GB	24GB	4GB

Table 1. Platforms used for evaluation.

Matlab developers to use GPU resources on AMD platforms; however, this tool only supports OpenCL, and it is not incorporated into Matlab's official release and has not been updated since January 2013 [8]. Therefore, we did not test Matlab on the AMD platform.

Our attack against the applications follows a consistent routine: The malicious application we built initializes the GPU memory and then monitors the usage of GPU memory. Subsequently, the victim application is launched followed by simulating a series of user operations, such as viewing a web page or viewing a PDF document. Next, the victim application is closed and the malicious application is reactivated because of the sudden increase of available memory. The GPU memory is instantly dumped and analyzed by the malicious application for image recovery. Finally, the malicious application saves the restored images into either RGBA or ARGB format; the malicious application decides on an output format during the step of data blocks pruning.

The overhead of each attack is bounded to the specifications of the platform, but it is in general unnoticeable. We ran our malicious application against each victim application five times and calculated the average time consumed in different steps. It took 75 to 95 ms for memory initialization and 110 to 130 ms for data block extraction and pruning on the AMD platform. The Nvidia platform incurred notably higher overhead. On Nvidia, the malicious application required 350 ms for memory initialization and 550 ms for data block extraction and pruning. We speculate that the greatest increases in overhead were due to the larger memory of the Nvidia platform. The overhead for layout inference is bounded to the size of the tile (the time complexity of Algorithm 1 is $O(n \log n)$ where n is the tile size). The largest tile we encountered was 15MB and could be processed in 13ms. Moreover, the highest number of tiles for a memory dump was no greater than 625 for all experiments. The overhead of this step in most cases would not exceed a second. This attack is capable of scanning even the largest GPU in several seconds, which would hardly be expected to raise the suspicion of a user. The highest CPU usage we observed during the inference phase was 45%.

5.2 Single-Machine Accuracy Assessment

Scale Ratio	Typical Size	Successfully Recovered	Recovered but not in Samples
1	1024*768	29	18
0.5	512*384	29	8
0.25	256*192	29	7
0.125	128*96	29	13
0.0625	64*48	29	28

Table 2. Accuracy test for the self-developed application.

Before testing the exploit against real-world applications, we evaluated the accuracy of our approach in reconstructing original images. Accordingly, we developed a toy application whose sole task is to load images into GPU memories. In particular, the application reads JPG files from disk, decodes them into bitmap format, stores them in GPU memory, and then exits without zeroing out the used memory region. The malicious application then attempts to reconstruct the original images from the uninitialized memory. We did not use other commercial or open-source applications because they could have split the images into pieces and rendered them in parallel.

The first set of test images consisted of 29 sample images from the INRIA Holidays dataset, which are widely used for evaluating computer vision algorithms [7, 17]. We began by evaluating the accuracy of recovering original sample images. Next, we zoomed out those images to different sizes and assess the impact of image size on our approach. Finally, we applied different types of transformations to the sample images to understand the limitations of our approach (i.e., which factors would impede the success of the malicious image reconstruction efforts).

Table 2 presents the test results for the first two evaluation tasks. Specifically, all of the original images were successfully recovered. When the size ratio of the image was scaled down from 1 to 0.0625 (the size was reduced to 64*48, the icon size), the result was not changed. All images were successfully restored, signifying that our approach is robust against images of various sizes. Notably, we also occasionally recovered images that had not been loaded by our application. We suspect these images were rendered by applications running simultaneously with our application.

Noise σ	1	5	10	20	30	40
Recovered #	29	29	28	26	25	23

Table 3. Successfully recovered images under different noise settings.

Next, we tested the capability of our approach in dealing with less meaningful images. Adding noise is a common means of obscuring the meaningful pieces within images. We applied Gaussian noise to the sample images and examined whether they could still be restored. We added Gaussian random number falling within the range of $N(0, \sigma)$ (σ is the noise standard deviation and the larger σ is, the more noise obstructs the signal) to each pixel of the original images before loading them into the GPU memory. Table 3 shows the number of successfully recovered images under different settings of σ.

As suggested by the results, our algorithm can recover images even when high values of Gaussian noise were applied as interference. When the noise standard deviation σ was increased to 40, the interfered images were barely recognizable to humans, but the malicious application still had a successful recovery rate of 79.3%. Figure 11a and Figure 11b show the the original image and a corresponding Gaussian-noise-distorted image with $\sigma = 40$. Both images could be correctly restored from memory dumps.

We also assessed the impact of other image transformations, including brightness and contrast adjustments. We increased and reduced the brightness and contrast of the 29 images by 80% separately, and the resulting images were unrecognizable. Figure 11c and Figure 11d depict the images after the brightness levels were adjusted. Remarkably, all such images were restored by our approach with a 100% success rate. This result strongly supports the robustness of our approach.

5.3 Virtualized Environment Experiments

Our accuracy study clearly shows that the GPU management system is vulnerable in a multiuser computer in a controlled environment. One must consider whether such a vulnerability also exists in a virtualized environment where multiple VMs share GPU resources in a time duplex manner. Therefore, we rented a passthrough-capable GPU, namely a Sapphire R7 250X, to set up a virtualized test bed. We did not run our test on a commercial cloud, such as Amazon EC2, to avoid breaching the privacy of other cloud users. The GPU of the AMD platform was temporarily replaced by the rented card to support GPU passthrough. We used QEMU 2.4.50 as the hypervisor to the host virtual machine.

Our evaluation routine was as follows: The attacker's VM was started first to initialize the GPU memory and then shut down. Next, the victim turned on his VM, used our self-developed toy program to load the 29 images to the GPU memory and then shut down. Finally, the attacker started his VM to extract the GPU memory and recover images. One restriction of the virtualized environment is that the two VMs cannot run simultaneously, because of the dedicated assignment of GPU resources to the VM. However, its impact on a real-world attack can be reduced if the adversary can profile the running time of the victim's VM ahead. Another restriction is that during the VM switching process, the physical machine cannot be restarted, lest the GPU and its memory be reset. To maximize the usage of machines, physical machines are rarely shutdown or restarted by cloud providers and the impact is therefore limited.

Our evaluation results revealed that, among the 29 images, 25 were completely recovered while 2 were completely missing and the remaining 2 images were recovered partially. The accuracy was lower than that in the single-machine context. We believe this was because the time gap between the termination of the victim app and the residues extraction was increased. When the attacker has a process running in parallel to the victim's, she can monitor the GPU memory usage and extract the residues immediately after the victim application terminates. By contrast, in a virtualized context, the attacker can only extract the residues at least after a VM switching process, which engenders considerable noise that may pollute the memory. However, the ratio of recovered images was still noteworthy, indicating that the threat to the virtualized environment cannot be neglected.

Next, we describe several concrete cases to demonstrate the impact of our attack.

5.4 Case 1: Google Chrome

Currently, increasing numbers of web applications are deployed to process users' personal information. Browser vendors design various mechanisms to protect users' data, such as private browsing. Such mechanisms are aimed at defending against malicious web pages or extensions planted by attackers but are powerless against adversaries capable of stealing information from GPU memories. The problem is exacerbated in up-to-date browsers wherein GPU-acceleration is intensively used. We present Google Chrome as an example to demonstrate the seriousness of this problem. Gmail is used to demonstrate what types of content can be recovered by our attack. In addition, we describe automated information extraction techniques against memory dumps from different web sites to assess the impact of the attack.

Recovered contents from Gmail. In this attack, we assumed the victim user logged into her Gmail account and the email titles were displayed. We ran an analysis routine against the GPU residues and recovered 113 images from the GPU memory dump, among which the largest had 512K pixels, the smallest had 926 pixels, and the average size of the images was 23.74 KB (PNG format). The images were manually classified based upon their visual positions in the page. The details of the leaked images are described as follows:

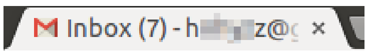

Inbox (7) - h z@ ×

Fig. 5. Browser tab showing Gmail.

• *Tab:* A browser tab for Google Chrome displays the favicon and the title specified by the web page. It can show which website is being visited by the user. Moreover, a Gmail page reveals much more information than just the name of a website. As shown in Figure 5, the email address and number of unread emails are also displayed in the tab. Leaking email address is of course undesirable for the victim because this can be exploited to send targeted phishing emails or for harvesting user's social profiles by querying popular social networking sites. In addition, this problem is not unique to Gmail, and equal or greater amounts of information could be disclosed from the tabs of other sites. For instance, Amazon displays the name of the product the user is viewing and YouTube displays the name of the video the user is watching.

Although our initial exploration indicates that critical information could be leaked, the extent of the inferred result should be interpreted with reservations. Google Chrome limits the number of characters displayed on a tab. The tab is squeezed when many tabs are opened (it begins to resize when more than 7 tabs are opened on a 14-inch laptop screen with a set resolution of 1600x1200). This design results in partial revelation of the title of a web site: for example, the Gmail tab displays only the first 14 characters of the user name (under the same screen setting); therefore, a long user name is not fully recovered. However, numerous users choose short user names [1] and knowing 14 characters is still a major breakthrough if the adversary plans a brute-force crack.

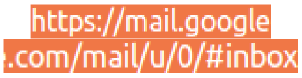

https://mail.google
.com/mail/u/0/#inbox

Fig. 6. Address bar of Gmail.

• *Address bar:* An image containing the address bar was also recovered by our program and is shown in Figure 6. Notably, the image does not show the whole region of the address bar or even the full URL. Chrome attempts to render the address with the GPU when the "Auto-Complete" is turned on and the user is typing. The recovered image reflects the characters that have been input. Although not fully displayed, this partial image can still indicate that the user is using Gmail. When the user is not typing in the address bar, a different type of image is generated, and an example is shown in Figure 1a. By collecting the leaked information from such images (and tab images), an adversary can partially reconstruct a targeted user's browsing history, which clearly violates user privacy. Previous research by Lee et al. [20] profiled 1000 website homepages and attempted to identify which sites had been visited. Our attack advances the field because potentially any site visited can be inferred without prior profiling and it is also resilient to content changes on the websites.

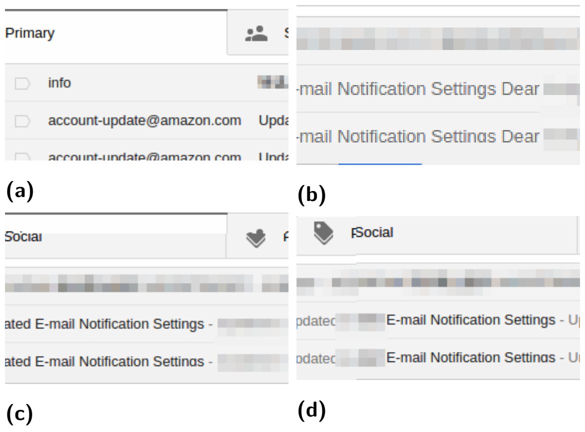

Primary
info
account-update@amazon.com Upda
account-update@amazon.com Upda
(a)
-mail Notification Settings Dear
-mail Notification Settings Dear
(b)
Social
ated E-mail Notification Settings -
ated E-mail Notification Settings -
(c)
Social
pdated E-mail Notification Settings - U
pdated E-mail Notification Settings - U
(d)

Fig. 7. Part of Gmail inbox.

	www.gmail.com	www.youtube.com	www.yahoo.com	www.facebook.com	www.twitter.com
Characters	2388	9184	690	6183	2110
Words	416	453	215	826	481
Faces	0	24	8	8	16

Table 4. Leakage from different websites

• *Page body:* Most of the images recovered can be attributed to the page body. Figure 7 shows one segment of Gmail inbox contents; the senders and the initial characters of the emails are displayed, which are obviously sensitive information belonging to the user. Because current web applications are designed to intensively deal with personal information, the threat could be more substantial if the malicious program is able to run on the victim's machine for a long time and restore images from different web pages. Among these recovered images, we also found seemingly meaningless textures such as the border of an object box. We suspect that they were peeled from the original objects because of the browser's splitting algorithm. They can be combined with their counterparts through a puzzle-solving algorithm.

Automated information extraction. All the images were manually examined for this Gmail case but the practice of manual examination is not scalable, particularly when numerous users are monitored or the tabs of Chrome are frequently closed and opened. We intended to reduce the attacker's workload by sending only sensitive images for analysis. In particular, automatically selecting sensitive images is quite challenging; it requires extensive knowledge of the user's background and the application's context. A more practical goal could be identifying the images that include texts and faces, which are already meaningful in common scenarios; that tasks can be automated.

We thus used Adobe PDF professional OCR module to find texts (the images must be converted into PDF first) and developed a Matlab program using a widely used computer vision system toolbox to recognize faces. Only the images containing either texts or faces are passed to the next step (i.e., analyzed by the attacker). We evaluated this tool on Gmail images and reduced the number of images of interest to 31 (out of 113 images in total) and all the images associated with tab, address bar, and inbox were identified. The overhead incurred in this process was also small, only costing several seconds in OCR and face recognition. We assumed that the modules would run on the attacker's server, but they could run on the victim's machines as well. For the latter setting, only the detected images are transmitted, which significantly reduces the network overhead and makes the attack even stealthier.

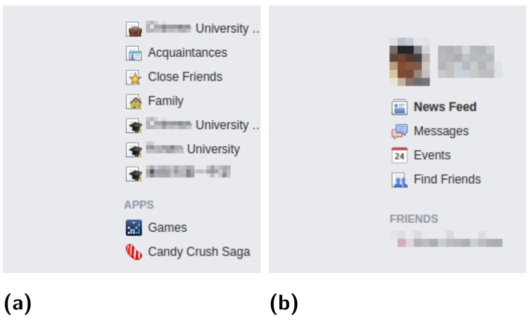


Fig. 8. List of attended universities and schools identified from a user's Facebook page.

Despite our attack against Gmail revealing that sensitive information could be successfully revealed, whether the issue is universal has yet to be clarified. Hence, we attempted to answer this question by evaluating our algorithm on four other highly popular websites: YouTube, Yahoo, Facebook and Twitter. Because no metric is available to quantify the sensitive data leaked, we measured the number of words and faces recognized from the images. As shown in Table 4, approximately hundreds of words and tens of faces could be identified for each web site. Although not all texts and faces were sensitive (e.g., we found that most of them were related to advertisements), the chances of leakage were still high. In particular, the social networking sites such as Facebook and Twitter tend to leak dangerously large amounts of useful information. Tweets and Facebook posts were discovered among the recovered images. Figure 8 shows a tile exhibiting a Facebook user's educational experience (sensitive personal information is mosaicked). Furthermore, we attempted to evaluate our attack against e-banking. The results revealed the **account balance**, **credit card account number**, and **transaction details**.

Discussion. Finally, we explored why segments were extracted rather than the whole web page. According to the design document [2], Chrome breaks each page into small tiles and allocates GPU resources for some of them based on their predefined priorities [9]. However, not all image segments were preserved in GPU memories when dumped by our attack program. In the real world, there are GPU memory restrictions that limit the number of tiles residing in the GPU and the memory manager is allowed to evict tiles from GPU memories. Therefore, not all segments can be recovered.

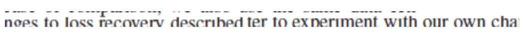
In addition to tests on Chrome, we tested our attack on Firefox, which is also under threat. Specifically, it was confirmed that Firefox also leaves residues in GPU memories [20]. We tested our attack on Gmail viewed through Firefox. Firefox was observed to not produce residue images related to the address bar and tab caption, but more severely, **it was observed to yield a relatively complete block showing the page body**. When Firefox viewed Gmail, it left a large image containing the sender, title and part of the email contents. Such leaked information is definitely also valuable to attackers.

5.5 Case 2: Adobe Reader

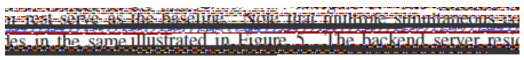
Adobe Reader uses GPUs to accelerate the rendering process of PDF documents. We considered the texts and graphs of a PDF as sensitive and tested whether they could be extracted by exploiting the residues of the application. We used a PDF of a research paper as an example, and the content recovered is described as follows.

Recovered content from the PDF. Results showed that both graphs and texts were rendered in the GPU. Similar to the Chrome case, segments of figures and texts were recovered. Notably, the segments did not only belong to the page shown in the foreground, but some of them also did belong to the pages rendered in the background.

• *Fragments of figures:* We identified that some of the recovered images were actually fragments of a given figure. We did not attempt to combine the fragments to recover the original figure; nevertheless certain algorithms may be able to achieve this goal through various strategies, such as by taking advantage of the similarities among the edges of neighboring fragments.


nges to loss recovery described ter to experiment with our own cha

(a) A normal line of text


n-test serve as the baseline. Note that multiple simultaneous se
les in the same illustrated in Figure 5. The backend server resic

(b) A line of text with noise

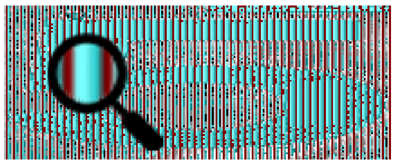
Fig. 9. Two typical images recovered from residues of Adobe Reader.

• *Lines of text:* We discovered that Adobe Reader separates text regions into long stripes. Figure 9a shows a normal line of text in its recovered state (some letters are incomplete) and Figure 9b shows one line of text with noise above and below. Although not fully recovered, the texts from the images can be easily read by an attacker. When excessive images are extracted from the GPU, the adversary can use OCR and natural language processing techniques to reduce the number of images requiring manual analysis.

5.6 Case 3: GIMP

This section presents the evaluation of a popular image processing software on Unix platforms, GIMP. Under its default configuration, GIMP does not rely on the GPU to render images. However, when the image is large, the user is recommended to turn on the GPU acceleration, which can be achieved by linking to a graphics library named `GEGL` when GIMP is started. A user can simply pass `"GEGL_USE_OPENCL=yes"` when launching GIMP. Our attack was tested under this setting. Specifically, we opened an image file, applied some different image filters (e.g., edge-laplace) for each run, and then closed the image file.

Recovered content from GIMP figures. Our evaluations indicated that the type of filter determines the outcome of the attack. For some filters, no information could be revealed from the image either before or after having applied the filters. By contrast, for other filters, the recovered images were actually close to the **whole** original images that were passed to GIMP, without any fragmentation.


(a) Before

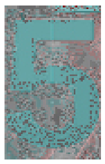

(b) After

Fig. 10. Images recovered from leakage with stripes before and after compression.

We also restored images with vertical stripes obscuring the original images (Figure 10a). This case was probably caused by the implementation of GEGL, which does not adopt a standard image format (RGBA and ARGB). Because such stripes could affect the step of information inference, we removed them with a "compressing" process: The width of the recovered image was shrunk to 25%, and a pixel in the new image was combined from 4 consecutive pixels arranged horizontally. The image derived from Figure 10a is shown in Figure 10b.

This case suggests that even if a developer does not use the standard RGBA or ARGB format, the sensitive information in the recovered images can still be extracted by tweaking the attack. The pseudo-periodicity still holds despite the loss of the color map information, which results in output images with correct alignment but inaccurate color. However, most the images can be recognized with techniques such as text recognition.

5.7 Case 4: Matlab

Matlab is widely used by academia and industry for scientific computing. It also provides various libraries to support image processing, and developers can capitalize on the power of GPUs with a parallel computing toolbox. We assumed that the developer loaded a picture from hard disk drive and then converted it into a GPU-compatible object named “gpuArray”. This image object was passed to the GPU for processing, and Matlab was closed when processing finished. Finally, the residues in the GPU memory were dumped and analyzed.

Recovered content. The loaded picture had been split by Matlab into fragments; therefore, the whole picture could not be directly recovered. However, the size of the fragments was still large enough to enable partial or full restoration of the original image through rearrangement. After the fragments had been combined together, we found that the generated picture was flipped along the diagonal of the original picture. This is because Matlab stores a picture as a matrix in a column-by-column manner, rather than a normal row-by-row fashion, thus automatically transposing the image in memory. Consequently our proposed algorithm can still be applied by simply transposing the image matrix back.

Discussion. Matlab is a frequently used computing tool, which is commonly used in many different fields. Matlab along with its Parallel Computing Toolbox is often run by developers on a high-performance computer with a powerful GPU. The high-performance computer is often shared by different users in many cases, because the computing resources are expensive and are wasted when the computer is idle. Hence, an attacker has a higher chance of obtaining the victim users’ images from these types of computers.

6 Discussion

Limitations. We chose four applications of interest for evaluation and demonstrated the efficacy of our attack. We did not evaluate more applications because of the considerable amounts of manual work involved in running victim applications under different settings and examining the results. However, we believe that our results suffice to demonstrate the unneglectable security issue underlying GPU memory management frameworks. The information extraction process is not fully automated yet. So far, we only applied text and face recognition for images rendered by Chrome. We are investigating more feasible mechanisms for this task. Although our approach is fairly robust, if images are loaded into memory without extra processing by the victim applications, then the parallel rendering techniques used by applications such as Chrome raise the difficulty of image recovery. Stitching together image segments is feasible, and we are in the process of exploring feasible algorithms (e.g., a puzzle-solving algorithm). However, the leaked information from the segments is already alarming.

Mitigation. Security-conscious users might propose a mitigation approach that automatically clears the used pages in global and private memory after the program exits. This solution would mitigate the threats in theory, but it would face grave obstacles to adoption. The performance degradation of this solution is huge, as identified in previous studies [10, 14, 20]. In the following paragraphs, we present two possible solutions which prevent GPU memory leaks without much impact on performance.

Manual clearing. Because complete and dynamic residue cleaning requires considerable resources and engenders performance degradation, selective clearing is a more practical solution. Given that the GPU structure cannot be completely redesigned in a short time, developers should manually clear the memory before releasing it. To maximize performance, they can devote resources to clear memory regions containing user’s sensitive data only. However, such a requirement is difficult to be fulfilled by every developer, especially when the applications maintained have colossal code bases (e.g., Chrome). Similar problems also exist in CPU structures and the issues regarding memory leaks from CPUs have been sufficiently studied, with many approaches being proposed. A developer could use a static analyzer to pinpoint a location in code where memory is not freed [13, 16] or identify the leaks in the runtime by monitoring the stacks or heaps [22, 28]. We hope the ideas in those studies could be leveraged to build detectors against GPU leaks.

Image layout obfuscation. The above preceding does not mitigate the problem completely. Even the best code analyzer or developer cannot capture every code location processing sensitive data. Alternatively, we suggest that developers obfuscate the image layout in the GPU memory. When an application delivers an image to a GPU, it could intentionally store the pixels in some irregular format instead of a sequential array. The obfuscation can be implemented in the complier or in the system library managing GPU memories. The key assumption of our scheme is that there should be high geometrical correlation between consecutive columns and rows. Apparently, the correlation is interrupted by layout obfuscation. This scheme would introduce some overhead inevitably, but compared with the scheme that involves blindly filling zeros, the overhead should be smaller since only images are subjected to this additional step. The technique may weaken the performance of memory compression, because the obfuscation breaks the local redundancy of the image, which consequently deteriorates the compression ratio.

Defense for virtual machine. To defend against attackers on a virtualized platform, it is sufficient for the hypervisor to clear the whole GPU memory space every time the VM switches. The overhead in this case is negligible, because 1) memory only needs to be cleaned when a VM switching occurs, and 2) VM switching process already consumes considerable time and the extra overhead is thus not prominent. Therefore, we recommend that all cloud service providers with GPU support in their infrastructure should clean their GPU memories during VM switching.

7 Related Works

GPU vulnerabilities. With the advancement of techniques for GPU computing, different security issues also emerge. Previous research has shown that the GPU security measures are far from perfect [18, 24]. Lombardi et al. conducted a comprehensive analysis regarding GPU usage in cloud environments and revealed several leakage problems. Pietro et al. discovered leakage in a CUDA framework, and their evaluation indicated that global memory, shared memory, and registers are all vulnerable [14]. Moreover, Clémentine et al. proposed an attack for acquiring leaked information from other virtual machines [21]. Ladakis et al. [19] implemented a stealthy keylogger using a GPU. The closest work to our research was that Lee et al. [20] were able to infer which website had been visited by a victim based on the color distributions of GPU memories.

Memory forensics. Memory forensics have been studied for a long time as a strategy for helping the government and police forces to collect electronic evidences from devices confiscated from criminals. In recent years, advances have been made in recovering images from main memory for forensic needs. Saltaformaggio et al. proposed a method for reconstructing Android APP GUI displays by reconstructing the GUI tree topology and reconstructing the drawing operations [25]. They also introduced a method for recovering photographic evidence produced by a smartphone camera by using the memory possessed by an intermediary service [26]. Recently, a method for reusing application's logic to recover images from computer memory [27] was proposed.

Image File Carving. Apart from leveraging program logic, file carving can be used to extract images from broken file systems [11, 12, 23]. The first step of file carving is locating the file signatures and metadata, and such information is not available in the GPU case. Furthermore, Guo et al. proposed a method for recovering jpeg files using their thumbnails [15] ,but our attack obtains images for which no thumbnails exist.

Those works requiring foreknowledge of the image for reconstruction (e.g., program logic, metadata, or thumbnails) are unsuitable for the GPU case where no such information can be found. By contrast, our algorithm can automatically recover images and does not rely on any foreknowledge of the target images.

8 Conclusion

In this paper, we prove that typical GPU memory management strategies are vulnerable, by proposing a novel attack for recovering images from the residues of other applications or other VMs in GPU memories. Our recovery technique was motivated by the observation that strong correlations exist between the rows and columns of an image. By evaluating highly popular applications, we demonstrate the severity of the security problems of GPU memory management. Sensitive information such as credit card numbers and email titles can be readily extracted, if it has been previously calculated by a GPU. The severity of this security flaw has been underestimated by previous researchers, and the security threats must be mitigated.

References

[1] Average username length. http://www.eph.co.uk/resources/email-address-length-faq/.

[2] The Chromium projects design documents: Gpu accelerated compositing in chrome. http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome.

[3] Delta color compression overview. http://gpuopen.com/dcc-overview/.

[4] Discrete fourier transform. http://www.robots.ox.ac.uk/~sjrob/Teaching/SP/l7.pdf.

[5] Fast Fourier transform. http://mathworld.wolfram.com/FastFourierTransform.html.

[6] GPU cloud computing. http://www.nvidia.com/object/gpu-cloud-computing-services.html.

[7] Inria holidays dataset. http://lear.inrialpes.fr/~jegou/data.php.

[8] Opencl-toolbox. https://code.google.com/p/opencl-toolbox/.

[9] Tile prioritization design of Chrome. https://docs.google.com/document/d/1tkwOlSlXiR320dFufuA_M-RF9L5LxFWmZFg5oW35rZk.

[10] XDC2012: Graphics stack security. https://lwn.net/Articles/517375/.

[11] N. A. Abdullah. *An improved file carver of intertwined jpeg images using X_myKarve*. PhD thesis, Universiti Tun Hussein Onn Malaysia, 2014.

[12] B. Chelliah, D. S. Vidyadharan, P. Shabana, and K. Thomas. Carving of bitmap files from digital evidences by contiguous file filtering. In *International Conference on Security in Computer Networks and Distributed Systems*, pages 234–239. Springer, 2012.

[13] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. *SIGPLAN Not.*, 42(6):480–491, June 2007.

[14] R. Di Pietro, F. Lombardi, and A. Villani. CUDA leaks: Information leakage in gpu architectures. *arXiv preprint arXiv:1305.7383*, 2013.

[15] H. Guo and M. Xu. A method for recovering jpeg files based on thumbnail. In *Control, Automation and Systems Engineering (CASE), 2011 International Conference on*, pages 1–4. IEEE, 2011.

[16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[17] H. Jégou, M. Douze, and C. Schmid. Hamming Embedding and Weak Geometry Consistency for Large Scale Image Search - extended version. Research Report 6709, Oct. 2008.

[18] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. Loh. Architectural vulnerability modeling and analysis of integrated graphics processors. In *Workshop on Silicon Errors in Logic-System Effects (SELSE), Stanford, CA*, 2012.

[19] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis. You can type, but you can¡¯t hide: A stealthy gpu-based keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.

[20] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 19–33, 2014.

[21] C. Maurice, C. Neumann, O. Heen, and A. Francillon. Confidentiality issues on a gpu in a virtualized environment. In *Proceedings of the 18th International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[22] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 397–407, New York, NY, USA, 2009. ACM.

[23] A. Pal and N. Memon. The evolution of file carving. *IEEE signal processing magazine*, 26(2):59–71, 2009.

[24] M. J. Patterson. *Vulnerability analysis of GPU computing*. PhD thesis, Iowa State University, 2013.

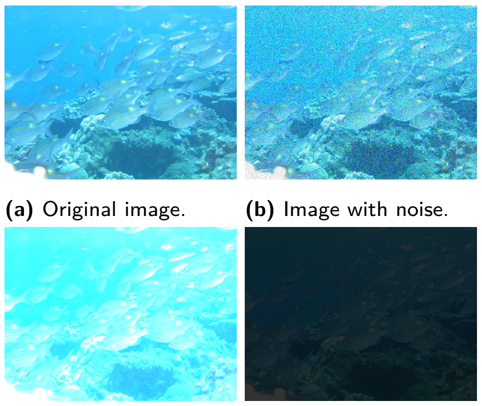
[25] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. Guitar: Piecing together android app guis from memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 120–132. ACM, 2015.

[26] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. Vcr: App-agnostic recovery of photographic evidence from android device memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 146–157. ACM, 2015.

[27] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu. Dscrete: automatic rendering of forensic information from memory images via application logic reuse. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 255–269. USENIX Association, 2014.

[28] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 151–160, New York, NY, USA, 2008. ACM.

Appendix – Accuracy Assessment Image Samples



(a) Original image. **(b)** Image with noise.

(c) Image with increased brightness. **(d)** Image with decreased brightness.

Fig. 11. Original image and the transformed versions.