PixelVault: Using GPUs for Securing Cryptographic Operations

Giorgos Vasiliadis
FORTH-ICS
gvasil@ics.forth.gr

Elias Athanasopoulos
FORTH-ICS
elathan@ics.forth.gr

Michalis Polychronakis
Columbia University
mikepo@cs.columbia.edu

Sotiris Ioannidis
FORTH-ICS
sotiris@ics.forth.gr

ABSTRACT

Protecting the confidentiality of cryptographic keys in the event of partial or full system compromise is crucial for containing the impact of attacks. The Heartbleed vulnerability of April 2014, which allowed the remote leakage of secret keys from HTTPS web servers, is an indicative example. In this paper we present PixelVault, a system for keeping cryptographic keys and carrying out cryptographic operations exclusively on the GPU, which allows it to protect secret keys from leakage even in the event of full system compromise. This is possible by exposing secret keys only in GPU registers, keeping PixelVault's critical code in the GPU instruction cache, and preventing any access to both of them from the host. Due to the non-preemptive execution mode of the GPU, an adversary that has full control of the host cannot tamper with PixelVault's GPU code, but only terminate it, in which case all sensitive data is lost. We have implemented a PixelVault-enabled version of the OpenSSL library that allows the protection of existing applications with minimal modifications. Based on the results of our evaluation, PixelVault not only provides secure key storage using commodity hardware, but also significantly speeds up the processing throughput of cryptographic operations for server applications.

Categories and Subject Descriptors

E.3 [**Data**]: DATA ENCRYPTION; D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

Keywords

GPU; SSL/TLS; trusted execution; isolation; tamper resistance

1. INTRODUCTION

Servers have always been an attractive target for attackers, especially when they host popular web sites and online services, as they typically contain a wealth of private user data and other sensitive information. Encryption can be used as an additional layer of protection for sensitive data, once a service has been compromised, but it is only effective as long as the keys involved in cryptographic operations are kept secret. In fact, keys themselves are often the target, as for example is the case with the infamous Heartbleed bug [9]. The exploitation of Heartbleed, a buffer over-read vulnerability in OpenSSL, allows attackers to read arbitrary contents from a server's memory, including TLS private keys. Besides attacks that leverage software vulnerabilities to disclose memory or take complete control of the host, key recovery attacks can also be mounted using direct memory access through Firewire [48] or PCI [57]. Moving one step further, it has been demonstrated that keys can be extracted by freezing memory chips and inspecting their contents [23].

Once the secret keys are leaked, attackers can impersonate the server (without triggering any browser warnings), or decrypt any past and future captured encrypted data (unless perfect forward secrecy is used). Defenses that involve the in-memory obfuscation of keys using dispersal techniques offer only partial protection, as attackers can eventually break the obfuscation scheme with adequate effort. To that end, it is crucial that, apart from the trusted operation of the underlying cryptographic implementation, secret keys and other sensitive information is safely stored and protected from leakage. It is important to ensure that a potential security flaw in a service will not allow an adversary to get access to secret keys, even if the service is fully compromised, as this can lead to further catastrophic consequences [9, 10].

In order to address this problem, researchers have proposed systems that store all sensitive information in CPU registers and never in main memory [21, 41, 56]. These approaches require a trusted and bug-free component for ensuring that an adversary cannot compromise part of the system and eventually extract the secret keys from the CPU registers. Unfortunately, however, with complex services consisting of databases, web servers, and a multitude of other software components and libraries, guaranteeing the absence of bugs that may lead to system compromise is rather unrealistic. For instance, a recent DMA attack against these systems has shown that secret keys can be extracted from the CPU registers into the target system's memory, and be retrieved using a normal DMA transfer [12].

Another possible research direction for solving this problem is through systems that support trusted computation in hostile operating systems [16, 26, 29]. These systems are designed as a generic solution for protecting computation performed by any application, even by non-sensitive ones, when the host is compromised. In such a setting, a process responsible for cryptographically signing a message would never expose its keys to the operating system, and therefore encryption remains functional even when the operating system is compromised. However, these systems require applications to

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660316 .

run on top of a hypervisor [16, 26], introducing significant performance overhead, or the addition of extra hardware abstraction layers and the re-compilation of the operating system [29]. Trusted Platform Modules (TPMs), on the other hand, do not provide useful support for the use case we consider. From a security standpoint, TPMs provide limited cryptographic support (current implementations support only RSA, SHA1, and HMAC) [8], while from a performance standpoint, their limited computational capabilities make them inappropriate for carrying out intensive and continuous cryptographic operations, such as handling a server's TLS connections [1].

In this paper, we explore an alternative approach to the problem of protecting a server's cryptographic keys, which takes advantage of the graphics card to exclusively i) store cryptographic keys and other sensitive information, and ii) carry out all cryptographic operations, without involving the CPU. Our prototype system, named *PixelVault*, provides native GPU implementations of the AES [19] and RSA [51] algorithms, and prevents key leakage even when the base system is *fully* compromised. This is possible by exposing private keys only in GPU registers, and keeping PixelVault's critical code exclusively in the GPU instruction cache, preventing this way even privileged host code from accessing any sensitive code or data. We have implemented a PixelVault-enabled version of the OpenSSL library, which allows the transparent protection of existing services without hardware modifications or operating system recompilation. Multiple services can use the same GPU to perform cryptographic operations, using the same or different certificates (and secret keys), while trust is always given to a single hardware entity—the GPU.

Our choice of the GPU for key storage is justified by its unique properties, including *(i) non-preemptiveness:* all program code running on the GPU is never context-switched, and therefore, there is no saved state in the host's memory that could include information associated with cryptographic keys; *(ii) on-chip memory operation only:* the running GPU code is tamper-resistant in on-chip memory, and the associated cryptographic keys are never stored in observable memory, but only in non-addressable memory, such as the registers of the GPU; *(iii) transparency:* the GPU is independent from the host, so no hardware, operating system, or application changes are required—just a modification of the standard cryptographic libraries used, such as OpenSSL, which essentially implies that legacy applications can fully take advantage of our system with minimal effort; *(iv) commodity component:* GPUs are commodity components and are cheaper than dedicated cryptographic hardware; *(v) performance:* GPUs achieve high computational performance for cryptographic operations, for applications in which they can be parallelized.

The main contributions of our work are the following:

1. We present the design of PixelVault, a system for keeping cryptographic keys and carrying out cryptographic operations exclusively on the GPU, which allows it to protect secret keys from leakage even in case the host is fully compromised.

2. We have implemented PixelVault using commodity GPUs (NVIDIA's GTX 480), and provide a PixelVault-enabled version of the OpenSSL library.

3. We evaluate our prototype implementation in terms of security and performance. Our analysis suggests that PixelVault not only provides better protection, but also outperforms CPU-based solutions in terms of processing throughput for server applications.

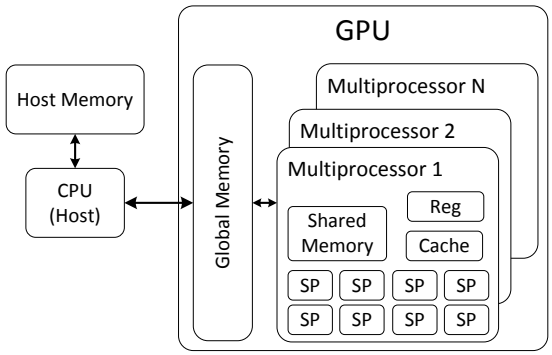


Figure 1: A simplified view of a typical graphics card memory hierarchy.

2. BACKGROUND

2.1 GPU Architecture Overview

The computational capabilities of modern graphics processing units in combination with their low cost makes them suitable for general-purpose applications beyond graphics rendering [52,55,60, 61]. GPUs contain hundreds of processing cores that can be used for general-purpose computation, facilitated by feature-rich frameworks for general purpose computing on GPUs (GPGPU). For our prototype implementation, we have chosen NVIDIA's CUDA [42], probably the most widely used GPGPU framework.

A fundamental difference between CPUs and GPUs is the decomposition of transistors in the processor. A GPU devotes most of its die area to a large array of arithmetic logic units (ALUs). In contrast, most CPU resources serve a large cache hierarchy and a control plane for the acceleration of a single CPU thread. A GPU executes code in a data-parallel fashion, so that the same code path is executed on different data at the same time. The code that the GPU executes is organized in units called *kernels*. To exploit parallelism, the same kernel is launched by a vast amount of GPU threads concurrently.

2.2 Memory Hierarchy

The NVIDIA CUDA architecture offers different memory spaces and types, as illustrated in Figure 1. The host is responsible for allocating memory for the GPU kernel from the *global*, *constant*, and *texture* memory spaces of the graphics card. Allocated memory can be accessed by the host through special functions provided by the CUDA driver, and is persistent across kernel launches by the same application. Both constant and texture memory are read-only, are initialized by the host, and contain separate caches, optimized for different uses. On devices with compute capability 2.x (and higher) global memory accesses are also cached in L1–L3 caches.

Each GPU thread maintains its own *local* memory area, which actually resides in global memory. Automatic variables declared inside a kernel are mapped to local memory. In implementations that do not support a stack, all local memory variables are stored at fixed addresses. The *parameter state space* (`.param`) is used to *(i)* pass arguments from the host to the kernel, *(ii)* declare formal input and return parameters for device functions called during kernel execution, and *(iii)* declare locally-scoped byte array variables that serve as function call arguments, typically for passing large structures by value to functions. The location of the parameter space is implementation-specific. In some implementations, kernel parameters reside in global memory, hence no access protection is provided between parameter and global space in this case.

Compute capability (version)	#registers per thread
1.x	128
2.x	63
3.0	63
3.5	255

Table 1: Maximum number of 32-bit registers per GPU thread for different levels of CUDA support (compute capability).

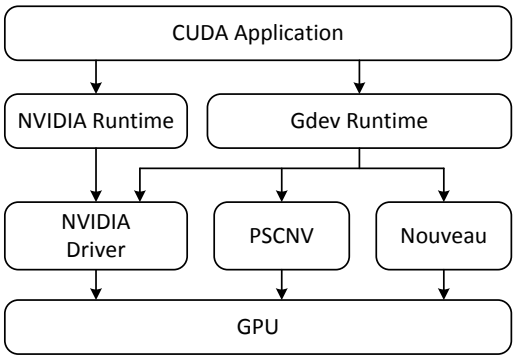


Figure 2: Structure of CUDA applications on top of the NVIDIA (closed-source) and Gdev (open-source) sets of GPGPU runtime and driver software.

The *shared memory* (comparable with scratchpad RAM in other architectures) provides very fast access, and is also shared between the threads that belong to the same block. The size of shared memory is 64KB per warp, and is also used by the hardware-managed L1 cache. Typical splits include either 16KB L1 / 48KB shared or 48KB L1 / 16KB shared. Finally, a set of *registers* (`.reg` state space) provides fast storage locations. The number of registers is limited, and will vary from platform to platform, as shown in Table 1. Registers differ from the other state spaces in that they are not fully addressable, i.e., it is not possible to refer to the address of a register. When the limit is exceeded, register variables will be spilled to global memory, causing changes in performance [47].

2.3 GPU Code Execution

A typical GPU kernel execution consists of the following four steps: *(i)* the DMA controller transfers input data from host memory to GPU memory; *(ii)* a host program instructs the GPU to launch the kernel; *(iii)* the GPU executes threads in parallel; and *(iv)* the DMA controller transfers the resulting data from device memory back to host memory. All these operations are performed by the CPU using architecture-specific commands.

Although the architectural details of GPUs are not publicly available, there is an ongoing research that tries to unveil how these operations are performed and to provide an in-depth understanding of their runtime mechanisms [32,38]. Specifically, the GPU exposes a memory-mapped region to the OS, which is the main control space of the GPU, and is used to send commands. For example, to copy data from host to device memory, a set of commands are sent to the GPU that specify the source and the destination virtual addresses, along with the mode of direct memory access (DMA). Similarly, when a kernel is launched, another set of commands is composed and sent to the GPU, specifying code and stack information.

CUDA applications can run either on top of the closed-source NVIDIA CUDA runtime, or on top of the open-source Gdev runtime [7]. The NVIDIA CUDA runtime relies on the closed-source kernel-space NVIDIA driver and a closed-source user-space library. Gdev also supports the NVIDIA driver, as well as the open source Nouveau [3] and PSCNV [6] drivers. Figure 2 illustrates the software stack of the CUDA and Gdev frameworks. Both frameworks support the same APIs: CUDA programs can be written using the runtime API, or the driver API for low-level interaction with the hardware.

3. DESIGN OBJECTIVES

Challenges. There are two characteristics of the GPU execution model that require careful consideration for designing a safe environment for cryptographic keys. First, GPU kernels typically run for a while, perform some computations and then terminate. Second, GPUs do not contain hierarchical protection domains (similar to the protection rings of CPUs).

Using this programming model, cryptographic keys should be transferred to the GPU every time there is a request for an operation, otherwise a malicious GPU kernel, executed in between, could attempt to extract sensitive information from the (unprotected) GPU space. Unfortunately, transferring the keys every time implies that they are already stored in a safe location, and are transferred securely to the GPU via the PCIe bus, which is not the case.

Our solution. To overcome these challenges, we propose a design that follows a different execution model from the typical GPGPU execution. Instead of spawning a GPU kernel execution every time a new cryptographic operation needs to be performed, the system uses a fully autonomous GPU kernel that runs indefinitely, without interruption. The GPU kernel continuously monitors a predefined host memory region (shared with the CPU) for new requests, performs the necessary computations, and transfers the results back. In addition, we ensure that all clear-text sensitive keys reside in device memory that cannot be accessed from the host at any time. Given the non-preemptive execution mode of GPUs, no other GPU kernel can be loaded for execution as long as our autonomous GPU kernel is running.

4. PixelVault

An overview of PixelVault's operation is illustrated in Figure 3. Applications use PixelVault's OpenSSL-compatible API to perform cryptographic operations. Private keys and other sensitive information are kept in encrypted form in a KeyStore that resides in GPU global, off-chip device memory. KeyStore entries are encrypted with a master key that is exclusively stored in GPU registers (Protected Space), and cannot be accessed in any way by the host. To perform a cryptographic operation, *(i)* the required (encrypted) key is fetched from the KeyStore into the protected space of the GPU registers, *(ii)* it is decrypted with the master key, and *(iii)* the actual operation is performed on the input data.

In the following, we describe in detail various aspects of PixelVault's architecture, and the design choices we made for ensuring that sensitive information cannot be leaked to the host even in case of full system compromise.

4.1 Non-Preemptive Execution

PixelVault is designed to keep secrets isolated from the host, which may be vulnerable and could be compromised. Therefore, to make PixelVault tamper resistant, it is important to ensure that all associated code is independent from the host, and completely decoupled from any other (probably untrustworthy) system module. Modern GPUs follow a non-preemptive execution model, which means that only a single kernel can occupy the GPU at any time, and the execution of a kernel can continue without interruption until

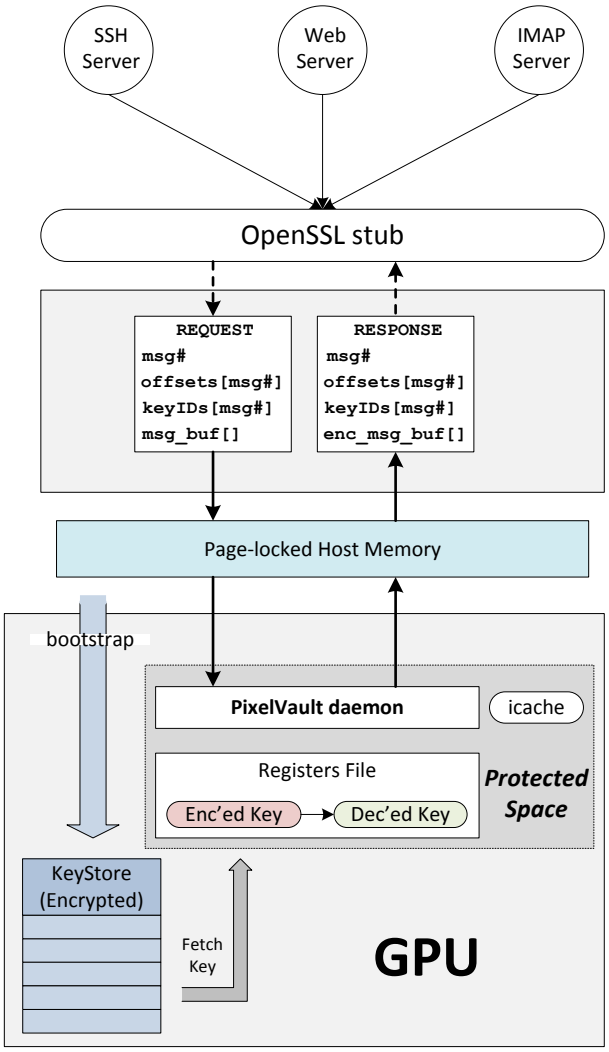


Figure 3: The architecture of PixelVault. Private and secret keys are kept encrypted in a KeyStore residing in global device memory. To obtain a key, the GPU kernel fetches the encrypted key from the KeyStore to the register space and decrypts it.

completion. This execution model is ideal for our purpose. Assuming a trusted bootstrap process (described in Section 4.5), we can enforce that *any* external interaction with PixelVault will immediately terminate its operation, preventing access to sensitive information. Execution can be resumed only by re-launching PixelVault with the same trusted bootstrap process.

Essentially, this means that PixelVault, once it has safely been initialized with all secrets loaded, will run uninterrupted indefinitely. Current GPU frameworks (such as CUDA and OpenCL), however, have not been designed for building applications that act autonomously. When forcing a kernel to run indefinitely (e.g., using an infinite `while` loop), the host process cannot transfer any data to (or from) the GPU. The reason is that, by default, only one *stream*[1] is utilized, forcing the host process to block, (busy-)waiting for the GPU kernel to finish its execution. To isolate completely the kernel execution from the host process, we explicitly create a second stream, which is used exclusively by the GPU kernel. In addition, we disable the *GPU kernel execution timeout*, used by most operating systems to kill any operations executed on the GPU for more than a few seconds [4].[2] As a result, the kernel runs indefinitely, while data transfers (to and from the device) can be performed from the first stream.

Using the above scheme, however, we cannot rely on the typical parameter-passing execution of GPU kernels, as described in Section 2, since the GPU kernel function runs continuously. Instead, we allocate a memory segment that is shared between the CPU and the GPU, as shown in Figure 3. This shared memory space is page-locked, to prevent it from being swapping to disk, and is accessed by the GPU directly, via DMA. The memory space is also declared as volatile, to ensure that all memory reads go directly to memory (and not through the cache). Requests for encryption and decryption are issued through this shared memory space. For example, to perform an encryption operation, an application running on the CPU places the data to be encrypted in the shared region, and sets the corresponding request parameter fields accordingly.

In the GPU, we spawn a large number of threads statically at startup, because the NVIDIA Fermi architecture that we use in this work does not support dynamic thread creation—this feature, called dynamic parallelism [43], was introduced in the more recent Kepler [45] architecture. As long as there is no work to be done, all threads remain idle, busy-waiting, except one (master thread) that continuously monitors the shared space for new requests. When a new request is available, the master thread is responsible for notifying all other threads by setting a special shared variable. Each thread is assigned an equal amount of work (typically to encrypt or decrypt a separate message using a desired key). To prevent out-of-bounds memory accesses, each thread computes the user requested message offset and length and verifies that it lies inside the page-locked memory region. Otherwise, a malicious user would be able to force the GPU kernel to write to non-permissive memory regions. When processing completes, each thread notifies the master thread by atomically increasing a shared variable. When all threads have finished, the master thread notifies the host by setting the response parameter fields accordingly.

4.2 On-chip Memory Operation Only

PixelVault avoids placing secrets in memory that can be easily inspected once a host is compromised. The GPU system we use offers a unified virtual address space mode (the so-called "Unified Virtual Addressing"), in which both the CPU and the GPU have access to the same address space. The virtual address space range is the same for all CUDA processes, and typically starts at address `0x600300000` (in 64-bit systems). Virtual addressing provides isolation between memory accesses from different processes. For example, a process cannot access the global device memory allocated from a different process, as it will have different virtual-to-physical mappings.

Still, an attacker could access the contents of the global device memory (as well as of the texture and constant memory) allocated by a different process in two ways: *(i)* by injecting malicious CPU code in PixelVault that would force it to transfer data from the GPU's global device memory to the host, and *(ii)* by killing PixelVault and iteratively allocating large segments from the global device memory; eventually, the segments used by PixelVault would be allocated too. Even if the attacker does not know the exact loca-

[1] A stream in CUDA is a sequence of operations executed on the device in the order in which they were issued by the host code.

[2] The use of the kernel execution timeout ensures proper display rendering, in cases where a GPU kernel execution takes a prohibitively long time.

```
#define NREGS 64

__global__
void regextract(int *buf)
{
 int val;
 asm(".reg .u32 r<1>;");
 asm("mov.u32 %0, r0;" :  "=r"(val));
 buf[threadIdx.x * NREGS + 0] = val;
}
```

```
 code for sm_10
 Function :  _Z5regextractPi
 /*0000*/ /*0x4000000100000007*/ IMUL32I.U16.U16 R0, R0L, 0x40;
 /*0008*/ /*0x30020001c4100780*/ SHL R0, R0, 0x2;
 /*0010*/ /*0x2000c80104200780*/ IADD R0, g [0x4], R0;
 /*0018*/ /*0xd00e0005a0c00781*/ GST.U32 global14 [R0], R1;
```

Figure 4: The GPU kernel source code for extracting the contents of a single register, and the corresponding `.text` assembly snippet produced by the `cuobjdump` tool [2].

tion of secret data, algebraic and statistical attacks can be used to extract cryptography keys even among gigabytes of data [53].

To overcome the lack of sufficient protections for the off-chip global device memory, one possibility is to store the secrets in memory hierarchies that cannot be accessed directly from the host. These types of memory include all kinds of auxiliary memory (texture cache, constant cache, L1–L3 cache, and shared memory) that can only be accessed from the scope of a GPU kernel. However, the contents of all these types of caches also reside in the off-chip global device memory, and more importantly, the data stored there cannot be managed by the programmer. It is not possible to ensure that PixelVault's data will remain in the cache indefinitely, as some of it can occasionally be evicted.

In contrast, shared memory (comparable to scratch-pad memory in other architectures) is managed directly by the programmer, by explicitly storing data into it. The shared memory has kernel scope life-cycle and the data stored there is shared between the threads of a block. A GPU kernel executed from a different context cannot retrieve them, because CUDA resets this memory during context initialization [49]. However, we have verified that the contents of shared memory can be retrieved by post-executed kernels that have been spawned from the same CUDA context.

Fortunately, data stored in GPU registers cannot be retrieved, as GPU registers are always initialized to zero every time a new kernel is loaded on the GPU for execution, even from the same CUDA context. This can be easily verified and demonstrated using the following experiment. We create thread blocks that are enough to occupy all available streaming cores, and initialize all available GPU registers with a predetermined constant value, using inline PTX assembly. We also mark the identifier of each stream processor (SP) of which the registers have been initialized, by reading the special purpose read-only register `smid`. If not all streaming processors have been marked, the creation of new thread blocks continues. To obtain the contents of the GPU registers afterwards, we first reset the currently running kernel and launch a new one, using the commands found in `gdev_nvidia_nvc0.c` file of the Gdev source tree [7]; other NVIDIA architectures require slightly different commands. We only used the minimum set of commands that are required for launching a new kernel. These include setting local, shared, and global memory space; transferring parameters to the constant memory space; setting grids, blocks, and barriers; and setting the number of registers that are needed. The second kernel, part of which is shown in Figure 4, simply reads the data from the registers and writes it to a buffer allocated in global device memory. Even when the GPU kernel is running as part of the same context, all GPU registers are always initialized to zero.

The summary of protection levels for each memory type is shown in Table 2. The global, constant, and texture memories provide the weakest form of protection, as the data stored there can be obtained—in certain occasions—even by different unprivileged processes. The shared memory secures data accesses from different processes, as its content is automatically reset whenever a new CUDA context is created. Still, an adversary that has full control of the PixelVault's process can terminate the GPU kernel, and acquire the contents of the shared memory by launching a malicious kernel from the same CUDA context. The malicious kernel would simply perform some cryptanalysis in the contents of the shared memory to obtain the secret keys. Similar attacks can be performed in the L1–L3 caches of the global device memory, as recent GPU architectures use the same hardware resources for the L1 cache and the shared memory. In contrast, the hardware always resets the GPU registers to zero every time a new kernel is loaded on the GPU for execution, even from the same CUDA context. As such, in PixelVault we only use registers to store secret and private keys in clear-text form. In all other kinds of memory, keys are stored in an encrypted form to prevent their exposure in case of leakage.

4.3 Preventing Code Modification Attacks

Normally, GPU code is initially stored in global device memory for the GPU to execute it. Still, there are three levels of instruction caching (icache), of sizes 4 KB, 8 KB, and 32 KB, respectively [62]. Therefore, it is feasible to load the code to the icache (by carefully exercising all different execution paths), and then completely erase the code from global device memory, by transferring dummy data to the corresponding global device memory region via DMA. The code then is not possible to be flushed from the hardware-managed instruction cache—all code runs indefinitely, and any interruption results in immediate termination, which erases any existing state (as discussed in Section 4.1).

As long as the code fits in the icache, the program executes autonomously, without fetching any new instructions from global device memory. Thus, the code is protected from tampering due to the fact that icache is not addressable (and hence it cannot be accessed) from the host, and most importantly, icache starts with a clean state whenever a new kernel is loaded for execution—any previous data is flushed. Consequently, adversaries cannot access the code and extract any key, even if they are able to launch a malicious kernel.

A limitation of this approach is that the total size of the code footprint should be small enough to fit in the dedicated icache. For-

Memory type	Protection
Global Memory	no protection; data can be acquired subsequently, even from a different CUDA context or process address space.
Constant Memory	no protection; data can be acquired subsequently, even from a different CUDA context or process address space.
Texture Memory	no protection; data can be acquired subsequently, even from a different CUDA context or process address space.
Shared Memory	contents can be acquired by a subsequent GPU kernel that executes in the same CUDA context.
L1-L3 Cache	contents can be acquired through Shared Memory, as L1 is used in common with Shared Memory.
Registers	full protection; registers automatically reset to zero on each GPU kernel execution.

Table 2: Protection levels of each GPU memory type.

tunately, this is the case for our RSA and AES implementations. The code footprint of the RSA encryption operation is 6.9 KB and of AES encryption and decryption is 7.5 KB.

4.4 Key Storage

Due to the small number of available registers in current GPU models (Table 1), only a few number of keys can be stored each time. To overcome this space restriction, we use a separate KeyStore array that can hold an arbitrary number of cryptographic keys. The KeyStore resides in the global device memory and is encrypted with a master key. The master key is stored in the GPU registers, and thus only the GPU kernel can decrypt the KeyStore and retrieve the actual keys. Therefore, even if adversaries manage to acquire the KeyStore array from global device memory, they would only get the encrypted contents, which are useless.

Each KeyStore entry is encrypted independently. To access an encrypted key, PixelVault first transfers it to the GPU registers and then decrypt it with the master key, which is permanently loaded in the on-chip registers. This prevents an attacker from accessing the keys by continuously snooping over the KeyStore array until the moment it gets decrypted, and avoids leaving any unencrypted copies of cryptographic keys in global device memory.

We should note that the KeyStore structure is only needed for services that maintain a large number of private and secret keys. In other cases, the KeyStore can be disabled, and PixelVault can be configured to hold all keys in registers. This has the added benefit of avoiding the extra overhead of fetching and decrypting the cryptographic keys used from the KeyStore array.

4.5 Key Management

An aspect of our design that needs careful consideration is the transfer of the KeyStore's master key to the GPU registers, and the loading of PixelVault's native code in the instruction cache of the GPU. These operations should be performed at an early stage of the bootstrapping phase, before launching any user process or connecting to the Internet—as also suggested in previous works [40, 41]—and preferably from a non-volatile storage device, such as an external USB flash drive. This approach exposes any sensitive information on the CPU only for a minimal amount of time, which for servers happens only rarely, whenever the system boots.

To exclude the possibility that the master key is not copied to an intermediate buffer, before finally being transferred to the GPU, we explicitly allocate a page-locked memory region that can be accessed directly from the GPU. Otherwise, if a regular memory buffer is used, the driver will have to copy the key to an internal page-locked memory region, to which we do not have direct access for erase it afterwards. After the key has been transferred to the memory space of the GPU, it stored in the GPU registers, and all instances of the key in the GPU's and host's memory are erased.

```
int GPU_AES_encrypt_cbc(int keyID,
    unsigned char *in, unsigned char *out,
    size_t nbytes, unsigned char *ivec);
int GPU_AES_decrypt_cbc(int keyID,
    unsigned char *in, unsigned char *out,
    size_t nbytes, unsigned char *ivec);

int GPU_AES_encrypt_cbc_batch(int* key,
    unsigned char *in, unsigned char *out,
    size_t *offsets, size_t *nbytes,
    unsigned char *ivec, size_t total);
int GPU_AES_decrypt_cbc_batch(int* keyID,
    unsigned char *in, unsigned char *out,
    size_t *offsets, size_t *nbytes,
    unsigned char *ivec, size_t total);
```

Figure 5: The OpenSSL-compatible API for the 128-bit AES-CBC cipher. The first two functions process a single message at a time and can be used transparently by legacy applications. The last two functions process a batch of messages at a time, and are better suited for throughput-oriented setups.

A user or an application can share certificates or secret keys with the service by transferring them through the shared memory space. Every new cryptographic key is stored sequentially in the KeyStore array, and is identified by its index during data encryption and decryption. The GPU uses the index to acquire the specified key from the KeyStore array and perform the requested operation. However, a service cannot access the keys directly, as they are stored encrypted. Therefore, even if an adversary manages to inject malicious CPU code in the address space of the service, it is impossible to acquire the clear-text keys.

5. IMPLEMENTATION

We have implemented PixelVault on Linux v3.5.0, on top of the NVIDIA CUDA architecture v4.2 using the NVIDIA driver v304.54. Our prototype implementation currently supports both RSA and AES, and provides an OpenSSL-compatible API that enables existing applications or services to easily be ported on top of PixelVault with minimal modifications.

5.1 AES

We have ported AES-128 on the GPU, by storing the key and all intermediate states in GPU registers. AES divides each plaintext message into 128-bit fixed blocks and encrypts each block into ciphertext with a 128-bit key. The encryption algorithm consists of 10 rounds of transformations. Each round uses a different round key generated from the original key using Rijndael's key schedule. We have chosen to derive the round key at each round, instead

```
int GPU_RSA1024_private_decrypt(char *in,
    char *out, int rsaKeyID, size_t *offsets,
    size_t *nbytes, size_t total);

int GPU_RSA1024_private_decrypt_batch(char *in,
    char *out, int *rsaKeyID, size_t *offsets,
    size_t *nbytes, size_t total);
```

Figure 6: The OpenSSL-compatible API for the 1024-bit RSA cipher. The first function process a single message at a time, and can be used transparently by legacy applications. The second function process a batch of messages at a time, and is better suited for high-performance setups.

of using pre-expanded keys. Although this approach incurs more computational overhead, it reduces the number of registers needed. Overall, we need 16 bytes for the key, 16 bytes for the round key, and 16 bytes for the input block (which is modified in-place over the rounds and eventually contains the output block). Part of the remaining registers are used for local variables. The only data that is written back to global, off-chip device memory after the input block has been processed is the output block. This restrictive policy ensures that no sensitive information about the key is leaked to global device memory.

Figure 5 shows PixelVault's API functions for the CBC mode of AES-128. The CBC-encryption mode has a dependency on the result of the previous block, hence the encryption of the blocks of a single message is axiomatically serialized. Nevertheless, each thread can perform encryption operations using a different AES key independently, as it contains its own register space. On the other hand, CBC decryption can be parallelized at the block level, as the result of the previous block is already known at decryption time. Other modes of AES (such as ECB, CBC, and CTR) can be implemented in a similar way without significant extra effort.

5.2 RSA

We have implemented the RSA decrypt function for 1024-bit keys (Figure 6). We focused on the performance of RSA decryption for two reasons. First, RSA typically requires several decryption operations per key. Second, decryption is heavily used at the server side, where runtime performance is more critical.

Our GPU implementation exploits parallelism at the message level, similarly to previous GPU-based implementations [25, 30, 58]. For modular exponentiation, we use the Montgomery Multiplication for Multi-Precision Integers (CIOS method) [34], and we apply the straightforward right-to-left method, similar to [58]. During exponentiation, each thread needs three temporary values of $(n+2)$ words each, where n is the size of the key in bits. The three temporary values are used as input and output in a round-robin fashion. Overall, $3 * (n + 2)$ words are required, which results in 408 words for 1024-bit keys. Unfortunately, there is not always enough space to hold all three temporary values in registers (see Table 1). One solution is to store the three temporary values in shared memory. Each multiprocessor features up to 48 KB of shared memory, which, in contrast to the off-chip global device memory, cannot be accessed by the host. Even if adversaries manage to stop PixelVault's autonomous kernel and run a malicious one, they would only retrieve a single static image of the shared memory.

Still, this does not pose a significant risk for two reasons: (a) this requires very precise timing of the attack, and (b) even if the right timing can be achieved, the obtained fraction of the key is too small to pose a key leakage risk. The reason is that for any n, only the least significant k bits of the key can be recovered, with a $O(2^k)$ complexity. For example, if we assume that a Meet-in-the-Middle attack is feasible for up to $k = 128$ in a reasonable time, then the least 128 significant bits of the key are exposed to the adversary. This amount of bits is far from being critical for revealing the entire key, given that the critical limit for RSA is at least one fourth of the key size [13]. To prevent even the above unlikely leak, we have implemented an optional mode in which the intermediate values are stored in shared memory in an encrypted form. Only those intermediate values that are needed are decrypted—within GPU registers only— but with an additional cost. The encryption and decryption of the intermediate states are performed in 16-byte chunks, using the master key.

From the performance standpoint, previous work has shown that the GPU is better utilized when several messages are processed at once [24, 25, 30], which is also true for our implementation. To achieve optimal performance, our GPU-based OpenSSL version processes many messages in bulk. This is achieved by buffering several messages and transferring them to the GPU at once for parallel processing. Recall that each thread can perform encryption and decryption operations using a different key independently, as it contains its own register space.

6. EVALUATION

6.1 Security Analysis

We now evaluate the security properties of PixelVault by describing possible attacks, and showing how our proposed design protects against them. For some of the attacks, we used the Gdev framework [7], as it is open-source and provides more insights about low-level operational details than the official closed-source CUDA runtime.

6.1.1 Host Memory Attacks

We have implemented RSA and AES in a way that nothing but the scrambled output block is ever written into host memory. This is feasible, as GPUs maintain their own discrete memory spaces for manipulating input data. When the GPU has performed the desired cryptographic operation, the resulting output is transferred back to host memory. In the meantime, GPU execution continues completely isolated from the CPU, without being affected by side effects of the OS or the hardware, such as interrupt handling, scheduling, swapping, and ACPI suspend modes. As a result, keys or any intermediate states are never transferred indirectly to host memory.

6.1.2 Extracting Intermediate States

As we described in Section 4.4, secret and private keys reside unencrypted only in GPU registers. However, encryption and decryption operations take place in global device memory, which is accessible from the host via the PCIe interconnect. Therefore, it is possible for an adversary to perform cryptanalysis from any intermediate states extracted from the global device memory.

To defend against cryptanalysis, we perform both AES and RSA exclusively in on-chip memory. In particular, after a plaintext block is read from global device memory, nothing but the scrambled output block is written back. Essentially, no valuable information about the key or intermediate state is visible in the global device memory at any time. To ensure that no intermediate state resides in global memory, we stage data to the on-chip shared memory, which is not accessible by the host. Even in case an adversary terminates the GPU program and successfully acquires the contents of

the shared memory, only a single intermediate state will be accessible, and more cannot be obtained for performing successful cryptanalysis. To acquire further intermediate states, an attacker needs to restart the autonomous PixelVault GPU kernel; this is not possible though, as only the administrator can re-execute PixelVault from a clean state, after transferring the master key and native code from an external device, as we described in Section 4.5.

6.1.3 CPU Code Injection

In a typical scenario, attackers can exploit software vulnerabilities and manage to inject code of their choice to a running service. Sensitive data, such as private keys, that are stored in the address space of the process, can be easily acquired. In contrast, hiding sensitive data in the on-chip memory space of the GPU using PixelVault prevents access even to fully privileged processes.

To verify this, we attached `cuda-gdb`, the CUDA debugger, to PixelVault using full-administrator privileges for tracing its execution. The `cuda-gdb` is very similar to `gdb` and allows tracing of both CPU and GPU variables, as well as the execution of arbitrary CPU and GPU code. Running PixelVault under a debugger allows us to transfer data from the off-chip global device memory. However, we are still not able to extract any key, as they are kept encrypted. Furthermore, we are not able to access any on-chip memory (i.e., shared memory and caches) even if PixelVault is compiled with debug-able device code (using both `-g` and `-G` flags). The reason is that the non-preemptive GPU execution does not allow adding breakpoints inside a kernel that is already running; to trace the execution of a kernel, the breakpoints have to be added before the kernel has been loaded on the GPU for execution. As we start the GPU kernel from a clean state, it is impossible for an attacker to trace the autonomous, self-contained GPU code of PixelVault.

6.1.4 GPU Code Injection

All GPU code is loaded in the global device memory before execution. The GPU code base of PixelVault is small, which can allows it to be formally verified, to prevent potential exploitation due to buggy code. However, accessing the code's memory region is still feasible, as the global device memory does not provide any access protection. An attacker could, for example, inject malicious GPU code by transferring it via PCIe to the appropriate memory region. The malicious code could contain commands for forcing the registers' contents to be written to the global device memory, where they could then easily be retrieved via the PCIe bus.

We have modified the Gdev framework to explicitly rewrite the memory region where native code is stored. Similar attacks can also be performed using the official CUDA debugger interface [46]. As we described in Section 4.3 though, PixelVault is tamper-resistant against GPU code modifications, as it forces all code to be loaded to the instruction cache. Even after erasing all PixelVault's native code from the global device memory, the GPU still executes the original, unmodified code of PixelVault from the instruction cache. Therefore, an attacker cannot overwrite PixelVault, because the instruction cache cannot be flushed without loading a new GPU kernel.

6.1.5 Simultaneous GPU Kernel Execution

Starting with the Fermi architecture [44] and onwards, different (relatively small) kernels of the same CUDA context can occasionally execute concurrently, allowing maximum utilization of GPU resources. However, all stream multiprocessors (SMs) are first filled with threads from the first kernel, and only if the remaining resources are sufficient, threads from a second kernel can be spawned. As a result, if all SMs are filled, threads from another kernel cannot execute before the initial kernel completes its execution. During initialization, PixelVault spawns a large number of threads that remain idle, busy-waiting, as we described in Section 4.1, occupying all available registers and shared memory. As a result, a malicious kernel cannot be launched simultaneously.

#Msgs	CPU	GPU [25]	PixelVault	PixelVault (w/ KeyStore)
1	1632.7	15.5	15.3	14.3
16	1632.7	242.2	240.4	239.2
64	1632.7	954.9	949.9	939.6
112	1632.7	1659.5	1652.4	1630.3
128	1632.7	1892.3	1888.3	1861.7
1024	1632.7	10643.2	10640.8	9793.1
4096	1632.7	17623.5	17618.3	14998.8
8192	1632.7	24904.2	24896.1	21654.4

Table 3: Decryption performance of 1024-bit RSA (#Msgs/sec).

6.1.6 Register Spilling In Global Device Memory

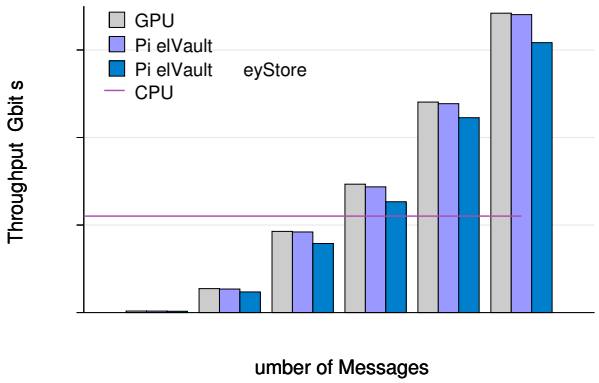
The registers that will be used by a GPU kernel are declared once, at compile time. As we can see in Table 1, the number of registers contained in GPUs is limited, and varies from platform to platform. When the number of declared registers exceeds the limit, the extra registers are mapped in global device memory, hence their contents can be exposed to adversaries. To rule out this possibility, we explicitly declare as many registers as the underlying hardware device provides. The number of declared registers serves as a heuristic for the compiler to decide when to spill registers during the compilation of the PTX code. By supplying the `--ptxas-options='-v'` flag to the `nvcc` compiler, we are explicitly notified if any spilling has occurred.

It would also be possible that registers could be spilled in global device memory when a context switch between different warps occurs. In contrast to CPUs, however, GPUs are non-preemptive processors, and thus the contents of GPU registers are never saved (in order to be restored later and continue running where it previously left off). Still, thread warps can be switched, e.g., when a warp is waiting for memory I/O another warp can be scheduled for running. According to NVIDIA, no state is saved when context switching between thread warps occurs, for performance reasons [36]. This is actually the reason that a large number of registers reduces the amount of thread parallelism.

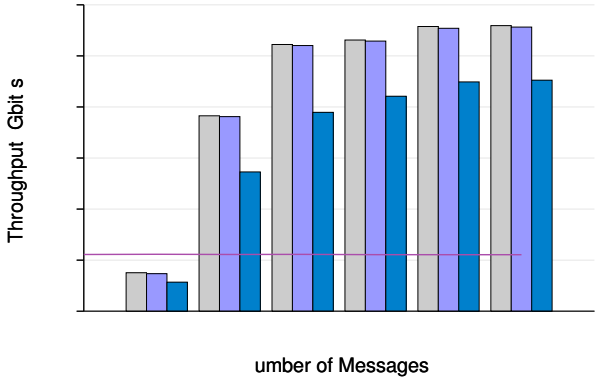
6.2 Performance Analysis

We now assess the performance of PixelVault in comparison to the standard CPU implementation (OpenSSL [5]). Our base system consists of two Intel Xeon E5520 Quad-core CPUs (2.27GHz, 8192KB L3-cache), 12GB of RAM, and a GeForce GTX480.

Table 3 shows the throughput of RSA on a single CPU core, on the GPU as reported by Harrison and Waldron [25], and using our PixelVault implementation. We evaluate PixelVault with and without the KeyStore structure. When the KeyStore structure is disabled, only a single RSA key is loaded on the registers (appropriate for simple setups that use only a single RSA key). We observe that the GPU performance is low when the number of messages is small, regardless of whether the KeyStore is used or not. With only one ciphertext message per invocation, the GPU has a throughput about two orders of magnitude worse compared to the CPU implementation. However, given enough parallelism, the GPU achieves a much higher throughput than the CPU. PixelVault has almost the same performance with the vanilla GPU-based RSA implementa-



(a) Encryption.



(b) Decryption.

Figure 7: Sustained throughput for 128-bit AES-CBC.

tion, as the autonomous execution of the GPU does not add any extra overhead. Enabling the KeyStore adds a small overhead—ranging between 1–15%—to the overall PixelVault performance. This is due to the copying of each key from the KeyStore array to the registers and its decryption. The RSA key is decrypted using the AES implementation that offers execution times proportionally smaller compared to the RSA implementation. In addition, a significant part of the overall time is spent on PCIe transfers, and thus the overhead introduced by the KeyStore mechanism is ameliorated.

We note that the implementation of the RSA algorithm is based on the work of Harrison and Waldron [25]. Further optimizations have been proposed since then, which could boost performance even further. For example, Jang et al. [30] report that with 16 messages, the performance of the GPU is equal to that of the CPU, in contrast to our implementation which needs about 110 messages to match the CPU performance. We are currently working on integrating these optimizations into our prototype implementation.

Figure 7 shows the performance of AES-CBC on a single CPU core (horizontal line), as well as on the GPU and using PixelVault (bars). We fix the message size to 16KB, the largest size supported by SSL, and increase the number of messages from 1 to 4096. The encryption mode of AES (Figure 7(a)) cannot be parallelized at the block level, and thus a sufficient number of messages is required to sustain acceptable performance. When using a single message, AES-enc achieves 17.4 Mbit/s, which gradually increases to 273.6 Mbit/s when processing 16 messages, and 3.4 Gbit/s for 4096 messages. Again, PixelVault achieves almost the same performance as the default GPU-based AES implementation. However, the performance decreases to 3 Gbit/s when the KeyStore is enabled, yielding a 13% decrease. The size of the AES keys is proportionally smaller compared to the size of the messages (indicatively, 16 bytes for AES-128), hence the copying and decryption of each key is quick, compared to the ciphering operations that follow. The CPU implementation achieves 1.1 Gbit/s and 1.2 Gbit/s for encryption and decryption, respectively, on a single CPU core.

In contrast, the decryption mode of AES ((Figure 7(b))) can be parallelized at the block level and achieves 753.2 Mbit/s even when processing a single message at a time. The peak performance of AES-dec is 5.5 Gbit/s when processing 4096 messages at once. The KeyStore adds a 20% overhead, limiting performance to 4.5 Gbit/s. Modes that can be parallelized at the block level, such as CTR, have a similar performance behaviour.

7. DISCUSSION AND LIMITATIONS

Dedicated GPU execution. PixelVault requires a dedicated GPU that is used exclusively for protecting secret keys and carrying out cryptographic operations. As a result, the GPU cannot be used by other programs or the OS for other purposes, e.g., for graphics rendering or general-purpose computations. Fortunately, recent advances in CPU architectures show that current CPU chips are already equipped with integrated graphics processors, e.g., the AMD APU [31] or the Intel HD Graphics [20]. In such cases, the integrated GPU can be used for performing any graphics-related operation; otherwise, a second, separate, GPU should be acquired and placed on a different PCIe slot. As PixelVault is mostly tailored to server applications, which typically run on headless machines, the requirement of a dedicated GPU is not a limiting factor.

Portability. Our design is based on current GPU architectures that maintain (primarily) two basic properties: (i) non-preemptiveness and (ii) on-chip memory operation only. These two properties are available to all official CUDA-enabled NVIDIA models (released after 2006 and onwards). We note though that our design may be generalized to other computational devices as well, as long as they maintain the same properties.

Misusing PixelVault for encrypting/decrypting messages. PixelVault cannot verify whether a request for an operation has been received from a benign or a malicious user. As a result, an attacker who has compromised the base system could leverage PixelVault to encrypt and decrypt messages. Still, the adversary cannot steal any key stored in PixelVault.

Generation of secret keys or key pairs at run-time. Many services require the creation of session keys or key pairs at run-time. For example, SSL-enabled services create a new secret key after the client has verified the server's certificate. Although the leakage of session keys is not considered as critical as the leakage of secret keys, creating secret keys or key pairs in a secure way is definitely a desirable functionality. PixelVault can easily provide this functionality by generating new keys in the on-chip memory, and securely storing them in the KeyStore structure. The service that requested the keys can refer to each generated key by its unique ID that is returned by PixelVault, and easily use them to encrypt or decrypt messages. We plan to implement this functionality in the near future.

TPMs. Trusted Platform Modules (TPMs) provide security-critical functions, such as secure storage and attestation of platform state and identities, and are mainly used to authenticate the base platform during the bootstrapping phase or generating hardware-protected key pairs. However, due to their limited storage space (the PC TPM specification mandates only 1,280 bytes of NVRAM [8]), their limited support of cryptographic algorithms (current version supports only RSA, SHA1, and HMAC) [8], and their low perfor-

mance (about 1.4 SSL handshakes per second [1]), they are not appropriate for carrying out intensive and continuous cryptographic operations, such handling a server's many concurrent TLS connections [1]. In contrast, PixelVault provides a fully programmable, secure, and fast framework for performing cryptographic operations, while ensuring that no secret or private key will leak even when the base system is fully compromised.

Denial-of-Service Attacks. Adversaries who have compromised the base system can easily disrupt the operation of PixelVault. For example, they can easily delete or modify input or output data by accessing the shared page-locked memory region, kill or suspend the execution of PixelVault, or terminate the interconnection of the GPU with the base system by sending a PCIe reset. As the main purpose of PixelVault is to protect secret keys, defending against these attacks is out of the scope of this work.

GPU Kernel Execution Timeouts. Due to the non-preemptive execution of GPUs, most operating systems use kernel execution timeouts to prevent system hangs when the GPU is also used for display rendering. Hence, any operation that is executed on a GPU with a display for more than a few seconds will be killed to ensure proper display rendering. PixelVault requires a dedicated GPU, hence we explicitly disable the kernel execution timeout through the graphics driver interface, to ensure that it will never terminate. An attacker might be able to terminate PixelVault's kernel by setting this timeout. However, this will only result in a DoS attack, similar to those described in the previous paragraph—no key will be leaked, given that GPU registers are erased upon termination.

Side-channel Attacks. It has been demonstrated that software side-channel attacks are possible based on inter-process leakage through the state of the memory cache of the CPU [22, 59]. These attacks allow an unprivileged process to attack other processes that run concurrently on the same processor, despite partitioning methods such as memory protection, sandboxing, and virtualization. Certain types of these attacks can be quite powerful, as they rely on merely monitoring the cache effects of cryptographic operations. PixelVault raises the bar against cache-based attacks, since *only one* kernel occupies the GPU at a time; no other GPU kernel can be executed in parallel to monitor the behavior of the cache.

In addition, timing attacks enable an attacker to extract secrets by observing the time it takes for a system to respond to various queries [11, 14, 35]. Although defending against this type of attacks is out of the scope of this work, a possible approach to enhance PixelVault against them is to implement all sensitive operations so that they consume a constant number of cycles, irrespectively of any combination of key and data, so as to make any timing based analysis hard [28].

Cold-boot Attacks. It has been demonstrated that keys can be extracted by freezing memory chips and inspecting their contents [23], an attack widely known as *cold boot attack*. It is hard to assess whether cold boot attacks are applicable on GPUs. We speculate that this might be possible, as graphics cards can be removed with the same ease as RAM DIMMs. PixelVault, however, is not vulnerable to cold boot attacks, because nothing sensitive is stored in DRAM—keys are only exposed in registers.

8. RELATED WORK

Many research works have focused on the implementation of cryptographic operations using GPUs [24, 25, 30]. Their data parallel architecture makes them attractive for the implementation of both symmetric and asymmetric cryptographic algorithms. The majority of these approaches has focused extensively on the implementation of widely-used cryptographic algorithms, such as AES and RSA. Cook et al. [18] describe an implementation of AES on an NVIDIA GeForce3 card, which provides little programmability. Szerwinski et al. [58] describe implementations of 1024 and 2048-bit modular exponentiations based on both the radix and RNS approaches, using an NVIDIA 8800GTS. Harrison et al [25] present a GPU implementation of a 1024-bit RSA decrypt primitive, outperforming a comparable CPU implementation by up to 4 times.

Besides performance, little focus has been placed on increasing the security of cryptographic implementations using GPUs, although the potential of modifying the GPU to a minimal secure computing base is not new. Cook et al [17] presented a mechanism to transfer encrypted video that is only decrypted once on the GPU. Based on this idea, our key insight is to demonstrate that the cryptographic operations executed on the GPU cannot only benefit in terms of performance, but also in terms of security, by keeping sensitive keys away from the CPU and the host memory.

Privilege separation is a good practice to restrict the number of operations executed with elevated privileges. Applications should be designed with the principle of least privilege, i.e., every operation should be executed with the minimum level of privileges. Provos et al. [50] demonstrate the value of privilege separation in OpenSSH. Privman [33] provides an API that can be used to integrate privilege separation into existing applications. Privtrans [15] allows the automatic integration of privilege separation, with the aid of a few annotations by the programmer. However, it has been shown that sensitive data or system objects (e.g., memory, the environment, memory mappings, file system information, and file descriptors) may still be leaked when, for instance, a trusted process of a partitioned application spawns an untrusted child process [54].

Many reseach efforts have recently focused on systems that support trusted computation in hostile operating systems [16, 26, 29]. These systems are designed towards a generic solution for protecting computation performed by any application, even by non sensitive ones, when the host is compromised. In such a setting, a process responsible for cryptographically signing a message would never expose its keys to the operating system, and therefore encryption will remain functional even when the operating system is compromised. Unfortunately, these systems require applications running in a hypervisor [16, 26], introducing significant performance overhead, or the addition of extra hardware abstraction layers and the re-compilation of the operating system [29]. We anticipate that these approaches will be eventually incorporated in commodity operating systems in the future. Nevertheless, in this paper, we seek for a less intrusive approach, that requires little modifications in current commodity systems, does not allow for trusted system-wide computation, but allows sensitive information, such as cryptographic keys, to remain secret, even when the host is compromised.

Similar to our purposes, recent work has pursued the idea of holding the cryptographic key solely in the registers of the CPU, for AES [41, 56] and RSA [21]. The secret key, then, cannot be traced in main memory, making cold boot attacks pointless. These approaches require a trusted and bug-free component for ensuring that an adversary cannot compromise part of the system and extract the keys from the CPU registers. Still, a DMA-capable adversary, with read and write access to the host physical memory, can extract the secret keys from the CPU into the target system's memory, from which they can be retrieved using a normal DMA transfer [12].

On the opposite aspect, many works talk about the danger of leaking data when offloading tasks to the GPU, in multi-user or virtualized environments [37, 49]. Di Pietro et al. [49] talk about the danger of leaking data from specific GPU memory hierarchies, namely the shared memory and global memory. They also mention that the registers contents can be exposed in cases where the developer declares more registers than the GPU contains; in such cases

where the GPU runs out of hardware registers, it can transparently leverage global memory ("register spilling" [39]). To rule out this possibility, PixelVault declares as many registers as the unterlying hardware device provides, and force the compiler to explicitly notify if any register spilling occurs. Similarly, Maurice et al. [37] analyze the behavior of GPU global memory and show that an adversary can recover data of a previously executed GPGPU application from the global device memory. In contrast with these works, we focus on memory hierarchies that provide a safe house for storing data, preventing *any* leakage. Such memories, include the hardware registers and the instruction caches, that are contained in modern graphics processors. By carefully leveraging these memory types in combination with the nonpreemptiveness execution model of the GPUs (that guarantees that no state will be stored somewhere else due to context-switching), we have managed to design a prototype architecture, namely PixelVault that is tamper resistant and also offers a secure environment for storing secrets.

Finally, Intel recently introduced SGX (Software Guard Extensions) [27], a set of new CPU instructions for establishing private memory regions. SGX allows an application to instantiate a protected container, which is a protected area in the application's address space that provides confidentiality and integrity even in the presence of privileged malware. SGX could potentially be used to implement similar functionality to PixelVault, using an SGX container for secure storage, and taking advantage of the CPU's cryptographic instructions. In the future, it would be interesting to compare the performance and characteristics of the two approaches.

9. CONCLUSION

We have presented the design and implementation of PixelVault, a GPU-based system that implements AES and RSA in a way that preserves the secrecy of keys even against attacks that fully compromise the host system. By taking advantage of the capabilities of modern graphics processing units (GPUs), PixelVault safely stores any sensitive information and offloads computationally-intensive cryptographic operations on the GPU. The underlying idea is to perform cryptographic operations entirely on the GPU, without involving the host or any memory that can be accessed by the host (even with full administrator permissions). This implies that (a) no secret key (nor the key schedule or intermediate states) ever get to host-accessible memory, and (b) GPU memory—where keys are stored—cannot be inspected while cryptographic operations are carried out. Our only requirement is that the system is initially bootstrapped in a trusted environment. Once GPU storage is initialized with the keys, PixelVault prevents key leakage even in case of full system compromise. As part of our future work, we plan to adapt our framework to other ciphers and application domains, and also to explore the design of a generic GPU-based framework for facilitating privilege partitioning of existing applications. We also plan to explore how our techniques could be applied in mobile and embedded devices. This may be harder to achieve though, as our design requires a dedicated GPU—otherwise the mobile device would not be able to render graphics properly.

Acknowledgments

We would like to thank our shepherd Roberto Di Pietro and the anonymous reviewers for their valuable feedback. This work was supported in part by the General Secretariat for Research and Technology in Greece with a Research Excellence grant, by the FP7-PEOPLE-2010-IOF project XHUNTER No. 273765, and by the FP7 projects NECOMA and SysSec, funded by the European Commission under Grant Agreements No. 608533 and No. 257007.

10. REFERENCES

[1] Benchmarking TPM-backed SSL. http://blog.habets.pp.se/2012/02/Benchmarking-TPM-backed-SSL.

[2] CUDA Binary Utilities. http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

[3] Nouveau: Accelerated Open Source driver for nVidia cards. http://nouveau.freedesktop.org/.

[4] NVIDIA Developer Forums - CUDA kernel timeout. https://devtalk.nvidia.com/default/topic/417276/cuda-kernel-timeout/.

[5] OpenSSL Project. http://www.openssl.org/.

[6] pscnv - PathScale NVIDIA graphics driver. https://github.com/pathscale/pscnv.

[7] shinpei0208 / gdev. https://github.com/shinpei0208/gdev.

[8] TCG PC Client Specific - TPM Interface Specification (TIS) Version 1.2. http://www.trustedcomputinggroup.org/files/resource_files/87BCE22B-1D09-3519-ADEBA772FBF02CBD/TCG_PCClientTPMSpecification_1-20_1-00_FINAL.pdf.

[9] The Heartbleed Bug. http://heartbleed.com/.

[10] Who holds the encryption keys? http://www.computerworld.com/s/article/9225414/Who_Holds_the_Keys_.

[11] D. J. Bernstein. Cache-timing Attacks on AES, 2004.

[12] E.-O. Blass and W. Robertson. TRESOR-HUNT: Attacking CPU-bound Encryption. In *ACSAC*, 2012.

[13] D. Boneh, G. Durfee, and Y. Frankel. An Attack on RSA Given a Small Fraction of the Private Key Bits. In *Lecture Notes in Computer Science*, volume 1514 of *Lecture Notes in Computer Science*, pages 25–34. Springer, 1998.

[14] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. In *USENIX Security*, 2003.

[15] D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *USENIX Security*, 2004.

[16] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS XIII*, 2008.

[17] D. Cook, R. Baratto, and A. Keromytis. Remotely Keyed Cryptographics Secure Remote Display Access Using (Mostly) Untrusted Hardware. In *ICICS*, 2005.

[18] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *CT-RSA*, 2005.

[19] J. Daemen and V. Rijmen. AES Proposal: Rijndael, 1998.

[20] Eliseo Hernandez. Accelerate Performance Using OpenCL with Intel HD Graphics. http://software.intel.com/en-us/articles/accelerate-performance-using-opencl-with-intel-hd-graphics.

[21] B. Garmany and T. Müller. PRIME: Private RSA Infrastructure for Memory-less Encryption. In *ACSAC*, 2013.

[22] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE S&P*, 2011.

[23] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.

[24] O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security*, 2008.

[25] O. Harrison and J. Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In *AFRICACRYPT*, 2009.

[26] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *ASPLOS*, 2013.

[27] Intel. Software Guard Extensions Programming Reference. `https://software.intel.com/sites/default/files/329298-001.pdf`.

[28] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with Constant Computational Overhead. In *STOC*, 2008.

[29] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *ASPLOS*, 2014.

[30] K. Jang, S. Han, S. Han, K. Park, and S. Moon. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *NSDI*, 2011.

[31] Jon Stokes. AMD reveals Fusion CPU+GPU, to challenge Intel in laptops. `http://arstechnica.com/business/2010/02/amd-reveals-fusion-cpugpu-to-challege-intel-in-laptops/`.

[32] S. Kato. Implementing Open-Source CUDA Runtime. 2013.

[33] D. Kilpatrick. Privman: A Library for Partitioning Applications. In *FREENIX*, 2003.

[34] C. Koc, T. Acar, and J. Kaliski, B.S. Analyzing and Comparing Montgomery Multiplication Algorithms. *Micro, IEEE*, 16(3):26–33, 1996.

[35] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96*, 1996.

[36] Luitjens, Justin and Rennich, Steven. CUDA Warps and Occupancy. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf`, 2011.

[37] C. Maurice, C. Neumann, O. Heen, and A. Francillon. Confidentiality Issues on a GPU in a Virtualized Environment. In *FC*, 2014.

[38] K. Menychtas, K. Shen, and M. L. Scott. Enabling OS Research by Inferring Interactions in the Black-box GPU Stack. In *USENIX ATC*, 2013.

[39] P. Micikevicius. Local Memory and Register Spilling. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf`.

[40] T. Müller, A. Dewald, and F. C. Freiling. AESSE: A Cold-boot Resistant Implementation of AES. In *EuroSec*, 2010.

[41] T. Müller, F. C. Freiling, and A. Dewald. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security*, 2011.

[42] NVIDIA. CUDA Programming Guide, version 4.0. `http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

[43] NVIDIA. Dynamic Parallelism in CUDA. `http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf`.

[44] NVIDIA. Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[45] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[46] NVIDIA Developer Zone. Flushing Instruction Cache on GPU. `https://devtalk.nvidia.com/default/topic/467841/flushing-instruction-cache-on-gpu/`.

[47] NVIDIA Developer Zone. PTX ISA :: CUDA Toolkit Documentation. `http://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

[48] D. R. Piegdon and L. Pimenidis. Targeting Physically Addressable Memory. In *DIMVA*, 2007.

[49] R. D. Pietro, F. Lombardi, and A. Villani. CUDA Leaks: Information Leakage in GPU Architectures. *ArXiv*, May 2013.

[50] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *USENIX Security*, 2003.

[51] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of ACM*, 21, February 1978.

[52] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *SOSP*, 2011.

[53] A. Shamir and N. v. Someren. Playing 'Hide and Seek' with Stored Keys. In *FC*, 1999.

[54] U. Shankar and D. Wagner. Preventing Secret Leakage from fork(): Securing Privilege-Separated Applications. In *ICC*, 2006.

[55] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. In *ASPLOS*, 2013.

[56] P. Simmons. Security through Amnesia: A Software-based Solution to the Cold-boot Attack on Disk Encryption. Technical report, 2011.

[57] P. Stewin and I. Bystrov. Understanding DMA Malware. In *DIMVA*, 2013.

[58] R. Szerwinski and T. Güneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *CHES*, 2008.

[59] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23, 2010.

[60] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *USENIX ATC*, 2014.

[61] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *CCS*, 2011.

[62] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *ISPASS*, 2010.