Security Applications of GPUs

Giorgos Vasiliadis
Foundation for Research and Technology – Hellas (FORTH)
Outline

• Background and motivation
• GPU-based Malware Signature-based Detection
  – Network intrusion detection/prevention
  – Virus scanning
• GPU-assisted Malware
  – Code-armoring techniques
  – Keylogger
• GPU as a Secure Crypto-Processor
• Conclusions
Outline

• Background and motivation
• GPU-based Malware Signature-based Detection
  – Network intrusion detection/prevention
  – Virus scanning
• GPU-assisted Malware
  – Code-armoring techniques
  – Keylogger
• GPU as a Secure Crypto-Processor
• Conclusions
Why GPU?

• General-purpose computing
  – Flexible and programmable
  – Portability

• Powerful and ubiquitous
  – Dominant co-processor
  – Constant innovation
  – Inexpensive and always-present

• Data-parallel model
CPU vs. GPU

CPU

Xeon X5550:
- 4 cores
- 731M transistors

GPU

GTX480:
- 480 cores
- 3,200M transistors
Single Instruction, Multiple Threads

- Example: Vector addition

CPU code

```c
void vecadd(
    int *A, int *B, int *C, int N)
{
    int i;
    //iterate over N elements
    for (i=0; i<N; ++i)
        C[i] = A[i] + B[i];
}

vecadd(A, B, C, N);
```
Single Instruction, Multiple Threads

• Example: Vector addition

CPU code

```c
void vecadd(
    int *A, int *B, int *C, int N)
{
    int i;
    //iterate over N elements
    for (i=0; i<N; ++i)
        C[i] = A[i] + B[i];
}

vecadd(A, B, C, N);
```

GPU code

```c
__global__ void vecadd(
    int *A, int *B, int *C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

//Launch N threads
vecadd<<<1, N>>>(A, B, C);
```

Giorgos Vasiliadis
Single Instruction, Multiple Threads

• Example: Vector addition

```c
void vecadd(
    int *A, int *B, int *C, int N)
{
    int i;
    //iterate over N elements
    for (i=0; i<N; ++i)
        C[i] = A[i] + B[i];
}

vecadd(A, B, C, N);
```

```c
__global__ void vecadd(
    int *A, int *B, int *C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

//Launch N threads
vecadd<<<1, N>>>(A, B, C);
```
**Single Instruction, Multiple Threads**

- Threads within the same *warp* have to execute the same instructions

- *Great for regular computations!*

Giorgos Vasiliadis
Outline

• Background and motivation
• **GPU-based Signature Detection**
  – Network intrusion detection/prevention
  – Virus matching
• **GPU-assisted Malware**
  – Code-armoring techniques
  – Keylogger
• **GPU as a Secure Crypto-Processor**
• Conclusions
Network Intrusion Detection Systems

• Typically deployed at ingress/egress points
  – Inspect *all* network traffic
  – Look for suspicious activities
  – Alert on malicious actions
Challenges (1)

- **Traffic rates** are increasing
  - 10 Gbit/s Ethernet speeds are common in metro/enterprise networks
  - Up to 40-100 Gbit/s at the core
Challenges (2)

- Ever-increasing need to perform more complex analysis at higher traffic rates
  - Deep packet inspection
  - Stateful analysis
  - 1000s of attack signatures
Designing NIDS and AVs

• Fast
  – Need to handle many Gbit/s
  – Scalable
    • The future is many-core

• Commodity hardware
  – Cheap
  – Easily programmable
Today: fast or commodity

• Fast “hardware” IDS/IPS
  – FPGA/TCAM/ASIC based
  – Usually, tied to a specific implementation
  – Throughput: High

• Commodity “software” NIDS/NIPS and AVs
  – Processing by general-purpose processors
  – Throughput: Low

IDS/IPS Sensors
(10s of Gbps)
~ US$ 20,000 - 60,000

IDS/IPS M8000
(10s of Gbps)
~ US$ 10,000 - 24,000

Open-source S/W
≤ ~1 Gbps
Single-threaded NIDS performance

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
(msg:“WEB-PHP horde help module arbitrary command execution attempt”;
flow:established,to_server; uricontent:" /services/help/"; pcre:” /[\?\x20\x3b\x26\]module=[a-zA-Z0-9]*[\^\x3b\x26]/U”); metadata:service http;

* PCRE: Perl Compatible Regular Expression
Single-threaded NIDS performance

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
(msg:“WEB-PHP horde help module arbitrary command execution attempt”;
flow:established,to_server; uricontent:” /services/help/”;
pcre:” [/\?\x20\x3b\x26]module=[a-zA-Z0-9]*[^\x3b\x26]/U”); metadata:service http;
Single-threaded NIDS performance

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80 (msg:"WEB-PHP horde help module arbitrary command execution attempt"; flow:established,to_server; uricontent:" /services/help/"; pcre:" /[\?\x20\x3b\x26]*[\^\x3b\x26]*/U"); metadata:service http;
Single-threaded NIDS performance

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
  (msg:“WEB-PHP horde help module arbitrary command execution attempt”;
  flow:established.to_server; uricontent:" /services/help/";
  pcre "(/[\?\x20\x3b\x26a-zA-Z0-9]*[^\x3b\x26]*/U")";
  metadata:service http;
Single-threaded NIDS performance

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
  (msg:"WEB-PHP horde help module arbitrary command execution attempt";
  flow:established,to_server; uricontent:" /services/help/"; pcre:"/[\?\x20\x3b\x26]module=[a-zA-Z0-9]*[^\x3b\x26]/U"); metadata:service http;
Single-threaded NIDS performance

• Vanilla Snort: 0.2 Gbit/s
Single-threaded NIDS performance

- Vanilla Snort: 0.2 Gbit/s
Problem #3: Pattern matching is the bottleneck

- On a Intel Xeon X5520, 2.27 GHz, 8 MB L3 Cache
  - String matching analyzing bandwidth per core: **1.1 Gbps**
  - PCRE analyzing bandwidth per core: **0.52 Gbps**
Offload pattern matching on the GPU

NIC → Preprocess → Pattern matching → Output

Strings pcre

Giorgos Vasiliadis
Pattern matching on the GPU

- Data level parallelism == Packet level parallelism
  - Uniformly one core for each reassembled packet stream
Pattern matching on the GPU

Both **string searching** and **regular expression matching** can be matched efficiently by combining the patterns into **Deterministic Finite Automata (DFA)**.
Pattern matching on the GPU

NVIDIA GTX 480 GPU
On an Intel Xeon X5520, 2.27 GHz, 8 MB L3 Cache
String matching analyzing bandwidth: \textbf{1.1 Gbps} \text{30 Gbps}
PCRE analyzing bandwidth: \textbf{0.52 Gbps} \text{8 Gbps}

Giorgos Vasiliadis
Pipelining CPU and GPU

• Double-buffering
  – Each CPU core collects new reassembled packets, while the GPUs process the previous batch
  – Effectively hides GPU communication costs
Multi-Parallel Network Intrusion Detection

- Vanilla Snort: 0.2 Gbit/s
- With multiple CPU-cores: 0.9 Gbit/s
- With GPU: 5.2 Gbit/s
Outline

• Background and motivation
• **GPU-based Signature Detection**
  – Network intrusion detection/prevention
  – **Virus matching**
• GPU-assisted Malware
  – Code-armoring techniques
  – Keylogger
• GPU as a Secure Crypto-Processor
• Conclusions
Anti-Virus Databases

- Contain thousands of signatures
  - ClamAV contains more than 60K signatures
Anti-Virus Databases

- ClamAV signatures are significantly longer than NIDS
  - Length varying from 4 to 392 bytes

> 80%

> 90%

Giorgos Vasiliadis
Anti-Virus Databases

• Memory requirements

~14 GB
~0.8 GB
Opportunity: Prefix Filtering

• Take the first $n$ bytes from each signature
  – e.g.
  Worm.SQL.Slammer.A:0:*:
  \[4e65742d576f726d2e57696e33322e536c616d6d65725554\]

• Compile all $n$-bytes sub-signatures into a single *Scanning Trie*

• The Scanning Trie can quickly filter clean data segments in linear time.
Scanning Trie

• Variable trie height

Depth 0

Depth 1

...
Longer prefix = Fewer matches
Longer prefix = More memory

![Graph showing total memory (MBs) against prefix length]

- Throughput (GBits/sec)
  - GrAVity
  - ClamAV (1x core)
  - ClamAV (8x cores)

Fig. 6. Memory requirements for the storage of the DFA as a function of the signature prefix length.

Fig. 7. Performance of GrAVity and ClamAV. We also include the performance number for ClamAV running on 8 cores. The CPU-only performance is still of magnitude less than the GPU-assisted. The numbers demonstrate that additional CPU cores offer less benefit than that of utilizing the GPU.

Throughput In this experiment we evaluate the performance of GrAVity compared to vanilla ClamAV. Figure 7 shows the throughput achieved for different prefix lengths. The overall throughput increases rapidly, raising at a maximum of 20 Gbits/s. A plateau is reached for a prefix length of around 10.

As the prefix length increases, the number of potential matches decreases, as shown in Figure 9. This results in lower CPU post-processing, hence overall application throughput increases. In the next section, we investigate in more detail the breakdown of the execution time.

Giorgos Vasiliadis
Virus Scanning on the GPU

• Each thread operate on different data
  – May overlap for spanning patterns, but …
  – … no communication/synchronization costs.
  – Highly scalable (million threads can run in parallel)
Execution Time Breakdown

- CPU time results in 20% of the total execution time, with a prefix length equal to 14
GPU vs CPU

- Up to 20 Gbps end-to-end performance

Giorgos Vasiliadis
Summary

• Both *Network Intrusion Detection* and *Virus Scanning* on the GPU are practical and fast!

• More technical details
  – See our *RAID’08, RAID’09, RAID’10, CCS’2011, and USENIX ATC’14* papers
Outline

• Background and motivation
• GPU-based Malware Signature Detection
  – Network intrusion detection/prevention
  – Virus scanning
• **GPU-assisted Malware**
  – Code-armoring techniques
  – Keylogger
• GPU as a Secure Crypto-Processor
• Conclusions
Motivation

• Malware continually seek new methods for hiding their malicious activity, ...
  – Packing/Polymorphism
  – Polymorphism
• ... as well as, hinder reverse engineering and code analysis
  – Code obfuscation
  – Anti-debugging tricks
• Is it possible for a malware to exploit the rich functionality of modern GPUs?
Proof-of-Concept GPU-based Malware

• Design and implementation of code armoring techniques based on GPU code
  – Self-unpacking
  – Run-time polymorphism

• Design and implementation of stealthy host memory scanning techniques
  – Keylogger
Outline

• Background and motivation
• GPU-based Malware Signature Detection
  – Network intrusion detection/prevention
  – Virus scanning
• **GPU-assisted Malware**
  – Code-armoring techniques
  – Keylogger
• GPU as a Secure Crypto-Processor
• Conclusions
Self-unpacking GPU-malware

![Diagram showing the process of self-unpacking GPU-malware]

- **GPU**: Decryptor, Packed Malware
  - Decryption
  - GPU-accessible address space
  - GPU execution

- **CPU**: Init, Packed Malware
  - mmap
  - CPU-accessible address space
  - CPU execution

- Code
- Data/Decrypted Code
- Code Execution

Giorgos Vasiliadis
Self-unpacking: Strengths

- Current analysis and unpacking systems cannot handle GPU code
- Exposes minimal x86 code footprint
- GPU can use extremely complex encryption schemes
Self-unpacking: Weaknesses

• Malware code lies unencrypted in main memory after unpacking
• Can be detected by dumping the memory

• Can we do better?
Runtime-polymorphic GPU-malware

Giorgos Vasiliadis
Run-time polymorphism: Strengths

• Only the necessary code blocks are decrypted each time
• GPU can use different encryption keys occasionally
  – Random-generated
• Newly generated encryption keys are stored in device memory
  – Not accessible from CPU
Outline

• Background and motivation
• GPU-based Malware Signature Detection
  – Network intrusion detection/prevention
  – Virus scanning
• GPU-assisted Malware
  – Code-armoring techniques
  – Keylogger
• GPU as a Secure Crypto-Processor
• Conclusions
Overall approach

• Scan kernel’s memory to locate the keyboard buffer
• Remap the memory page of the buffer to user space
• Set the GPU to periodically read and scan them for sensitive information (e.g., credit card numbers)
• Unmap the memory in order to leave no traces
• GPU periodically collects newly-typed keystrokes
How the GPU access host memory

User-space

Kernel-space

Physical Mem.

Giorgos Vasiliadis
How the GPU access host memory

User-space
- CUDA Runtime Library
- User Virtual Address

Kernel-space
- Kernel Virtual Address

GPU
- GPU Buffer
- Keybd Buffer

Physical Mem.
How the GPU access host memory

- **User-space**
  - CUDA Runtime Library
  - User Virtual Address

- **Kernel-space**
  - Kernel Virtual Address

- **DMA**
  - GPU Buffer
  - Keybd Buffer

- **Physical Mem.**

Giorgos Vasiliadis
**Opportunity:** Remap process’ virtual memory to sensitive physical pages

![Diagram](image_url)
Opportunity: Remap process’ virtual memory to sensitive physical pages

- CUDA Runtime Library
- User Virtual Address
- Kernel Virtual Address
- GPU Buffer
- Keybd Buffer

User-space
Kernel-space

DMA

GPU
Physical Mem.
Implementation

• Use polling to catch keystrokes
  – “wake up” GPU process periodically through the CPU controller process

• Simple state machine translates keystrokes into ASCII characters

• Store keystrokes into Video RAM
CPU Utilization

![Graph showing the relationship between CPU utilization and kernel invocation interval. The x-axis represents the kernel invocation interval in milliseconds (msecs), ranging from 0.001 to 1000. The y-axis represents CPU utilization in percent, ranging from 0 to 100. The graph shows a downward trend as the kernel invocation interval increases.]
CPU Utilization

Kernel invocation interval (msecs)

CPU utilization (percent)

Fastest Typists
GPU Utilization

Giorgos Vasiliadis
Current Prototype Limitations

• Requires a CPU process to control its execution
  – Future GPGPU SDKs might allow us to drop the CPU controller process

• Requires administrative privileges
  – For installing and using the module
  – However the control process runs in user-space
    • No OS modification needed or data structure manipulation, in order to hide
Summary

• GPUs offer new ways for robust and stealthy malware
  – We demonstrated how a malware can increase its robustness against detection using the GPU
    • Unpacking / Runtime polymorphism
  – Presented a fully functional and stealthy GPU-based keylogger
    • Low CPU and GPU usage
    • No device hooking

• Graphics cards may be a promising new environment for future malware
Outline

• Background and motivation
• GPU-based Malware Signature Detection
  – Network intrusion detection/prevention
  – Virus scanning
• GPU-assisted Malware
  – Code-armoring techniques
  – Keylogger
• **GPU as a Secure Crypto-Processor**
• Conclusions
Motivation

• Modern cryptography is based on keys

• **Problem:** Secret keys **may remain unencrypted** in CPU Registers, RAM, etc.
  – Memory disclosure attacks
    • Heartbleed
  – DMA/Firewire attacks
  – Physical attacks
    • Cold-boot attacks
  – ...
PixelVault Overview

- Runs encryption securely outside CPU/RAM
- Only on-chip memory of GPU is used as storage
- Secret keys are never observed from host
Cryptographic Processing with GPUs

- GPU-accelerated SSL
  - [CryptoGraphics, CT-RSA’05]
  - [Harrison et al., Sec’08]
  - [SSLShader, NSDI’11]
  - ...

- High-performance
- Cost-effective
Cryptographic Processing with GPUs

- GPU-accelerated SSL
  - [CryptoGraphics, CT-RSA’05]
  - [Harrison et al., Sec’08]
  - [SSLShader, NSDI’11]
  - …

- High-performance
- Cost-effective

Can we also make it secure?
Implementation Challenges

• How to isolate GPU execution?

• Who holds the keys?

• Where is the code?
Implementation Challenges

• How to isolate GPU execution?

• Who holds the keys?

• Where is the code?
Autonomous GPU execution

• Force GPU program to run indefinitely
  – i.e., using an infinite \texttt{while} loop

• GPUs are non-preemptive
  – No other program can run at the same time

• We use a \texttt{shared memory segment} for communication between the CPU and the GPU
Shared Memory between CPU/GPU

- **Page-locked** memory
  - Accessed by the GPU directly, via DMA
  - Cannot be swapped to disk

- Processing requests are issued through this shared memory space
Shared Memory between CPU/GPU

- GPU continuously monitors the shared space for new requests
Shared Memory between CPU/GPU

- When a new request is available, it is transferred to the memory space of the GPU
Shared Memory between CPU/GPU

- The request is processed by the GPU
Shared Memory between CPU/GPU

- When processing is finished, the host is notified by setting the response parameter fields accordingly.
Autonomous GPU execution

- Non-preemptive execution
- Only the output block is being written back to host memory
Implementation Challenges

- How to isolate GPU execution?
- Who holds the keys?
- Where is the code?
Who holds the keys?

- GPUs contain different memory hierarchies of ...
  - different sizes, and ...
  - different characteristics
Who holds the keys?

• GPUs contain different memory hierarchies of ...
  – different sizes, and ...
  – different characteristics
Support for an arbitrary number of keys

- We can use a separate KeyStore array that holds an arbitrary number of secret keys.

Encrypted keys are stored in GPU global device memory:

- Copy to registers.

Each key is decrypted in registers during encryption/decryption:

KeyStore

GPU Registers File

Master Key

Enc’ed Key

Dec’ed Key

Giorgos Vasiliadis
Implementation Challenges

• How to isolate GPU execution?

• Who holds the keys?

• Where is the code?

```
mov.u32 %r2, 0;
setp.le.s32 %p1, %r1, %r2;
mov.s32 %r5, %r4;
add.u32 %r6, %r1, %r4;
@%p1 bra $Lt_0_1282;
mov.s32 %r8, %r3;
xor.b32 %r10, %r7, %r9;
st.global.u8 [%r5+0], %r10;
add.u32 %r5, %r5, 1;
setp.ne.s32 %p2, %r5, %r
```
Where is the code?

• GPU code is initially stored in global device memory for the GPU to execute it
  – An adversary could replace it with a malicious version
Prevent GPU code modification attacks

- Three levels of instruction caching (icache)
  - 4KB, 8KB, and 32KB, respectively
  - Hardware-managed

- **Opportunity**: Load the code to the icache, and then erase it from global device memory
  - The code runs indefinitely from the icache
  - Not possible to be flushed or modified
PixelVault Crypto Suite

• Currently implemented algorithms
  – AES-128
  – RSA-1024

• Implemented completely using on-chip memory (i.e. registers, scratchpad memory)
  – The only data that is written back to global, off-chip device memory is the output block
AES-128 CBC Performance

- **Encryption**: 
  - Up to 13% overhead on GPU execution

- **Decryption**: 
  - Up to 20% overhead on GPU execution

---

Giorgos Vasiliadis

---

89
AES-128 CBC Performance

![Graph showing throughput for encryption and decryption with different numbers of messages for GPU, PixelVault, PixelVault (w/ KeyStore), and CPU.]

- **GPU**: 3x-4x faster than CPU for a sufficient number of messages.
- **Intel Nehalem single core (2.27GHz)**

**Encryption**

**Decryption**

Giorgos Vasiliadis
### 6.1.6 Register Spilling In Global Device Memory

The registers that will be used by a GPU kernel are declared by the programmer. If the number of declared registers exceeds the amount of registers contained in GPUs, the extra registers are mapped in global device memory.

It would also be possible that registers could be spilled in global device memory when a context switch between different warps occurs, for performance reasons. According to NVIDIA, no state is saved when context switching.

### 6.2 Performance Analysis

The GPU code base of PixelVault is small, which can result in a high performance footprint. The GPU registers are volatile, and thus the contents of GPU registers are never saved (in contrast to CPU registers, which can be saved for later restoration). Still, thread warps can be switched, e.g., when a warp is waiting for memory I/O another warp can be scheduled for running.

### 6.1.5 Simultaneous GPU Kernel Execution

As we described in Section 6.1.4, the instruction cache cannot be flushed without loading a new GPU code. Therefore, an attacker cannot overwrite PixelVault, because the instruction cache 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.

Sensitive data, such as private keys, that are stored in the address space of both CPU and GPU variables, as well as the execution of arbitrary GPU code by transferring it via PCIe to the appropriate memory region where native code is stored. Similar attacks can be performed by forcing the registers' contents to be written to the global device memory, region. The malicious code could contain commands for forcing the registers to execute an arbitrary GPU instruction. The GPU code base of PixelVault is small, which can provide an attacker with a significant performance advantage over the vanilla GPU-based RSA implementation.

### RSA 1024-bit Decryption

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

- PixelVault adds an 1%-15% overhead over the default GPU-accelerated RSA.

---

Giorgos Vasiadis

91
RSA 1024-bit Decryption

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

- Still faster than CPU when batch processing >128 messages
PixelVault Features

• Prevents key leakages
  – Even when the base system is fully compromised

• Requires just a commodity GPU
  – No OS kernel modifications or recompilation

• High-performance cryptographic operations
Limitations

• Require trusted bootstrap

• Dedicated GPU execution

• Misusing PixelVault for encrypting/decrypting messages

• Denial-of-Service attacks

• Side-channel attacks
Summary

• Cryptography on the GPU is not only fast …
• … *but* also **secure**!
  – Preserves the secrecy of keys even when the base system is fully compromised

• More technical details
  – See our ACM CCS’2014 paper
Outline

• Background and motivation
• GPU-based Malware Signature Detection
  – Network intrusion detection/prevention
  – Virus scanning
• GPU-assisted Malware
  – Code-armoring techniques
  – Keylogger
• GPU as a Secure Crypto-Processor
• Conclusions
Conclusions

• GPUs have diverse security applications
  – Both for defense and offense
  – NDIS, AV, crypto-devices, secure processors, etc.
  – Generic library with functionality for various applications
  – Combine high-performance with programmability

• Future work
  – Adapt to other application domains
  – Apply to mobile and embedded devices
  – Utilize integrated CPU-GPU designs

• Credits to:
  – Sotiris Ioannidis, Lazaros Koromilas, Michalis Polychronakis, Spyros Antonatos, Evangelos Ladakis, Elias Athanasopoulos, Evangelos Markatos
GPUs for Security

Giorgos Vasiliadis
Foundation for Research and Technology – Hellas (FORTH)
thank you!