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

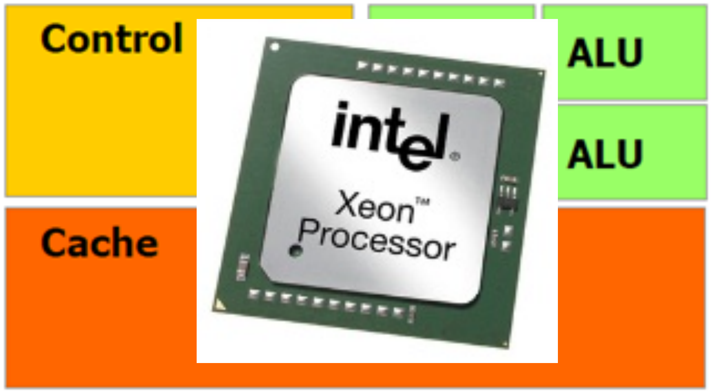
Outline

- **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

```
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);
```

Single Instruction, Multiple Threads

- Example: Vector addition

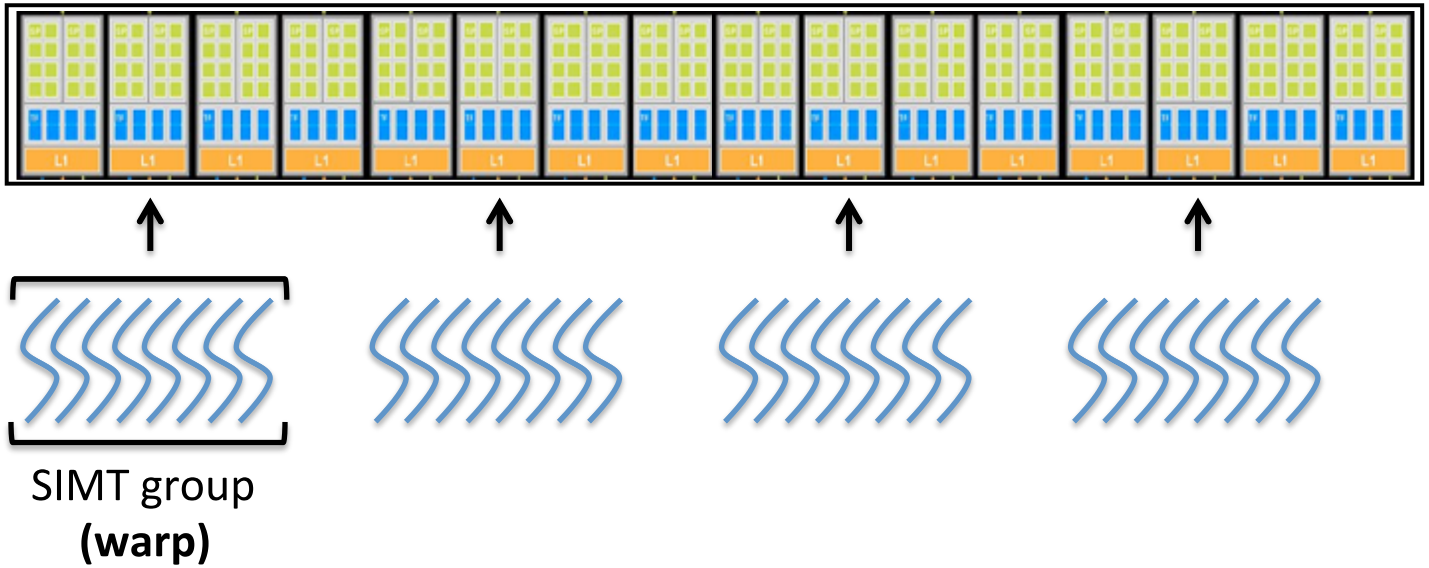
CPU code

```
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

```
__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 **I**nstruction, **M**ultiple **T**hreads



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

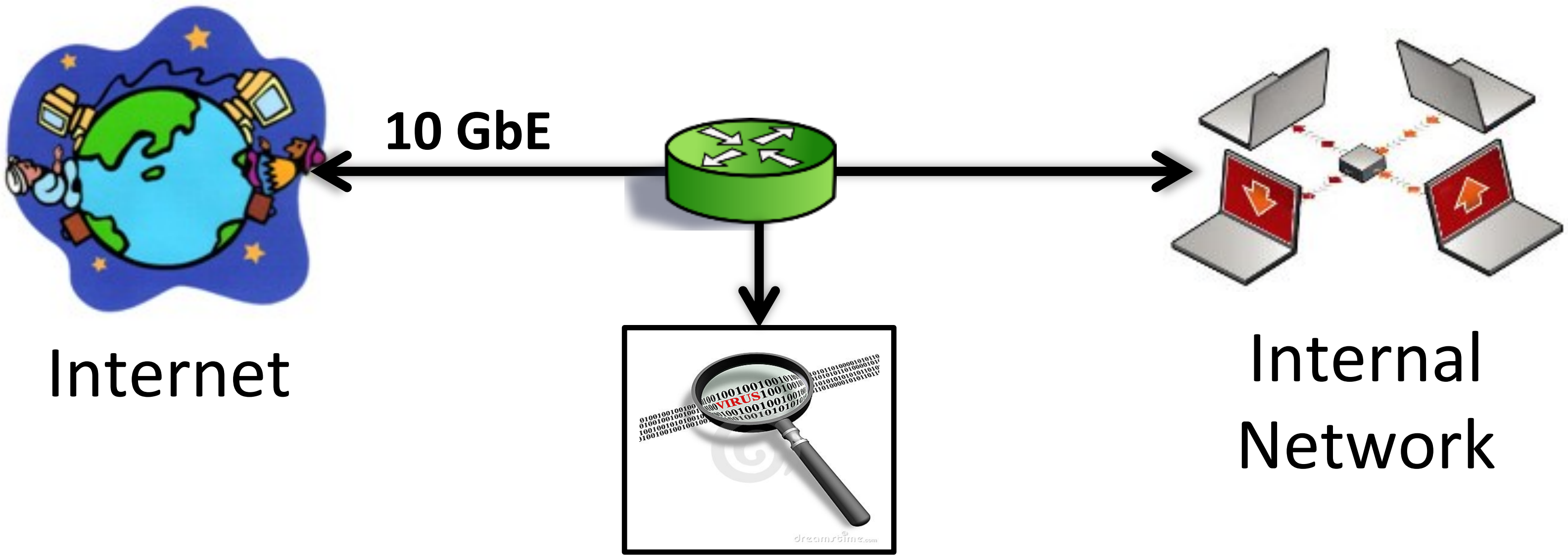
- *Great for regular computations!*

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

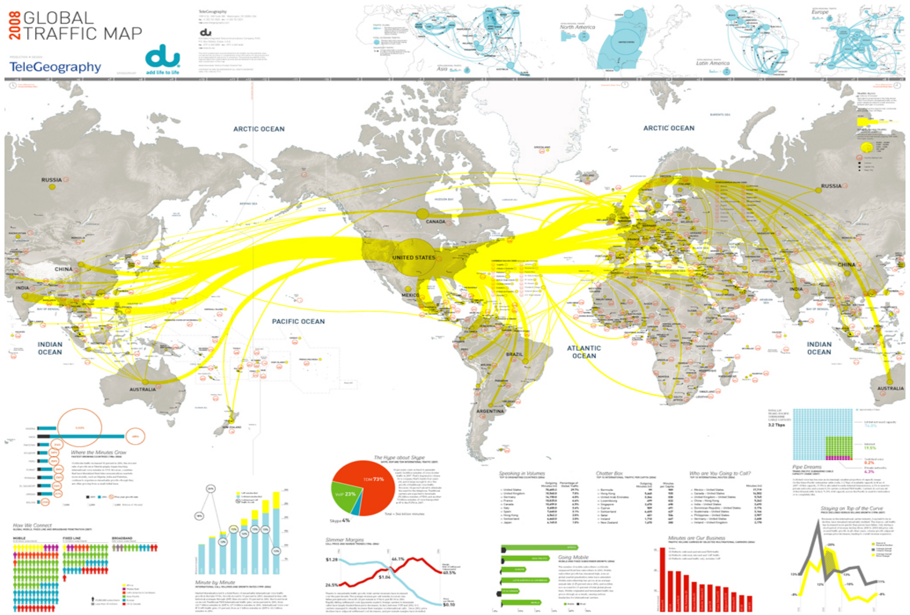
Network Intrusion Detection Systems

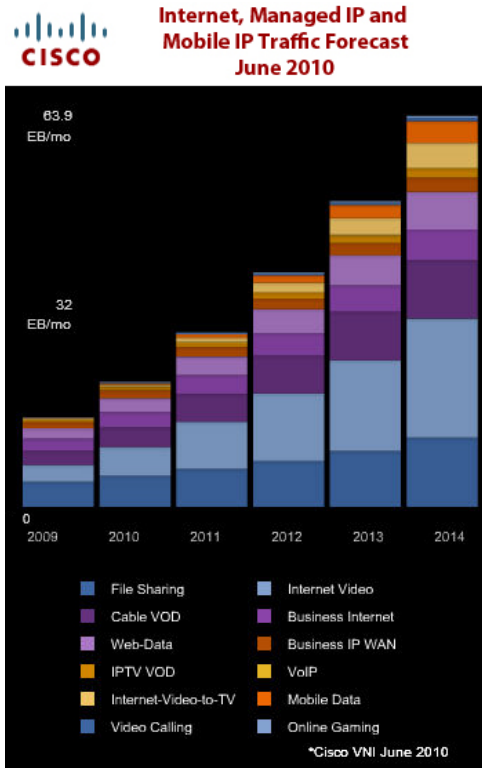
- Typically deployed at ingress/egress points
  - Inspect *all* network traffic
  - Look for suspicious activities
  - Alert on malicious actions



Challenges (1)

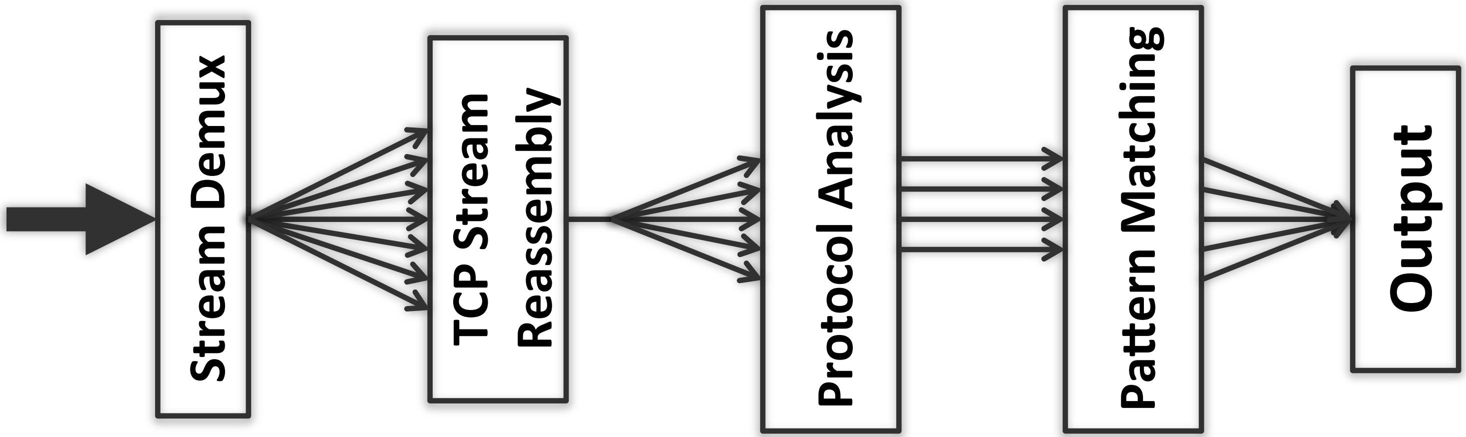
- ***Traffic rates*** are increasing
  - 10 Gbit/s Ethernet speeds are common in metro/ enterprise networks
  - Up to 40-100 Gbit/s at the core





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

IDS/IPS Sensors
(10s of Gbps)

~ US$ 20,000 - 60,000

IDS/IPS M8000
(10s of Gbps)

~ US$ 10,000 - 24,000

- Commodity "software" NIDS/NIPS and AVs
  - Processing by general-purpose processors
  - Throughput: Low

Open-source S/W

≤ ~1 Gbps

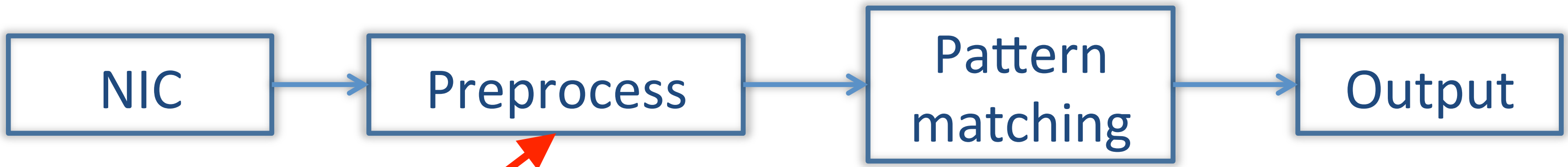
Single-threaded NIDS performance

NIC → Preprocess → Pattern matching → Output

```
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;
```

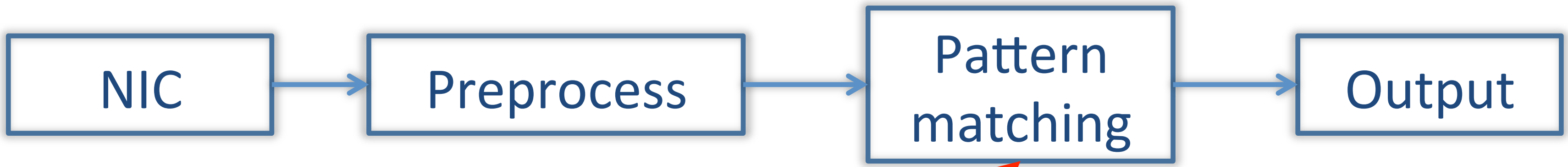
* PCRE: Perl Compatible Regular Expression

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]module=[a-zA-
Z0-9]*[^\x3b\x26]/U"); metadata:service http;
```

Single-threaded NIDS performance

NIC → Preprocess → Pattern matching → Output

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

NIC → Preprocess → Pattern matching → Output

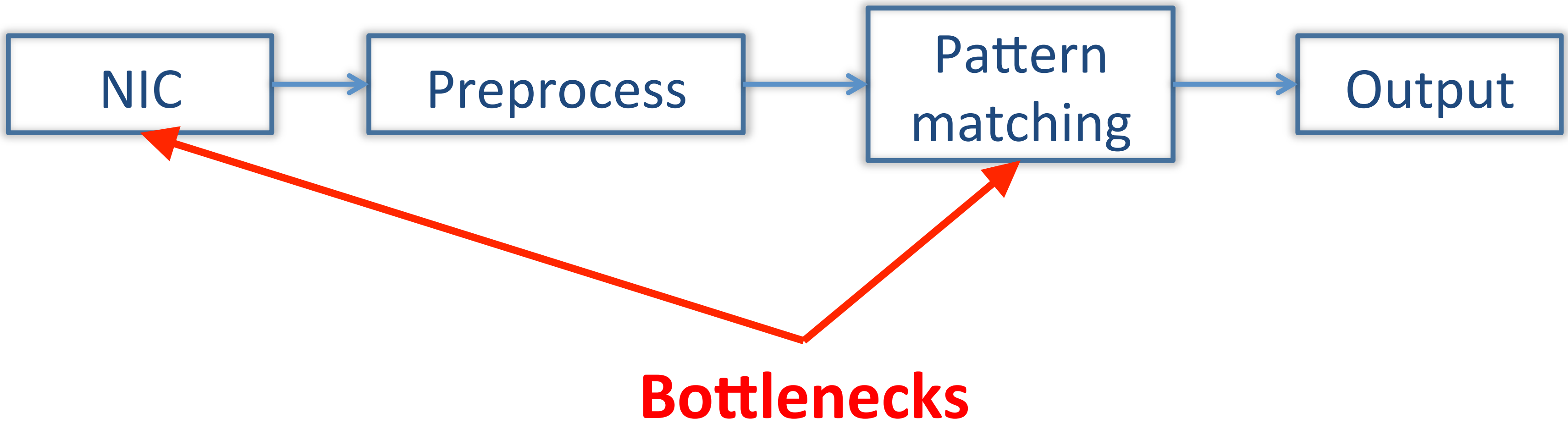
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
(msg:“WEB-PHP horde help module arbitrary command execution attempt”;
flow:established,to_server; uricontent:” /services/help/"; pcre:” /[\?\x20\x3b\x26]module=[a-zA-Z0-9]*[^\x3b\x26]/U"); metadata:service http;

Single-threaded NIDS performance

NIC → Preprocess → Pattern matching → Output

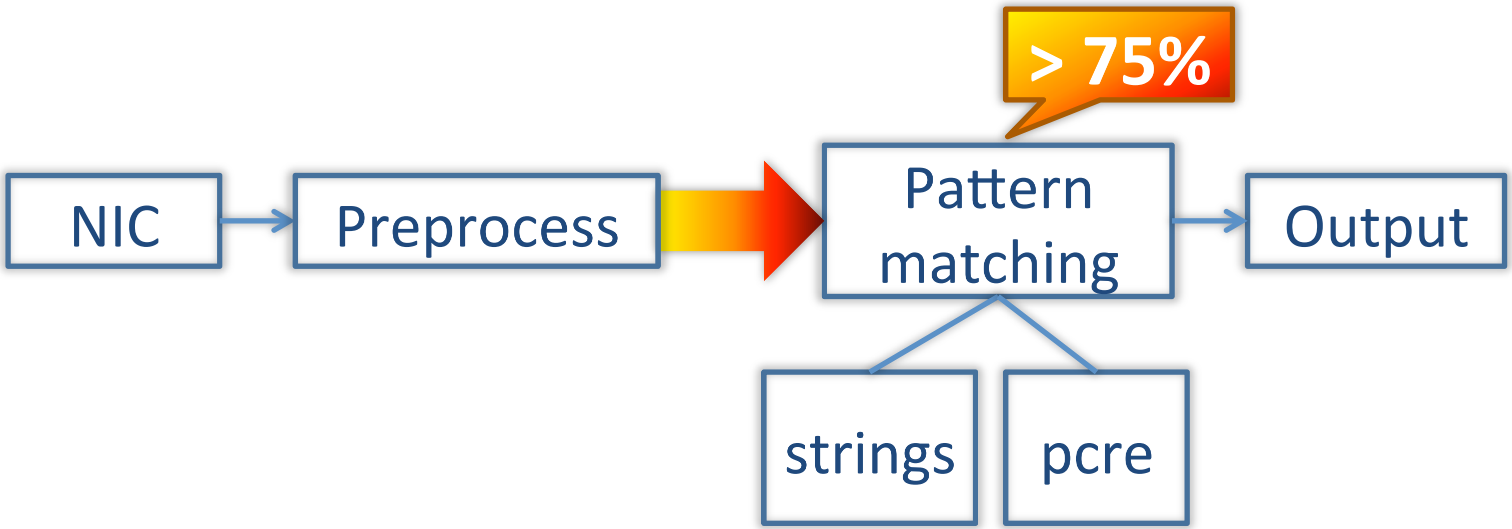
- **Vanilla Snort: 0.2 Gbit/s**

Single-threaded NIDS performance



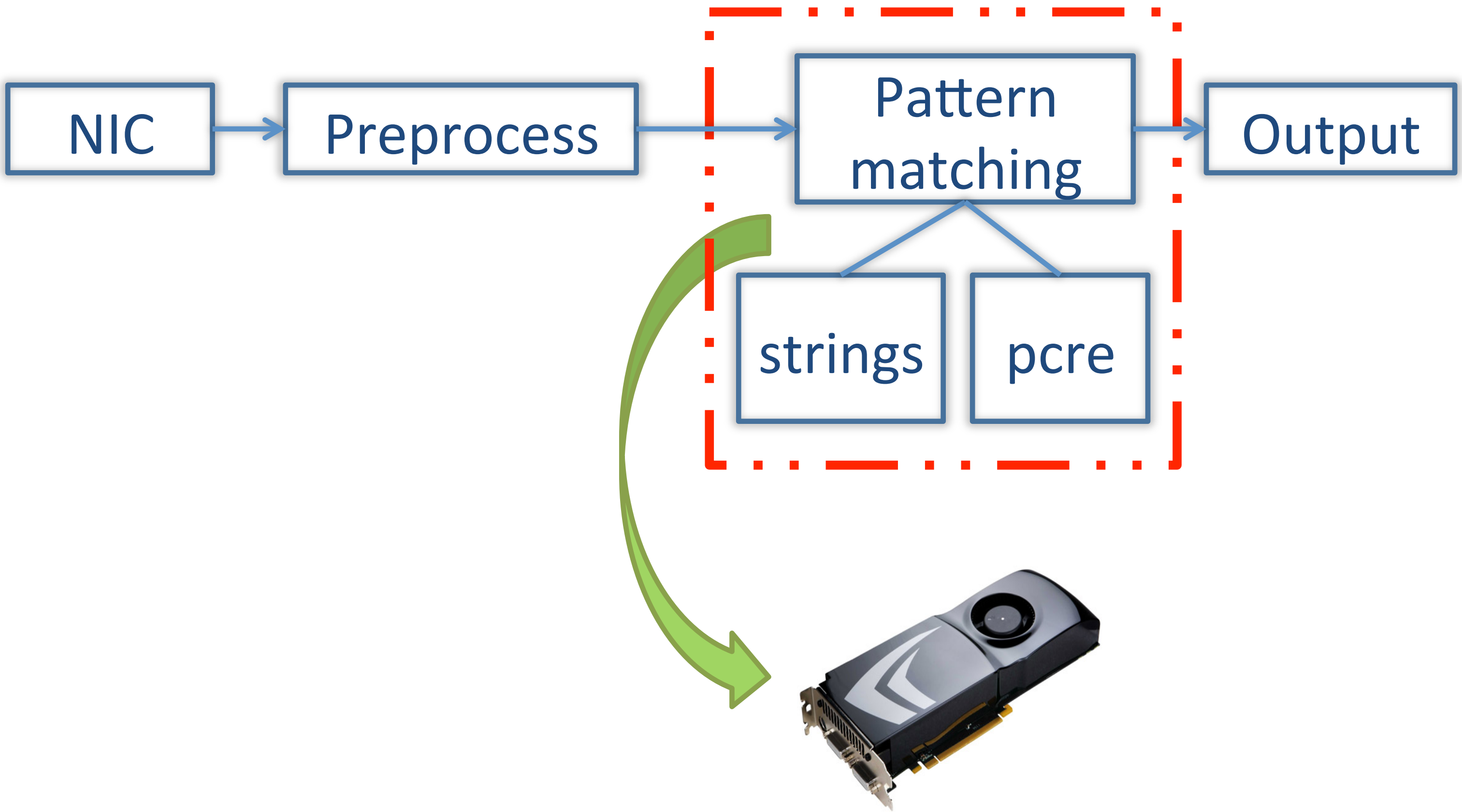
- **Vanilla Snort: 0.2 Gbit/s**

Problem #3: Pattern matching is the bottleneck

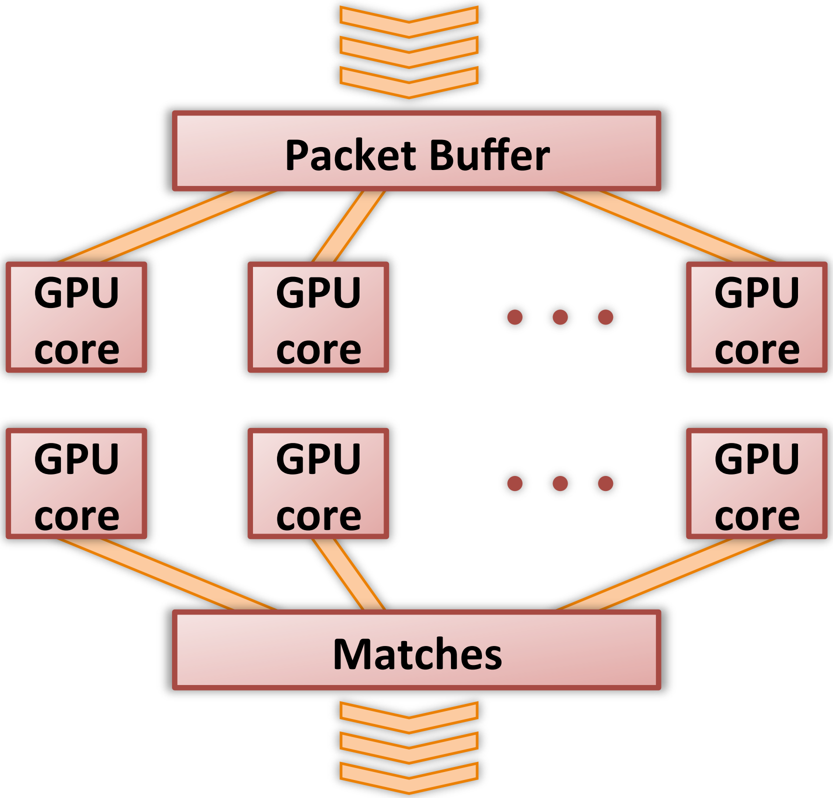


- On a Intel Xeon X5520, 2.27 GHz, 8 MB L3 Cache
  – String matching analyzing bandwidth per core: **1.1 Gbps**
  – PCRE analyzing bandwidth per core: **0.52 Gbps**

Offload pattern matching on the GPU



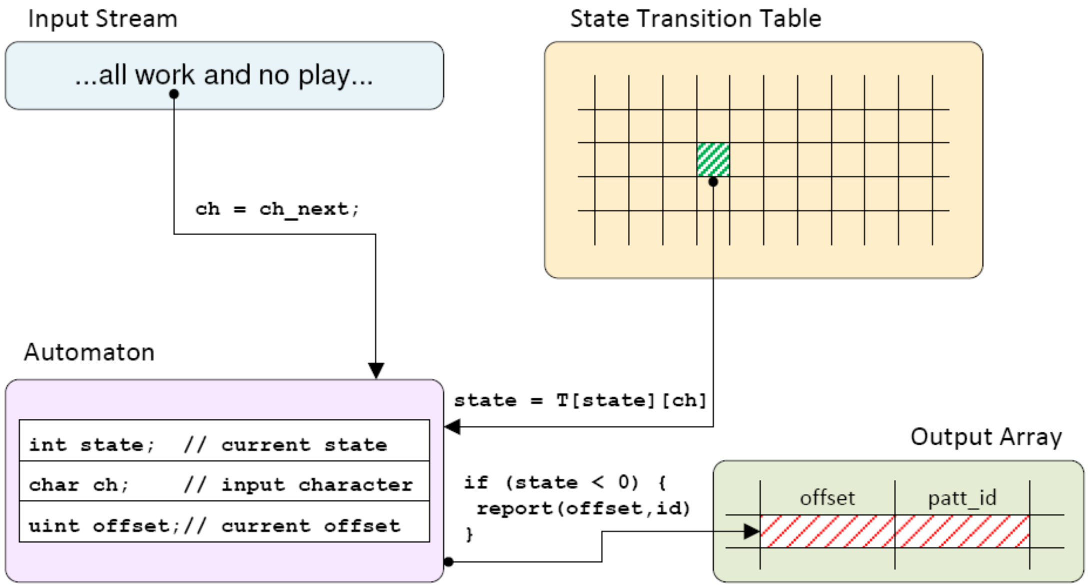
Pattern matching on the GPU



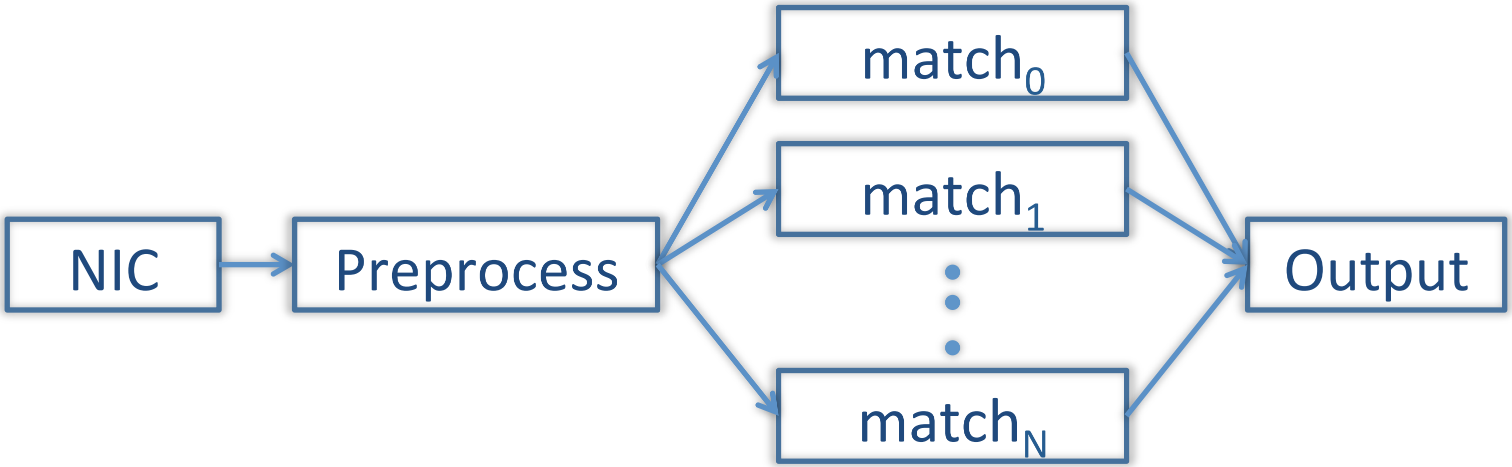
- Data level parallelism == Packet level parallelism
  - Uniformly one core for each reassembled packet stream

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



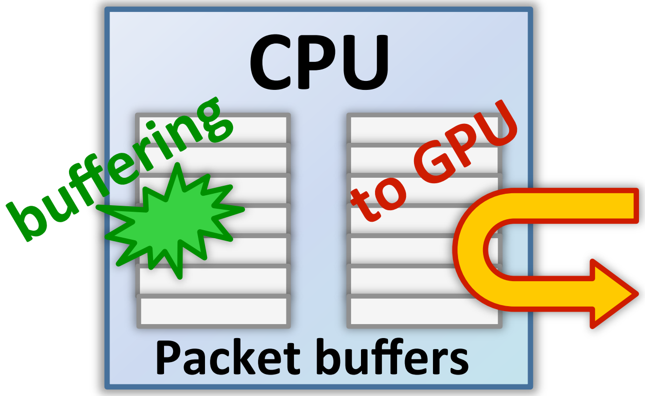
NVIDIA GTX 480 GPU

On an ~~Intel Xeon X5520, 2.27 GHz, 8 MB L3 Cache~~

String matching analyzing bandwidth: ~~1.1 Gbps~~ **30 Gbps**

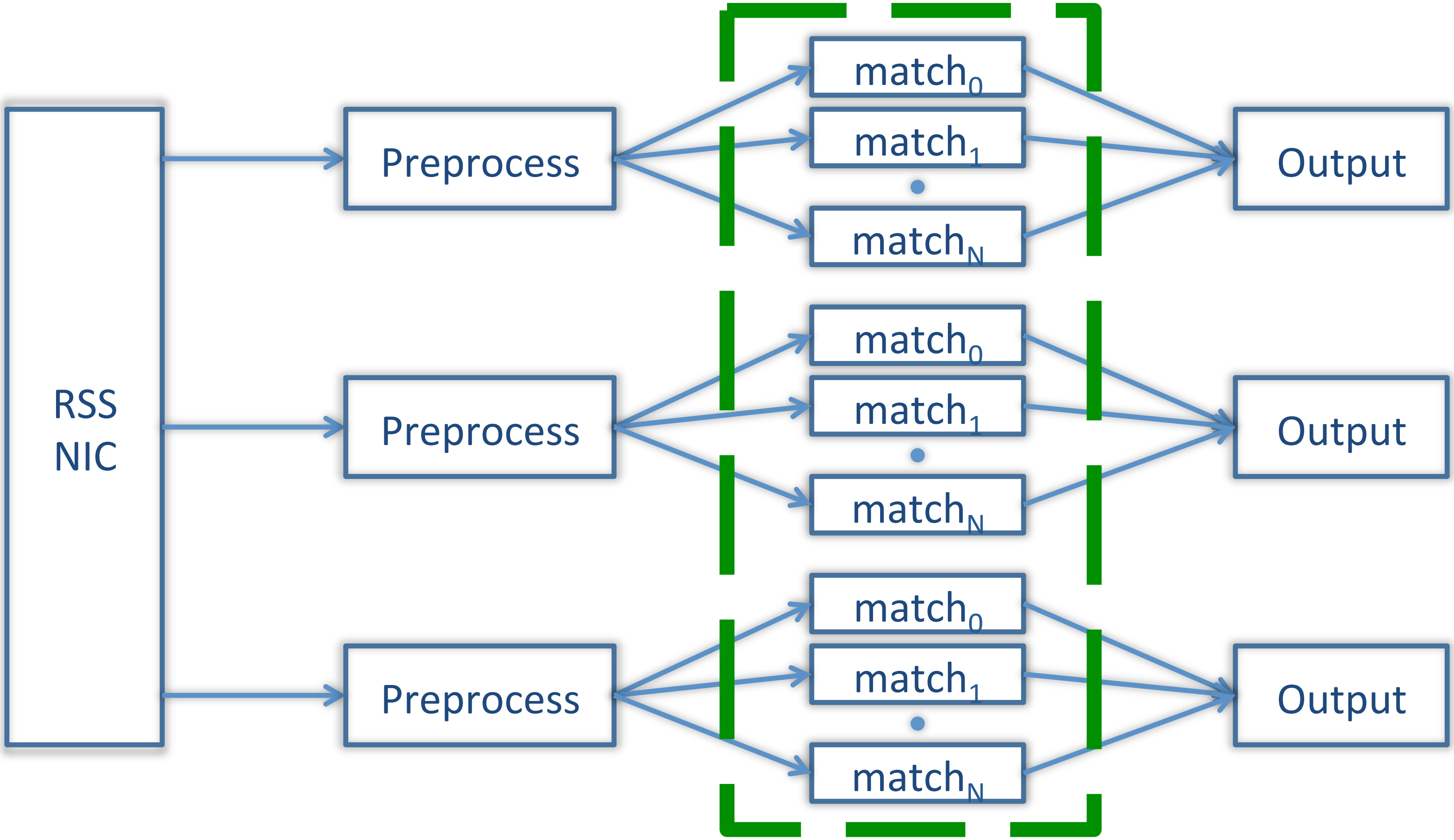
PCRE analyzing bandwidth: ~~0.52 Gbps~~ **8 Gbps**

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



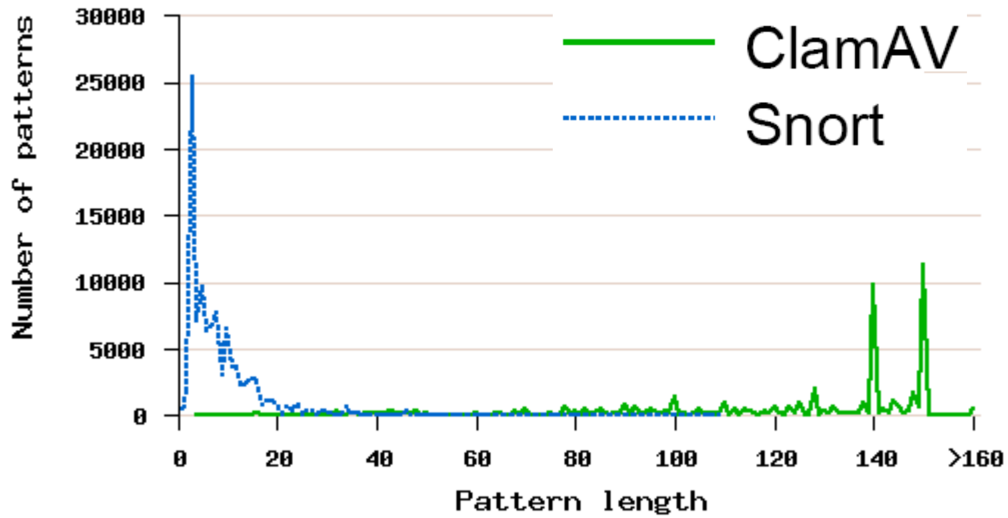
- Vanilla Snort: 0.2 Gbit/s
- With multiple CPU-cores: 0.9 Gbit/s
- **With GPU: 5.2 Gbit/s**

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

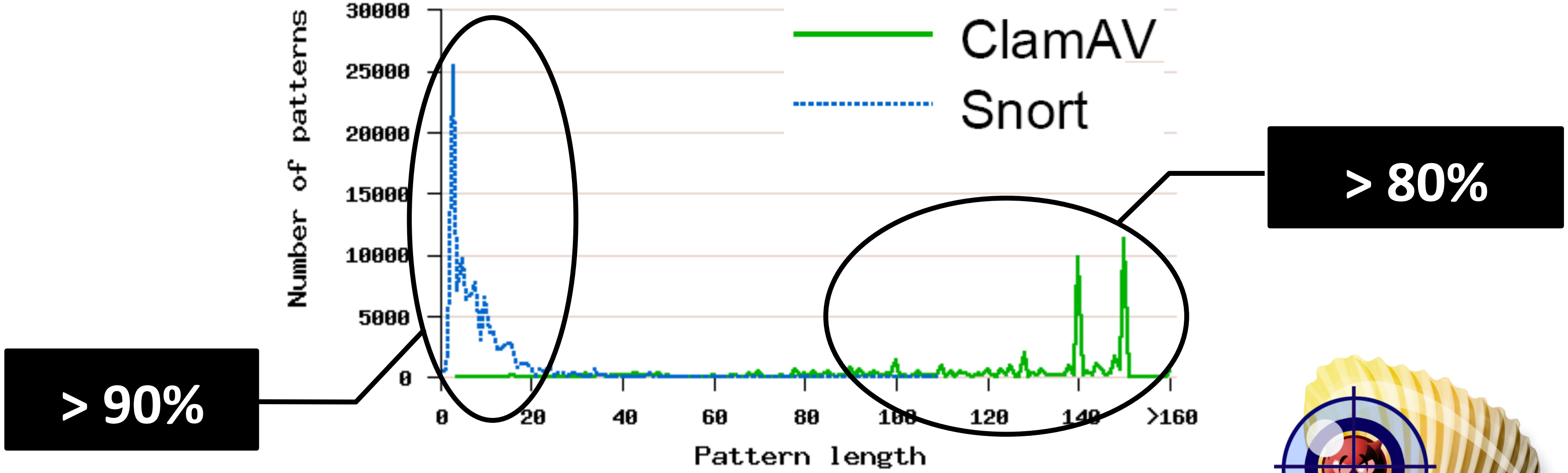
Anti-Virus Databases

- Contain thousands of signatures
  - ClamAV contains more than 60K signatures



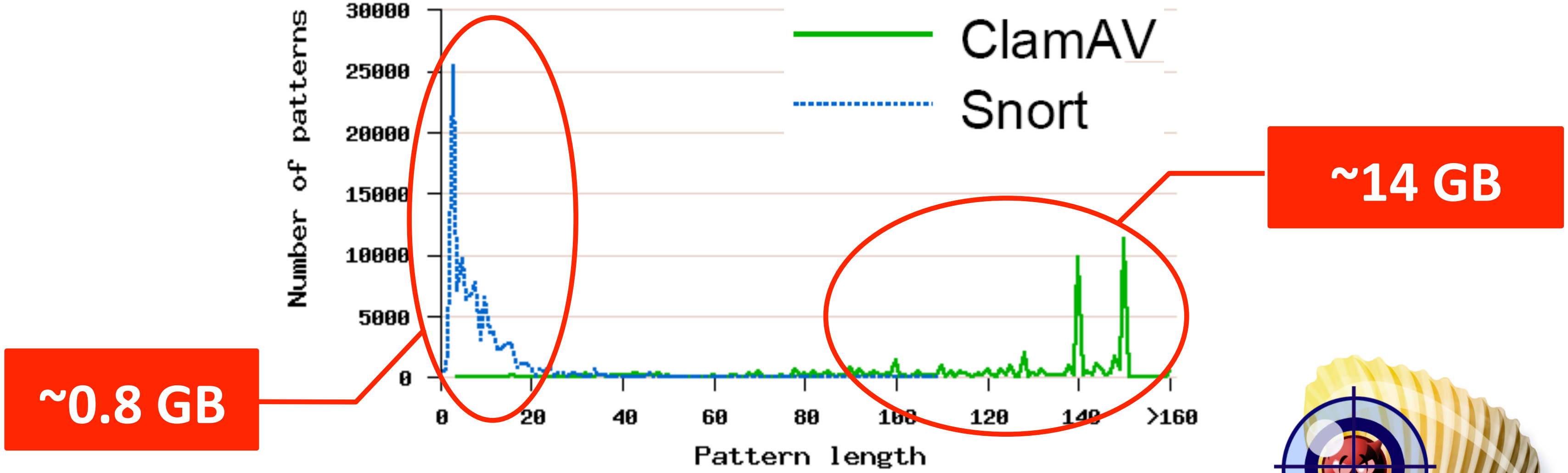
Anti-Virus Databases

- ClamAV signatures are significant longer than NIDS
  – length varying from 4 to 392 bytes



Anti-Virus Databases

- Memory requirements



Opportunity: Prefix Filtering

- Take the first *n* bytes from each signature
  - e.g.

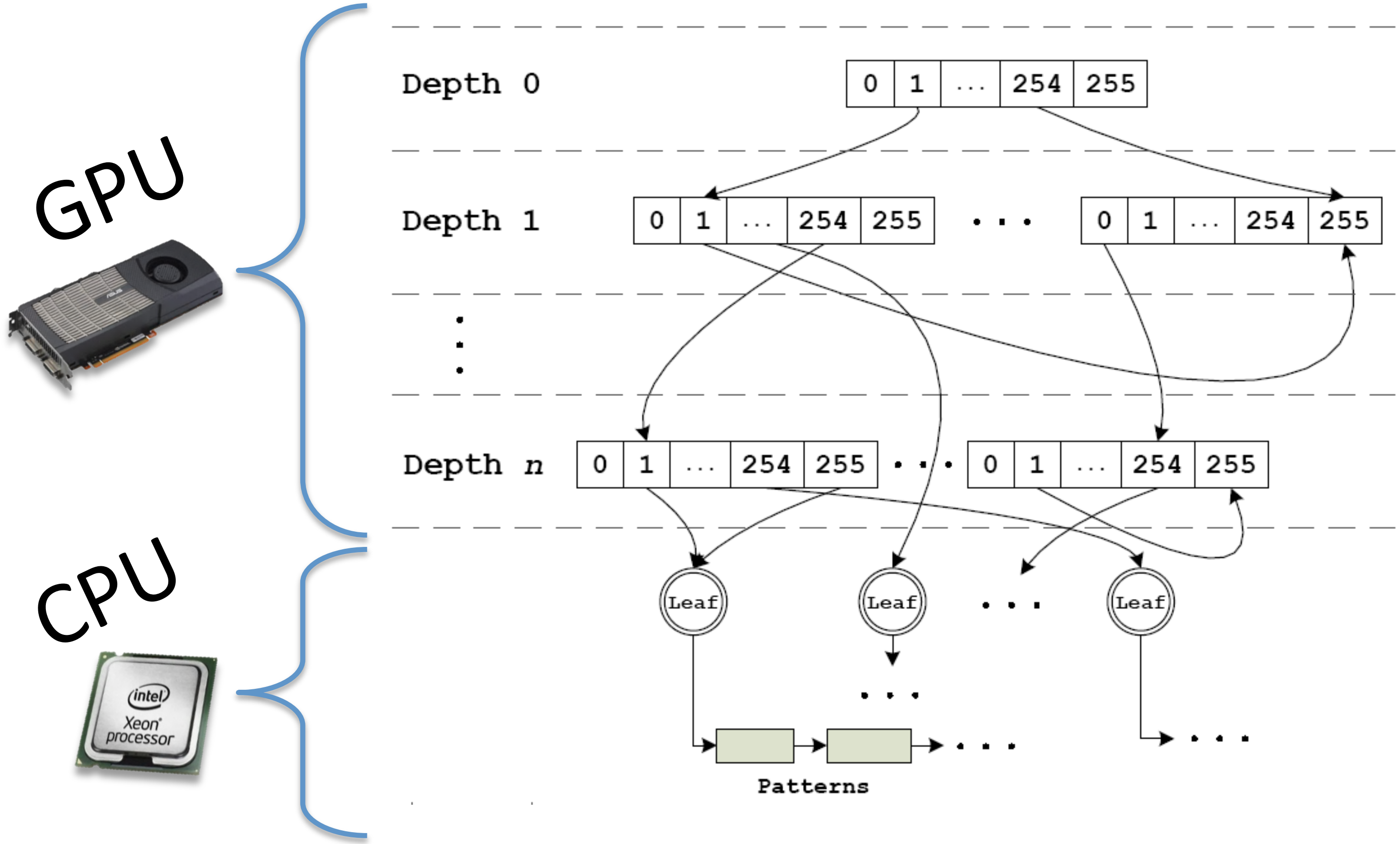
    Worm.SQL.Slammer.A:0:*:

    `4e65742d576f726d2e57696e33322e536c616d6d65725554`

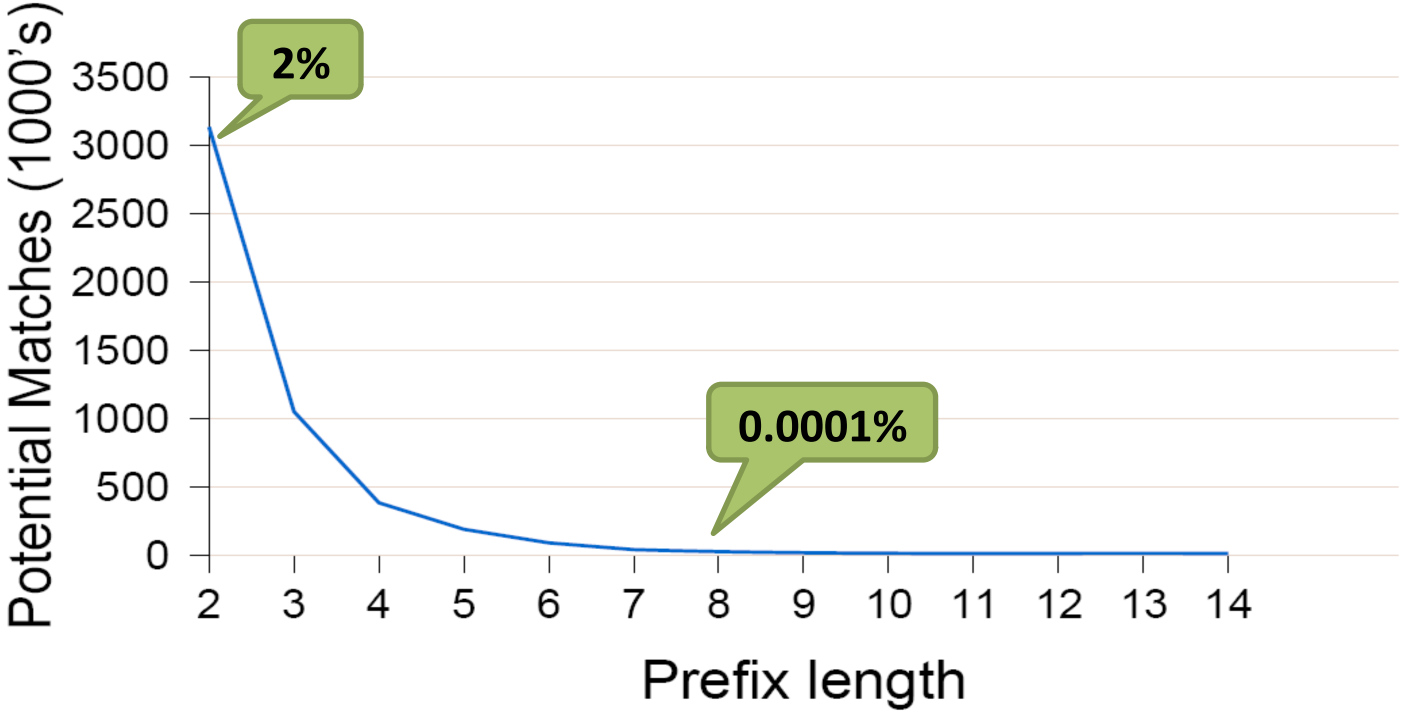
- Compile **all** *n*-bytes sub-signatures into **a single** *Scanning Trie*
- The Scanning Trie can quickly filter clean data segments in linear time.

Scanning Trie

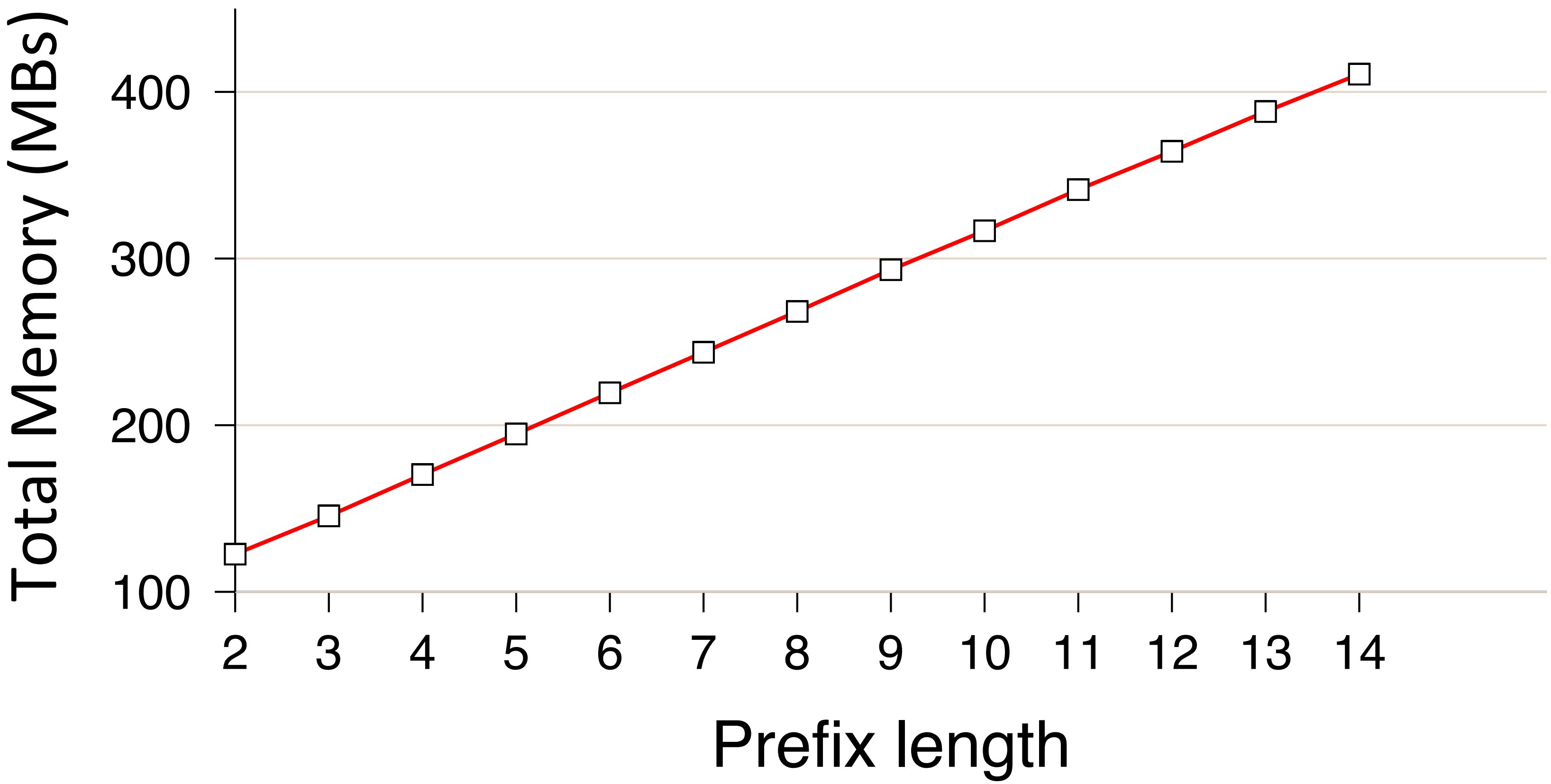
- Variable trie height



Longer prefix = Fewer matches

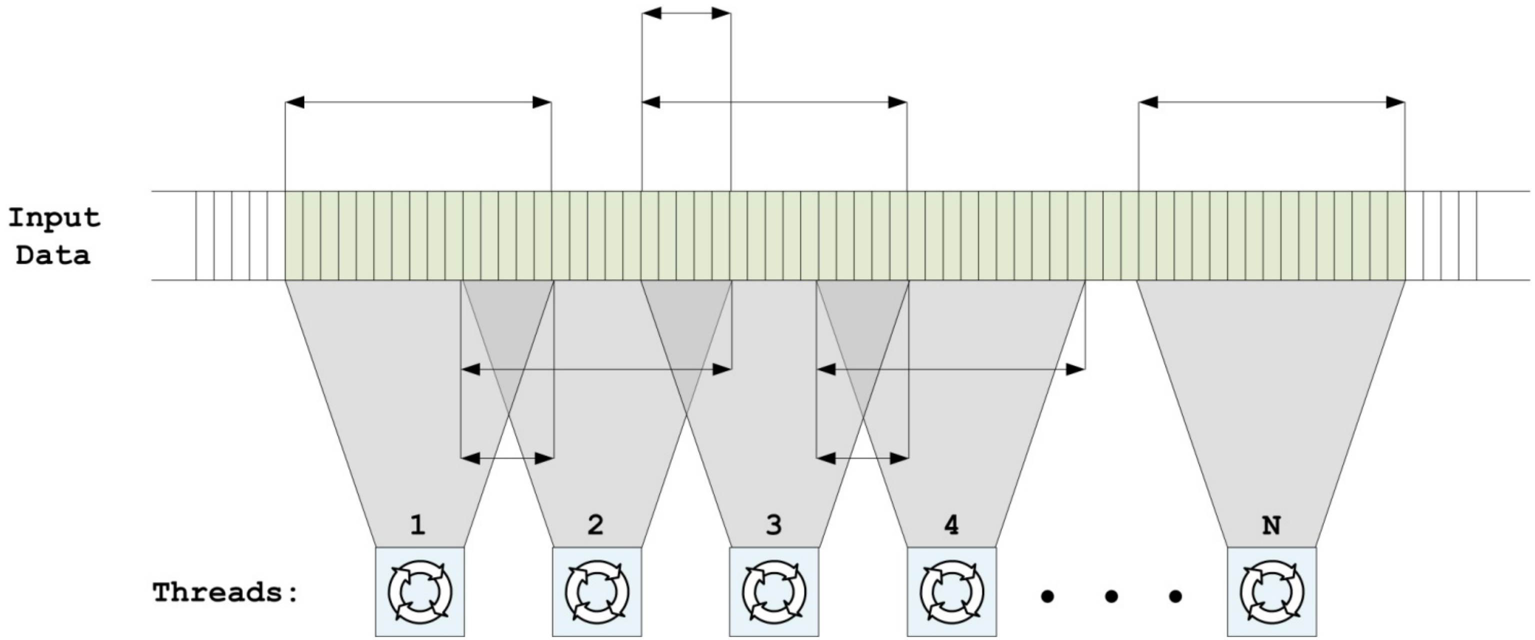


Longer prefix = More memory

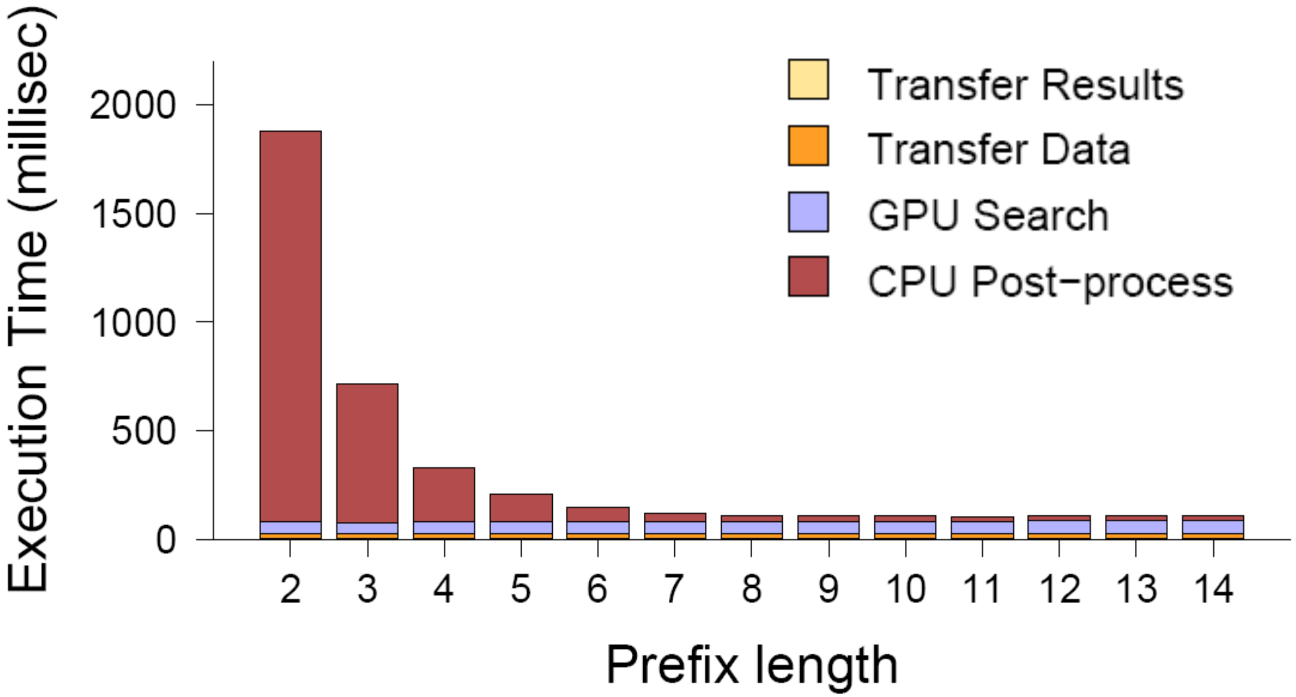


Virus Scanning on the GPU

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

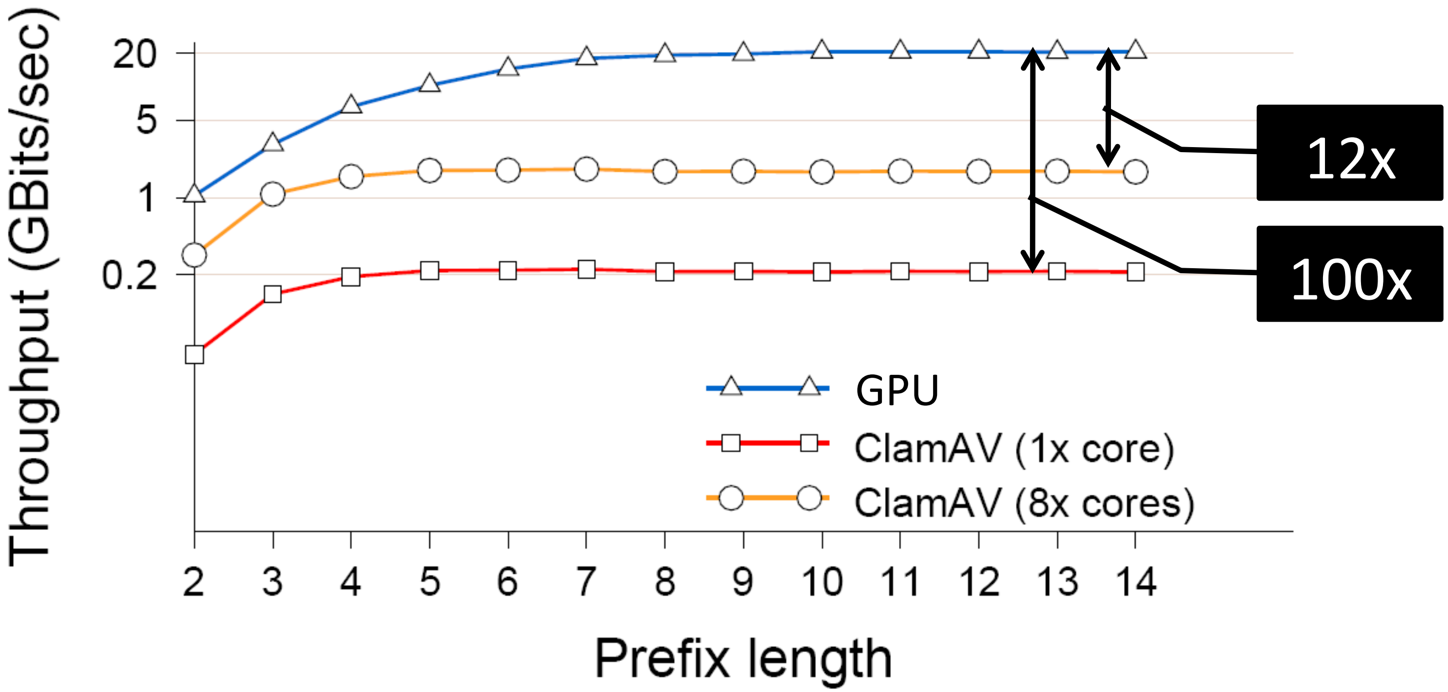


Execution Time Breakdown



- 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

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

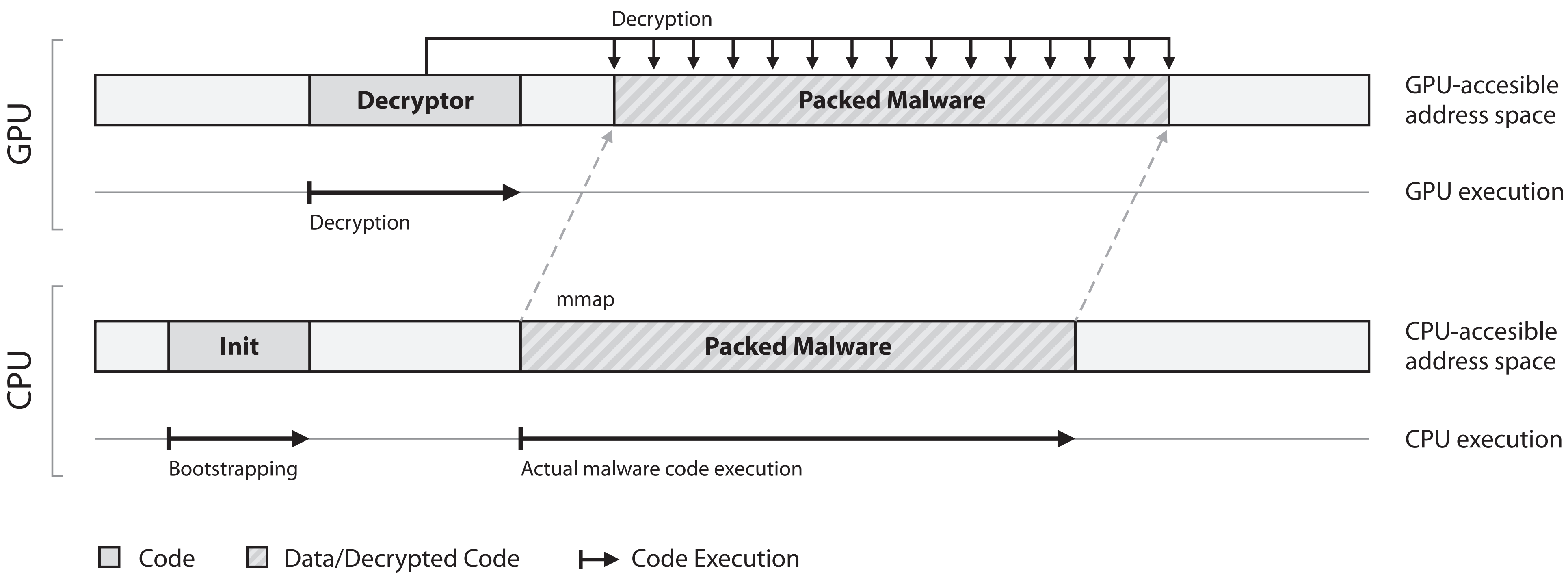
- Design and implementation of **code armoring** techniques based on GPU code
  - Self-unpacking
  - Run-time polymorphism

- 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



Self-unpacking: Strengths

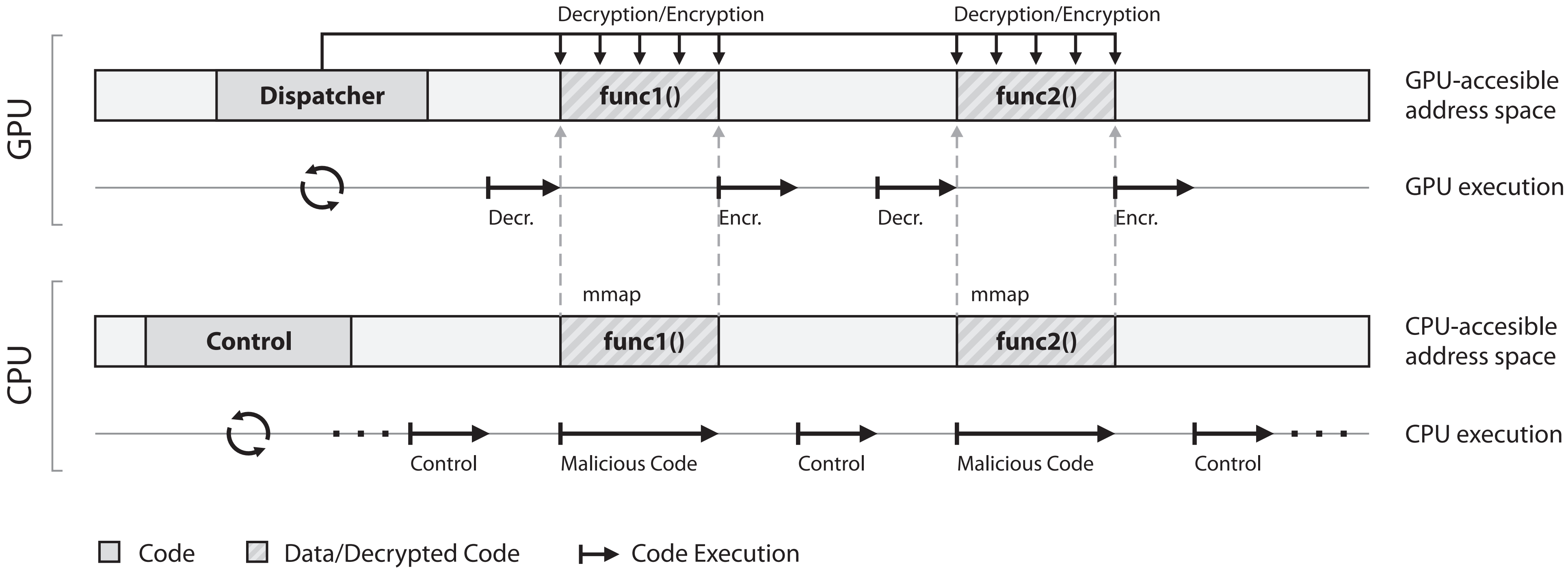
- Current analysis and unpacking systems cannot handle GPU code
- Exposes minimal x86 code footprint
- GPU can use extremely complex encryption schemes

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



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

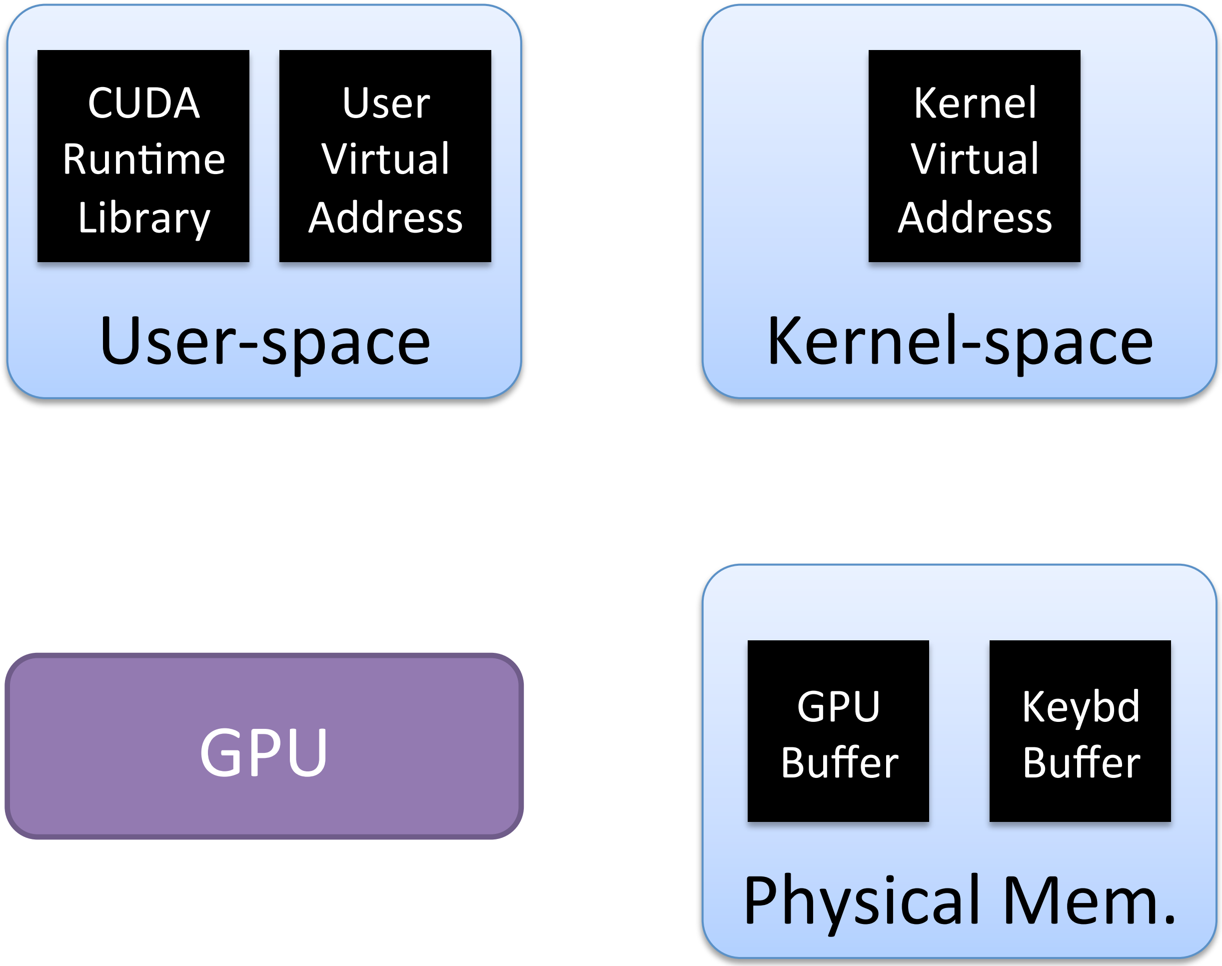
Outline

- 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

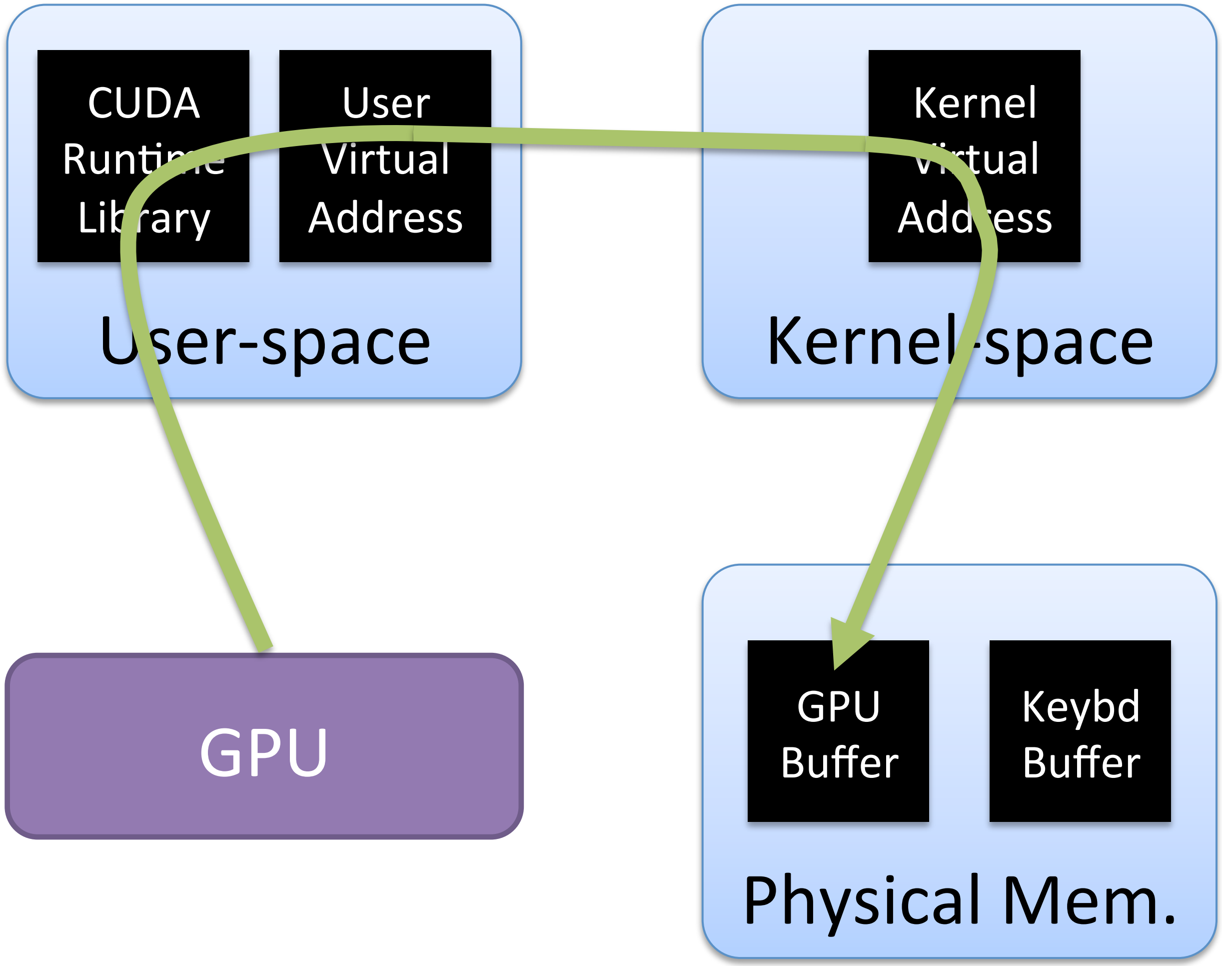
Overall approach

- 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

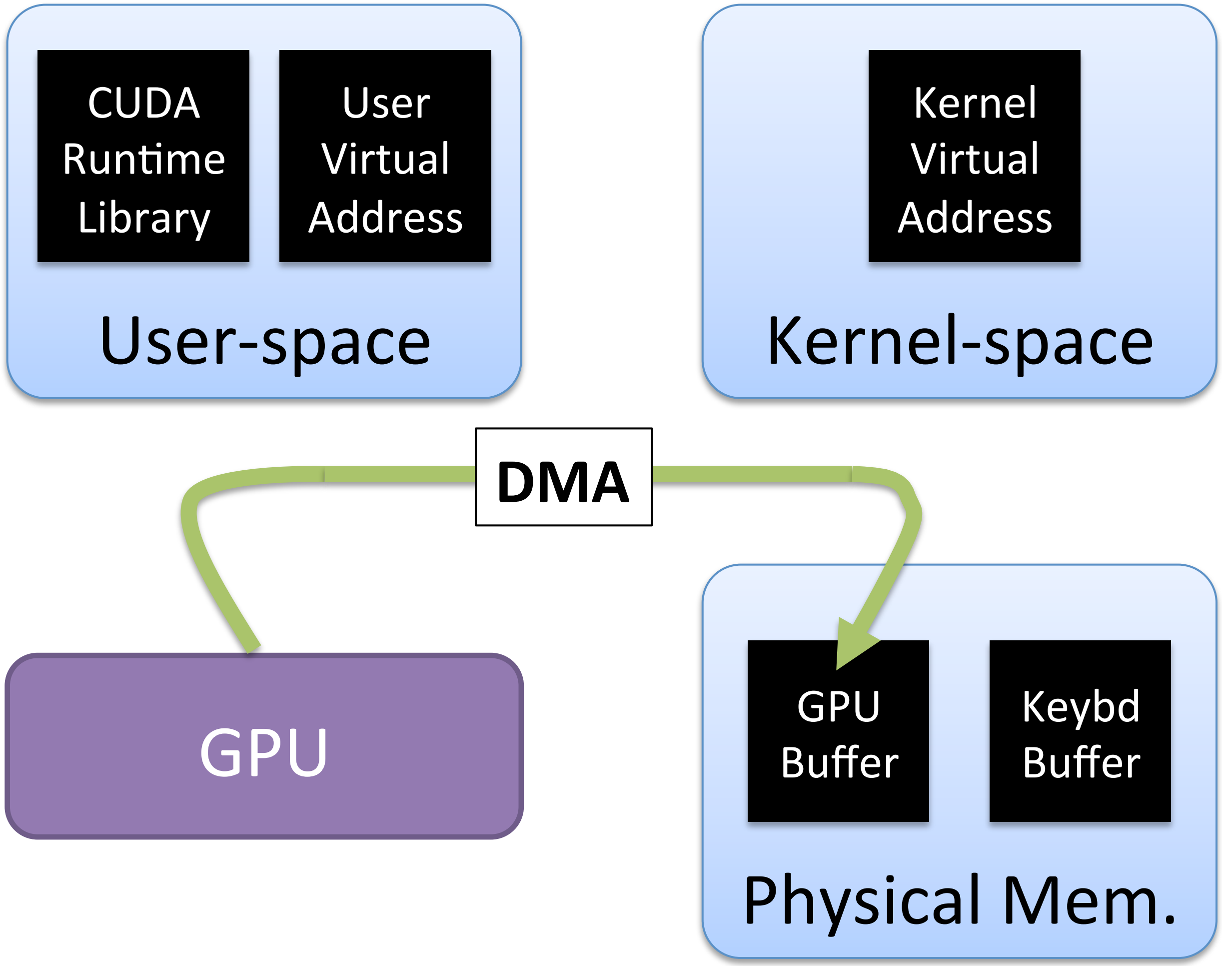
How the GPU access host memory



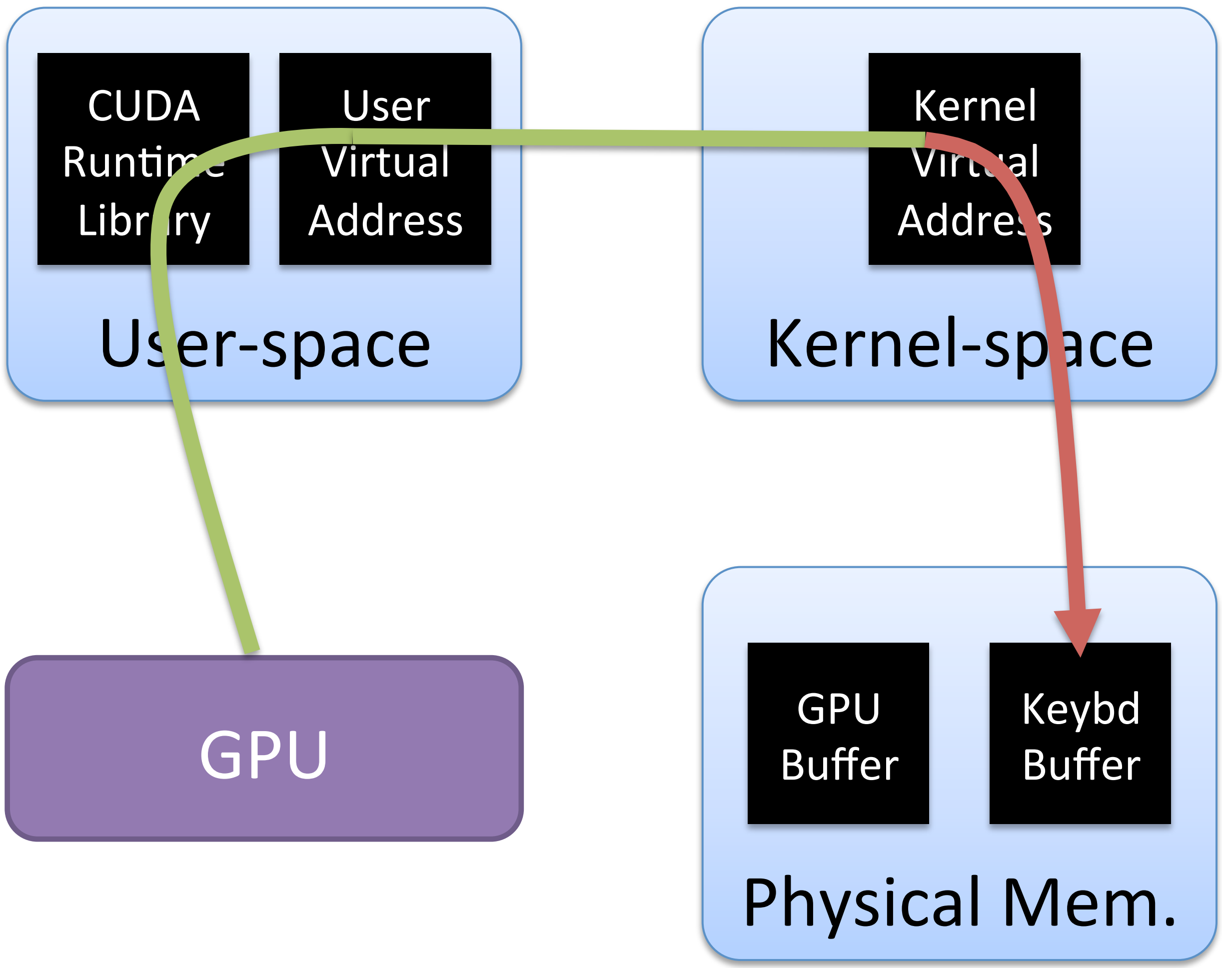
How the GPU access host memory



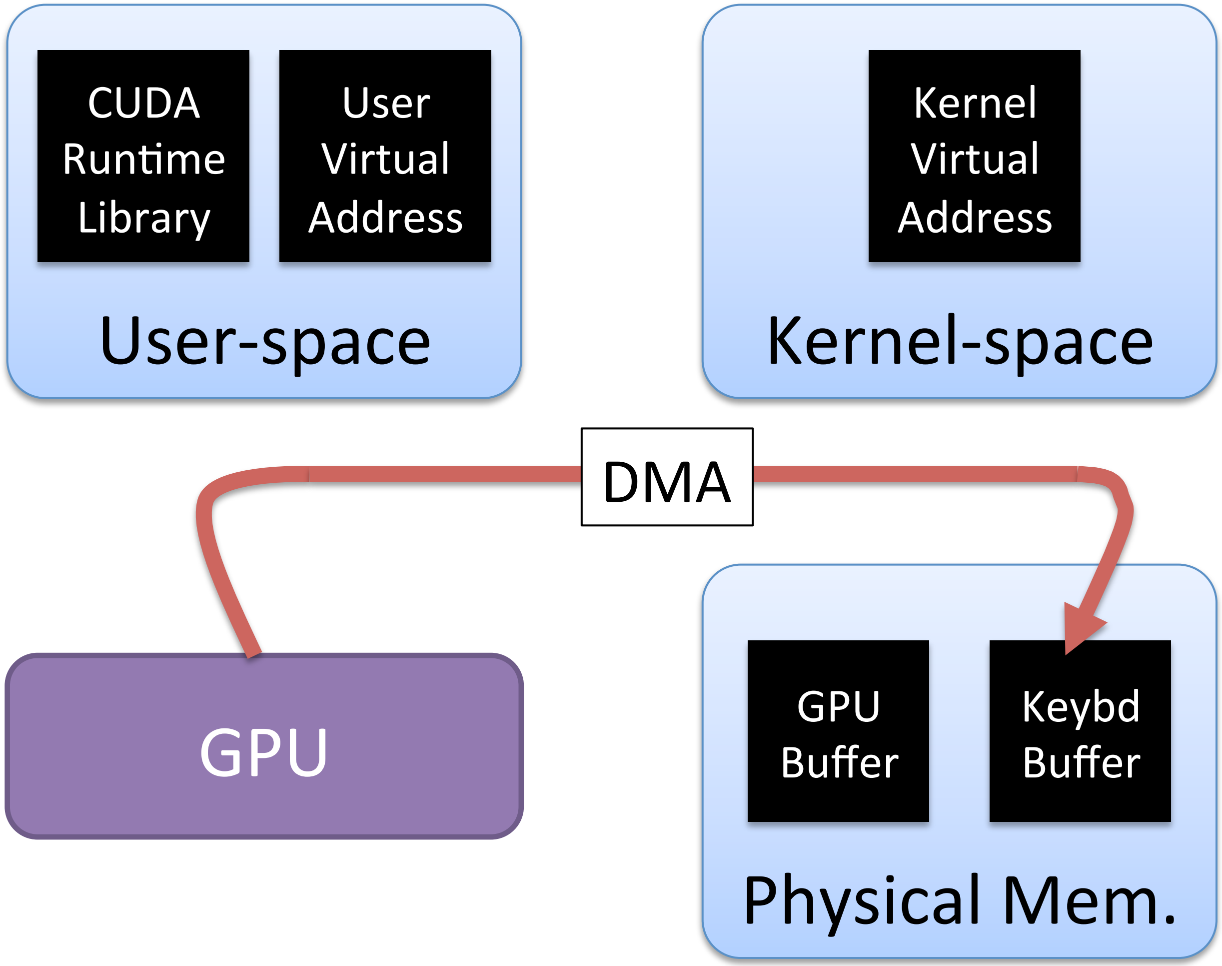
How the GPU access host memory



Opportunity: Remap process' virtual memory to sensitive physical pages



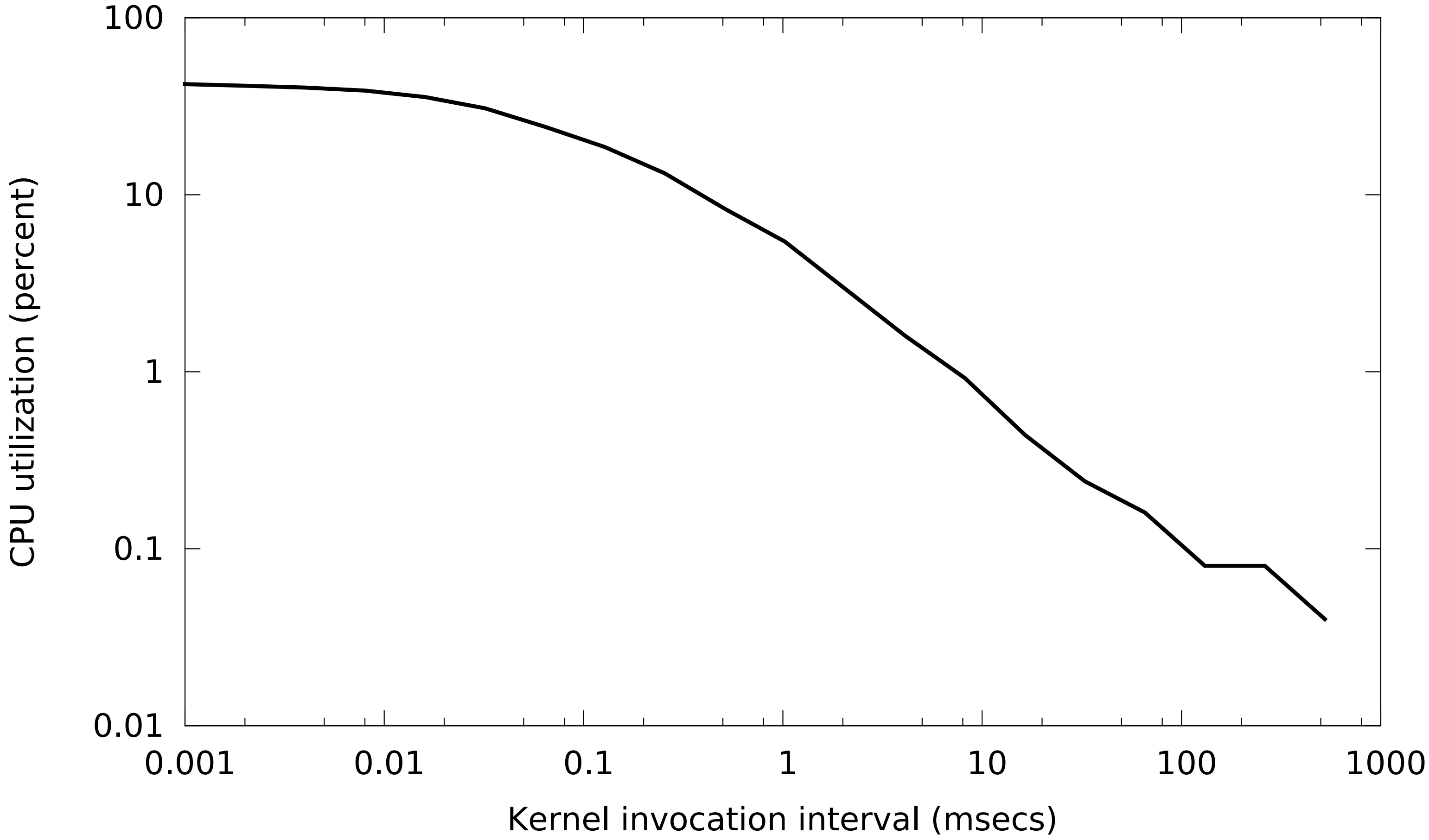
Opportunity: Remap process' virtual memory to sensitive physical pages



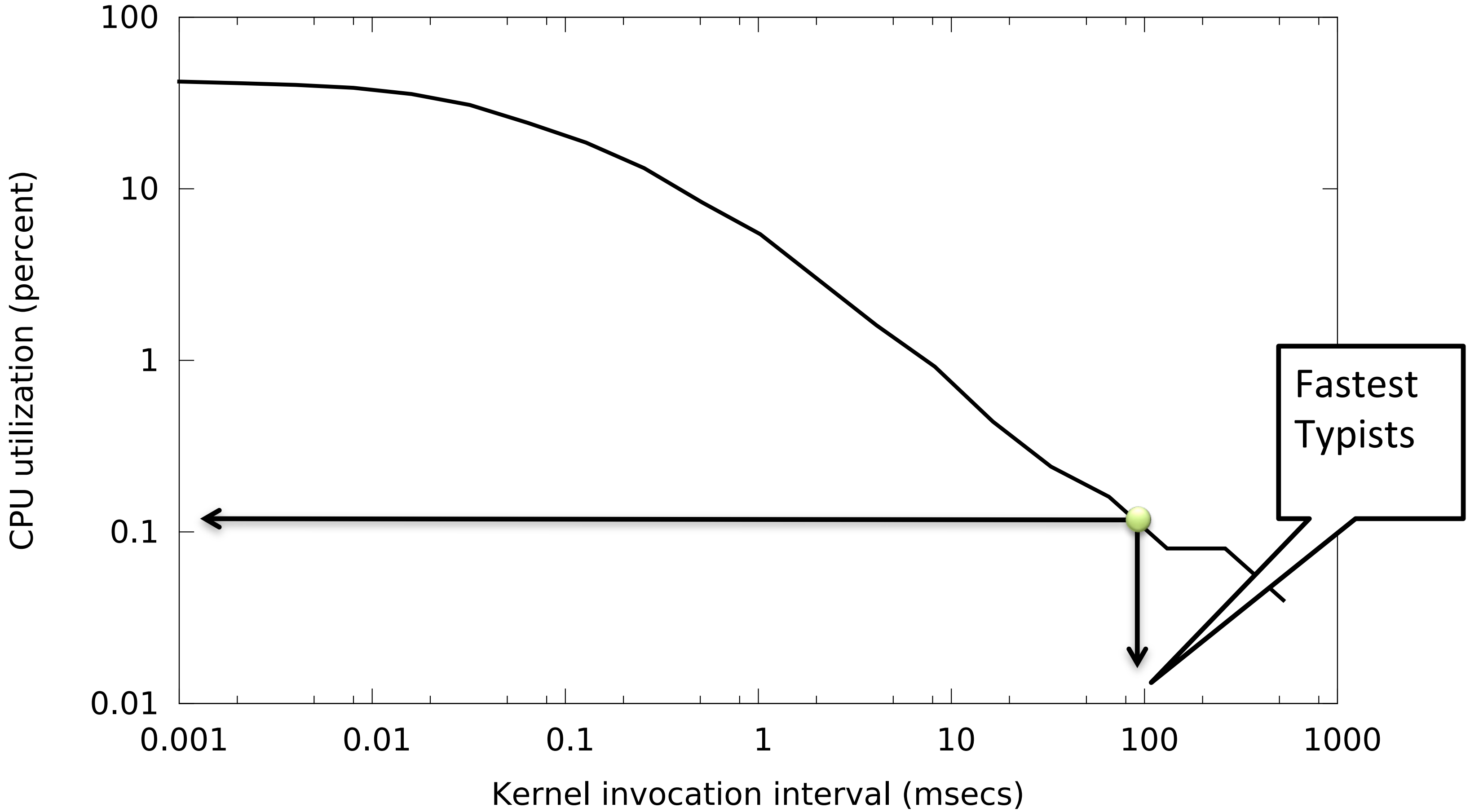
Implementation

- 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

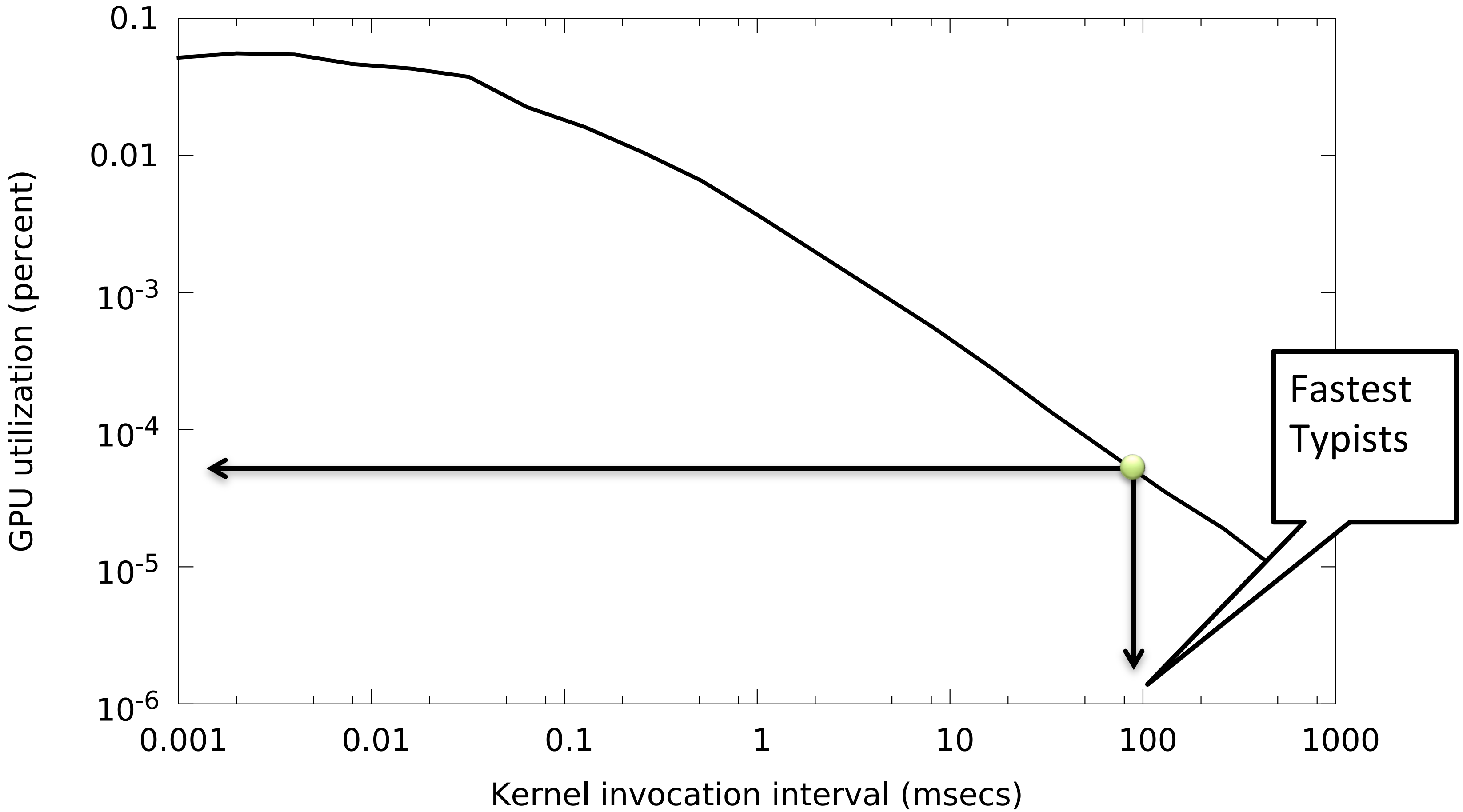
CPU Utilization



CPU Utilization



GPU Utilization



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

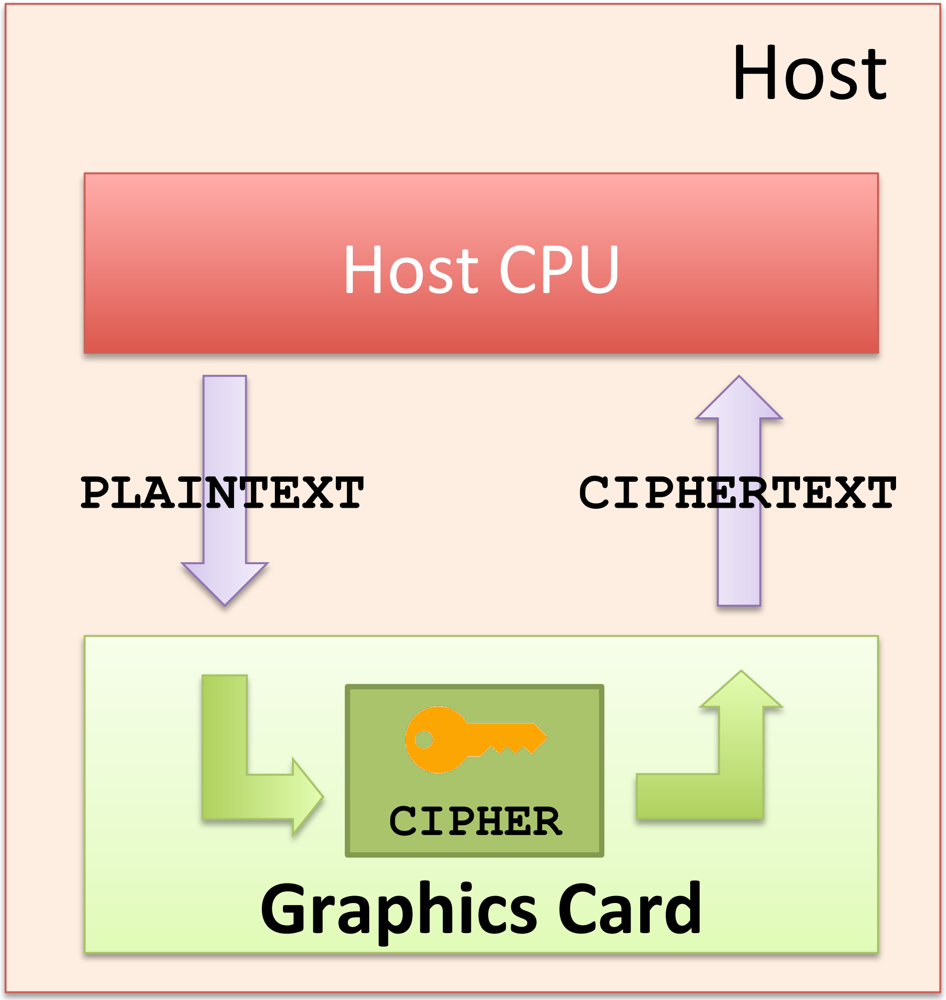
- Background and motivation
- GPU-based Malware Signature Detection
  - Network intrusion detection/prevention
  - Virus scanning
- GPU-assisted Malware
  - Code-armoring techniques
  - Keylogger
- **GPU as a Secure Crypto-Processor**
- Conclusions

Motivation

- Modern cryptography is based on keys

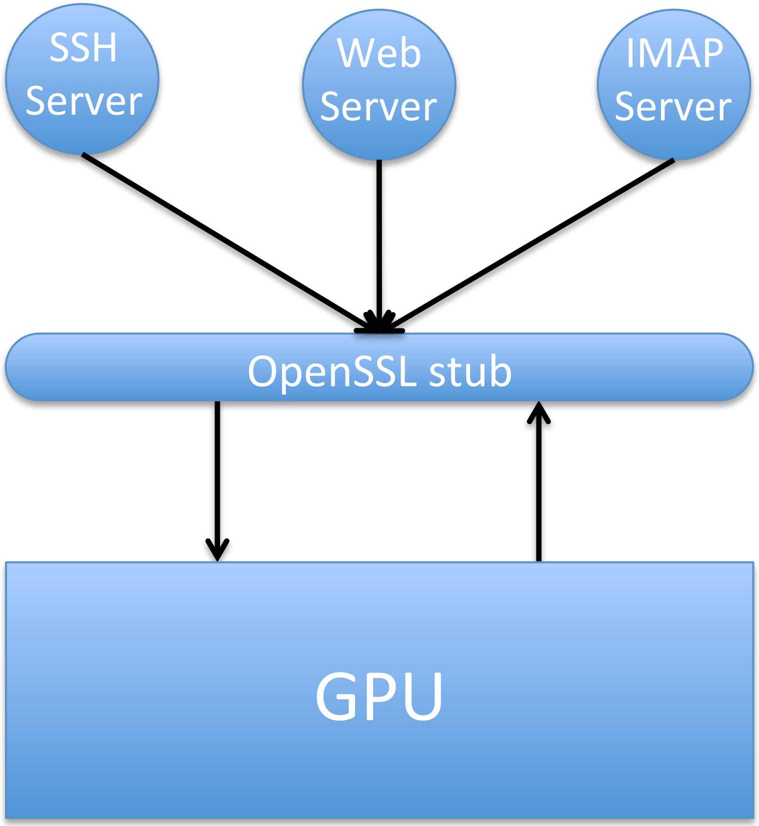
- **Problem:** Secret keys may remain unencrypted in CPU Registers, RAM, etc.
  - Memory disclosure attacks
    - Heartbleed
  - DMA/Firewire attacks
  - Physical attacks
    - Cold-boot attacks
  - …

PixelVault Overview



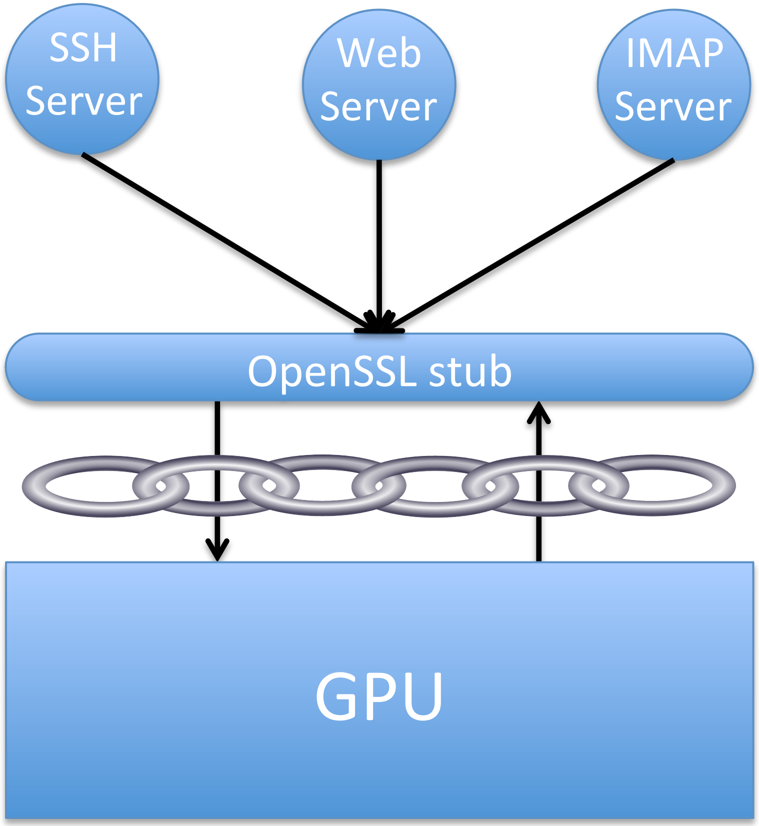
- Runs encryption securely outside CPU/RAM
- Only on-chip memory of GPU is used as storage
- Secret keys are never observed from host

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?

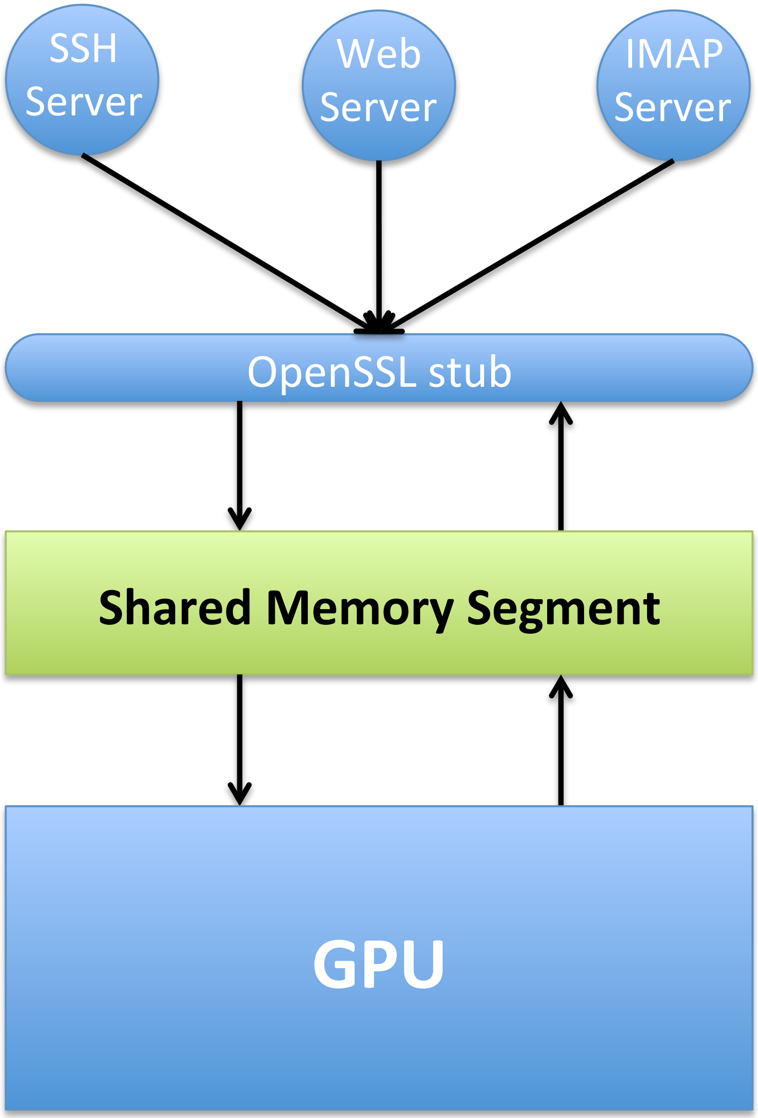
- Who holds the keys?

- Where is the code?

Autonomous GPU execution

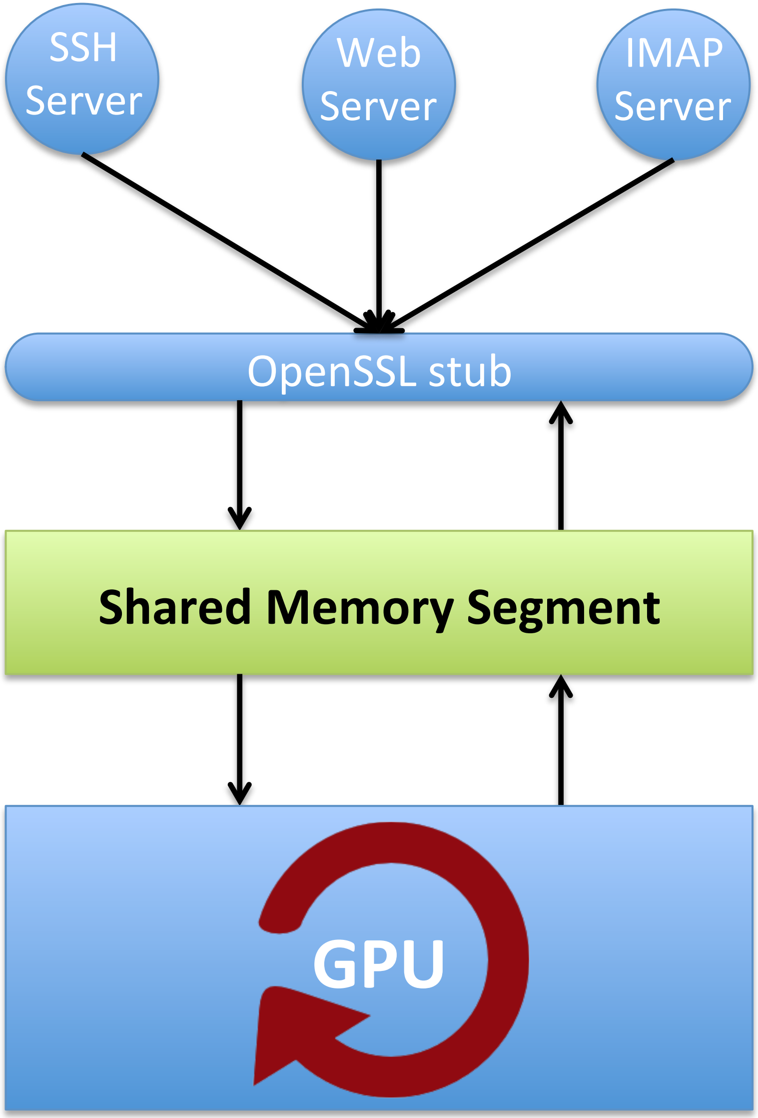
- Force GPU program to run indefinitely
  - i.e., using an infinite **`while`** loop
- GPUs are non-preemptive
  - No other program can run at the same time
- We use a shared memory segment for communication between the CPU and the GPU

Shared Memory between CPU/GPU



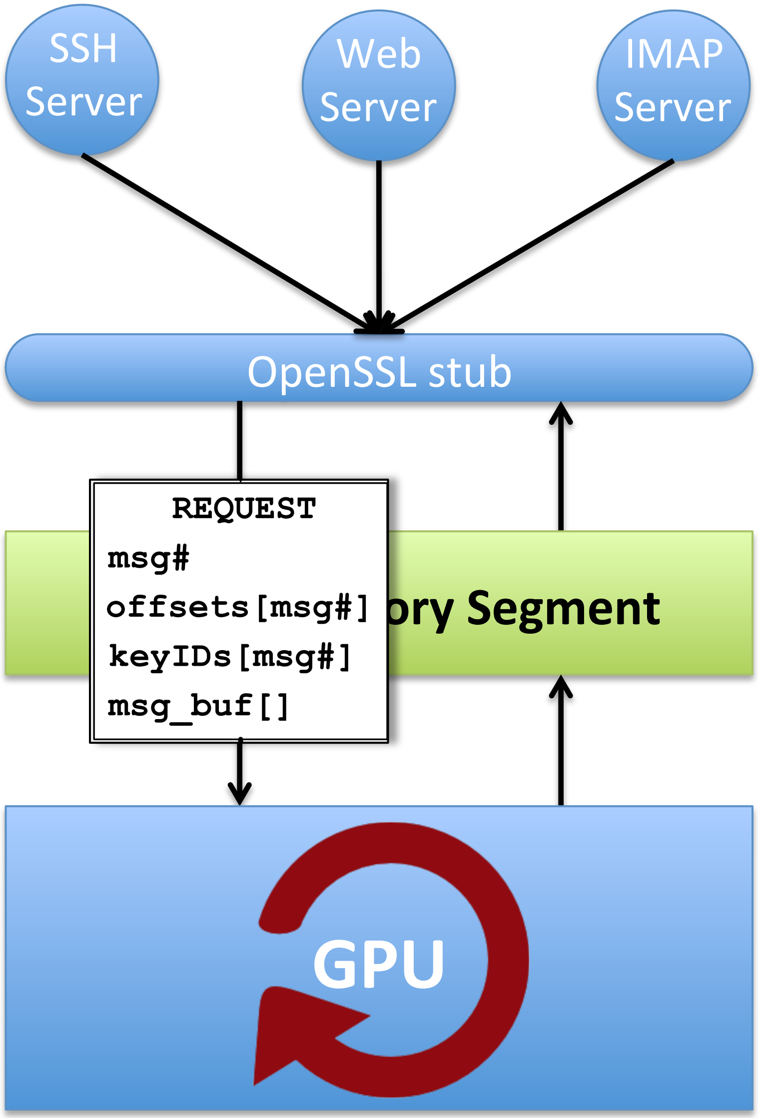
- *Page-locked* memory
  - Accessed by the GPU directly, via DMA
  - Cannot be swapped to disk
- Processing requests are issued through this shared memory space

Shared Memory between CPU/GPU



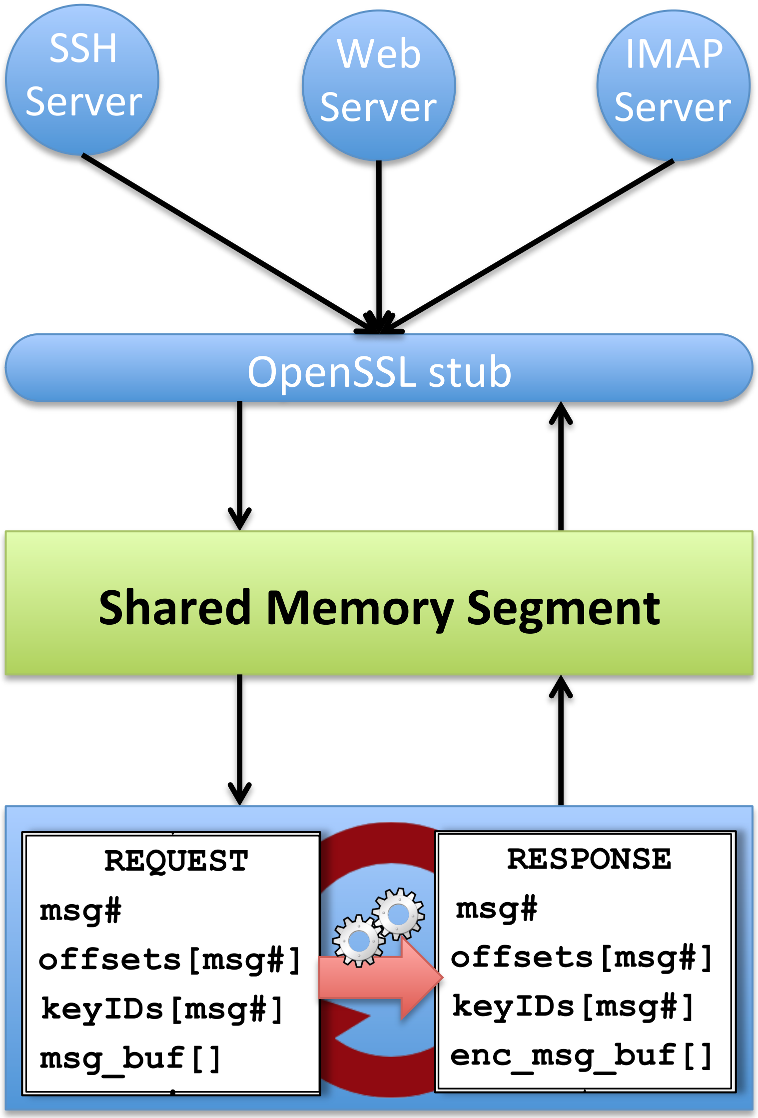
- GPU continuously monitors the shared space for new requests

Shared Memory between CPU/GPU



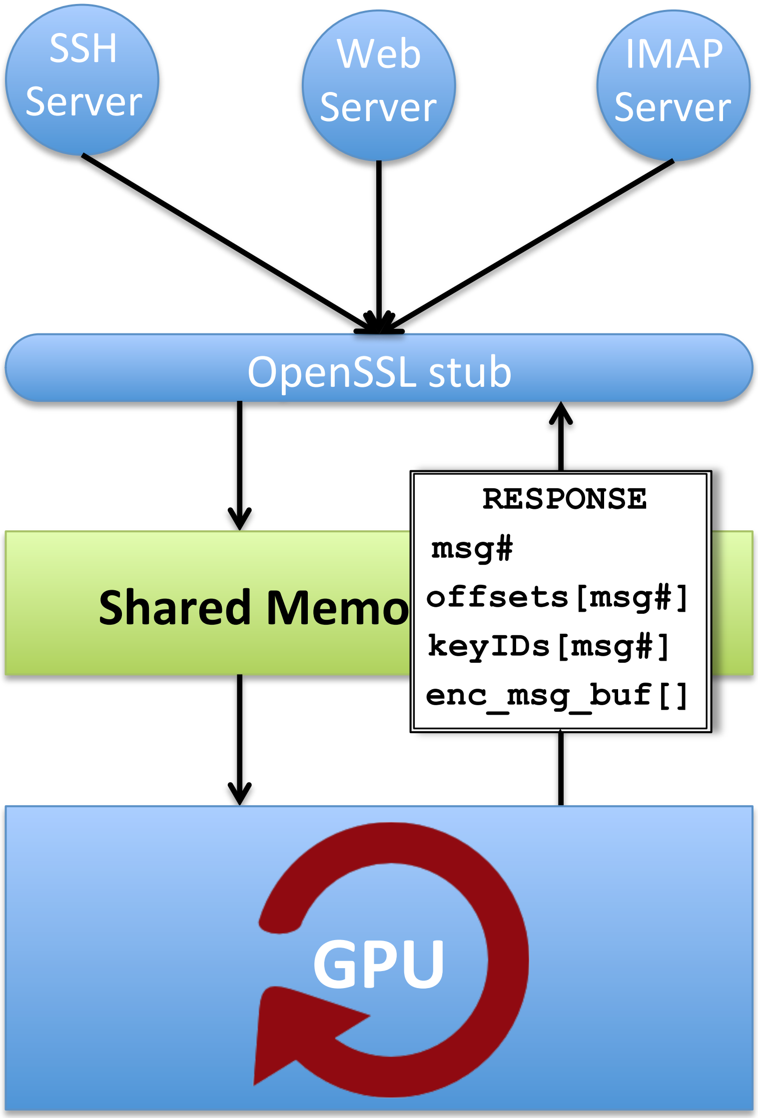
- When a new request is available, it is transferred to the memory space of the GPU

Shared Memory between CPU/GPU



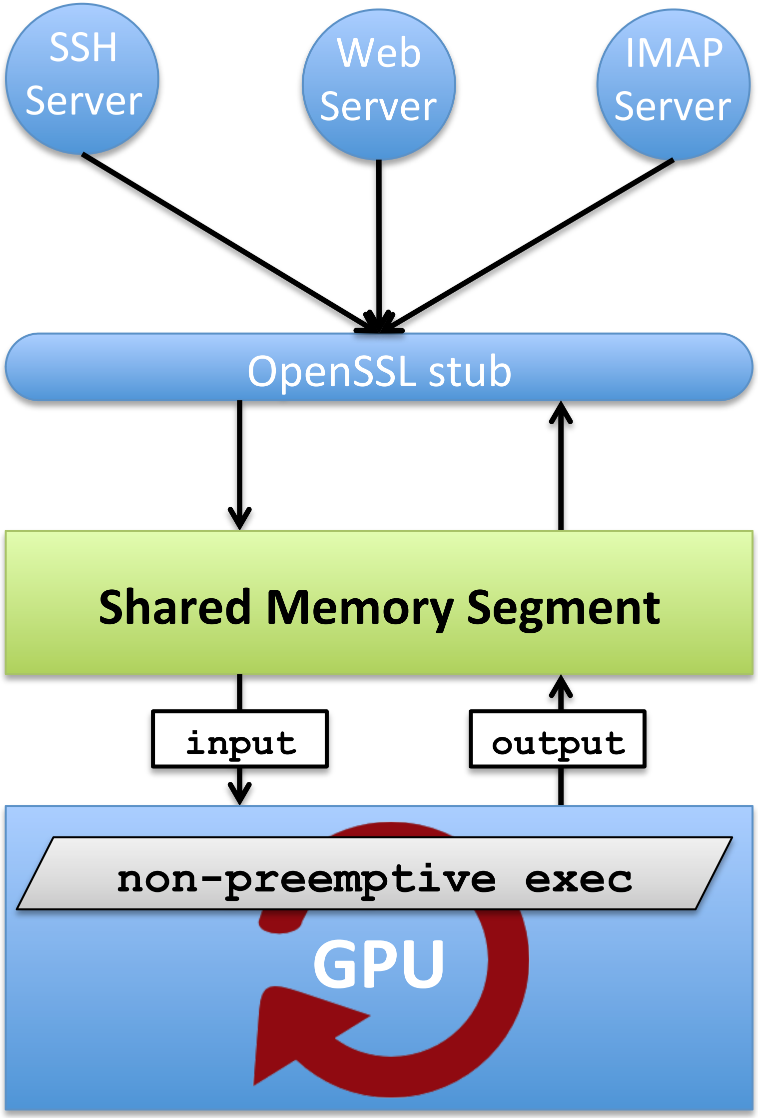
- The request is processed by the GPU

Shared Memory between CPU/GPU



- When processing is finished, the host is notified by setting the response parameter fields accordingly

Autonomous GPU execution

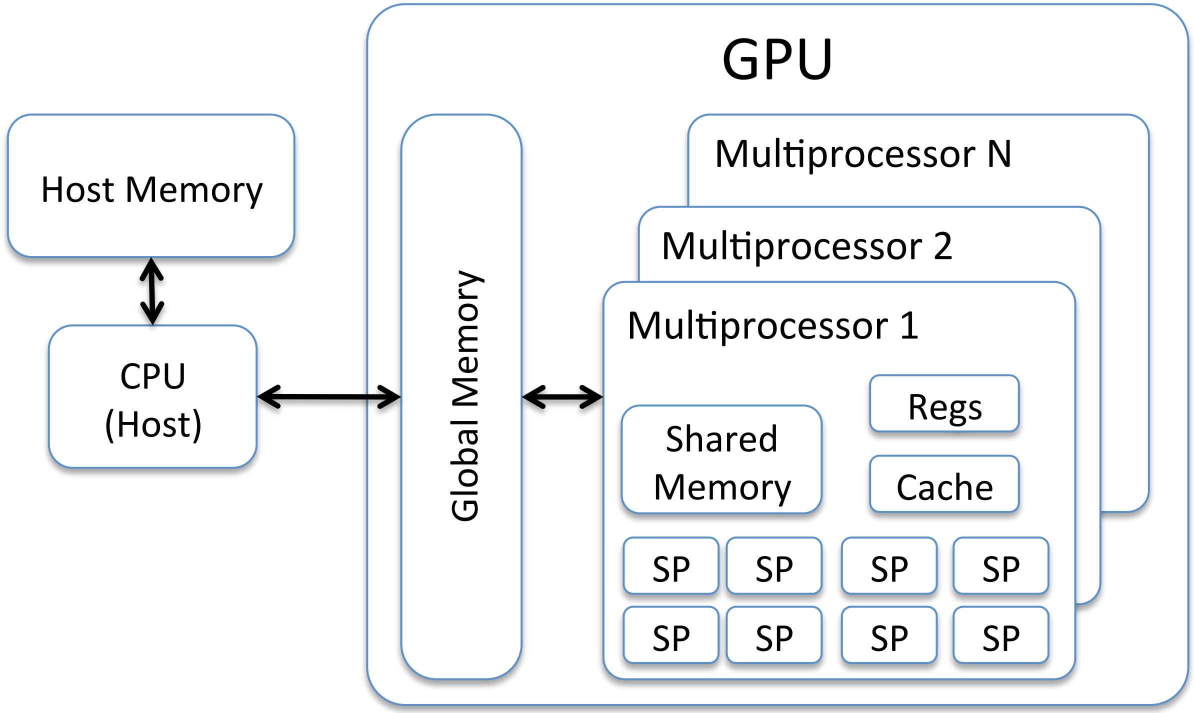


- Non-preemptive execution
- Only the output block is being written back to host memory

Implementation Challenges

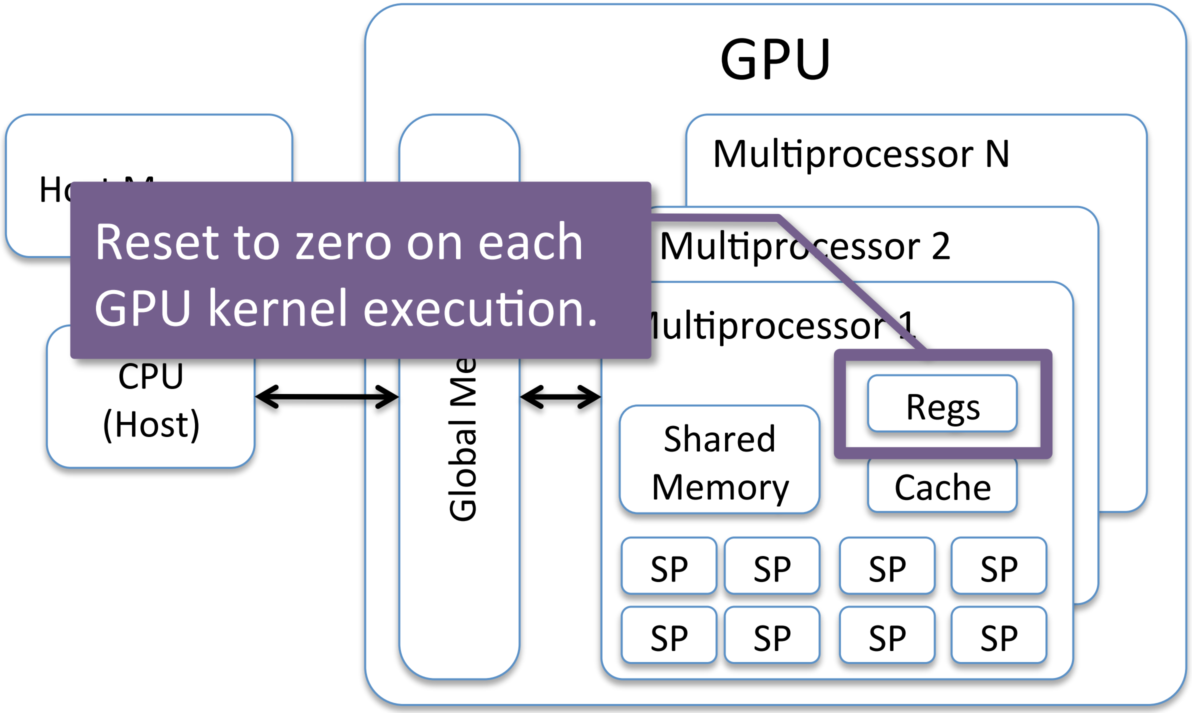
- How to isolate GPU execution?
- Who holds the keys?
- Where is the code?

Who holds the keys?



- GPUs contain different memory hierarchies of …
  - different sizes, and …
  - different characteristics

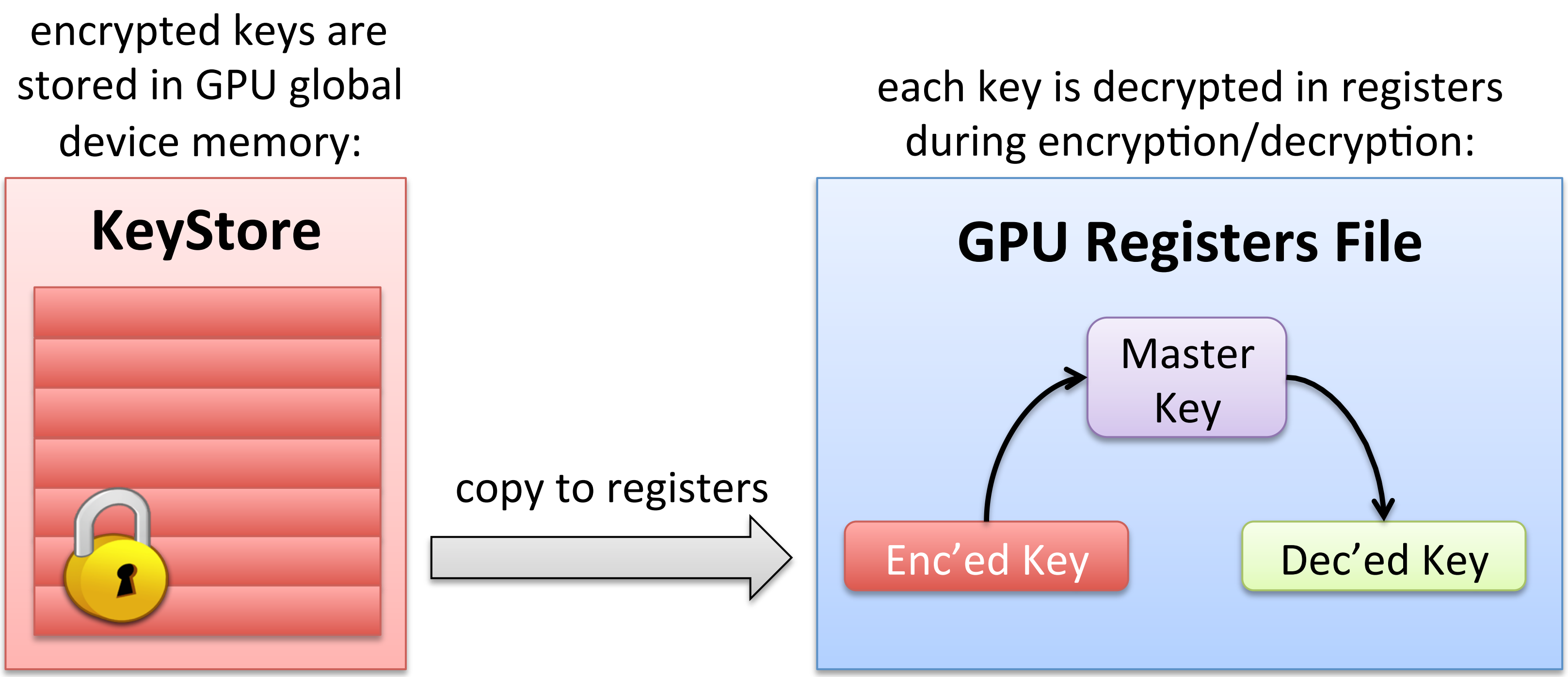
Who holds the keys?



- 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



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

Global Device Memory

```
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
```

Prevent GPU code modification attacks

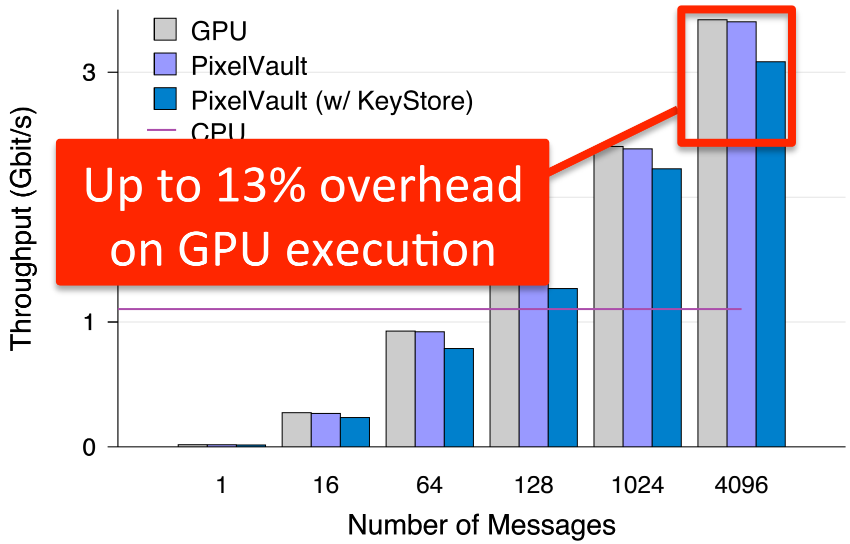
- Three levels of instruction caching (icache)
  - 4KB, 8KB, and 32KB, respectively
  - Hardware-managed

- **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

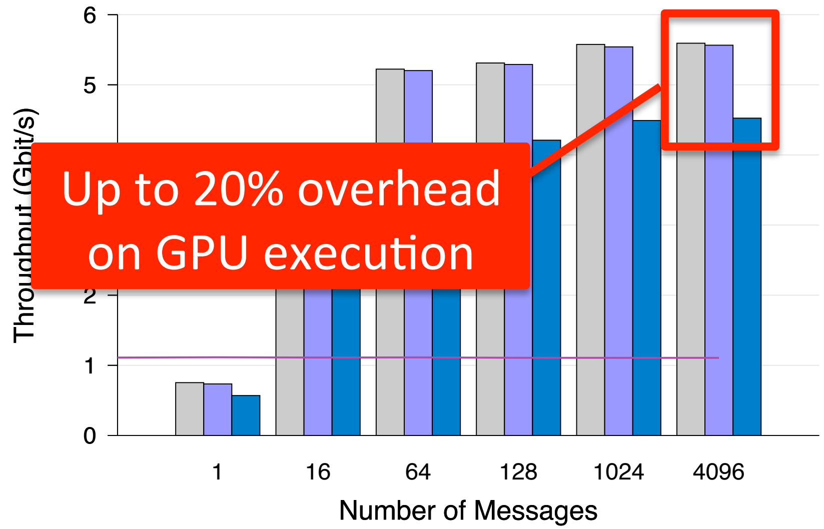
PixelVault Crypto Suite

- 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

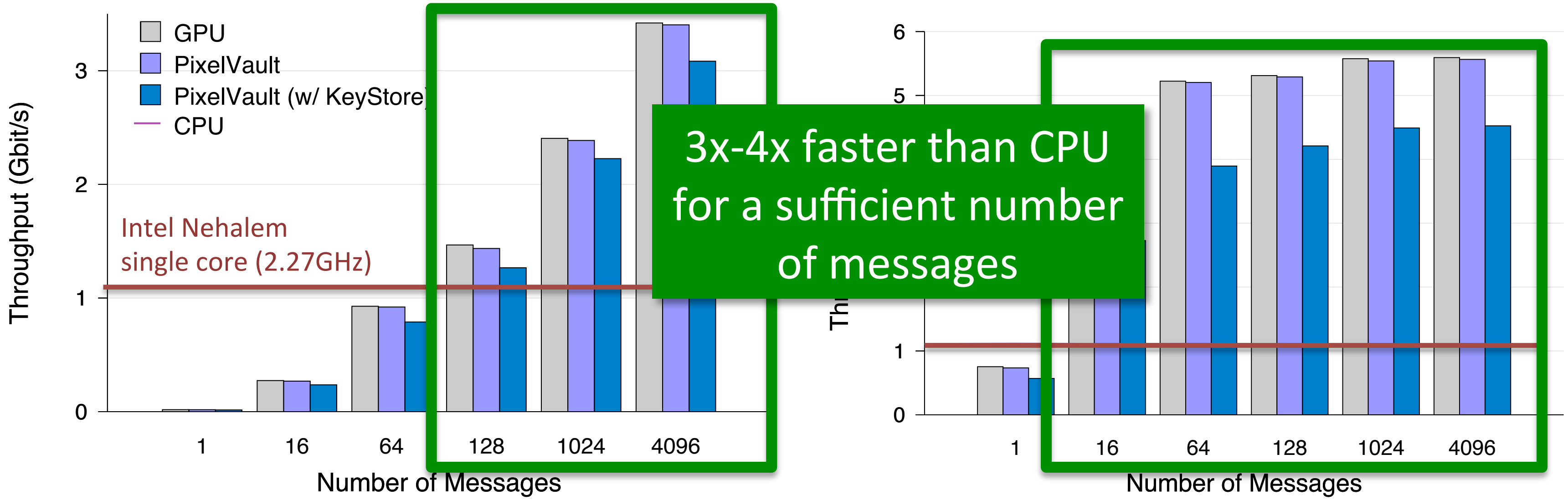


Encryption



Decryption

AES-128 CBC Performance



Encryption

Decryption

RSA 1024-bit Decryption

#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

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

RSA 1024-bit Decryption

#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

- 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

    "PixelVault: Using GPUs for Securing Cryptographic Operations"

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!