Security Applications of GPUs

Sotiris Ioannidis
Foundation for Research and Technology – Hellas (FORTH)

Outline

- **Background and motivation**
- GPU-based, signature-based malware detection
  - Network intrusion detection/prevention
  - Virus scanning
- GPU-assisted malware
  - Code-armoring techniques
  - Keylogger
- GPU as a secure crypto-processor
- Conclusions

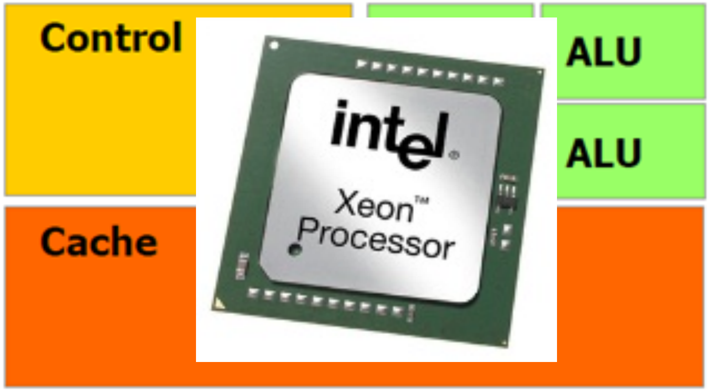
GPU = Graphics Processing Unit

- The heart of graphics cards
- Mainly used for real-time 3D game rendering
  - Massively parallel processing capacity

Why GPU?

- General-purpose computing
  - Flexible and programmable
  - Portability
- Powerful, ubiquitous, affordable
  - Dominant co-processor
  - Constant innovation
  - Inexpensive and always-present
- Data-parallel model

CPU vs. GPU



CPU

GPU

Xeon X5550:
4 cores
731M transistors

GTX480:
480 cores
3,200M transistors

Single **I**nstruction, **M**ultiple **T**hreads

- 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

```
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 Instruction, Multiple Threads

- Example: vector addition

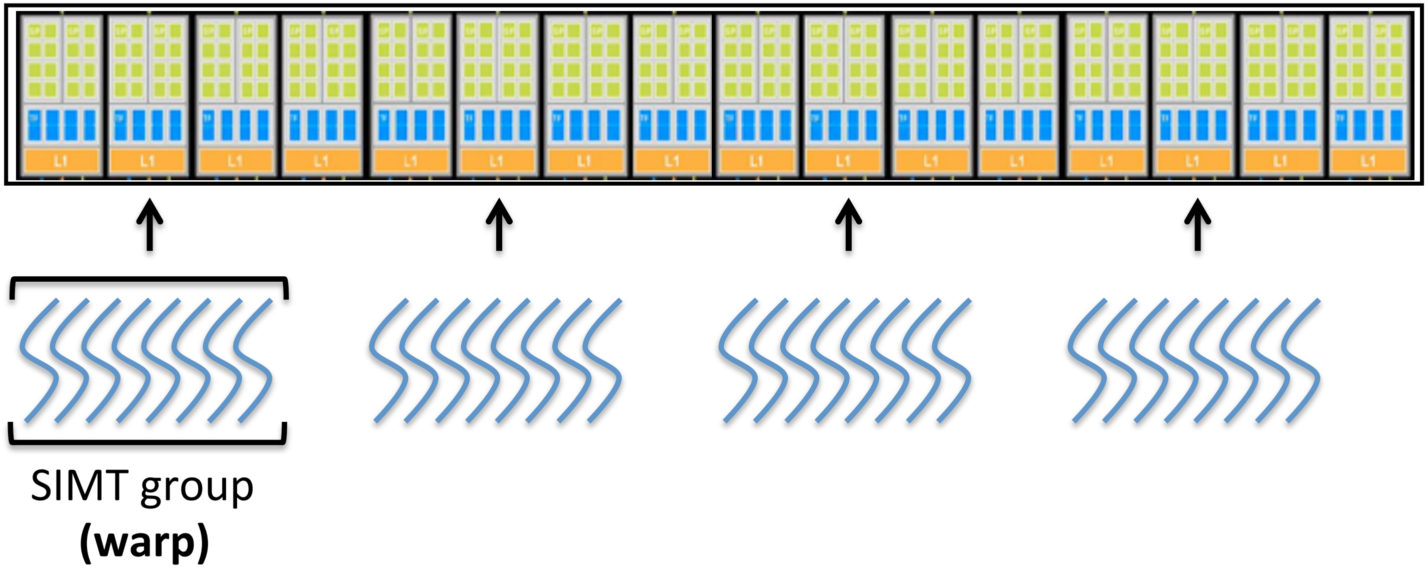
CPU code

```
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

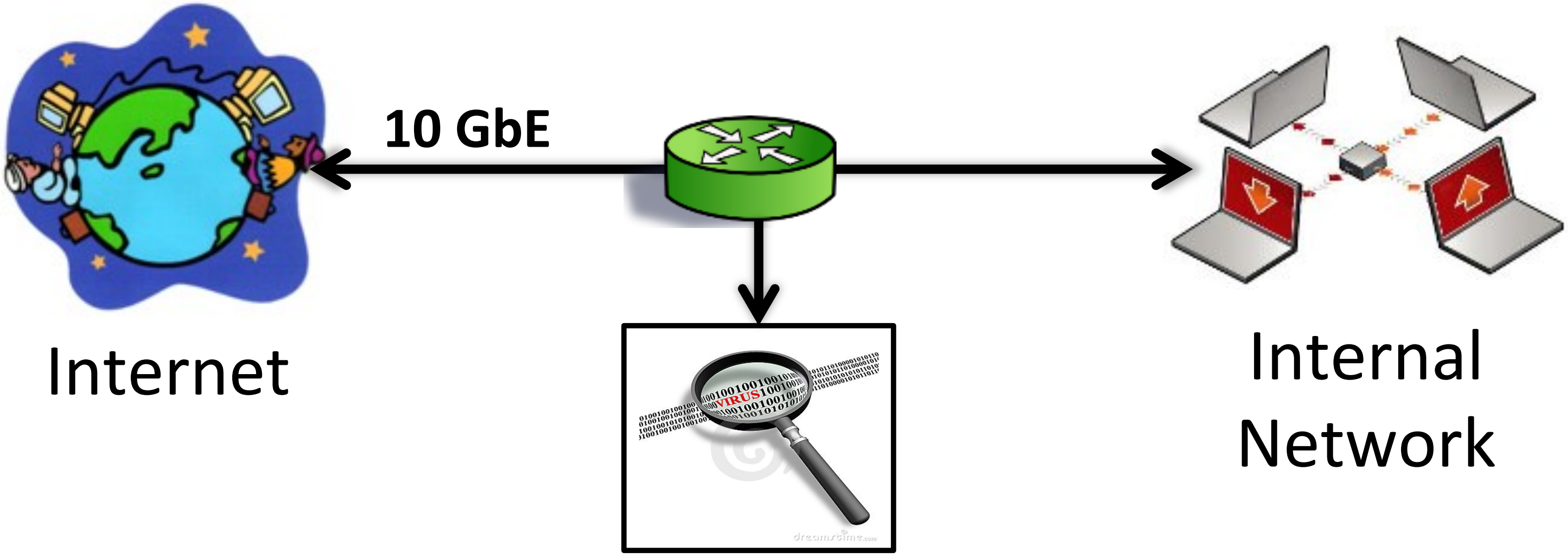
- *Great for regular computations!*

Outline

- Background and motivation
- **GPU-based, signature-based malware detection**
  - **Network intrusion detection/prevention**
  - Virus scanning
- GPU-assisted malware
  - Code-armoring techniques
  - Keylogger
- GPU as a secure crypto-processor
- Conclusions

Signature-based Detection

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



10 GbE

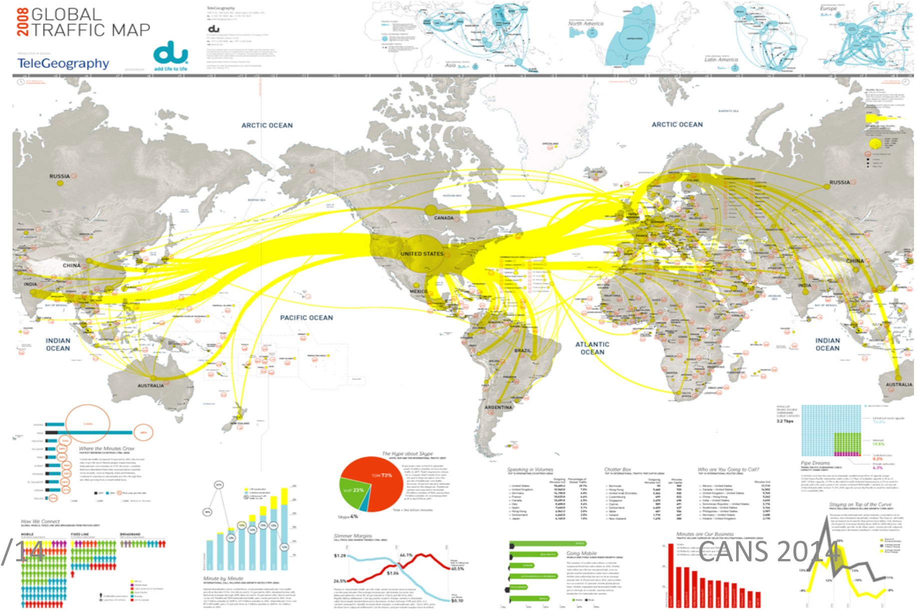
Internet

Internal
Network

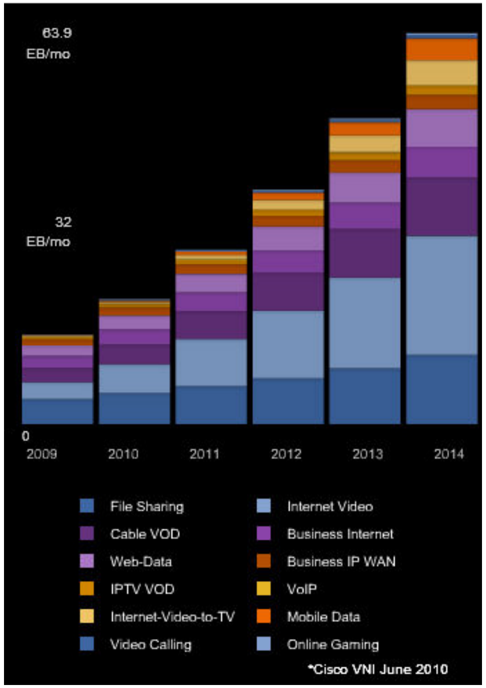
NIDS

Challenges (1)

- ***Traffic rates*** are increasing
  - – 10 Gbit/s Ethernet speeds are common in metro/ enterprise networks
  - – More than 40 Gbit/s at the core

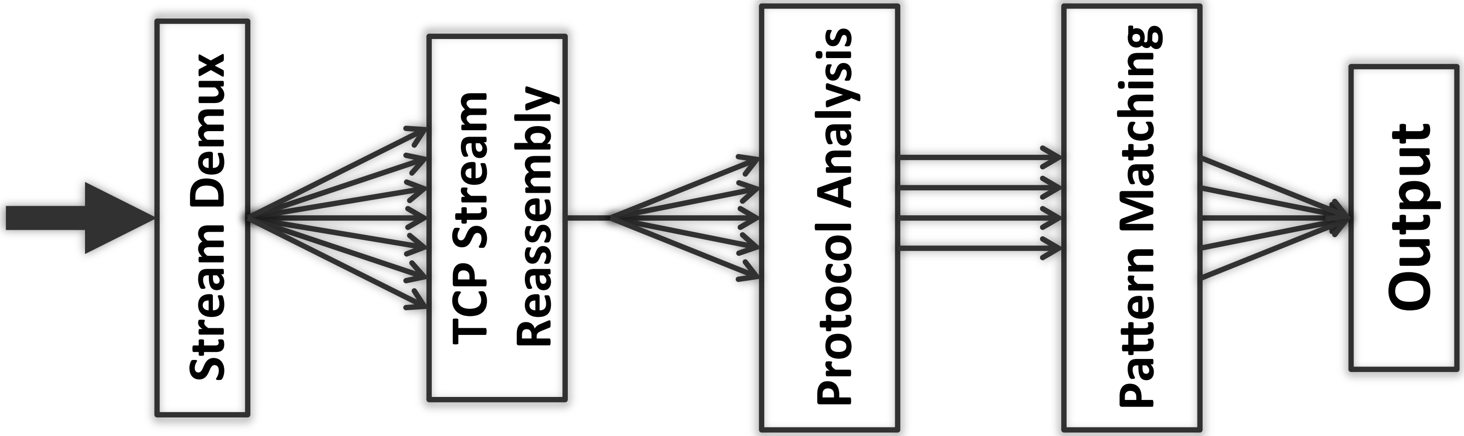


Internet, Managed IP and Mobile IP Traffic Forecast
June 2010



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

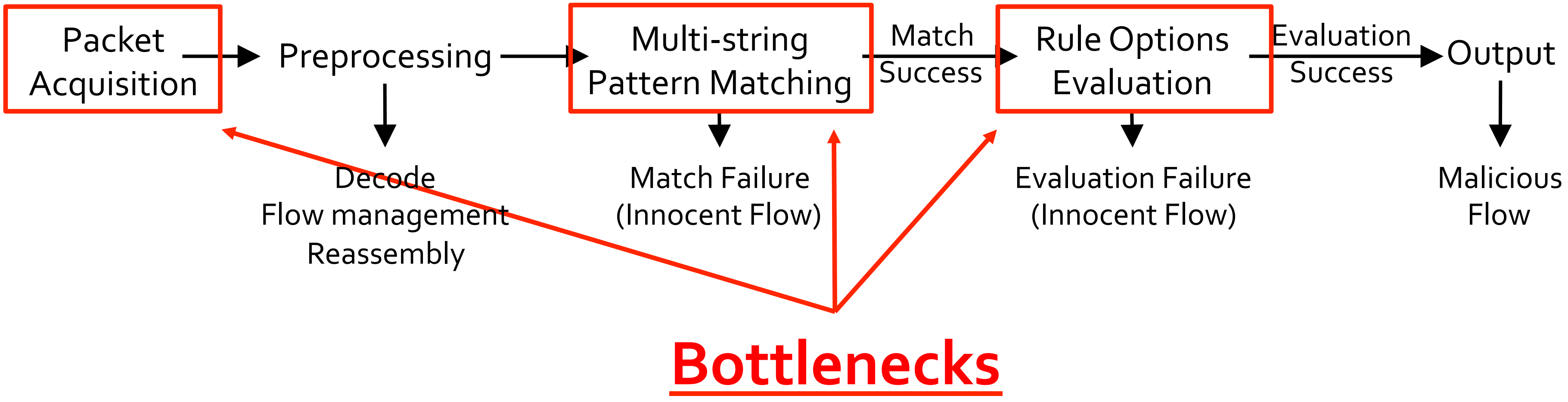
- Commodity "software" NIDS/NIPS and AVs
  - Processing by general-purpose processors
  - Throughput: Low

Open-source S/W

≤ ~1 Gbps

Typical Signature-based NIDS Architecture

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
(msg:“possible attack attempt BACKDOOR optix runtime detection" content:"/whitepages/page_me/
100.html"; pcre:"/body=\x2521\x2521\x2521Optix\s+Pro\s+v\d+\x252E\d+\S+sErver\s+Online
\x2521\x2521\x2521/")
```



Packet Acquisition

Preprocessing

Decode
Flow management
Reassembly

Multi-string Pattern Matching

Match Failure
(Innocent Flow)

Match Success

Rule Options Evaluation

Evaluation Failure
(Innocent Flow)

Evaluation Success

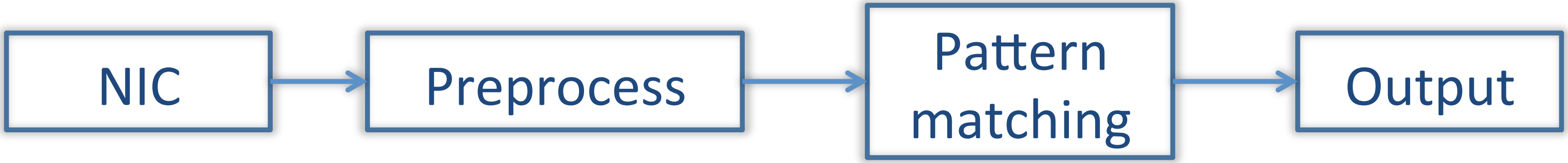
Output

Malicious Flow

Bottlenecks

* PCRE: Perl Compatible Regular Expression

Single-threaded NIDS performance

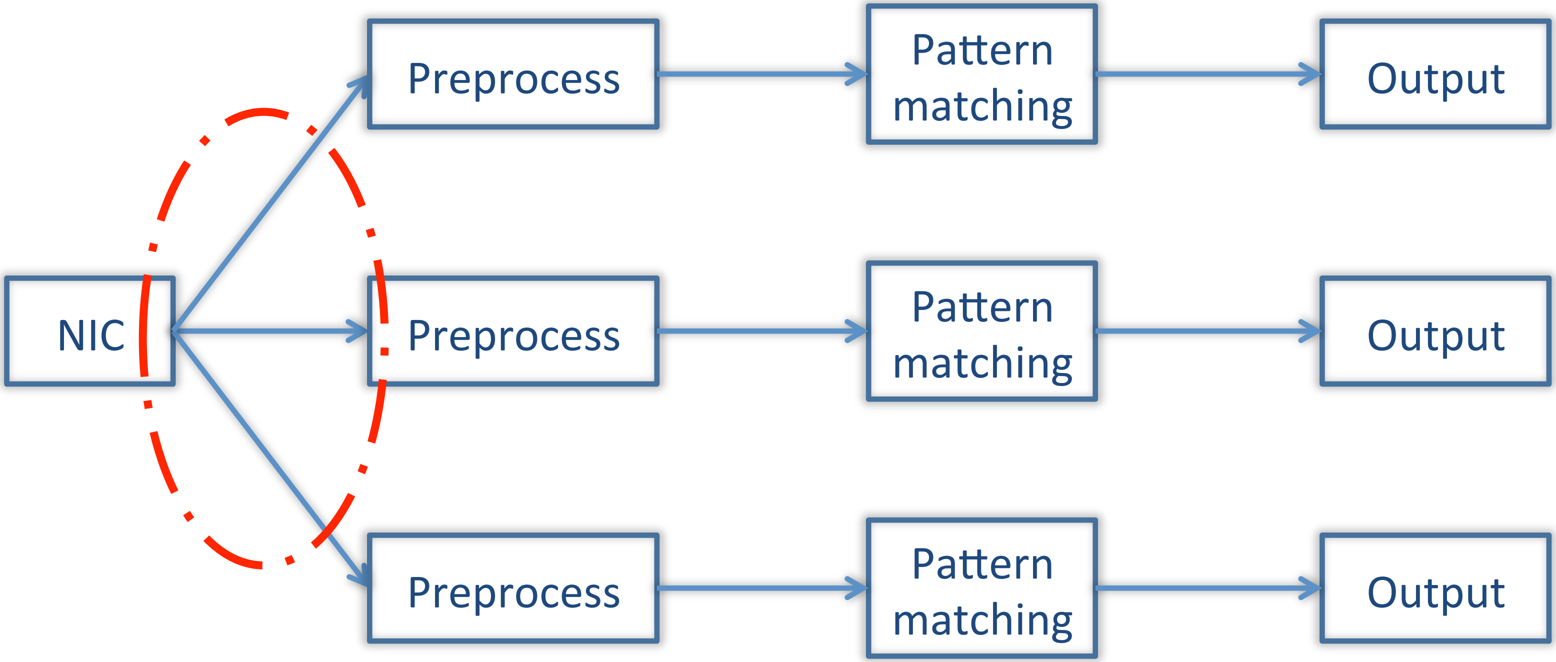


- **Vanilla Snort: 0.2 Gbit/s**

Problem #1: Scalability

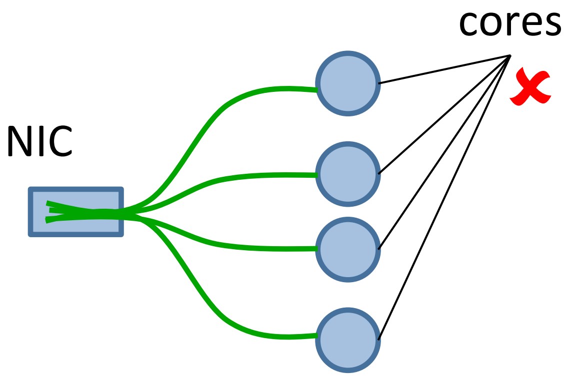
- Single-threaded NIDS have limited performance
  - Do not scale with the number of CPU cores

Multi-threaded performance

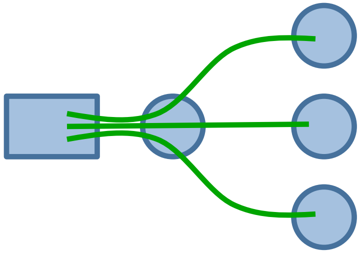


- Vanilla Snort: 0.2 Gbit/s
- **With multiple CPU-cores: 0.9 Gbit/s**

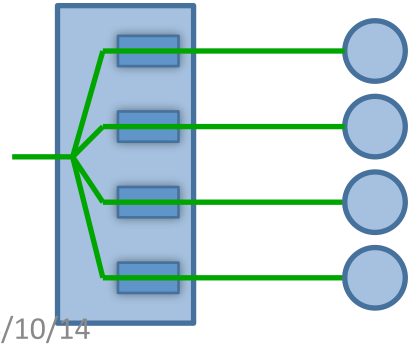
Problem #2: How to split traffic



- ✗ Synchronization overheads

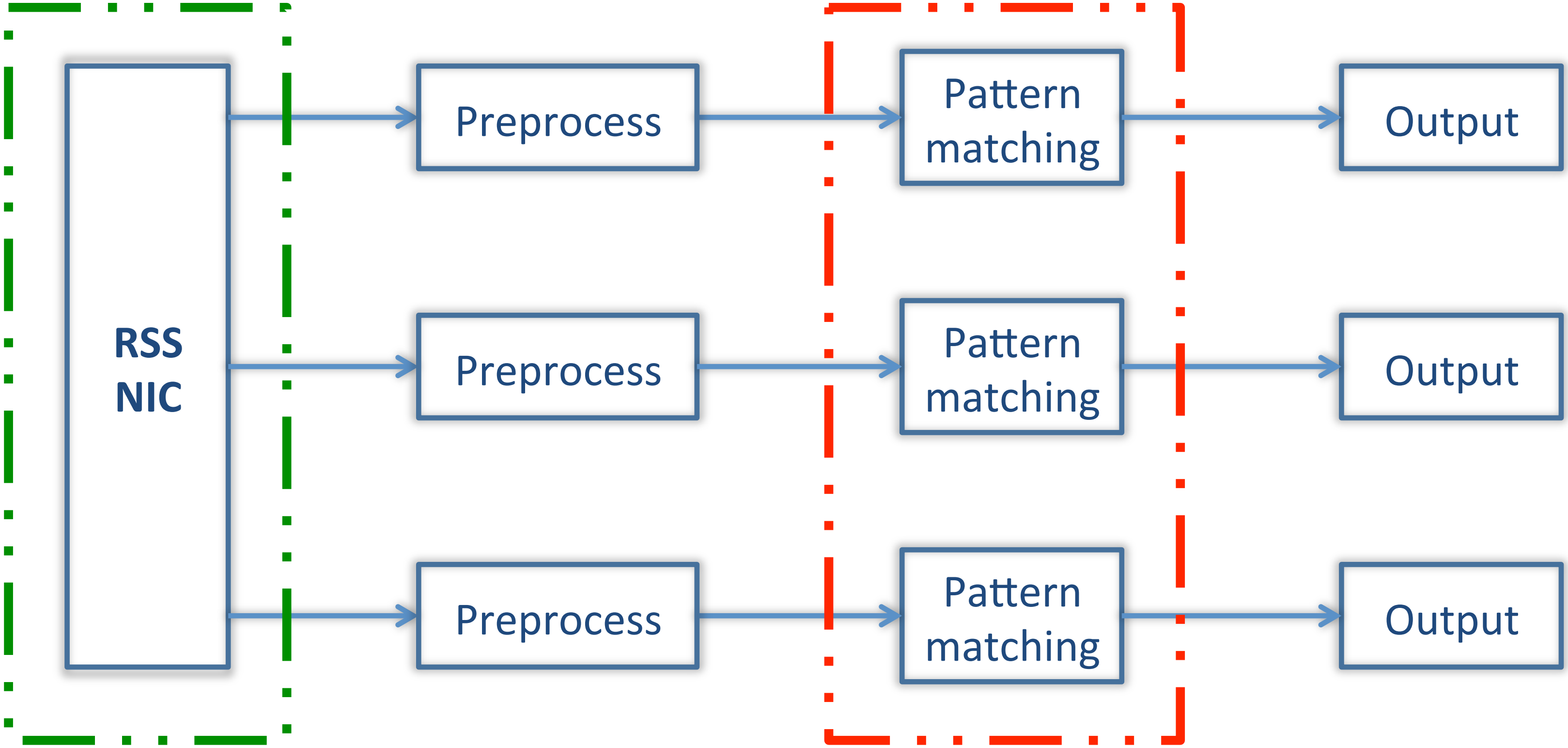


- ✗ Cache misses



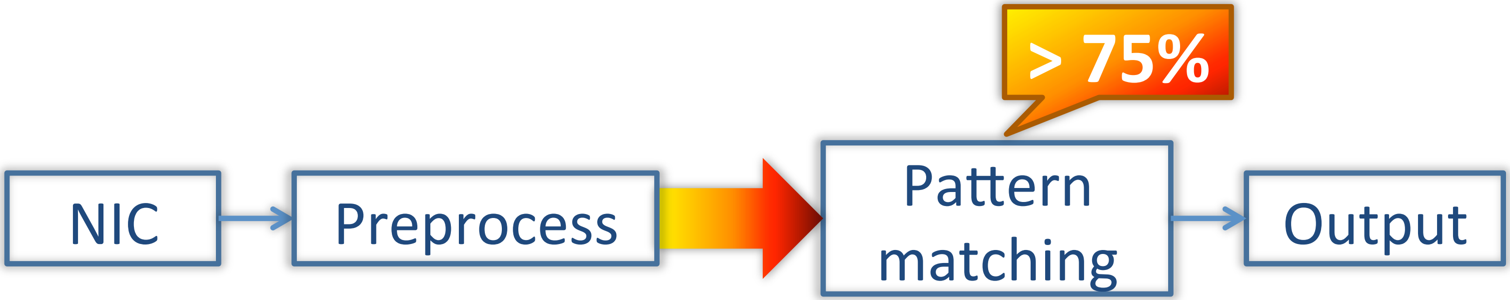
- ✓ Receive-Side Scaling (RSS)

Multi-queue performance



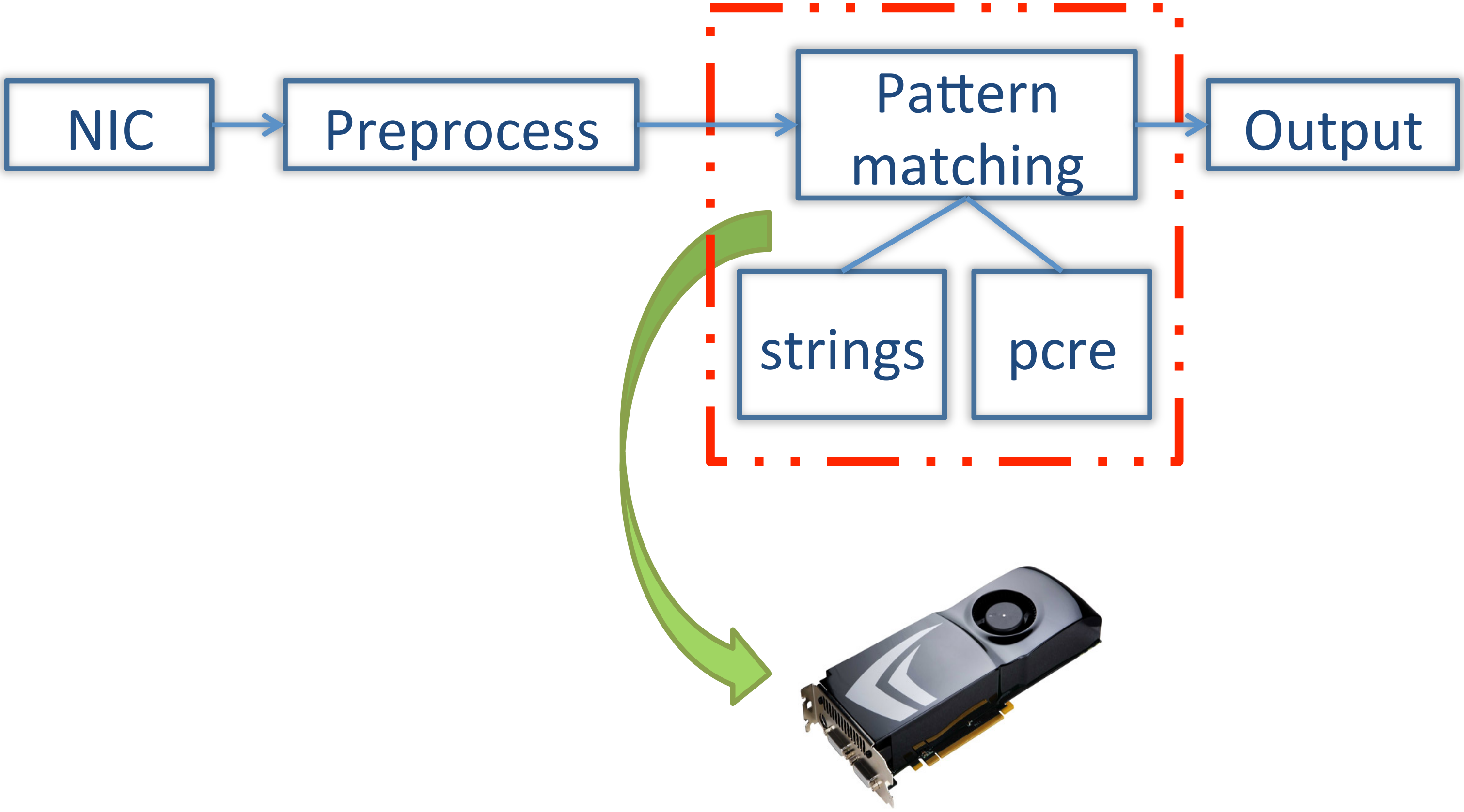
- Vanilla Snort: 0.2 Gbit/s
- With multiple CPU-cores: 0.9 Gbit/s
- **With multiple Rx-queues: 1.1 Gbit/s**

Problem #3: Pattern matching is the bottleneck



- On an 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

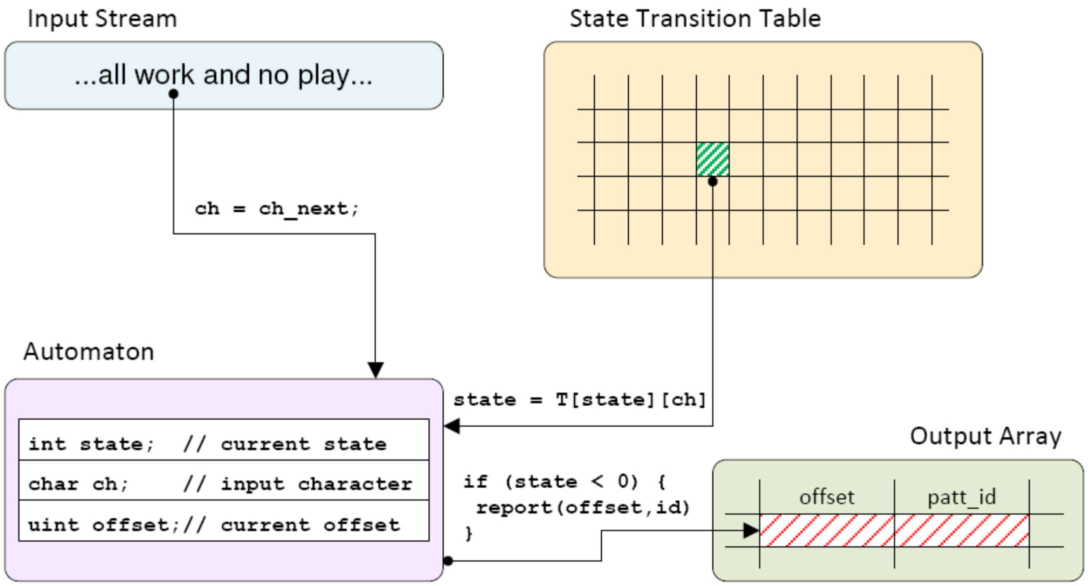
strings

pcre

Output

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)*



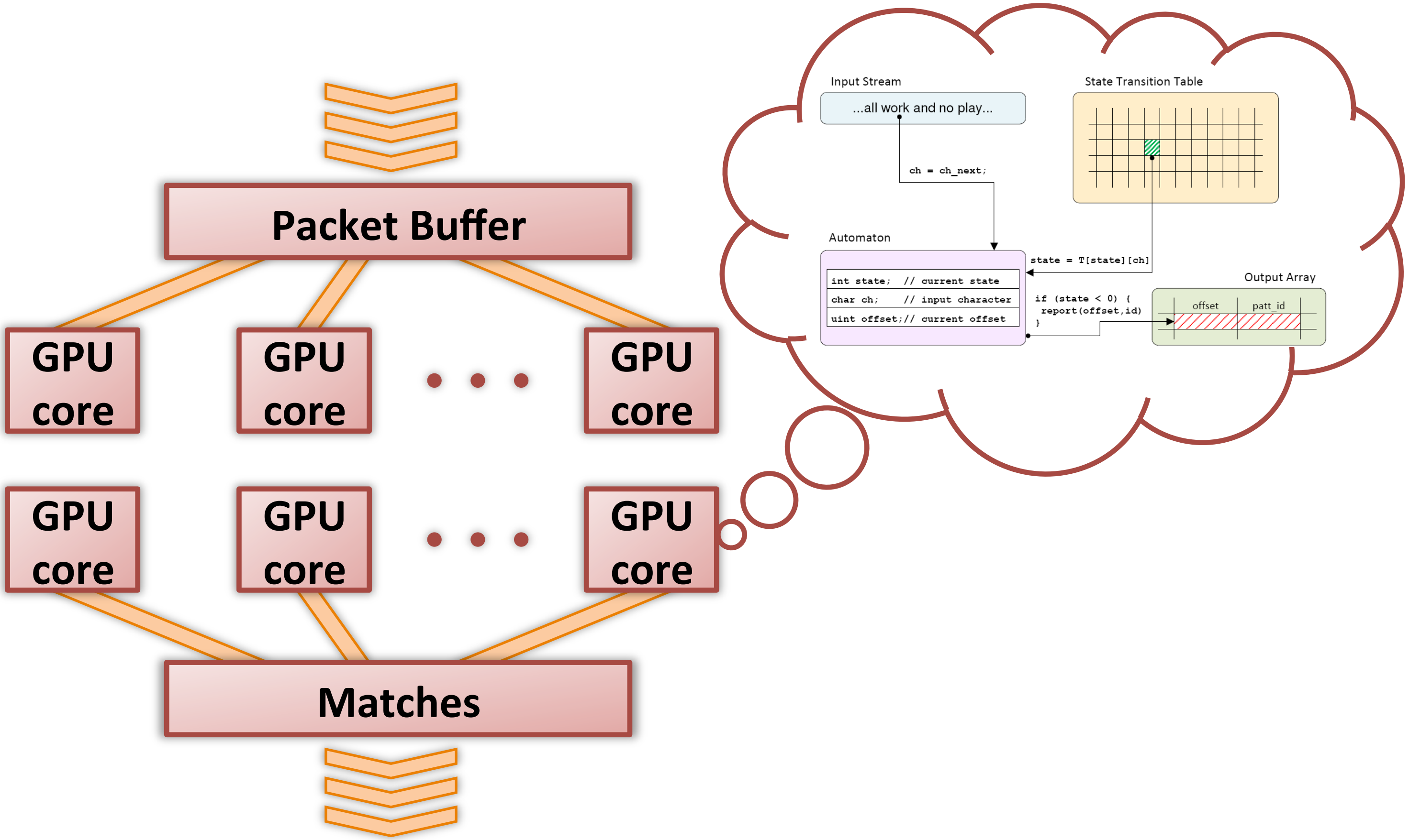
```
ch = ch_next;
```

```
state = T[state][ch]
```

```
int state;   // current state
char ch;     // input character
uint offset;// current offset
```

```
if (state < 0) {
 report(offset,id)
}
```

Pattern matching on the GPU



Packet Buffer

GPU core　GPU core　…　GPU core

GPU core　GPU core　…　GPU core

Matches

```
ch = ch_next;
```

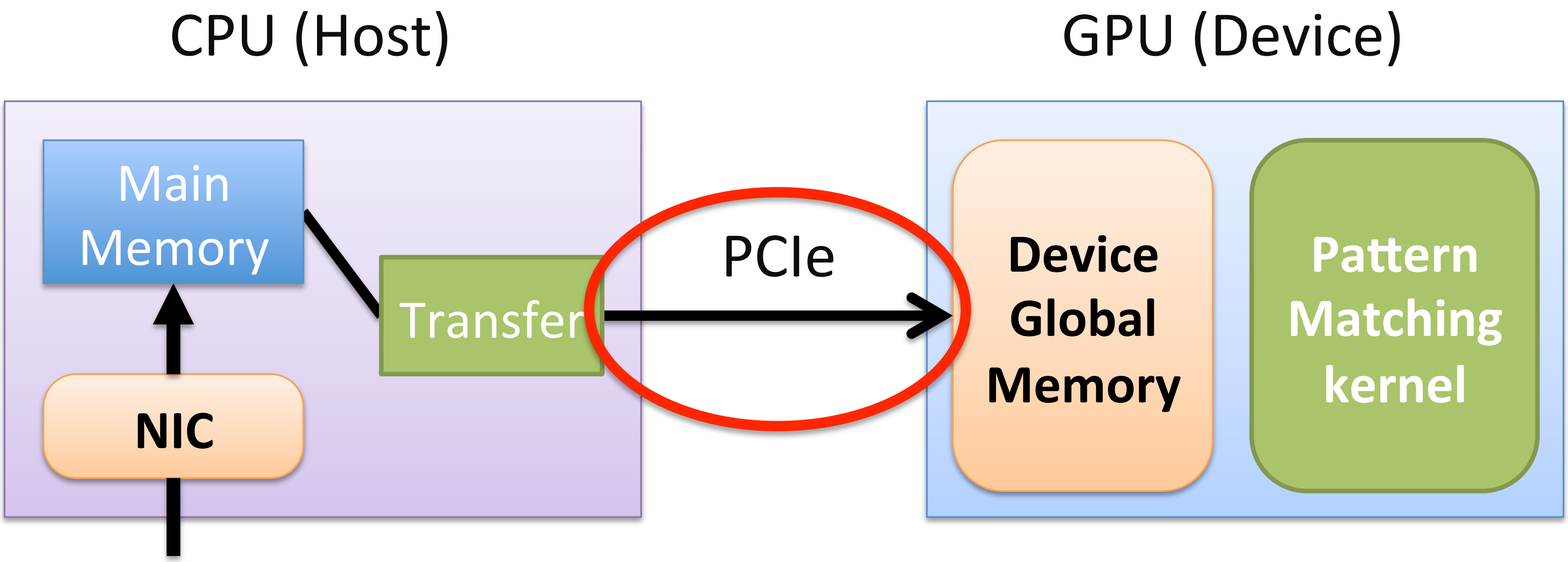
```
state = T[state][ch]
```

```
int state;  // current state
char ch;    // input character
uint offset;// current offset
```

```
if (state < 0) {
 report(offset,id)
}
```

- Uniformly one core for each reassembled packet stream

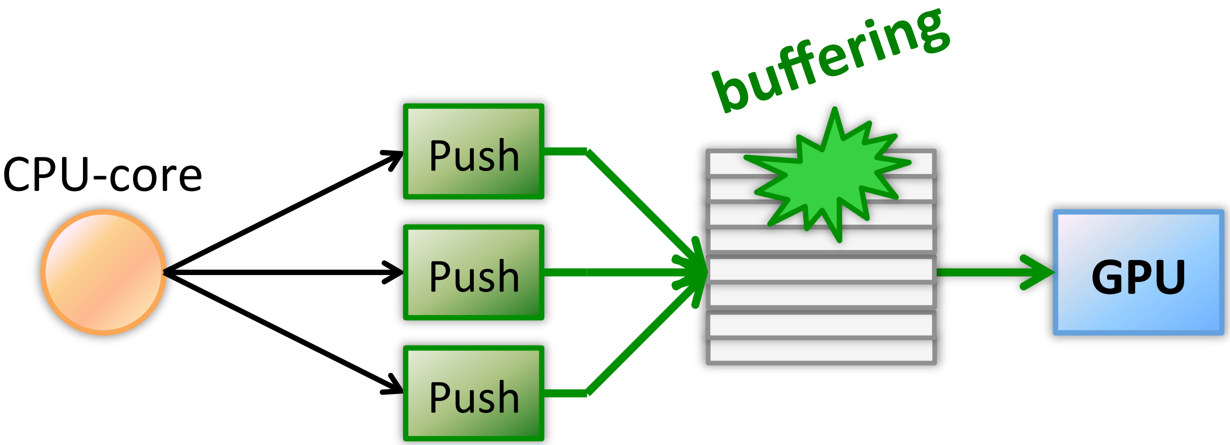
Multiple data transfers



- Several data transfers between different devices

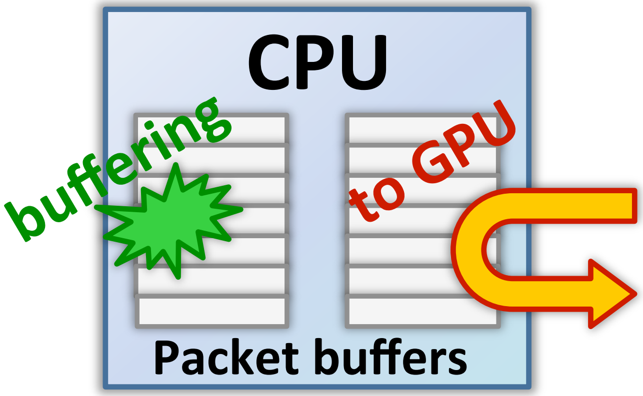
  Are the data transfers worth the computational gains offered?

Transferring to GPU



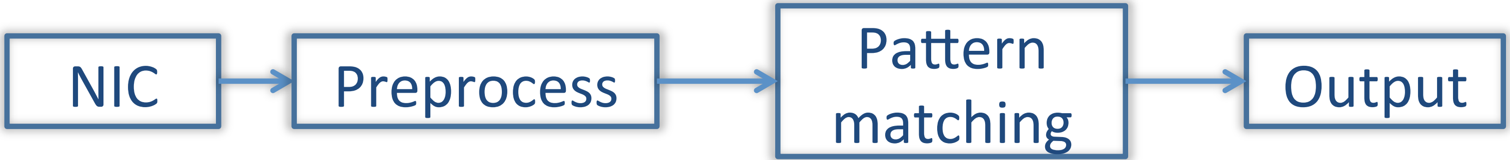
- Small transfer results to PCIe throughput degradation
  - ➔Many reassembled packets are batched into a single buffer

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

Pattern matching on the GPU

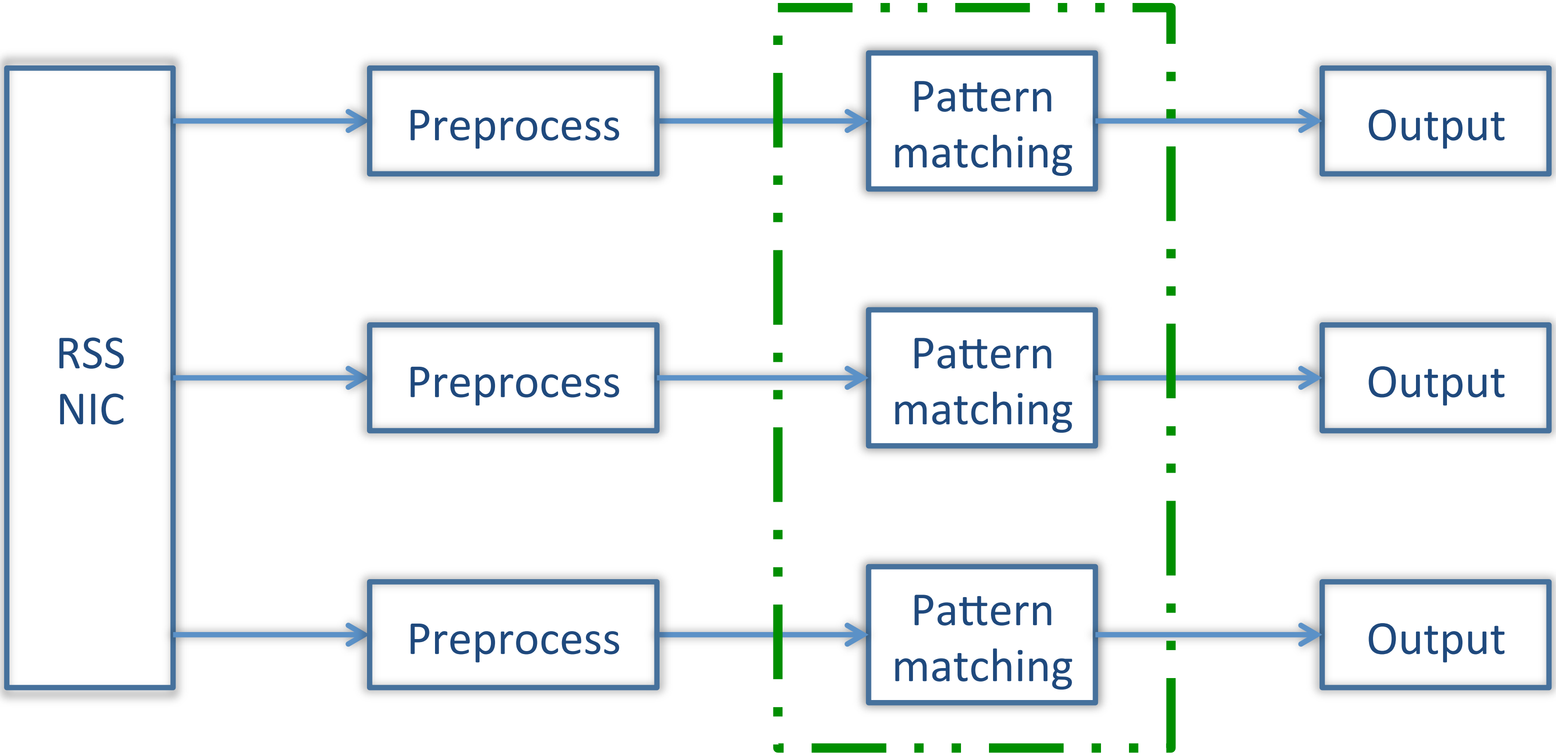


NVIDIA GTX 480 GPU

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**

Offloading pattern matching to the GPU



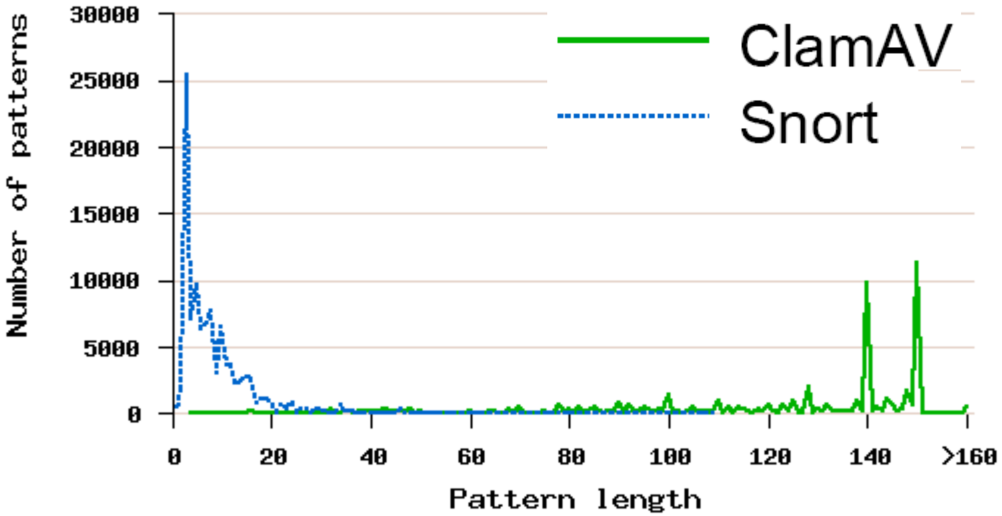
- Vanilla Snort: 0.2 Gbit/s
- With multiple CPU-cores: 0.9 Gbit/s
- With multiple Rx-queues: 1.1 Gbit/s
- **With GPU: 5.2 Gbit/s**

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

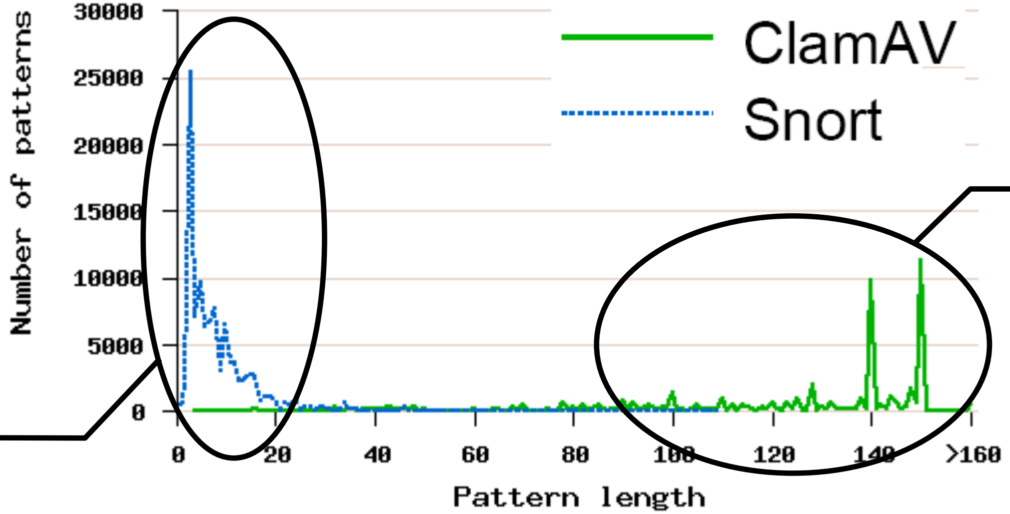
Anti-Virus Databases

- 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

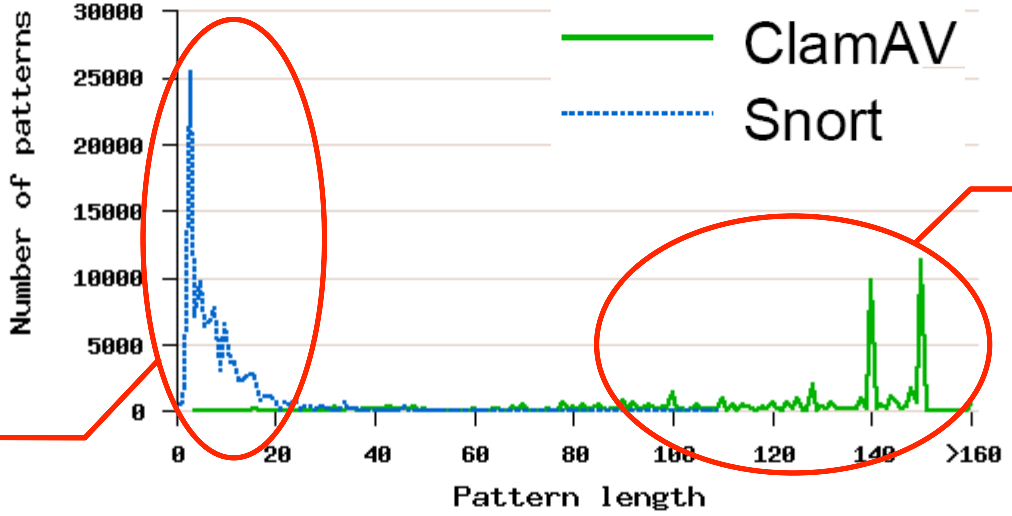


> 90%

> 80%

Anti-Virus Databases

- Memory requirements



~14 GB

~0.8 GB

Opportunity: Prefix Filtering

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

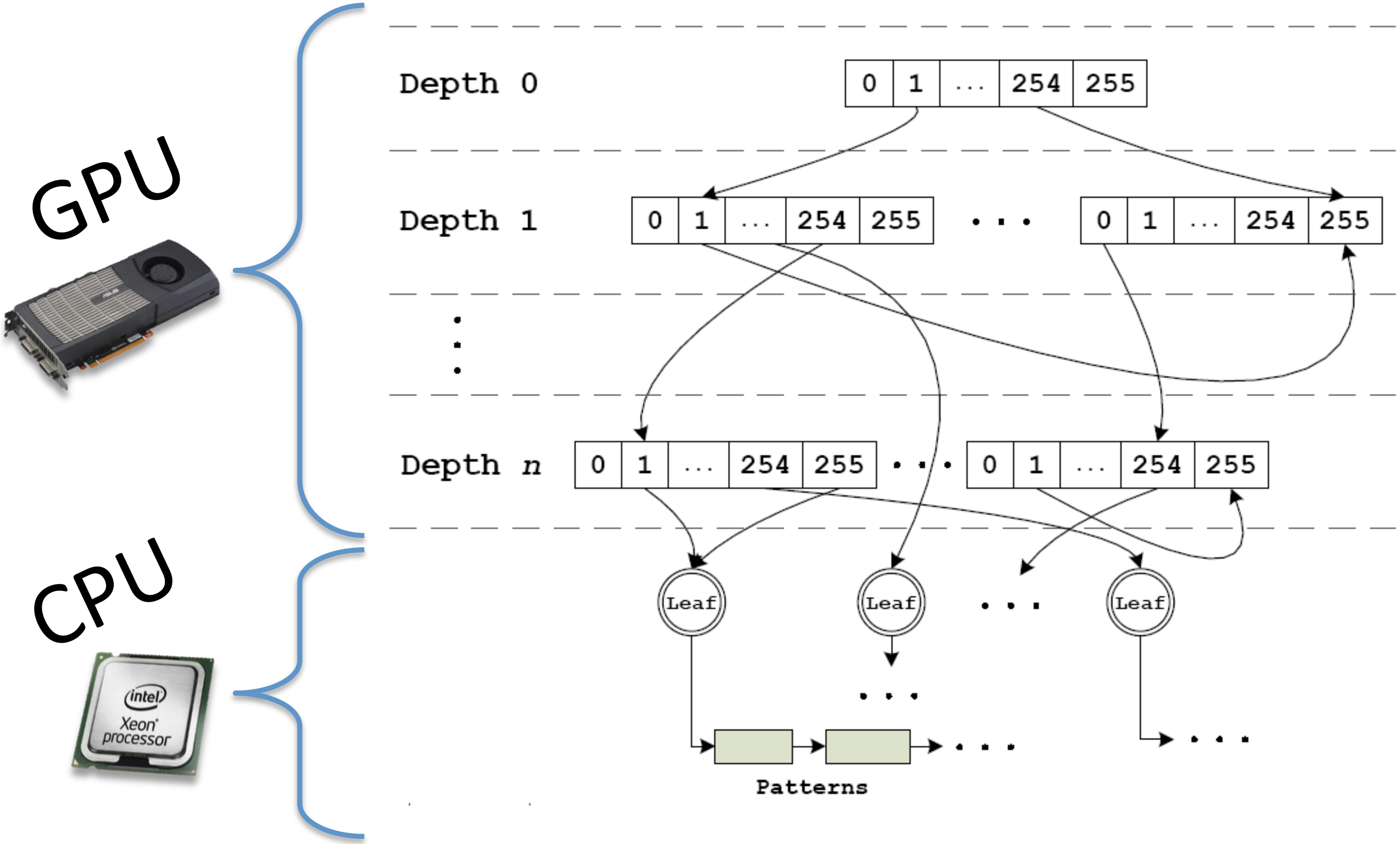
Worm.SQL.Slammer.A:0:*:

`4e65742d576f726d2e57696e33322e536c616d6d65725554`

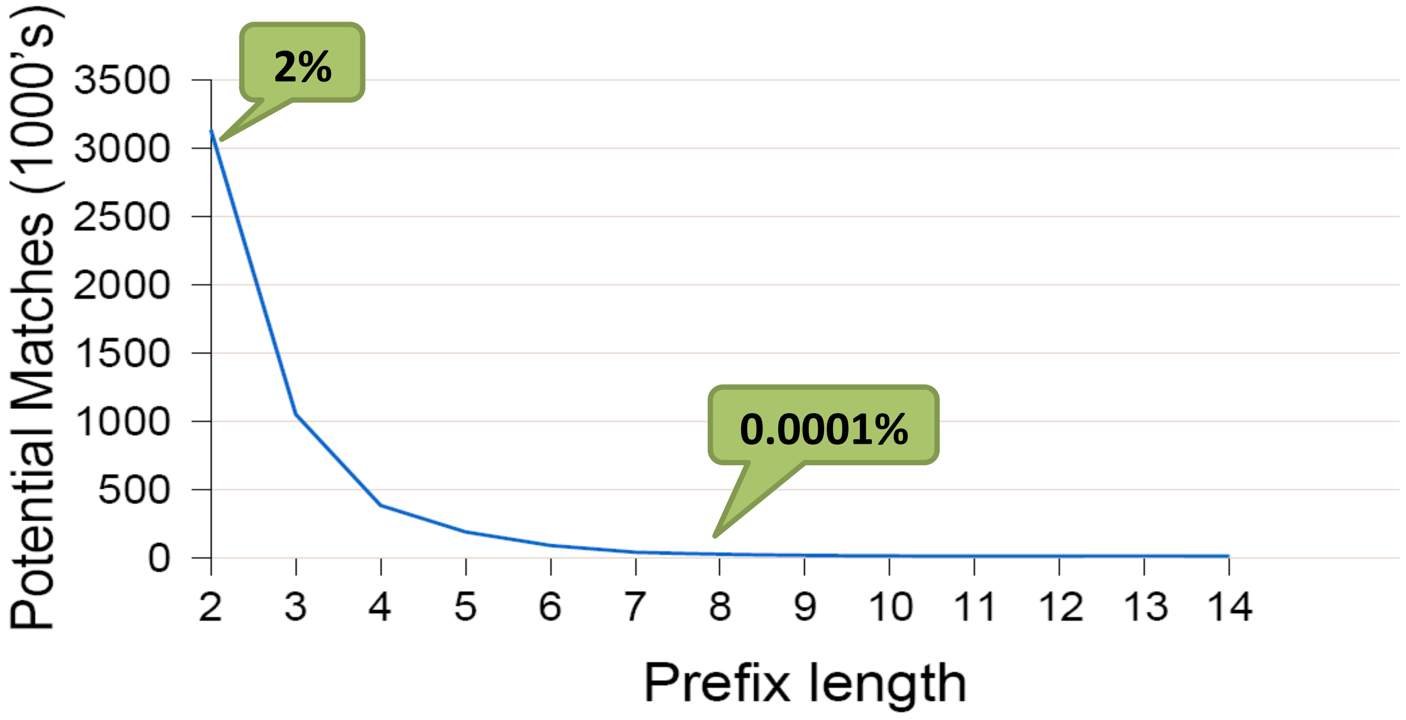
- Compile **all** *n*-bytes sub-signatures into **a single** *Scanning Trie*
- The Scanning Trie can quickly filter clean data segments in linear time.

Scanning Trie

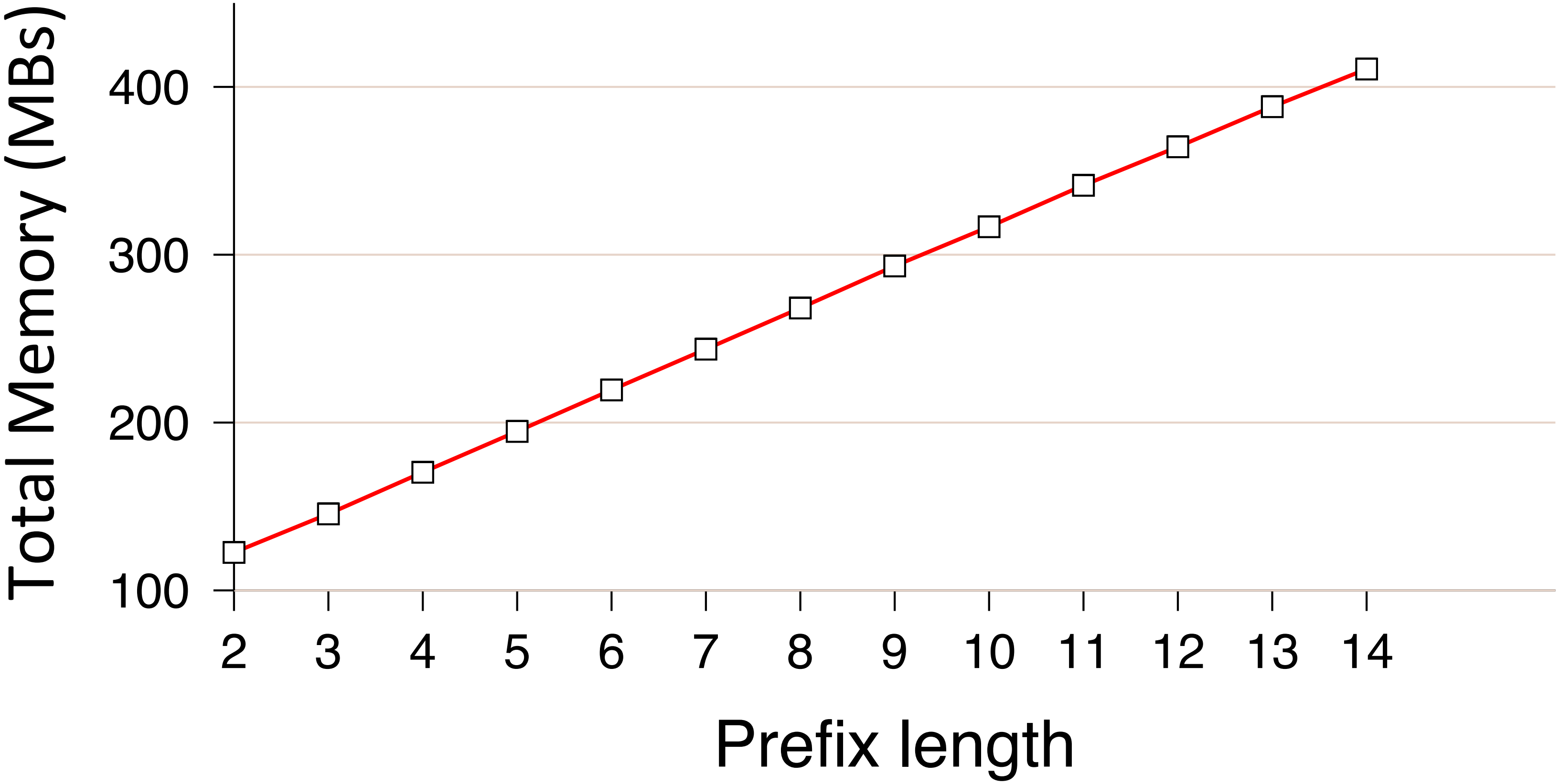
- Variable trie height



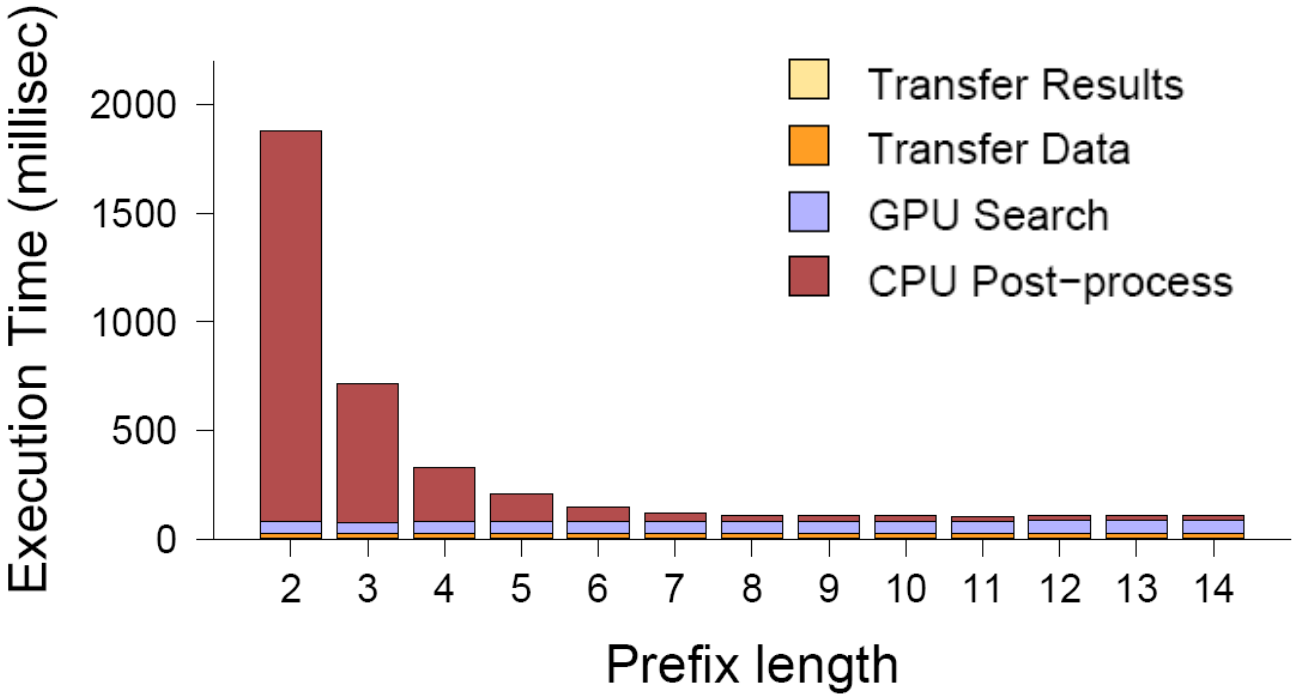
Longer prefix = Fewer matches



Longer prefix = More memory

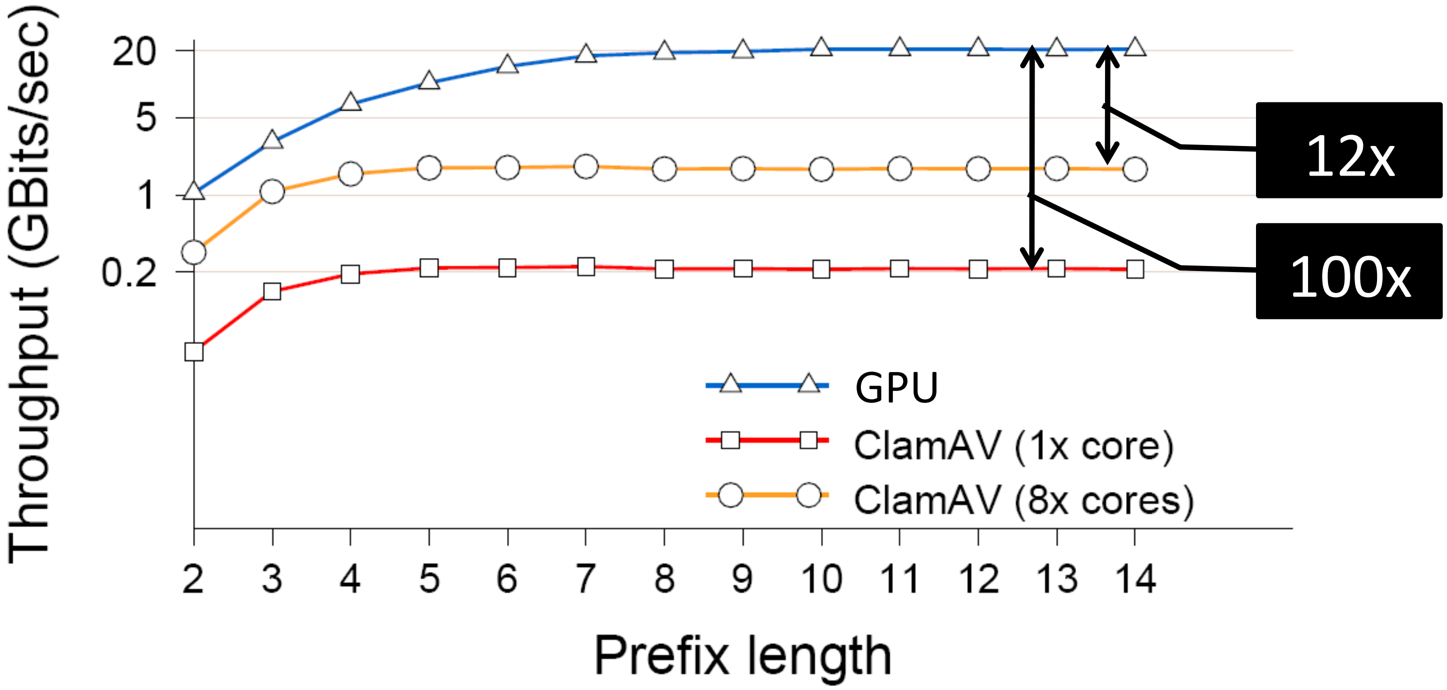


Execution Time Breakdown



- 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!**

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

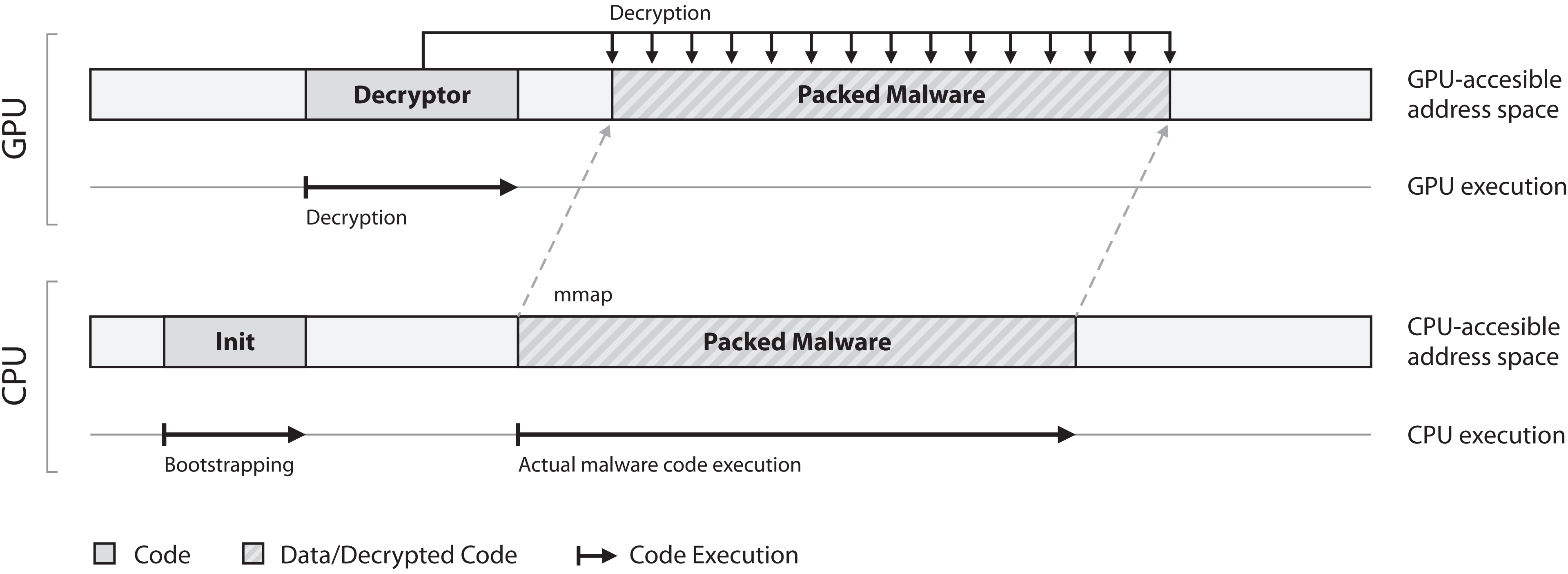
- Malware continually seek new methods for hiding their malicious activity, …
  - Packing
  - Polymorphism
- … as well as, hinder reverse engineering and code analysis
  - Code obfuscation
  - Anti-debugging tricks
- Is it possible for a malware to exploit the rich functionality of modern GPUs?

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

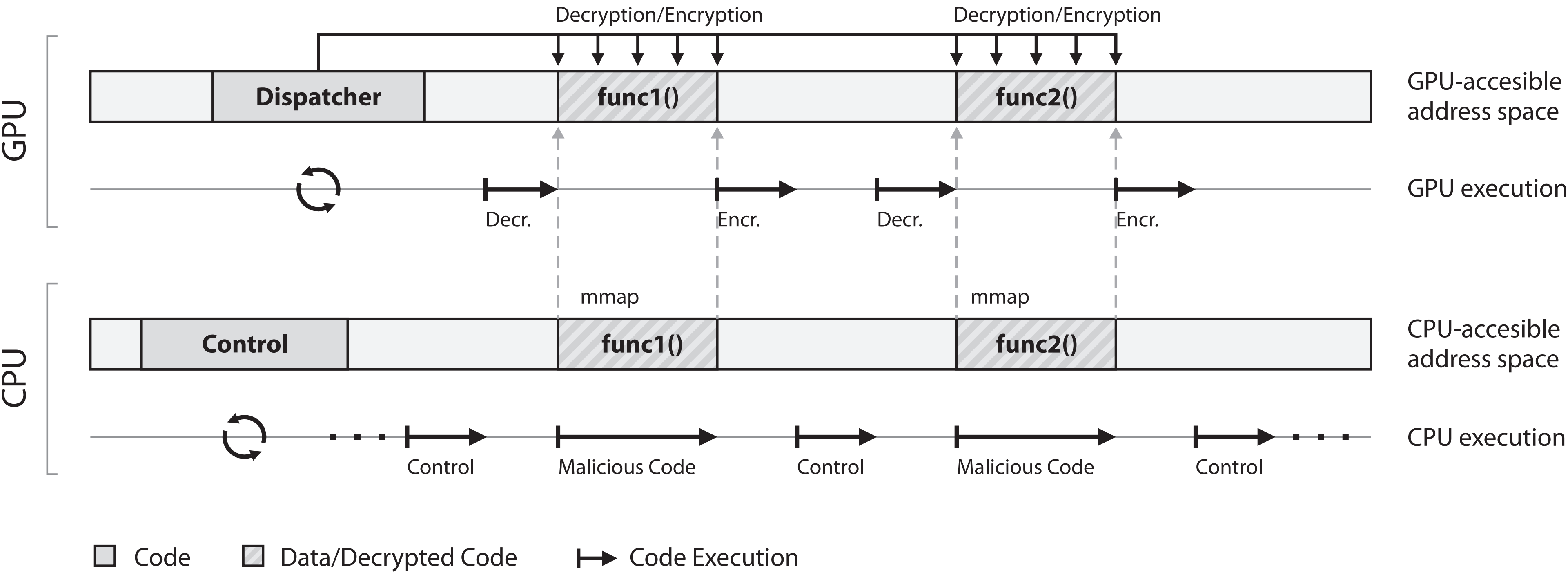
Self-unpacking GPU-malware



Self-unpacking: Strengths

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

Runtime-polymorphic GPU-malware



Runtime-polymorphism: Strengths

- GPU can use different encryption key every time
  - Random-generated
- Each encryption key is stored in device memory
  - Not accessible from CPU

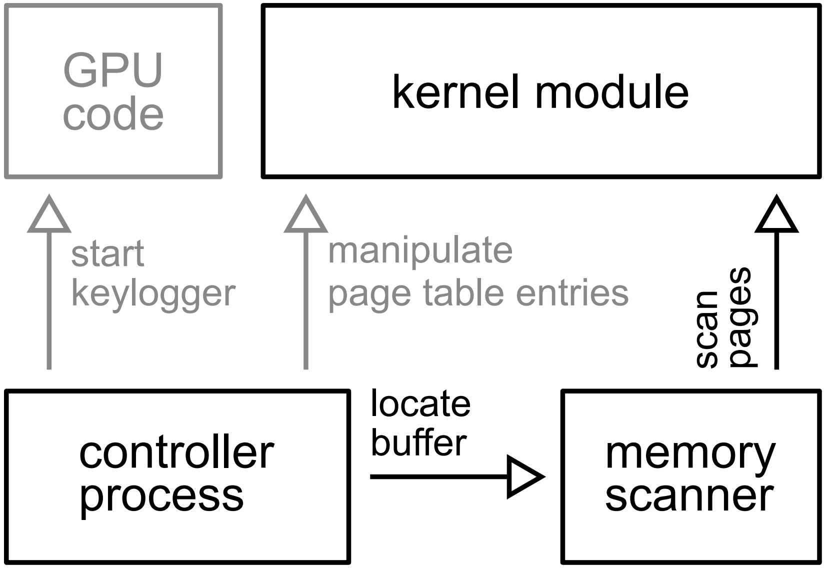
GPU-keylogger

- 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

Implementation

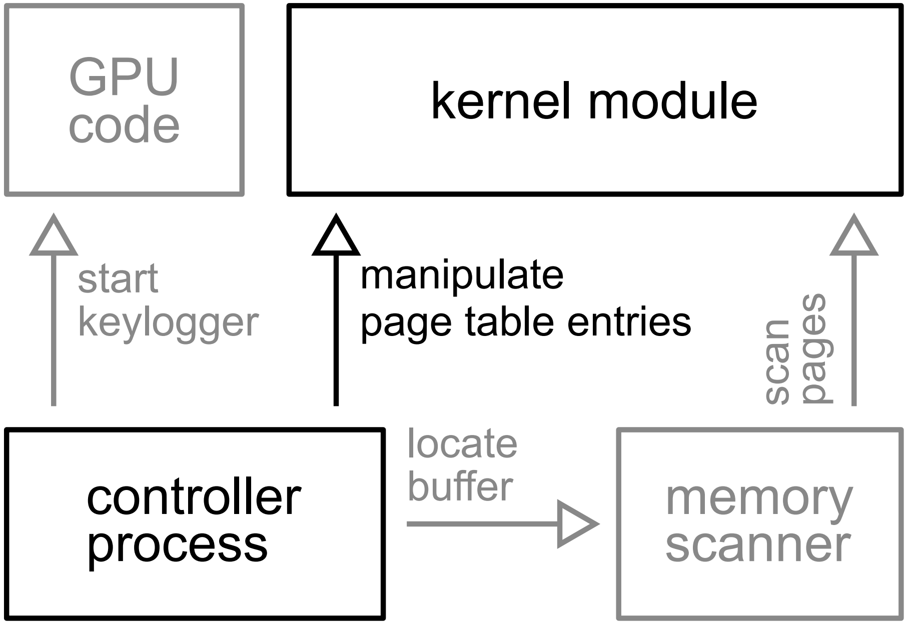
Step 1: **Locate the keyboard buffer**

- *Keyboard buffer dynamically changes address after system rebooting or after unplugging and plugging back in the device*



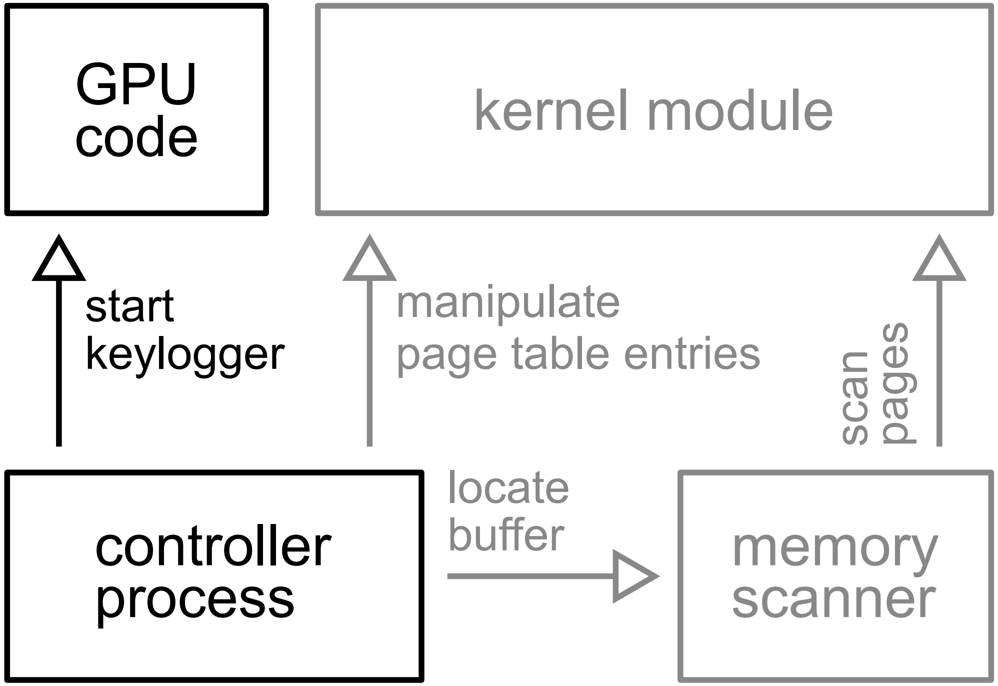
Implementation

Step 2: **Configure the GPU to constantly monitor buffer contents for changes**



Implementation

Step 3: **Start GPU process & Capture keystrokes**



Possible Defenses

- Monitoring GPU access patterns
  - Multiple/repeated DMAs from the GPU to system RAM
- Monitoring GPU usage
  - Unexpected increased GPU usage

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 kernel injection needed or data structure manipulation, in order to hide

Summary

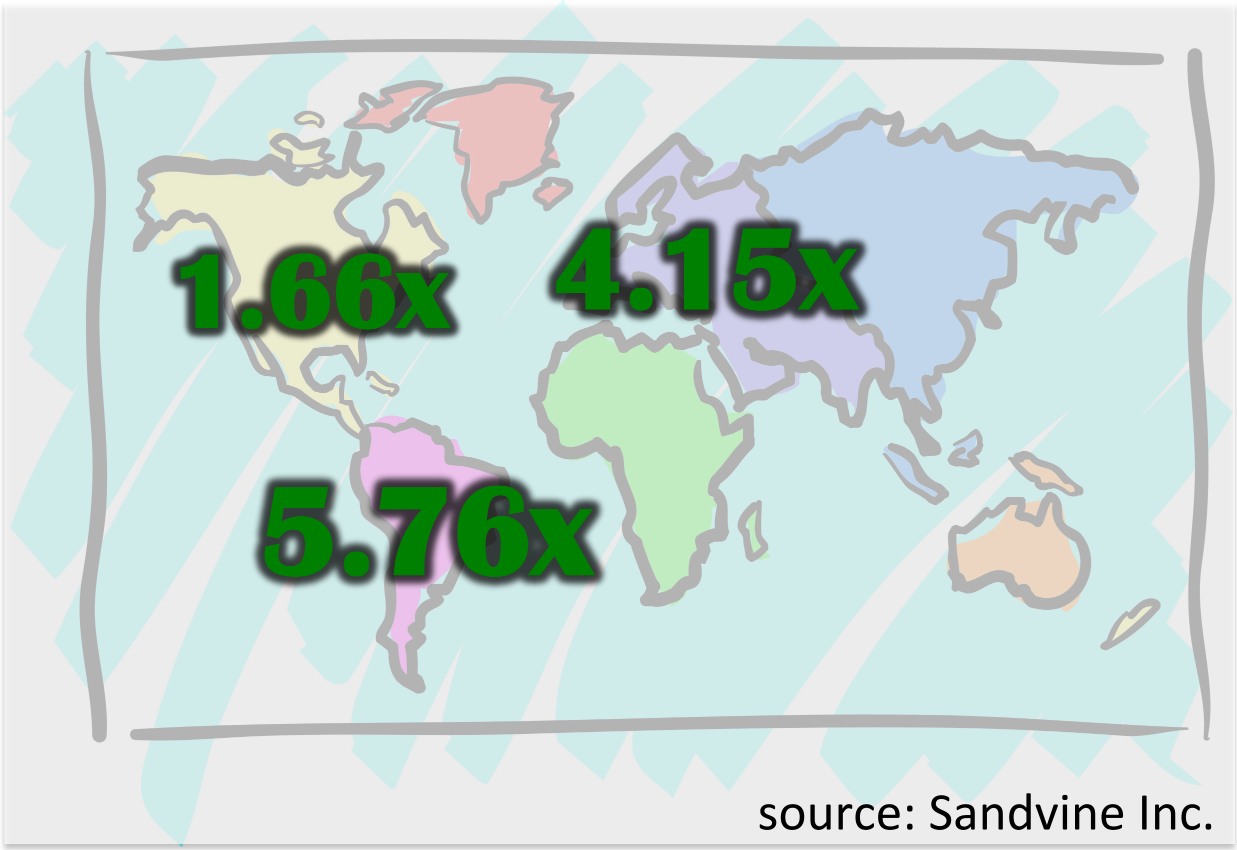
- GPUs offer new ways for robust and stealthy malware
  - We demonstrated how a malware can increase its robustness against detection using the GPU
    - Unpacking
    - Run-time polymorphism
  - Presented a fully functional and stealthy GPU-based keylogger
    - Low CPU and GPU usage
    - No device hooking
    - No traces left after exploitation
    - User mode application; No kernel injection needed

- 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

Last years increase of SSL traffic



source: Sandvine Inc.

“We should encrypt the entire internet”

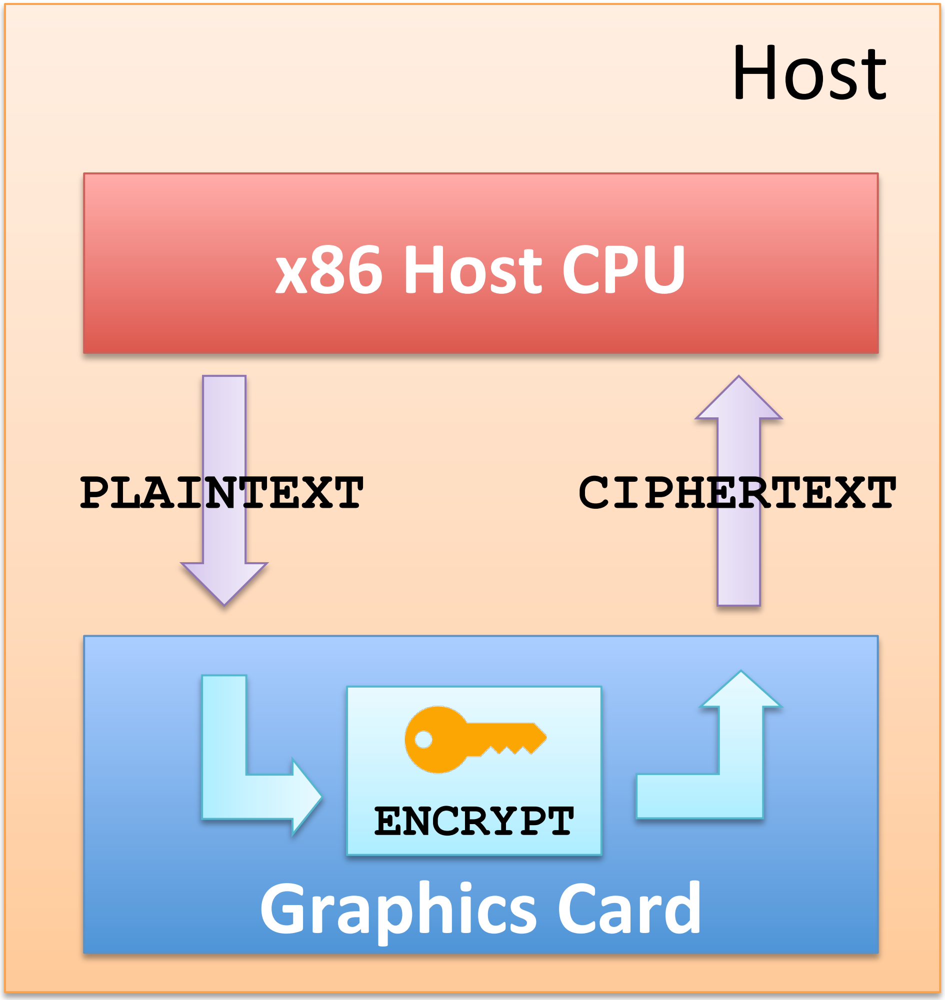
-- Matt Cutts, Head of Google's Webspam team

Motivation

- Secure Sockets Layer (SSL) is a de-facto standard for secure communication
  - Authentication, confidentiality, integrity

- Cryptographic keys may remain unencrypted in CPU Registers, RAM, HDD, etc.
  - Memory attacks
  - DMA/Firewire attacks
  - Heartbleed attack
  - Cold-boot attacks

PixelVault Overview



- Runs encryption securely outside CPU/RAM
- Secret keys and states never observed from host
- Instead, only GPU's non-addressable memory is used as storage

PixelVault Features

- Prevent key leakages
  - Even when the base system is fully compromised
- Requires just a commodity GPU
  - No OS kernel modifications or recompilation
- Provides strong security guarantees
  - Even against local root attackers

Limitations

- Require trusted bootstrap
- Dedicated GPU execution
- Misusing PixelVault for encrypting/decrypting messages
- Denial-of-Service attacks
- Side-channel attacks

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 ciphers and application domains
  - Apply to mobile and embedded devices
  - Utilize integrated CPU-GPU designs

- Credits to:
  - Giorgos Vassiliadis, Lazaros Koromilas, Michalis Polychronakis, Spyros Antonatos, Vagelis Ladakis, Elias Athanasopoulos, Evangelos Markatos