GPU Taint Tracking

Ari B. Hayes
Rutgers University

Lingda Li
Brookhaven National Lab

Mohammad Hedayati
University of Rochester

Jiahuan He
Rutgers University

Eddy Z. Zhang
Rutgers University

Kai Shen
Google

Presented by Eddy Z. Zhang
Rutgers University

July 12, 2017
1 Vulnerability of GPUs
Sensitive Data on the GPU

- Many GPU applications use sensitive data:
  - Machine learning, data encryption, computer vision.
Sensitive Data on the GPU

- Many GPU applications use sensitive data:
  - Machine learning, data encryption, computer vision.

Face Recognition
Input

Face Recognition
Leaked Features
Memory Protection

- Virtual Memory
  - Address Space Layout Randomization
  - Process Isolation
  - Page Protection
- Bounds Checking
- Memory Erasure

None of these are fully available on the GPU!
Memory Protection

• Without address space layout randomization, an attacker can predict where GPU data is stored. [Patterson, ISU thesis 2013]
• Without process isolation, an attacker can peek into another GPU process, steal encryption keys. [Pietro+, TECS 2016]
• Without page protection and bounds checking, an attacker can force a GPU program to write to non-permissive memory regions. [Vasiliadis+, CCS 2014]
• Without a reliable way to control or erase GPU thread-private memories, a user cannot keep their data contained. [Pietro+, TECS 2016]
GPU Memory

- **SM**: Registers, Shared Mem, L1 Cache
- **GPU**: L2 Cache
- **RAM**: Local Memory, Global Memory
- **CPU Memory**
Global memory

- Easily accessible to an attacker.
Local Memory

- Used for spilled registers; inaccessible to programmer
- Accessible by attacker through global memory
Shared Memory & L1 Cache

- Shared mem is accessible to attacker after function ends
- On some GPUs, L1 cache can leak into shared memory
Register File

- Designed to be inaccessible to programmer.
- Accessible to attackers after GPU function finishes.
Dynamic Taint Analysis

- Common technique for monitoring sensitive data
- Marks (taints) sensitive data and tracks taint at runtime
- Has extensive CPU work with various implementations:
  - Compile-time instrumentation [Lin+, ICC 2010]
  - Dynamic instrumentation [Kemerlis+, VEE 2012]
  - Emulation [Bosman+, RAID 2011]
  - Virtual machine [Enck+, TOCS 2014]
- Not previously attempted for GPU programs
Challenges of GPU Taint Tracking

- Must track several memory types
- Dynamic instrumentation infeasible
  - Lack of support from OS or driver;
  - Cannot intercept/modify instructions on the fly.
- Emulation is unappealing
  - Up to 1000x slowdown [Farooqui+, GPGPU 2011]
- Virtual machines are unhelpful
  - Cannot monitor data in GPU
Our Contributions

- First GPU dynamic taint tracking system.
  - Compile-time binary instrumentation
  - Dynamic tracking
  - GPU-specific optimizations to minimize overhead.
  - Filter out unnecessary tracking instructions
  - Improves tracking performance by 5 to 20 times
Taint Tracking
Taint Tracking

- Maintains taint map; **one taint bit** for each memory location.
- Monitors instructions & operands, propagating taint values.

Original code

```c
void foo() {
    b = a;
    d = b + c;
}
```

Taintedness propagation

```c
void foo_taint_tracking() {
    taint(b) = taint(a);
    taint(d) = taint(b) || taint(c);
}
```
Our Taint Tracking System

Binary Analysis

Two Pass Analysis
  Forward Pass
  Backward Pass

Binary Instrumentation

Tracking Filter
Analysis
Our Taint Tracking System

- Binary Analysis
  - Basic Blocks & CFG
  - Forward Pass
  - Backward Pass

- Binary Instrumentation

- Tracking Filter

GPU Program
GPU Behavior

- We observe that not everything needs to be tracked.
- Some GPU data is untaintable or cannot spread taint.
  - Thread ID
  - Grid Size
  - Constant memory
  - Loop Iterators
  - Immediate values
- These operands and instructions can be identified by analyzing the basic blocks and control flow graph.
Our Taint Tracking System

GPU Program -> Binary Analysis -> Binary Instrumentation

Basic Blocks & CFG

Two Pass Analysis
- Forward Pass
- Backward Pass

Tracking Filter

Taintability & Reachability
Two Pass Analysis

- Backward pass
  - Identifies & marks taint sinks
  - Propagates markings backward
- Forward pass
  - Identify & marks potential taint sources
  - Propagates markings forward
- Two-pass analysis
  - Combine markings from both passes
Backward Pass

Block4:

R0 = R1 + R2;  reachable = \{R1, R2, R3\}
R1 = R1 + R3;  reachable = \{R1, R2, R3\}
R0 = [R1];     reachable = \{R1, R2, R3\}
R2 = R3 * R2;  reachable = \{R1, R2, R3\}
[R1] = R2;     reachable = \{R1, R2, R3\}
R0 = R1 * R3;  reachable = \{R1, R3\}
BRA block5;    reachable = \{R0, R3\}
Backward Pass

Block 4:

\[
R_0 = R_1 + R_2; \\
R_1 = R_1 + R_3; \\
R_0 = [R_1]; \\
R_2 = R_3 \times R_2; \\
[R_1] = R_2; \\
R_0 = R_1 \times R_3; \\
\text{BRA block 5}\]

reachables:

- \{R_1, R_2, R_3\}
- \{R_1, R_2, R_3\}
- \{R_1, R_2, R_3\}
- \{R_1, R_2, R_3\}
- \{R_1, R_2, R_3\}
- \{R_1, R_3\}
- \{R_0, R_3\}
Backward Pass

Block 4:

R0 = R1 + R2;
R1 = R1 + R3;
R0 = [R1];
R2 = R3 * R2;
[R1] = R2;
R0 = R1 * R3;

reachability = {R1, R2, R3}
reachability = {R1, R2, R3}
reachability = {R1, R2, R3}
reachability = {R1, R2, R3}
reachability = {R1, R2, R3}
reachability = {R1, R3}
reachability = {R0, R3}
Backward Pass

Block4:

R0 = R1 + R2;  
R1 = R1 + R3;  
R0 = [R1];  
R2 = R3 * R2;  
[R1] = R2;  
R0 = R1 * R3;  
BRA block5;

reachable = \{R1, R2, R3\}  
reachable = \{R1, R2, R3\}  
reachable = \{R1, R2, R3\}  
reachable = \{R1, R2, R3\}  
reachable = \{R1, R2, R3\}  
reachable = \{R1, R3\}  
reachable = \{R0, R3\}
Forward Pass

Block4:

\[
\begin{align*}
R0 &= R1 + R2; \\
R1 &= R1 + R3; \\
R0 &= \{R1\}; \\
R2 &= R3 \times R2; \\
\{R1\} &= R2; \\
R0 &= R1 \times R3; \\
\text{BRA block5;}
\end{align*}
\]

\[
\text{taintable} = \{R1\}
\]

\[
\text{taintable} = \{R0, R1\}
\]

\[
\text{taintable} = \{R0, R1\}
\]

\[
\text{taintable} = \{R0, R1\}
\]

\[
\text{taintable} = \{R0, R1\}
\]

\[
\text{taintable} = \{R0, R1\}
\]
Forward Pass

Block4:

\[
\begin{align*}
R0 &= R1 + R2; \\
R1 &= R1 + R3; \\
R0 &= [R1]; \\
R2 &= R3 \times R2; \\
[R1] &= R2; \\
R0 &= R1 \times R3; \\
\text{BRA block5;} \\
taintable &= \{\text{R1}\} \\
taintable &= \{R0, R1\} \\
taintable &= \{R0, R1\} \\
taintable &= \{R0, R1\} \\
taintable &= \{R0, R1\} \\
taintable &= \{R0, R1\} \\
taintable &= \{R0, R1\} \\
taintable &= \{R0, R1\}
\end{align*}
\]
Forward Pass

Block 4:

\[
\begin{align*}
R0 &= R1 + R2; \\
R1 &= R1 + R3; \\
R0 &= [R1]; \\
R2 &= R3 \times R2; \\
[R1] &= R2; \\
R0 &= R1 \times R3; \\
\text{BRA block 5};
\end{align*}
\]

taintable = \{R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}
Block4:

\[ R0 = R1 + R2; \]
\[ R1 = R1 + R3; \]
\[ R0 = [R1]; \]
\[ R2 = R3 \times R2; \]
\[ [R1] = R2; \]
\[ R0 = R1 \times R3; \]

BRA block5;

taintable = \{R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}

taintable = \{R0, R1\}
4 Instrumentation
Our Taint Tracking System

GPU Program → Binary Analysis → Binary Instrumentation → New Assembly

Basic Blocks & CFG → Two Pass Analysis

Forward Pass → Tracking Filter → Backward Pass

Taintability & Reachability
Naive Tracking Code

Block4:

\[ R_0 = R_1 + R_2; \]

\[ R_1 = R_1 + R_3; \]

\[ R_0 = [R_1]; \]

\[ R_2 = R_3 \times R_2; \]

\[ [R_1] = R_2; \]

\[ R_0 = R_1 \times R_3; \]

BRA block5;
Naive Tracking Code

Block4:

\[
R_0 = R_1 + R_2;
\]

\[
t(R_0) = t(R_1) | t(R_2)
\]

\[
R_1 = R_1 + R_3;
\]

\[
t(R_1) = t(R_1) | t(R_3)
\]

\[
R_0 = [R_1];
\]

\[
t(R_0) = t([R_1])
\]

\[
R_2 = R_3 * R_2;
\]

\[
t(R_2) = t(R_3) | t(R_2)
\]

\[
[R_1] = R_2;
\]

\[
t([R_1]) = t(R_1) | t(R_2)
\]

\[
R_0 = R_1 * R_3;
\]

\[
t(R_0) = t(R_1) | t(R_3)
\]

BRA block5;
Naive Tracking Code

Block4:

\[
\begin{align*}
R_0 &= R_1 + R_2; \\
t(R_0) &= t(R_1) \mid t(R_2) \\
R_1 &= R_1 + R_3; \\
t(R_1) &= t(R_1) \mid t(R_3) \\
R_0 &= [R_1]; \\
t(R_0) &= t([R_1]) \\
R_2 &= R_3 \ast R_2; \\
t(R_2) &= t(R_3) \mid t(R_2) \\
[R_1] &= R_2; \\
t([R_1]) &= t(R_1) \mid t(R_2) \\
R_0 &= R_1 \ast R_3; \\
t(R_0) &= t(R_1) \mid t(R_3)
\end{align*}
\]

BRA block 5;
Naive Tracking Code

Block4:

\[ R_0 = R_1 + R_2; \]
\[ t(R_0) = t(R_1) \mid t(R_2) \]
\[ R_1 = R_1 + R_3; \]
\[ t(R_1) = t(R_1) \mid t(R_3) \]
\[ R_0 = [R_1]; \]
\[ t(R_0) = t([R_1]) \]
\[ R_2 = R_3 \cdot R_2; \]
\[ t(R_2) = t(R_3) \mid t(R_2) \]
\[ [R_1] = R_2; \]
\[ t([R_1]) = t(R_1) \mid t(R_2) \]
\[ R_0 = R_1 \cdot R_3; \]
\[ t(R_0) = t(R_1) \mid t(R_3) \]
BRA block5;
Naive Tracking Code

Block4:

\[ R_0 = R_1 + R_2; \]
\[ t(R_0) = t(R_1) | t(R_2) \]
\[ R_1 = R_1 + R_3; \]
\[ t(R_1) = t(R_1) | t(R_3) \]
\[ R_0 = [R_1]; \]
\[ t(R_0) = t([R_1]) \]
\[ R_2 = R_3 * R_2; \]
\[ t(R_2) = t(R_3) | t(R_2) \]
\[ [R_1] = R_2; \]
\[ t([R_1]) = t(R_1) | t(R_2) \]
\[ R_0 = R_1 * R_3; \]
\[ t(R_0) = t(R_1) | t(R_3) \]
BRA block5;
Naive Tracking Code

Block 4:

```plaintext
R0 = R1 + R2;
t(R0) = t(R1) | t(R2)
R1 = R1 + R3;
t(R1) = t(R1) | t(R3)
R0 = [R1];
t(R0) = t([R1])
R2 = R3 * R2;
t(R2) = t(R3) | t(R2)
[R1] = R2;
t([R1]) = t(R1) | t(R2)
R0 = R1 * R3;
t(R0) = t(R1) | t(R3)
BRA block 5;
```
Filtered Tracking Code

Block4:

\[ R_0 = R_1 + R_2; \]
\[ t(R_0) = t(R_1) | t(R_2) \]
\[ R_1 = R_1 + R_3; \]
\[ t(R_1) = t(R_1) | t(R_3) \]
\[ R_0 = [R_1]; \]
\[ t(R_0) = t([R_1]) \]
\[ R_2 = R_3 * R_2; \]
\[ t(R_2) = t(R_3) | t(R_2) \]
\[ [R_1] = R_2; \]
\[ t([R_1]) = t(R_1) | t(R_2) \]
\[ R_0 = R_1 * R_3; \]
\[ t(R_0) = t(R_1) | t(R_3) \]

BRA block 5;
Filtered Tracking Code

Block4:

\[ R_0 = R_1 + R_2; \]
\[ t(R_0) = t(R_1) \mid t(R_2) \]
\[ R_1 = R_1 + R_3; \]
\[ t(R_1) = t(R_1) \mid t(R_3) \]
\[ R_0 = [R_1]; \]
\[ t(R_0) = t([R_1]) \]
\[ R_2 = R_3 \ast R_2; \]
\[ t(R_2) = t(R_3) \mid t(R_2) \]
\[ [R_1] = R_2; \]
\[ t([R_1]) = t(R_1) \mid t(R_2) \]
\[ R_0 = R_1 \ast R_3; \]
\[ t(R_0) = t(R_1) \mid t(R_3) \]

BRA block 5;
Filtered Tracking Code

Block 4:

\[ R_0 = R_1 + R_2; \]
\[ R_1 = R_1 + R_3; \]
\[ R_0 = [R_1]; \]
\[ R_2 = R_3 \times R_2; \]
\[ [R_1] = R_2; \]
\[ \tau([R_1]) = \tau(R_1) \]
\[ R_0 = R_1 \times R_3; \]
\[ \tau(R_0) = \tau(R_1) \]
BRA block 5;
Our Taint Tracking System

- Binary Analysis
  - Basic Blocks & CFG
- Two Pass Analysis
  - Forward Pass
  - Backward Pass
    - Taintability & Reachability
- Tracking Filter
  - New Assembly

GPU Program

Binary Instrumentation
  - Basic Blocks
Efficient Taint Map

- Taint map is typically kept completely in RAM.
- Off-chip memory is very slow on the GPU.
- Better to keep part of the taint map in on-chip memory.
  - We keep register taintedness in the register file.
  - Registers are 32 bits, so every 32 tracked registers adds only one register of overhead.
5 Evaluation
Methodology

- Binary code is converted to assembly with `cuobjdump`.
- Our compiler **Orion** analyzes assembly and adds taint tracking (and erasure) code to assembly.
- New assembly is converted into binary based on `asfermi & MaxAs`.
- Taint map allocation can be done indirectly through CPU, using `LD_PRELOAD` to intercept `cudaMalloc` calls.
- Evaluated on NVIDIA **GTX 745**, compute capability 5.0.
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Domain</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>im2col</td>
<td>Machine Learning</td>
<td>Caffe</td>
</tr>
<tr>
<td>ReLUForward</td>
<td>Machine Learning</td>
<td>Caffe</td>
</tr>
<tr>
<td>MaxPoolForward</td>
<td>Machine Learning</td>
<td>Caffe</td>
</tr>
<tr>
<td>FDTD3d</td>
<td>Numerical Analysis</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>Financial Analysis</td>
<td>CUDA SDK</td>
</tr>
<tr>
<td>SSLShader</td>
<td>Cryptography</td>
<td>[Jang+, NSDI 2011]</td>
</tr>
<tr>
<td>needle</td>
<td>Bioinformatics</td>
<td>Rodinia</td>
</tr>
</tbody>
</table>
Results - Runtime with Tracking

Normalized execution time

- naive
- reg-in-reg
- forward-filter
- backward-filter
- two-pass-filter
- fully optimized

Tasks:
- im2col
- ReLUForward
- MaxPoolForward
- FDTD3d
- BlackScholes
- SSLShader
- nw
Results - Runtime with Tracking

Geomean is 24.41X
Results - Runtime with Tracking

Geomean is 5.19X
Results - Runtime with Tracking

Geomean is 17.84X
Results - Runtime with Tracking

Geomean is 7.38X
Results - Runtime with Tracking

Geomean is 2.80X
Results - Code Size with Tracking

Normalized code size

naive
reg-in-reg
forward-filter
backward-filter
two-pass-filter
fully optimized

im2col
ReLUForward
MaxPoolForward
FDTD3d
BlackScholes
SSLShader
nw
Memory Erasure

- After adding tracking code, we can also add erasure code.
  - On-chip memory can only be reliably erased via binary instrumentation.
- We have GPU threads clear their own registers and shared memory, as well as thread-private data in local memory.
- The final taint map identifies global memory with sensitive data, so that it can be erased.
## On-Chip & Thread-Private Erasure

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Memories</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>im2col</td>
<td>Reg</td>
<td>0.26%</td>
</tr>
<tr>
<td>ReLUForward</td>
<td>Reg</td>
<td>0.33%</td>
</tr>
<tr>
<td>MaxPoolForward</td>
<td>Reg</td>
<td>0.59%</td>
</tr>
<tr>
<td>FDTD3d</td>
<td>Reg, Shared</td>
<td>5.10%</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>Reg</td>
<td>0.40%</td>
</tr>
<tr>
<td>SSLShader</td>
<td>Reg, Local</td>
<td>0.41%</td>
</tr>
<tr>
<td>needle</td>
<td>Reg, Shared</td>
<td>13.05%</td>
</tr>
</tbody>
</table>

Naive erasure is up to nine times slower!
Conclusion
Conclusion

- We present the first GPU dynamic taint tracking system.
  - Two pass filtering eliminates tracking code.
  - GPU-specific optimizations to minimize overhead.
  - Clears memory the programmer cannot.
  - Improves tracking performance by 5X to 20X.
Questions?