

GPUs Assisted Malware Detection for Mobile Devices

Jared Carlson

CONTENTS

I	Introduction	4
II	Overview of Algorithms	4
II-A	Signature Checks	4
II-B	Memory Analysis	4
II-C	Dynamic Disassembly	4
II-D	Encryption	5
III	Signature Checking	5
III-A	General Techniques	7
IV	Memory Analysis	7
IV-A	Texture Atlas	9
IV-B	Heap Visualization	9
V	Dynamic Disassembly	11
V-A	Why This Works Even Better for Mobile	12
V-B	Implementation	12
V-C	In Practice	17
VI	Encryption	18
VII	Environment Considerations	18
VII-A	Restrictive Environments	18
VII-B	Unrestrictive Environments	18
VIII	Device and Platform Information	19
IX	Optimizations and Other Tricks	20
IX-A	Generalities	20
IX-B	Our Code	20
X	Performance	20
XI	Hardware Integration	21
XI-A	Shader Trends	23

XII	Strategies	23
XIII	GPU Assisted Malware	23
XIV	Summary	24
XV	Landscape and Likely Developments	24
XV-A	CUDA	24
XV-B	RenderScript	24
XV-C	OpenCL	24
XVI	Future Research	25
XVII	Conclusion	25
XVIII	Acknowledgment	25
	References	25
	Appendix A: Example Code	26
	Appendix B: Project Templates	26
B-A	iOS	26
B-B	Android	31

LIST OF FIGURES

1	Looking at an Attack Signature	5
2	We sweep a masking texture over our payload or other memory region, performing texture operations to find our signature	8
3	Sample Screenshot of Memory as a Texture	10
4	Performance Characteristics	21
5	Flow diagram of online JIT	22
6	Relationships between LLVM and Renderscript Core Libs	22

LIST OF TABLES

I	Device Characteristics	19
---	----------------------------------	----

LISTINGS

1	Sample ObjC code to Generate Texture From File	6
2	Signature Check on Star Exploit	6
3	Simple Memory Object to Manage Dynamic Memory	7
4	Watching the Heap	9
5	GL code to read in our GPU results	10
6	Simple GDB Capture of GPU results	10
7	Sample Shader that can test for a simple byte signature	11
8	Simple Shader Identifies Signature as Red	11
9	Script to grab ARM opcodes from Executables	12
10	Sample Disassembly	14
11	Python Script to Exercise Disassembler	14
12	Main for creating disassembly tables	15

13	Simple Shader for Square Matrix Multiplication	17
14	Code Fragment from rsdProgram.cpp within frameworks/base/libs/rs/driver	22
15	Render Method for iOS project	26
16	Sample disassembly using the Accelerate framework	27
17	Simple NDK usage for rendering	31
18	To Move into Vectorized C++ for the NEON Processor	36
19	"Cuda Example Program"	36
20	"OpenCL Example"	37
21	"OpenCL via Python"	40

Abstract

This paper, produced as part of a DARPA CFT grant, discusses how to use the GPU and other vectorized processors for cyber tasks, starting with defensive measures. We address mobile devices for their novelties, tight integration with CPUs and the difficulty of coming up with OTA updates or other vendor concerns.

The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This is in accordance with DoDI 5230.29, January 8, 2009

Index Terms

Security, ARM, GPU, iOS, Android, OpenGL, Vector, SIMD, DARPA

I. INTRODUCTION

GPUs have seen rapid growth over the past decade, with rapid improvement in their integration into mobile devices. This has been powered by gaming sales, and other user interface desires from the consumer market. But within the hacking world we can see how capable these chips are and how they've recently been used for aiding malware proliferation [12]. What's even more interesting in mobile devices is the integration within a System-on-Chip (SoC) or Package-on-Package (PoP) chipset.

Mobile devices, with a diverse set of communication channels have RF for bluetooth, cellular and other bands such as NFC for certain configurations. Current top-of-the-line system chips will typically be comprised of an ARM CPU, GPU as well as a SIMD Accelerator, a NEON chip for example. Our goal was to see what can we do with these other chips could offer for malware defense schemes. By doing this, we are effectively increasing the defense surface, forcing an attacker to understand the system and not just the CPU. Naturally the first choice is the GPU but recent SoC's (System on a Chip) include a NEON chip, and SDK's and other tools are being developed to allow developers increasing access and capabilities to leverage the processors outside of the the CPU. Examples of these on iOS and Android are the Accelerate framework as well as the Renderscript toolset provided on Android.

II. OVERVIEW OF ALGORITHMS

The main challenge when dealing with the GPU, or the NEON processor for that matter, is that we need to focus on SIMD (Single Instruction Multiple Data) based algorithms and how to incorporate them into malware defense. This means that we want to explore some existing capabilities and see how they map to the GPU, as well as tasks that seem especially suited for GPU assistance. With this in mind, we explored the following schemes to see what could be done on the GPU or other SIMD processors.

A. Signature Checks

A staple of anti-virus software, signature checks are a basic capability that we want to explore using the GPU. We also want to establish that we can reuse this basic premise for other, related tasks such as pattern recognition.

B. Memory Analysis

More advanced attacks could alter in-memory objects, spray a payload into the heap or other tasks to exploit the program. We want to show how small modifications in code management can lend itself to allow GPU tracking and integrity checks.

C. Dynamic Disassembly

Being able to disassemble incoming instructions would allow for a variety of analysis tasks to consume the disassembled instruction sets and make decisions or alter existing parameters. Here we want to show that we can leverage vectorizable code and if necessary that the building blocks to handle this on the GPU are present today, and likely increasingly accessible tomorrow.

To test, on iOS for example we can use a method to create a texture from a payload.

Listing 1: Sample ObjC code to Generate Texture From File

```

1 // load a signature
2 - (GLuint) loadSignatureFromFile:(NSString*)file
3 {
4     NSString * pathToFile = [[NSBundle mainBundle] pathForResource:file ofType:@"bin"];
5     NSData * contentsOfFile = [[NSData alloc] initWithContentsOfFile:pathToFile];
6     NSUInteger numBytes = [contentsOfFile length];
7
8     NSUInteger pow = [self fitPowerOf2:(4 * numBytes / 3 )] * 2;
9
10    if ( pow > (dimension*dimension) )
11    {
12        NSLog(@"File contents doesn't fit inside texture...");
13        return 0;
14    }
15
16    unsigned char * data = (unsigned char*) calloc( sizeof(unsigned char) * dimension*←
17        dimension*4, 1 );
18    unsigned char * buffer = (unsigned char*) [contentsOfFile bytes];
19    NSUInteger index = 0;
20
21    for (NSUInteger y = 0; y < dimension; y++ ) {
22        for (NSUInteger x = 0; x < dimension; x++) {
23            int byteIndex = (dimension*4 * y) + x * 4;
24
25            if ( index > numBytes ) goto endloop;
26            memcpy(&data[byteIndex], &buffer[index], 3);
27            data[byteIndex+3] = 0xff;
28            index += 3;
29        }
30    }
31    endloop:
32
33    GLuint texName;
34    glGenTextures(1, &texName);
35    glBindTexture(GL_TEXTURE_2D, texName);
36    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
37    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
38    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
39    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, dimension, dimension, 0, GL_RGBA, ←
40        GL_UNSIGNED_BYTE, data);
41
42    return texName;
43 }

```

Using this method we simply create two textures and perform difference operations to see if we can find the signature. The shader is extremely simple, simply subtracting the signature texture from the payload.

If we watch the application, while within GDB, (and truncating some of @comex's payload) we can see the effect.

Listing 2: Signature Check on Star Exploit

```

1
2 // relevant part of frag shader
3 // offset is a uniform we control from the CPU, allowing us to "sweep"
4 gl_FragColor = texture2D( Texture, TexCoordOut ) - texture2D( Mask, TexCoordOut + offset←
5     );
6
7 // placing breakpoint immediately after:
8 glReadPixels(0, 0, dimension, dimension, GL_RGBA, GL_UNSIGNED_BYTE, bytes);
9 //
10 // offset (0,0)

```

```

10 (gdb) x/20 bytes
11 0x6f33000: 0xff000000 0xffac420f 0xff002047 0xff000000
12 0x6f33010: 0xff090000 0xff5b1a00 0xff000000 0xff1c002c
13 0x6f33020: 0xff761900 0xff080900 0xff000001 0xff250f00
14 0x6f33030: 0xff000e00 0xff0d0013 0xff432500 0xff00005e
15 0x6f33040: 0xff1d8c1e 0xffc23646 0xffc36bb4 0xffbdc2b5
16
17 // offset(x,0)
18 (gdb) x/20 bytes
19 0x6f33000: 0xff070503 0xff000000 0xff000000 0xff000000
20 0x6f33010: 0xff000000 0xff000000 0xff000000 0xff000000
21 0x6f33020: 0xff000000 0xff000000 0xff000000 0xff000000
22 0x6f33030: 0xff000000 0xff000000 0xff000000 0xff000000
23 0x6f33040: 0xff000000 0xff520000 0xffc36bb4 0xffbdc2b5

```

We can see that we quickly find a matching "black" region where the signature fragment matches the payload across those pixels. At this point the application could decide to dismiss the payload, labeling it as "suspicious" or run another malware detection scheme.

A. General Techniques

We can borrow graphical techniques to optimize our simple shader. Using techniques such as a texture atlas we can check for multiple signatures, not just the one we showed above, within a single texture fragment. In OpenGL we can even get periodic boundary (or repeating) boundary conditions making our texture atlas scheme more efficient.

Algorithm 1 Signature Checking Algorithm

```

1: texture ← payload-to-check
2: Create atlas of various signatures
3: second-texture ← atlas-of-signatures
4: draw-call initiates GPU calculation
5: for each texel do
6:   The FOR loop is implicit as shader takes care of this
7:   for each atlas region do
8:     Frag Operation with i-th atlas region
9:   end for
10: end for

```

An illustration serves best...

IV. MEMORY ANALYSIS

A common tactic for exploitation is a heap spray or some other data corruption. With access to dynamic memory we can investigate structures and other objects that may be compromised due to a bug, heap spray, and eventually lead to exploitation due to a NULL dereferencing or similar bug. This strategy is basically a generalization of a "guard", but we also mention that visualization and other runtime information could be gathered by using the GPU to analyze dynamic memory.

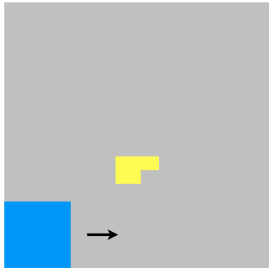
Let's take a quick look at an example of how we can set up our application to allow GPU integrity checking.

Listing 3: Simple Memory Object to Manage Dynamic Memory

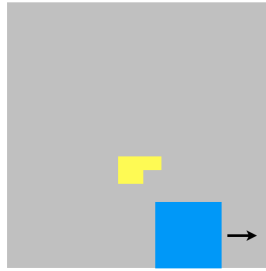
```

1
2 class MemoryObject
3 {

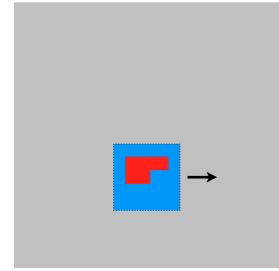
```



((a)) We start sweeping our masking texture along the byte sequence



((b)) The mask continues along...



((c)) The mask encounters the signature as texture operations reveal an indicator

Fig. 2: We sweep a masking texture over our payload or other memory region, performing texture operations to find our signature

```

4   unsigned char leading[8];
5   vector<string> objects;
6   // other objects
7   unsigned char trailing[8];
8
9
10  // static methods
11  static void Generator(unsigned char ptr[8]) {
12      static unsigned char start = 0;
13      const unsigned char interval = 0x02;
14      start += interval;
15      for (int i=0; i<8; i++)
16          ptr[i] = start;
17  }
18
19
20  public:
21
22      // constructor
23      MemoryObject(void) { Generator(leading); objects.push_back("Testing"); Generator(↔
          trailing); }
24
25      // destructor
26      ~MemoryObject(void){
27          memset(leading, '\0', 8);
28          memset(trailing, '\0', 8);
29      }
30
31
32  };

```

To build our understanding, we start with a look at dynamic memory in a simple example.

Listing 4: Watching the Heap

```

1
2 (gdb) list 295
3 290
4 291 // populate the heap..
5 292 - (void) populateHeap
6 293 {
7 294     // current pointer to dynamic memory, here, I'll actually just use the first ↔
    object I want to track.
8 295     heapPtr = (void*) malloc( 0 );
9 296
10 297     // create a couple of objects following our pointer so that we can track them...
11 298     first  = new MemoryObject;
12 299     second = new MemoryObject;
13
14 (gdb) x/20 first
15 0x6d40330: 0x02020202 0x02020202 0x06d40350 0x06d40354
16 0x6d40340: 0x06d40354 0x04040404 0x04040404 0x00000000
17 0x6d40350: 0x06d3fc5c 0x00000000 0x00000000 0x00030000
18 0x6d40360: 0x07b91364 0x07b91364 0x07b91000 0x07b91378
19 0x6d40370: 0x07b91378 0x07b91378 0x07b91378 0x07b91378
20
21
22 (gdb) x/20 second
23 0x6d3fc70: 0x06060606 0x06060606 0x06d40460 0x06d40464
24 0x6d3fc80: 0x06d40464 0x08080808 0x08080808 0x00020000
25 0x6d3fc90: 0x012adb70 0x010012c0 0x01000009 0x00000002
26 0x6d3fca0: 0x00000000 0x00040000 0x012b0b40 0x06d3fcc0
27 0x6d3fcb0: 0x06d3f210 0x00000000 0x00000000 0x00000000

```

One of the advantages of this approach is that similar to a texture atlas we can map various regions of memory, they don't need to be contiguous! For the GPU, we can populate a texture, or several textures, based on the heap or some other pointers in memory and allow the GPU to look for object signatures, or pattern searches with differing memory locations that would verify memory integrity, and help protect against an attack such as a heap spray. For example if we find similar patterns in two differing locations on the heap this may signal a payload being sprayed.

While this may seem somewhat contrived to some readers, this is not an uncommon way to represent contiguous memory. LLVM (Low Level Virtual Machine) has a *MemoryObject*, which is used within memory transactions - here we take a simplistic representation and tie this into our cyber analysis. What this allows us to do is offload tracking dynamic memory objects during execution, with only minimal interaction to track changes.

A. Texture Atlas

In the above discussion we brought up the concept of an atlas, which is familiar to those in the graphics world where one image has several pieces that will be used in a game or other application, so for example a single image file may have a character in various poses to allow a single image to contain the "running" or other actions the character might have. Similar to a texture atlas we can mark various regions of memory (within limits) and store them to a texture. We can mark region boundaries with a byte signature, such as a single color channel, and within the region we store our more used functions, data or other models and references.

B. Heap Visualization

Returning to our example, in listing 4, the first and subsequent objects we create, have a leading and trailing signature we can find these using a fragment shader (discussed and shown later), we can at some point move these objects into a texture to watch them.

For example in the object,

```

1 (gdb) x/20 first
2 0x6d40330↵
   : 0x02020202 0x02020202 0x06d40350 0x06d40354
3 0x6d40340↵
   : 0x06d40354 0x04040404 0x04040404 0x00000000
4 0x6d40350↵
   : 0x06d3fc5c 0x00000000 0x00000000 0x00030000
5 0x6d40360↵
   : 0x07b91364 0x07b91364 0x07b91000 0x07b91378
6 0x6d40370↵
   : 0x07b91378 0x07b91378 0x07b91378 0x07b91378

```

If we have a passthrough shader, then we can read back the pixel data and investigate the bytes after the draw call. A brief code snippet shows just how simple this can be.

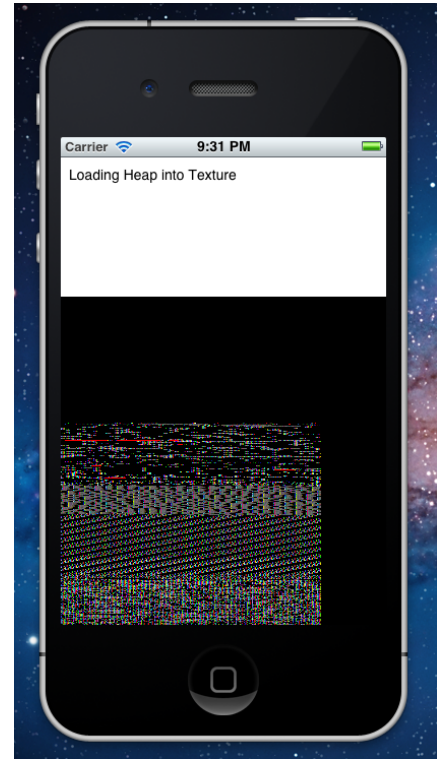


Fig. 3: Sample Screenshot of Memory as a Texture

Listing 5: GL code to read in our GPU results

```

1 // capture the data for analysis...
2 memset(bytes, '\0', dimension*dimension*4+1);
3 glReadPixels↵
  (0, 0, 256, 256, GL_RGBA, GL_UNSIGNED_BYTE, bytes);

```

Listing 6: Simple GDB Capture of GPU results

```

1 (gdb) x/10 bytes
2 0xb7fb000↵
   : 0xff020202 0xff020202 0xff204000 0xff06de5b
3 0xb7fb010↵
   : 0xffde5b24 0xff5b2406 0xff0406de 0xff040404
4 0xb7fb020: 0xff040404 0xff000000

```

We have a signature beginning at 0x6d40330, followed by the start and end of the `vector<string>` iterators (start, and finish), and then finally we have our trailing signature marking the boundary of the object of interest. To use this in practice, we can add a destructor that alters these signatures and so mark the texture as having invalidated the object. With this in mind, if we test the texture for a valid object being manipulated then we can see that the manipulation will likely result in an invalid object being used, and a possible target for a heap spray.

Listing 7: Sample Shader that can test for a simple byte signature

```

1  varying lowp vec4 DestinationColor;
2
3  varying lowp vec2 TexCoordOut;
4  uniform sampler2D Texture;
5  uniform sampler2D Mask;
6  uniform lowp vec2 offset;
7
8  // assume we have our signature in four bytes or less...
9  bool isValidSignature( lowp vec4 pixel )
10 {
11     lowp float norm;
12     norm = dot(pixel.rgb , pixel.rgb );
13     if ( norm > 0.0 )
14     {
15         // for now we just see if the all RGB channels match, this means
16         if ( pixel.r == pixel.g && pixel.g == pixel.b )
17             return true;
18     }
19     return false;
20 }
21
22 // basic shader to operate
23 // on multiple texture operations..
24 void main(void) {
25
26     if ( isValidSignature( texture2D(Texture, TexCoordOut) ) )
27     {
28         gl_FragColor    = vec4( 1, 0, 0, 1 );
29     }
30     else
31     {
32         gl_FragColor    = vec4( 0, 0, 0, 1 ) * texture2D( Texture, TexCoordOut);
33     }
34 }
35 }

```

With this very simple shader we can see immediate results, as the leading and trailing signatures become red pixels, marked for analysis!

Listing 8: Simple Shader Identifies Signature as Red

```

1  (gdb) x/100 bytes
2  0x6f99000:  0xff0000ff  0xff0000ff  0xff000000  0xff000000
3  0x6f99010:  0xff000000  0xff000000  0xff000000  0xff0000ff
4  0x6f99020:  0xff0000ff  0xff000000  0xff000000  0xff000000

```

So in this case, we know we can use a vectorized algorithm to find multiple signatures, markers or other interesting features within the region-of-interest.

V. DYNAMIC DISASSEMBLY

This is fairly straightforward in that we make the assumption that, simply put, if we better understand a payload, dataset, and other objects that may be executed than we can use higher level algorithms such as artificial intelligence and a variety of other techniques.

Inspired by the paper by [13], we decided that mobile systems might be a better target than the original idea applied to x86. This is because x86 has variable length instruction codes where as ARM (if we disregard Thumb for the moment) has a fixed instruction length.

A. Why This Works Even Better for Mobile

The approach we've taken works particularly well for mobile or ARM, because with our alpha, green, blue and red channels available in the texture, we have 4 bytes, or 32 bits. Well of course ARM instructions are 32 bit! Thumb is 16 and so for any instruction we can operate on both ARM and Thumb instruction inside a pixel of the fragment shader. This is in contrast to x86, with variable length instructions, which seriously challenges vectorization attempts of the algorithms on a GPU or other SIMD processor as we might have to deal with data loss. In other words, the convenience of an ARM instruction fitting into a texel is a very happy convenience and not something to take for granted with other instruction sets.

B. Implementation

To generate a table that we can include in our project we can build a variety of ARM codes, especially relevant to iOS/Android and then dump the text section to an instruction file which we can process for statistical relationships.

Listing 9: Script to grab ARM opcodes from Executables

```

1  #!/bin/bash
2
3  # test to make sure the instructions directory is here..
4  if [ ! -d "instruction-contents" ]; then
5      echo "[+] Creating instruction contents directory"
6      mkdir "instruction-contents"
7  fi
8
9
10 # otool -t simple | cut -d " " -f 2-4
11 for i in `ls`
12 do
13     type=`file $i`
14     needle='executable arm'
15     if [[ "$type" == *"$needle"* ]]; then
16         echo "Found arm executable -> $type"
17         # parse out the file name we'll write to...
18         fileprefix=`echo $type | cut -d ':' -f 1`
19         filename="${fileprefix}.instr"
20         `otool -t $fileprefix | cut -d " " -f 2-5 > "instruction-contents/${filename}"`
21     fi
22 done

```

With the instructions in a simple text file, we can create a Trie and a generic controller to find the opcodes, as well as the order of these instructions and create probabilistic tables of what likely instruction sequences should be. With our known opcodes we can construct a matrix to identify incoming operations. One item of interest is that we don't have to have this for the entire platform. For example if we were going to employ dynamic disassembly for a browser, we would want probability tables that are representative of what we should expect, so generate tables from WebKit source for example. If we started encountering lower probability sequences we might suspect that we have foreign code. With this in mind, let's explain the algorithm in a bit more detail.

We want to find the maximal probability of an opcode given the incoming instruction, or in matrix form, we have:

$$\begin{pmatrix} Op_{str} & 0 & 0 & 0 \\ 0 & Op_{add} & 0 & 0 \\ 0 & 0 & Op_{mov} & 0 \\ 0 & 0 & 0 & Op_{ldr} \end{pmatrix} \times \begin{pmatrix} inst_1 & inst_2 & inst_3 & inst_4 \\ inst_1 & inst_2 & inst_3 & inst_4 \\ inst_1 & inst_2 & inst_3 & inst_4 \\ inst_1 & inst_2 & inst_3 & inst_4 \end{pmatrix} = \begin{pmatrix} p_{1,str} & p_{2,str} & p_{3,str} & p_{4,str} \\ p_{1,add} & p_{2,add} & p_{3,add} & p_{4,add} \\ p_{1,mov} & p_{2,mov} & p_{3,mov} & p_{4,mov} \\ p_{1,ldr} & p_{2,ldr} & p_{3,ldr} & p_{4,ldr} \end{pmatrix}$$

To find our best match, or if perhaps our table is incomplete, we can take the maximal value for each row, which corresponds to the best matching operation, and if this value is above some threshold we declare that we "know" the opcode. If not, we declare we have an unknown operation.

To look at the instruction in a contextual sense, we look at patterns within subregions. First we can construct a transition matrix (details to follow) and use this to guess our unknown instructions, or we could look holistically at a "best fit" instruction giving known operations prior to and following an unknown.

The former case is fairly straightforward in that we walk our currently constructed sequence and when we reach a given unknown we multiply the against the the transition matrix to obtain the most probable instruction path. The best way to make use of resources in this fashion is to populate a matrix of transitions, each row corresponding to a state entering transition, and then multiply this matrix against the transition matrix.

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{pmatrix} = \begin{pmatrix} p_{31} & p_{32} & p_{33} & p_{34} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{41} & p_{42} & p_{43} & p_{44} \\ p_{11} & p_{12} & p_{13} & p_{14} \end{pmatrix}$$

In the latter case, we use the powers of the transition matrix in reverse.

State Transition

$$S \cdot P = S'_p$$

$$S \cdot P^2 = S''_p$$

But if we are only missing a single operation we know the outcome and hence some properties of S''_p

This means we can look forwards or backwards

$$S \cdot P = S''_p \cdot P^{-1}$$

This means that we can calculate the most likely path to our now known state, and deduce a reasonable approximation for our prior operation. Any component of S''_p that doesn't include a path to our known operation can be eliminated, and then we take the maximal probability remaining as our accepted conclusion. Each column then represents either a known instruction or an equally weighted unknown. We can

then create a transition matrix, which contains our probabilities of the next instruction given the previous. We can then find our successive probability of a sequence by collapsing the matrix product, in essence giving us a "best match" over instruction paths.

Let's test this using a simple source and then using Python we'll remove various opcodes and let our disassemble predict the correct instruction.

Listing 10: Sample Disassembly

```

1 $ python simulate_dynamic_dissembly.py
2 Read 44 instructions from the file sim.instructions
3 replacing 0x3c
4   with 0x00
5
6 replacing 0xaf
7   with 0xaf
8
9 replacing 0x00
10  with 0x00
11
12 replacing 0x28
13  with 0x28
14
15 replacing 0x58
16  with 0x58
17
18 replacing 0x08
19  with 0x08
20
21 replacing 0x00
22  with 0x00
23
24 * * * 1 Errors in 8 unknown operations * * *
```

And the Python code to exercise our disassembler for testing and enhancements is straightforward.

Listing 11: Python Script to Exercise Disassembler

```

1 #!/usr/env python
2
3 import subprocess, random, simplejson
4 # to install simplejson: pip install simplejson
5
6 # First we read in our instruction file that we want to use
7 filename = 'sim.instructions'
8 instructions = open(filename).readlines()
9 print "Read ",len(instructions)," instructions from the file ",filename
10
11 # now let's get rid of ~8 instructions (roughly 20% from our given file)
12 n = 8
13 random.seed(24) # for reproducability, if needed
14 samples = list(instructions)
15 for xx in xrange(0,n):
16     index = random.randint(0,len(instructions)-1)
17     samples[index] = 'xxx'
18
19 # Now we have a few unknown ops, let's use our C-code to on-the-fly generate
20 # disassembly using probability tables
21 last_instruction = False
22 application = './optimization'
23 args = ''
24 digraph = 'simple.digraph'
25 recreated = []
```

```

26 index = 0
27 for instr in samples:
28     if ( instr == 'xxx' ):
29         # call out...
30         try:
31             prob_instruction = subprocess.check_output([application, last_instruction, ↵
                 digraph])
32             instr = prob_instruction
33             print "replacing %s with %s " % (instructions[index], instr)
34         except:
35             print "Error in our C-code, time to debug..."
36         # append...
37         recreated.append(instr)
38         last_instruction = instr
39         index+=1
40
41 # Now we test our disassembly to see how much we got right...
42 errors = 0
43 for i in xrange(0, len(instructions)):
44     if recreated[i] != instructions[i]:
45         errors+=1
46
47 print "* * * ", errors, " Errors in ", n, " unknown operations * * *"

```

Listing 12: Main for creating disassembly tables

```

1  #include <iostream>
2  #include <algorithm>
3  #include <fstream>
4  #include <unistd.h>
5  #include <getopt.h>
6  #include "trie.h"
7  #include "util.h"
8  #include "controller.h"
9  #include "matrix.h"
10
11
12 using std::fstream;
13 using std::cout;
14 using std::endl;
15 using std::string;
16
17 int main( int argc, char *argv[] )
18 {
19
20     /*
21      * Argument error check
22      */
23     if ( argc < 2 )
24     {
25         cout << "Usage: " << argv[0] << " <infile> [additional file or file patterns]" << ↵
             endl;
26         exit(0);
27     }
28
29     /*
30      * Parse argument
31      */
32     int option_index, c;
33     bool dump_opcode   = false;
34     bool combine_files = false;
35     bool gen_digraph   = false;
36     string outFileName = "temp.txt";
37     string treeName     = "";

```

```

38     bool dump_all      = false;
39
40     static struct option options[] = {
41         {"opcodes",0,0,0},
42         {"outfile",1,0,0},
43         {"combine",0,0,0},
44         {"digraph",0,0,0},
45         {"all",1,0,0},
46         {NULL,0,NULL,0},
47     };
48
49     while( (c = getopt_long( argc, argv, "a:cgo:x", options, &option_index) ) != -1 )
50     {
51         switch ( c )
52         {
53             case 'a':
54                 cout << "Dumping all instructions.." << endl;
55                 dump_all = true;
56                 treeName = optarg;
57                 break;
58             case 'x':
59                 cout << "opcode output..." << endl;
60                 dump_opcode = true;
61                 break;
62             case 'o':
63                 cout << "outfile named..." << endl;
64                 outFileNames = optarg;
65                 break;
66             case 'c':
67                 cout << "combine files..." << endl;
68                 combine_files = true;
69                 break;
70             case 'g':
71                 cout << "generating digraph..." << endl;
72                 gen_digraph = true;
73                 break;
74             default:
75                 cout << "unknown " << c << endl;
76                 break;
77         }
78     }
79
80     }
81
82
83     vector<string> files_to_process = parse_args( argc, argv );
84
85     Controller * jobController = new Controller;
86     Trie * tree;
87
88     for (auto fstring = files_to_process.begin();
89          fstring != files_to_process.end(); ++fstring)
90     {
91         if (!jobController->ProcessFile( (*fstring) ) )
92             cout << "Error processing file " << (*fstring) << endl;
93         else {
94             cout << "Processed file " << (*fstring) << endl;
95             tree = jobController->TreeForFile( (*fstring) );
96             cout << "Processed " << tree->NumberOfInstructions() << " instructions" << endl;
97             cout << "Processed " << tree->NumberOfUniqueInstructions() << " unique instructions" << endl;
98         }
99     }
100 }
101

```



```

102
103     /*
104     * look across our tries
105     */
106     size_t ninstructions = jobController->NumberOfUniqueInstructionAcrossTrees( );
107     cout << "Batch: " << ninstructions << " unique instructions" << endl;
108     cout << "Writing to file: " << outFileName << endl;
109
110     if ( dump_all )
111     {
112     if ( dump_opcode ) jobController->WriteOpcodes();
113     jobController->ListAllInstructionsToFile( treeName, outFileName );
114     }
115     else if ( dump_opcode && !gen_digraph )
116     {
117     jobController->WriteOpcodes();
118     cout << "Dumping opcodes - mode: " << jobController->Mode() << endl;
119     jobController->ListInstructionsToFile( outFileName );
120     }
121     else if ( gen_digraph )
122     {
123     cout << "Writing DiGraph..." << endl;
124     if ( !jobController->WriteDiGraphToFile( outFileName ) )
125         cout << "Unable to write digraph to " << outFileName << endl;
126     else
127         cout << "Written..." << endl;
128     }
129     else
130     {
131     jobController->ListInstructionsToFile( outFileName );
132     }
133     cout << "\n";
134
135     /*
136     * Cleanup
137     */
138
139     delete jobController;
140
141     return 0;
142 }

```

Using this code-set we can create a table of transitional probabilities.

C. In Practice

Dynamic disassembly is clearly best with floating point precision and other numerical routines available to the system. For example if floating point precision isn't available, in libraries such as Renderscript, Accelerate or in an optimized C++ library such as Eigen, then we have to be careful for signed/unsigned overflow operations and how to round our decimal representations to a byte, for example we could round 0.50 to 0x7f as an unsigned byte that can be used a fractional number. An example of how to perform generic matrix multiplication on the GPU follows.

Listing 13: Simple Shader for Square Matrix Multiplication

```

1 void main( ) {
2     // we invert, row <-> column
3     lowp vec2 transpose = TexCoordOut.ts;
4     lowp vec2 original  = TexCoordOut.st;
5     // multiply the components of the two textures..
6     gl_FragColor = texture2D( matrix_one, original ) * texture2D( matrix_two, transpose );
7 }

```

With the above shader, if the diagonals RGB were 0xff and 0x7f, then the resultant fragment would have 0x7f along it's diagonals as well. It's simply $0.5 \cdot 1.0 = 0.5$. Keep in mind that we could perform the calculations on separate channels, giving us a separate range $[0, 1]$ for each of the four channels within RGBA.

VI. ENCRYPTION

Especially within mobile applications, live memory manipulation and capture has been done. On iOS, cypcript can be used to inspect the objective-c runtime. In that case, we could use the GPU to vectorize our encryption process, as well as dynamically loading a new encryption scheme by updating the shader (OTA for example). A basic algorithm is outlined as:

Algorithm 2 Encryption

```

1: INPUT: Uniforms as keys, etc; Shaders as encryption methods
2: Bind data to be encrypted to texture
3: Link frag shader as a "block" encryption call
4: Draw call, renders encrypted data to off screen texture
5: if insufficient rounds then
6:   use newly rendered texture as input to above
7:   repeat
8: end if
9: if want to change block method then
10:  link new frag shader
11:  repeat using new shader program
12: end if
13: End

```

Employing encryption can help hide sensitive information from malware, or if signed we could follow where the encrypted payload moves throughout the operating system, performing a taint analysis with the encrypted data to minimize exposure.

VII. ENVIRONMENT CONSIDERATIONS

For simplicity's sake we consider beta algorithms for two environments and actually implement in one. The two environments are termed *restrictive*, meaning an environment where we are forced to play by the Vendor's policies; and *unrestrictive*, meaning an environment where we have control and are not forced to adhere to any vendor policies.

In the commercial environment we would expect developers and the vast majority of development efforts to work in the *restrictive* environment. This is contrast to the DoD, for example which could contract with a vendor for a particular Android version and include various mechanisms into the firmware, meaning an *unrestrictive environment*.

A. Restrictive Environments

Restrictive environments are the focus for the Milestone 3 deliverable as it demonstrates feasible approaches within the current marketplace. The details can be seen in the source code which is attached as part of the deliverable.

B. Unrestrictive Environments

An unrestrictive environment has a number of advantages, primarily being able to manipulate the kernel and graphics driver code for additional resource capabilities. For example, on Android this means that compiled shaders could be cached for reuse as needed, allowing an optimization not available within the SDK.

VIII. DEVICE AND PLATFORM INFORMATION

TABLE I: Device Characteristics

Characteristic	iOS (iPhone 4S)	ASUS Transformer Prime
Max. Number of Textures	8	16
Max. texture size	4096	2048
Version	OpenGL ES 2.0 IMGSGX543-63.24	OpenGL ES 2.0 14.01002
Vendor	Imagination Technologies	NVIDIA Corporation
Renderer	PowerVR SGX 543	NVIDIA Tegra 3
Extensions	GL_OES_depth_texture GL_OES_depth24 GL_OES_element_index_uint GL_OES_fbo_render_mipmap GL_OES_mapbuffer GL_OES_packed_depth_stencil GL_OES_rgb8_rgba8 GL_OES_standard_derivatives GL_OES_texture_float GL_OES_texture_half_float GL_OES_texture_half_float_linear GL_OES_vertex_array_object GL_EXT_blend_minmax GL_EXT_color_buffer_half_float GL_EXT_debug_label GL_EXT_debug_marker GL_EXT_discard_framebuffer GL_EXT_occlusion_query_boolean GL_EXT_read_format_bgra GL_EXT_separate_shader_objects GL_EXT_shader_texture_lod GL_EXT_shadow_samplers GL_EXT_texture_filter_anisotropic GL_EXT_texture_rg GL_APPLE_framebuffer_multisample GL_APPLE_rgb_422 GL_APPLE_texture_format_BGRA8888 GL_APPLE_texture_max_level GL_IMG_read_format GL_IMG_texture_compression_pvrtc	GL_NV_platform_binary GL_OES_rgb8_rgba8 GL_OES_EGL_sync GL_OES_fbo_render_mipmap GL_NV_depth_nonlinear GL_NV_draw_path GL_NV_texture_npot_2D_mipmap GL_OES_EGL_image GL_OES_EGL_image_external GL_OES_vertex_half_float GL_OES_mapbuffer GL_NV_draw_buffers GL_NV_multiview_draw_buffers GL_EXT_Cg_shader GL_EXT_packed_float GL_OES_texture_half_float GL_OES_texture_float GL_EXT_texture_array GL_OES_compressed_ETC1_RGB8_texture GL_EXT_texture_compression_latc GL_NV_texture_compression_latc GL_EXT_texture_compression_dxt1 GL_EXT_texture_compression_s3tc GL_NV_texture_compression_s3tc GL_EXT_texture_filter_anisotropic GL_NV_get_tex_image GL_NV_read_buffer GL_NV_shader_framebuffer_fetch GL_NV_fbo_color_attachments GL_EXT_bgra GL_EXT_texture_format_BGRA8888 GL_EXT_unpack_subimage GL_NV_pack_subimage GL_NV_texture_compression_s3tc_update GL_NV_read_depth GL_NV_read_stencil GL_EXT_robustness GL_OES_standard_derivatives GL_NV_EGL_stream_consumer_external GL_NV_coverage_sample GL_EXT_occlusion_query_boolean

As the reader can see from the demonstration, our example iOS code only uses a small texture as we want to create a demonstration application that allows for visualization of our intent and actions. However, in practice we have more resources at our disposal, as we can augment our 256 square texture to a 4096 ($x16^2$) square texture as well as rendering off screen. In addition, one has to be careful with background interference. In our sample application we limit this by setting the alpha channel to 0xff (1.0f) and only using the RGB (Red/Green/Blue) channels for cyber tasks. This isn't a significant hurdle but something to be aware of.

IX. OPTIMIZATIONS AND OTHER TRICKS

A. Generalities

One advantage of the GPU environment is that many of the tricks that we might like, such as repeating boundaries (often necessary in numerical environments such as PDE's) are available via simple API calls. As we sweep our mask through the region-of-interest we can use the repeating background as a performance gain, meaning that we only need to sweep half-way through the region in some cases (it depends on sizing, etc).

There are other graphical techniques that might allow us to "zoom in" and out of various regions of interest. For example, if we take a block of memory to be executed, and let's assume for the moment that this is sufficiently large that we would need a compression method or perhaps mipmap leveling (or both!) to store as much as possible. If we focus on approximate disassembly, which is our basis for the idea, then if we choose a compression method which leaves enough of the high frequency instructions or opcodes then we can use our probabilistic detection schemes on the compressed data. The losses, or inaccuracies,¹ would be independent, due to the nature of the design (because it's probabilistic!). As we said, if we thought we wanted to "zoom in" on a specific region, then we could do a vertex transformation to bring that mipmap level into view. Each region-of-interest could then be checked for signatures, investigated for instructions, etc.

B. Our Code

In our code we use off screen rendering to hold results in a texture that we can quickly process. The metaphor or a workspace of a texture is appropriate here, and so an engineer could decide whether certain applications might benefit from a larger workspace or more workspaces. Some very quick thoughts to provoke the reader are, for example in our two devices where the iPhone 4S has larger texture sizes available to us and the ASUS tablet has more textures, if we have a set of signatures we want to test against the the workspace it likely the higher value as we can tile the signatures and not have to exchange framebuffers and texture objects. In contrast if we have more complicated algorithms that might require intermediate staging then the number of textures might make ease the development process. An example here might be a complicated encryption scheme that uses a texture to hold intermediate results in a chained block cipher.

X. PERFORMANCE

Performance can be investigated by running more developed, "Beta" quality applications, running GPU cyber tasks via the aforementioned off-screen rendering technique. Our two applications of choice were chosen to contrast how a developer might use the GPU; namely continuously for application monitoring via shader memory tacking or as single event to scan for malicious signatures after an event however costly that shot might be.

Figure 4 summarize the power cost of continuously monitoring an area of application memory, Figure 4(b), as well as a performance of a vectorized scan of a payload after download, Figure 4(a).

As we can see the cost of the both of these techniques is somewhat minimal and over repeated usage (on the order of tens of app launches) the power cost is minimal, comparable to other applications. Scanning a payload obviously has a time cost but we should remind the reader that we're not just able to scan a single signature in that time period, we're able to scan many. Speaking to monitoring memory via the GPU, this is another benefit of a mobile device in that we expect application usage to be brief, less than a minute and so with a simple OpenGL surface being rendered offscreen we would estimate the cost to be similar to a simple game. With the event driven model, the payload is scanned effectively, with a small cost to the setup of the OpenGL context and scanning, which involves sweeping our set of signatures-of-interest

¹For example, using a DCT, Direct Cosine Transform, generally causes a loss of high frequency or background information. This is not what our scheme would want as the background information, or opcodes give context to our guess as to what opcodes would be between

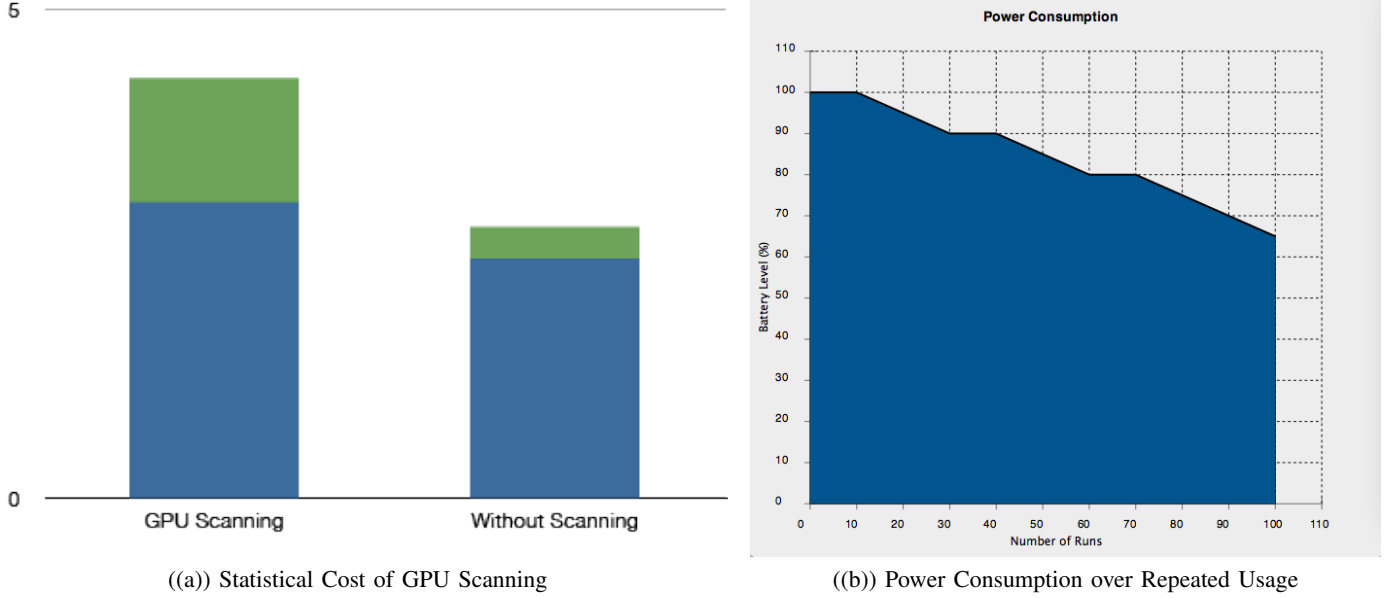


Fig. 4: Performance Characteristics

through the payload. The average time to load our payload is seen in blue (comparing with/without a scan) and the green represents a single standard deviation. The expected time increase is the cost of an additional operation and the variance can be understood in terms of when we find a signature-of-interest.

In summary, the performance costs are acceptable today and will likely be even more advantageous in upcoming years.

XI. HARDWARE INTEGRATION

How vectorized processors are being used in every day applications is of great interest and the state of the art is currently Android's Renderscript. Renderscript uses LLVM to compile C99 code into LLVM bitcode (a .bc file) as well as Java reflection classes. These are all packaged as part of the apk and then the device itself will compile the bitcode into machine code as well as cache it for later use. As part of the OS, libbcc is the on-device compiler that will generate code for the GPU or DSP processor as well as integration glue for the Dalvik JVM [14].

For best results the ideal placement would be allow for "on-the-fly" alteration of the bitcode (this would necessitate a recache of the newly compiled machine code for ICS as this a feature to save on-load time) and then update the application, or system. From the operating system this should not be a problem and would allow Android to fully utilize the other processors for cyber tasks however this also means that code insertion could be that much easier for malicious tasks.

The call to libbcc and the related tasks are handled with the on device renderscript framework. Renderscript contains a runtime, which helps to bind variables to reflected Java classes from generated shaders or other GPU programs. The runtime has a related driver code which is linked to the libbcc code, this bridge allows for compilation, propagation of metadata and other information to pass between the JVM and GPU code. The actual device code is generated from within an ExecutionEngine, the details can be found within, frameworks/compile/libbcc/lib/ExecutionEngine/ see the files bcc.pp and Compiler.cpp for instance. It's of note that this is a heavily stripped down compiler as compared to many LLVM backend systems. The engine links the bitcode to system libraries as well as extract the necessary information to create vectorized code for the GPU/DSP. The process can be summarized in Figure 5, which is a slide from Liao Shih-wei's presentation at *Linux Foundation: Collab-Summit*[14], and outlines the interplay of bitcode and reflected Java for interplay.

Online JIT Compiler Flow

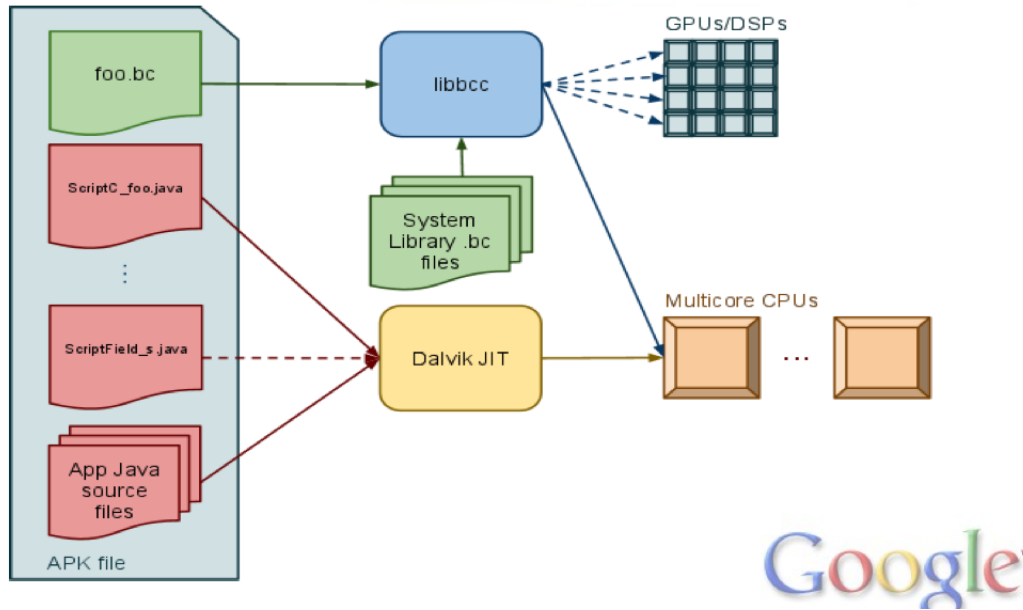


Fig. 5: Flow diagram of online JIT

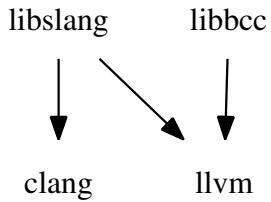


Fig. 6: Relationships between LLVM and Renderscript Core Libs

llvm-rs-cc is a driver on top of libclang and is run on the host and performs many aggressive optimizations. As a result, libbcc on the device can be lightweight and focus on machine-dependent code generation for the input bitcode. It would be of interest to investigate the security of this lightweight compiler with obvious privileges. The architecture of libclang and libbcc is depicted in figure 6.

The renderscript runtime functions as you might expect in that it creates and binds vertex and frag programs to the GPU from the bitcode. The runtime is bound to the libbcc code, being able to extract meta data from the bcc (BitCode Compiler) as needed as well as creating generic GPU materials. For example, we see the following snippet within the runtime creating a vertex shader.

Listing 14: Code Fragment from rsdProgram.cpp within frameworks/base/libs/rs/driver

```

1
2 bool rsdProgramVertexInit(const Context *rsc, const ProgramVertex *pv,
3                           const char* shader, uint32_t shaderLen) {
4     RsdShader *drv = new RsdShader(pv, GL_VERTEX_SHADER, shader, shaderLen);
5     pv->mHal.drv = drv;
6
7     return drv->createShader();
  
```

8 }

These frameworks are non-existent in other parts of mobile platforms and there's reason to believe that like the desktop, capabilities and runtime access will be expanded for more general capabilities. Here again we can see the potential of a graphics based capability: a dynamic architecture capable of altering itself on the device; fast, powerful processing capabilities; with an increasingly broad set of capabilities.

While it's out of scope for this project, if there was sufficient time and resources the RenderScript framework could be used to allow OTA shader delivery that use GPU processing for cyber tasks. For a quality product this needs to be integrated with the hardware capabilities of the device but the framework does allow for shader caching and recall.

A. Shader Trends

The reason for the run-time compilation of shaders for both iOS and Android is to mitigate the device specifics, allowing the developer to work with the core technology, namely OpenGL ES, and not worry about the details and varieties of driver versions, etc. OpenGL developers have and do complain about not being to work directly with the driver and not being allowed to load the pre-compiled shader binaries directly into the graphics hardware. While allowing for binary access to a graphics chip and the device driver, residing in the kernel, may sound like a worrisome security choice it's important to note that the graphics industry is by far the largest market genre for revenue within the mobile application industry - in other words games mean money and this is highly influential. It's fairly easy to make the case that iOS's security measures have been largely brought about, not for device security but for an economic monopoly and therefore if money is a driver this is an area to watch in upcoming years.

XII. STRATEGIES

While each of these techniques adds some level of protection, if we want to truly gain substantially from GPU malware assistance, it's more likely to employ a combination of ideas. Let's explore some of this here.

As we know we can detect signatures, inspect memory and other relevant objects directly on the GPU, as well as take advantage of the GPU and other SIMD processors for analysis tasks such as dynamic disassembly. From these tasks we can start to form a strategy. For example, if we don't want to consume too much power, we could use the GPU/NEON processors while tethered (as power is not concern there and security is) to disassemble incoming instructions. When the browser is about to relinquish control to a Flash application or PDF viewer, then a burst of activity could be used to both perform some signature matching as well as set up memory tracking. Similarly if a password or other scheme is needed, it could be encrypted using the GPU.

Most significantly, these checks can be done by outside processing power, and dynamically. As shader can be compiled at runtime, and a new shader could be introduced simply be over-writing to an existing file and recompiling (which is typically done by simply restarting the parent process) this allows for a very flexible strategy that could be updated Over-The-Air (OTA).

XIII. GPU ASSISTED MALWARE

It's noteworthy that this same defensive strategy could be employed by an attacker for offensive capabilities. As we've proven that we do pattern searching, even if somewhat basic, along with dynamic disassembly then we can imagine constructing ROP gadgets for exploitation using similar techniques. While this is worrisome, the essence of the work is to increase the defensive surface and we've done this. Attackers will continue to be creative and come up with new exploitation methods and to play our part in the game we augment the resources at our disposal.

XIV. SUMMARY

This research shows that the GPU, as well as other processors, could be used for cyber tasks that could make attacks more difficult. While we have shown a series of algorithms and code that demonstrate how to do this, this is likely just a starting point. GPU and other SIMD processors will only get more powerful as mobile devices continue to evolve and in particular the demand for compelling user interfaces and high-end graphics. This means that while this paper might be state-of-the-art today, it will likely serve as an entry point for future research for tomorrow.

XV. LANDSCAPE AND LIKELY DEVELOPMENTS

Expanding on the minimal components we've discussed to this point can be extrapolated by looking at desktop SDK's such as CUDA and OpenCL. The same parallelization and general techniques are the core of these APIs but there are, of course, more access and control points in the desktop environment. However, one significant point to make is that in the development of heuristic algorithms, not only can we prototype using these more generalized APIs but we can also use them to train input or other peripheral sets that can be used by the GPU - for example using optimization algorithms for pseudo-disassembly, often uses probability tables which could be generated on any machine. By leveraging graphics kernels (operations) and data sets made for vectorized algorithms this offers a faster path to migrate general cyber algorithms to the GPU itself.

A. *CUDA*

CUDA is NVIDIA's offering to generalize GPU capabilities for tasks outside of graphics and rendering. A couple of key concepts that CUDA uses have become common place within the graphics community.

- kernel – An operation to be applied throughout the geometry, in other words to all elements.
- Device and Host Memory interaction. CUDA has various calls to transfer memory from the host (assumed to be a PC) and the GPU.
- Thread Batching – A batch of threads that operates with a limited amount of shared memory.

One of the noteworthy items of CUDA is the utilization of memory. Of particular interest is CUDA's Uniform Virtual Addressing (UVA) which uses the address as a way to detect which device houses a region of memory. This is designed to aid in scaling GPU tasks. CUDA hardware uses a set of SIMD multiprocessors with on-chip shared memory, the SIMD behavior can be grouped by threads into units called warps. Further details for hardware and software can be found at the following CUDA Overview [8].

A sample CUDA program can be seen in Listing 19 of Appendix B.

B. *RenderScript*

Android's migration to Honeycomb meant a leap forward in Renderscript, which now moves more towards a CUDA-like API - see <http://android-developers.blogspot.com/2011/02/introducing-renderscript.html>. This is another reason to pursue the current state of GPGPU research on the desktop. While a guess, it looks like Android will be an active research and development area for third party graphics and chip makers to investigate embedded and mobile development.

C. *OpenCL*

OpenCL is not solely dedicated to the GPU but more broadly about balancing the GPU and CPU to optimize performance. OpenCL takes advantage of vectorized libraries that will run efficiently on the GPU, a reason why many within the iOS community have concluded that OpenCL will come to iOS based on the libraries within the Accelerate Framework.

OpenCL also uses a terminology similar to CUDA, where a kernel is typically a name given to a parallelized operation. see Listings 20 and 21 within Appendix B for examples.

XVI. FUTURE RESEARCH

This project was inspired by the work others have shown in hacking embedded chips in cars, appliances and other electronics. The conversation sprung up in terms of what could be utilized to exploit or aid the CPU; naturally the GPU was and is the first option, but others are likely potential resources as well. NEON processors and similar vectorized, auxiliary capabilities will be available in future mobile devices.

XVII. CONCLUSION

This project has demonstrated that GPUs and other vectorized processors can play a vital role in cyber tasks for mobile devices. We have presented psuedo-algorithms and alpha implementations for a variety of tasks, such as signature checking, memory tracking, encryption, dynamic disassembly. We have demonstrated that the landscape for GPU driven cyber tasks is likely to provide more opportunity and that with the existing capabilities we can provide meaningful systems to mitigate malware and monitor the device. Research in this area is highly recommended and will likely continue to be driven by ever-increasing GPU capabilities.

XVIII. ACKNOWLEDGMENT

The author is thankful to the following contributors: Alan Stone, Rob Dingwell and Seth Landsman of the *MITRE Corporation*. In addition the author wishes to thank Zachary Carlson of *Lycos*, Ayal Spitz of *PatientKeeper*, George Gal and Dan Rosenberg of *Virtual Security Research* and Thomas Cannon and Andrew Hoog of *viaForensics* for comments and thoughts as well as support regarding this project. We certainly want to thank *DARPA* for its support and interest in the research.

REFERENCES

- [1] Apple, Apple Guidelines for OpenGL ES https://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/BestPracticesforShaders/BestPracticesforShaders.html#//apple_ref/doc/uid/TP40008793-CH7-SW3
- [2] Ubergizmo, *NVIDIA Tegra 3* <http://www.ubergizmo.com/2011/11/nvidia-tegra-3/>
- [3] Streamcomputing.eu, *Is OpenCL coming to Apple iOS?* <http://www.streamcomputing.eu/wp-content/uploads/kalins-pdf/singles/is-opencl-coming-to-apple-ios.pdf>
- [4] ARM Holdings, *Mali Graphics Hardware* <http://www.arm.com/products/multimedia/mali-graphics-hardware/index.php>
- [5] Qualcomm Developer Network, *Adreno Graphics Processing Units* <https://developer.qualcomm.com/discover/chipsets-and-modems/adreno>
- [6] NVIDIA, *Meet 'CARMA' - The CUDA on ARM Development Kit* <http://blogs.nvidia.com/2011/12/meet-carma-the-cuda-on-arm-development-kit/>
- [7] anandtech.com *Motorola Droid X: Thoroughly Reviewed* <http://www.anandtech.com/show/3826/motorola-droid-x-thoroughly-reviewed/5>
- [8] Bastien Chopard, *CUDA Overview* <http://cui.unige.ch/~chopard/GPGPU/2-cuda-overview.pdf>
- [9] Davi, L., Sadeghi, A.R., Winandy, M. Sytem Security Lab - Ruhr University Bochum, Germany *ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks*
- [10] Polychronakis, M., Keromytis, A.D., Columbia University *ROP Payload Detection Using Speculative Code Execution*
- [11] Apple, Apple Guidelines for OpenGL ES https://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/BestPracticesforShaders/BestPracticesforShaders.html#//apple_ref/doc/uid/TP40008793-CH7-SW3
- [12] G. Vasiliadis, M. Polychronakis, and S. Ioannidis *GPU-Assisted Malware* <http://dcs.ics.forth.gr/Activities/papers/gpumalware.malware10.pdf>
- [13] Shah, Abhishek, *Approximate Disassembly using Dynamic Programming*, 2010, Master's Projects. Paper 8. http://scholarworks.sjsu.edu/etd_projects/8
- [14] Shih-wei Liao, *Linux Foundation: Collab-Summit*, April 7th, 2011 https://events.linuxfoundation.org/slides/2011/lfcs/lfcs2011_llvm_liao.pdf

APPENDIX A EXAMPLE CODE

APPENDIX B PROJECT TEMPLATES

A. iOS

Apple's iOS is fairly straightforward to set up for GPU experimentation. The simplest form simply incorporates the OpenGL ES framework.

Listing 15: Render Method for iOS project

```

1 - (void)render:(CADisplayLink*)displayLink
2 {
3     glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
4     glEnable(GL_BLEND);
5
6     glClearColor(0, 104.0/255.0, 55.0/255.0, 1.0);
7     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
8     glEnable(GL_DEPTH_TEST);
9
10    float top = 320.0f;
11    float bottom = 0.0f;
12    float left = 0.0f;
13    float right = 320.0f;
14    float aspect = 1.0f; //(float) self.frame.size.width / (float) self.frame.size.height;
15    float projection[16] = {
16        2.0/(right-left), 0.0f, 0.0f, 0.0f,
17        0.0f, 2.0f/((top-bottom)*aspect), 0.0f, 0.0f,
18        0.0f, 0.0f, -2.0f/1000.0f, -3.0f,
19        0.0, 0.0, 0.0, 1.0f
20    };
21    float modelview[16] = {
22        1.0f, 0.0f, 0.0f, 0.0f,
23        0.0f, 1.0f, 0.0f, 0.0f,
24        0.0f, 0.0f, 1.0f, 0.0f,
25        0.0f, 0.0f, 0.0f, 1.0f
26    };
27
28    GetGLError();
29
30
31    glUniformMatrix4fv(_projectionUniform, 1, GL_FALSE,
32        projection);
33    glUniformMatrix4fv(_modelViewUniform, 1, GL_FALSE,
34        modelview);
35
36    GetGLError();
37
38    glViewport(0, 0, self.frame.size.width, self.frame.size.height);
39    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
40    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
41
42    GetGLError();
43
44    glVertexAttribPointer(_positionSlot, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
45    glVertexAttribPointer(_colorSlot, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (GLvoid*)(←
46        sizeof(float)*2) );
47    glVertexAttribPointer(_texSlot, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
48        (GLvoid*)(sizeof(float)*6) );
49
50    GetGLError();

```

```

51     glActiveTexture(GL_TEXTURE0);
52     glBindTexture(GL_TEXTURE_2D, texture);
53     glUniformli(sampler, 0);
54
55     GetGLError();
56
57
58     glDrawElements(GL_TRIANGLES, 6,
59                   GL_UNSIGNED_BYTE, 0);
60
61
62     // capture the data for analysis...
63
64     glReadPixels(0, 0, 320, 480, GL_RGBA, GL_UNSIGNED_BYTE, bytes);
65
66     // analyze our results
67     [self analyzePixelData:bytes];
68
69     [_context presentRenderbuffer:GL_RENDERBUFFER];
70
71 }

```

The key points, for experimentation is to setup a frame area, which could be offscreen but in this case is not, which can be easily scanned (via *glReadPixels*. Once this area is setup we can send various objects, in form of texture and vertex data to the GPU via draw calls while reading results.

Sample code will illustrate the finer details but before the reader does this, it's important to contrast this with a simple example using the Accelerate framework. This framework is used to vectorize various mathematical calculations, leveraging the GPU as well as the NEON processor.

Listing 16: Sample disassembly using the Accelerate framework

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <math.h>
4  #include <stdlib.h>
5  #include <Accelerate/Accelerate.h>
6
7  typedef struct {
8      char **opcodes;
9      unsigned int numberOpcodes;
10     float * elements;
11     unsigned int numberElements;
12 } matrix;
13
14 matrix *read_matrix_from_file( const char * filename );
15 void print_matrix( FILE * file, matrix * m );
16 int destroy_matrix( matrix * m );
17 void populate_vector( float * vector, int nels, char ** opcodes, char * code );
18 void print_vector( float * result, int nels );
19 void print_result( float * result, int nels, char **opcodes );
20 char *choose_opcode( float * result, int nels, char **opcodes );
21
22 int main( int argc, char * argv[] )
23 {
24
25     /*
26     * BLAS documentation
27     * https://developer.apple.com/library/ios/#DOCUMENTATION/Accelerate/Reference/BLAS\_Ref/
28     * Reference/reference.html
29     */
30
31     /*
32     * Here's a really good source...

```

```

32     * http://www.prism.gatech.edu/~ndantam3/cblas-doc/doc/html/cblas\_8h.html
33     */
34
35     /*
36     * doc's for LAPACK functions can be found at:
37     * http://www.netlib.org/lapack/lug/node147.html#22228
38     */
39     int solve_for_instruction = 0;
40     char * matrixfile = NULL;
41     matrix * m; // = read_matrix_from_file( "matrix.txt" );
42     // print_matrix( stdout, m );
43     // destroy_matrix( m );
44
45
46     if ( argc == 3 ) {
47         solve_for_instruction = 1;
48         matrixfile = argv[2];
49     } else if ( argc == 1 ) {
50         matrixfile = NULL;
51     } else {
52         printf("Usage %s [previous opcode] [matrix file]\n",argv[0]);
53         exit(1);
54     }
55
56     __CLPK_integer info;
57     float alpha = 1.0f;
58     float beta = 1.0f;
59
60     if ( solve_for_instruction )
61     {
62         m = read_matrix_from_file( matrixfile );
63
64         __CLPK_integer mn = (__CLPK_integer) m->numberOpcodes;
65         float * vec, * result;
66         vec = (float*) malloc( sizeof(float) * m->numberOpcodes );
67         result=(float*)malloc( sizeof(float) * m->numberOpcodes );
68
69         // populate the vector...
70         populate_vector( vec, m->numberOpcodes, m->opcodes, argv[1] );
71
72         cblas_sgemv( CblasColMajor, CblasNoTrans, mn, mn, alpha, m->elements,
73                     mn, vec, 1, beta, result, 1 );
74
75         char * chosen_opcode = choose_opcode( result, m->numberOpcodes, m->opcodes );
76         printf("%s\n",chosen_opcode);
77         // print_vector( result, m->numberOpcodes );
78
79         free( vec );
80         free( result );
81         destroy_matrix( m );
82         return 0;
83     }
84
85     matrix * a = read_matrix_from_file( "matrix.txt" );
86     __CLPK_integer dim = (__CLPK_integer) a->numberOpcodes;
87     float * testA = a->elements;
88
89     __CLPK_integer n = 3;
90
91     float A[9] = { 2.0f, 0.0f, 0.0f, // first column
92                   0.0f, -1.0f, 0.0f, // second column
93                   0.0f, 0.0f, 2.0f };
94
95     __CLPK_integer ipiv[3];
96

```

```

97     sgetrf_(&n, &n, A, &n, ipiv, &info);
98     printf(" dim = %d, n = %d\n", dim, n);
99     if ( info != 0 ) {
100         printf("sgetrf failed with error code %d\n", (int)info);
101         return 0;
102     }
103
104     __CLPK_integer inc = 1;
105
106     float c[3] = {1.0f, 1.0f, 1.0f };
107     float b[3] = {2.0f, -3.0f, 4.0f };
108
109     char transpose = 'N';
110     __CLPK_integer nrhs = 1;
111
112     sgetrs_(&transpose, &n, &nrhs, A, &n, ipiv, b, &n, &info);
113     printf(" n = %d\n", n);
114     if ( info != 0 ) {
115         printf("sgetrs failed with error code %d\n", (int)info);
116         return 0;
117     }
118
119     printf("b = [ %f %f %f ]\n", b[0], b[1], b[2]);
120     printf("c = [ %f %f %f ]\n", c[0], c[1], c[2]);
121     printf("\n");
122
123     printf(" testA * b \n");
124     cblas_sgemv( CblasColMajor, CblasTrans, n, n, alpha, testA, n, b, 1, beta, c, 1);
125     printf("x = [ %f %f %f ]\n", c[0], c[1], c[2]);
126
127     printf(" b \n");
128     printf("x = [ %f %f %f ]\n", b[0], b[1], b[2]);
129
130     printf(" scaling b by 1.5 \n");
131     cblas_sscal( n, 1.5f, b, 1 );
132     printf("x = [ %f %f %f ]\n", b[0], b[1], b[2]);
133
134     printf(" A * b \n");
135     cblas_sgemv( CblasColMajor, CblasNoTrans, n, n, alpha, A, n, b, 1, beta, c, 1 );
136     printf("x = [ %f %f %f ]\n", c[0], c[1], c[2]);
137     destroy_matrix( a );
138     return 0;
139 }
140
141
142 matrix *read_matrix_from_file( const char * filename )
143 {
144     FILE * fhandle = fopen( filename, "r" );
145     if ( fhandle == NULL )
146         return NULL;
147
148     matrix * pmtrix = (matrix*) malloc( sizeof(matrix) );
149     float num;
150
151     // read in the first line, containing the opcodes...
152     unsigned int commas = 0, linelength = 0;
153     int c;
154
155     while ( (c = fgetc(fhandle)) != EOF ) {
156         linelength++;
157         if ( c == ',' ) commas++;
158         if ( c == '\n' ) break;
159     }
160
161     // rewind the file...

```

```

162     rewind(fhandle);
163     pmtrx->numberOpcodes = commas+1;
164     pmtrx->numberElements= pmtrx->numberOpcodes * pmtrx->numberOpcodes;
165     pmtrx->opcodes = (char**) malloc( sizeof(char*) * pmtrx->numberOpcodes );
166     pmtrx->elements= (float*) malloc( sizeof(float) * pmtrx->numberElements );
167
168     char * cptr, * line = (char*) malloc( sizeof(char) * linelength );
169     fgets( line, linelength, fhandle );
170
171     // read in our opcodes...
172     cptr = strtok( line, "," );
173     c = 0;
174     while ( cptr != NULL )
175     {
176         pmtrx->opcodes[c++] = strdup( cptr );
177         cptr = strtok(NULL, ",");
178     }
179
180     free( line );
181
182     c = 0;
183     while ( fscanf(fhandle, "%f ", &num) )
184     {
185         pmtrx->elements[c++] = num;
186         if ( c == pmtrx->numberElements )
187             break;
188     }
189
190     fclose(fhandle);
191     // done...
192
193     return pmtrx;
194
195 }
196
197 void print_matrix( FILE * file, matrix * m )
198 {
199     int i, j;
200     int num = (int) sqrtf( m->numberElements );
201     for (i=0; i<m->numberOpcodes; i++)
202         fprintf(file, "%8s ", m->opcodes[i] );
203     fprintf(file, "\n");
204
205     for (i=0; i<num; i++)
206     {
207         for (j=0; j<num; j++)
208             fprintf(file, "%8f ", m->elements[i*num+j]);
209         fprintf(file, "\n");
210     }
211 }
212
213 int destroy_matrix( matrix * m )
214 {
215     int i;
216     for (i=0; i<m->numberOpcodes; i++)
217         free( m->opcodes[i] );
218
219     free( m->elements );
220     free( m->opcodes );
221
222     m->numberOpcodes = 0;
223     m->numberElements= 0;
224
225     m = NULL;
226

```

```

227     return 0;
228 }
229
230 void populate_vector( float * vector,
231                     int nels,
232                     char ** opcodes,
233                     char * code )
234 {
235     int i;
236     for (i=0; i<nels; i++) {
237         if ( strcmp(opcodes[i],code,4) == 0 )
238             vector[i] = 1.0f;
239         else
240             vector[i] = 0.0f;
241     }
242     // end of function
243 }
244
245
246 void print_vector(float * result, int nels)
247 {
248     int i;
249     for (i=0; i<nels; i++)
250         printf(" %8f ",result[i]);
251 }
252
253 void print_result(float * result, int nels, char **opcodes )
254 {
255     int i;
256     // print out json result, can import into python easily...
257     printf("{");
258     for ( i=0; i<nels; i++ )
259     {
260         if ( result[i] > 0.0f )
261             printf("'s':%f",opcodes[i],result[i]);
262     }
263     printf("}");
264 }
265
266
267 char *choose_opcode( float * result, int nels, char **opcodes )
268 {
269     // get a rand num [0,1]
270     float num = ( (float) rand() / (float) RAND_MAX );
271     float sum = 0.0f;
272     int i;
273     for (i=0; i<nels; i++)
274     {
275         sum += result[i];
276         if ( num < sum )
277             return opcodes[i];
278     }
279     return NULL;
280 }

```

Note the use of the BLAS routines, for matrix calculations. The *cblas* routines allow hardware acceleration, allowing us to leverage SIMD processors.

B. Android

Listing 17: Simple NDK usage for rendering

```

1 #include <stdio.h>

```

```

2  #include <stdlib.h>
3  #include <math.h>
4
5  #include <jni.h>
6  #include <android/log.h>
7  #include <GLES2/gl2.h>
8  #include <GLES2/gl2ext.h>
9
10 #include <sys/types.h>
11 #include <android/asset_manager.h>
12 #include <android/asset_manager_jni.h>
13 #include <assert.h>
14
15 #define LOG_TAG    "libglsl2"
16 #define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)
17 #define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_TAG, __VA_ARGS__)
18
19
20 static void printGLString( const char *name, GLenum s )
21 {
22     const char *v = (const char*) glGetString(s);
23     LOGI("GL %s = %s\n", name, v);
24 }
25
26 static void checkGlError( const char * op )
27 {
28     GLint error;
29     for (error = glGetError(); error; error = glGetError() ) {
30         LOGI("after %s() glError (0x%x)\n", op, error );
31     }
32 }
33
34 int screenWidth;
35 int screenHeight;
36
37 int analyzeFrameBuffer( unsigned char * data )
38 {
39     int i, j;
40     FILE * tmpfile = fopen("/sdcard/tmp/results.txt", "w");
41     if ( tmpfile == NULL )
42         return 1;
43
44     for (i=0; i<screenHeight; i++)
45     {
46         for (j=0; j<screenWidth; j++)
47         {
48             fprintf(tmpfile, " %c ", data[i*screenWidth+j]);
49         }
50         fprintf(tmpfile, "\n");
51     }
52     LOGI("WROTE GPU buffer to file");
53     fclose( tmpfile );
54
55     return 0;
56 }
57
58 GLuint loadShader(GLenum shaderType, const char* pSource) {
59     GLuint shader = glCreateShader(shaderType);
60     if ( shader ) {
61         glShaderSource(shader, 1, &pSource, NULL);
62         glCompileShader( shader );
63         GLint compiled = 0;
64         glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
65         if ( !compiled ) {
66             GLint infoLen = 0;

```



```

67     glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);
68     if ( infoLen ) {
69         char * buf = (char*) malloc( infoLen );
70         if ( buf ) {
71             glGetShaderInfoLog(shader, infoLen, NULL, buf);
72             LOGE("Could not compile shader %d:\n%s\n", shaderType, buf);
73             free(buf);
74         }
75         glDeleteShader(shader);
76         shader = 0;
77     }
78 }
79 }
80 return shader;
81 }
82
83 GLuint createProgram( const char * pVertexSource, const char * pFragmentSource )
84 {
85     GLuint vertexShader = loadShader( GL_VERTEX_SHADER, pVertexSource );
86     if ( !vertexShader ) {
87         return 0;
88     }
89
90     GLuint pixelShader = loadShader( GL_FRAGMENT_SHADER, pFragmentSource );
91     if ( !pixelShader ) {
92         return 0;
93     }
94
95     GLuint program = glCreateProgram();
96     if ( program ) {
97         glAttachShader(program, vertexShader);
98         checkGlError("glAttachShader");
99         glAttachShader(program, pixelShader );
100        checkGlError("glAttachShader");
101        glLinkProgram(program);
102        GLint linkStatus = GL_FALSE;
103        glGetProgramiv(program, GL_LINK_STATUS, &linkStatus);
104        if ( linkStatus != GL_TRUE ) {
105            GLint bufLength = 0;
106            glGetProgramiv(program, GL_INFO_LOG_LENGTH, &bufLength);
107            if ( bufLength ) {
108                char * buf = (char*) malloc( bufLength );
109                if ( buf ) {
110                    glGetProgramInfoLog(program, bufLength, NULL, buf);
111                    LOGE("Could not link program:\n%s\n", buf);
112                    free(buf);
113                }
114            }
115            glDeleteProgram(program);
116            program = 0;
117        }
118    }
119    return program;
120 }
121
122
123 GLuint gProgram;
124 GLuint gvPositionHandle;
125
126 char *readShader( const char *filename )
127 {
128
129 }
130
131 char * gVertexShader;

```

```

132 char * gFragmentShader;
133
134 int setupGraphics(int w, int h) {
135
136     printGLString("Version", GL_VERSION);
137     printGLString("Vendor", GL_VENDOR);
138     printGLString("Renderer", GL_RENDERER);
139     printGLString("Extensions", GL_EXTENSIONS);
140
141     screenWidth = w;
142     screenHeight = h;
143
144     LOGI("setupGraphics(%d,%d)", w, h);
145     gProgram = createProgram( gVertexShader, gFragmentShader );
146     if ( !gProgram ) {
147         LOGE("Could not create program.");
148         return 0;
149     }
150
151     gvPositionHandle = glGetAttribLocation(gProgram, "vPosition");
152     checkGLError("glGetAttribLocation");
153     LOGI("glGetAttribLocation(\"vPosition\") = %d\n", gvPositionHandle);
154
155     glViewport(0, 0, w, h );
156     checkGLError("glViewport");
157     return 1;
158 }
159
160
161 const GLfloat gVertices[] = {0.0f, 0.5f, -0.5f, -0.5f, 0.5f, -0.5f };
162
163 void renderFrame( ) {
164
165     unsigned char * bufferdata;
166
167     glClearColor(0,0,0,1.0f);
168     checkGLError("glClearColor");
169     glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );
170     checkGLError("glColor");
171
172     glUseProgram(gProgram);
173     checkGLError("glUseProgram");
174
175     glVertexAttribPointer( gvPositionHandle, 2, GL_FLOAT, GL_FALSE, 0, gVertices );
176     checkGLError("glVertexAttribPointer");
177     glEnableVertexAttribArray(gvPositionHandle);
178     checkGLError("glEnableVertexAttribArray");
179     glDrawArrays( GL_TRIANGLES, 0, 3 );
180     checkGLError("glDrawArrays");
181
182     LOGI("Called draw functions");
183     LOGI("Screen width,height are %d,%d", screenWidth, screenHeight);
184     // post draw call, we call read pixels to obtain our result
185     // note that we need depth of 4...
186     bufferdata = (unsigned char*) malloc( sizeof(unsigned char) * screenWidth * screenHeight←
        * 4 );
187     glReadPixels( 0, 0, screenWidth, screenHeight, GL_RGBA, GL_UNSIGNED_BYTE, bufferdata );
188
189     LOGI("Read Pixels");
190
191     // pass this buffer data for analysis...
192     analyzeFrameBuffer( bufferdata );
193     LOGI("Analyzed...");
194
195     // clean up

```

```

196     free( bufferdata );
197
198 }
199
200 /*
201  JNIEXPORT void JNICALL Java_com_gotottheboard_gls12_NativeGLLib_init(JNIEnv * env, ↵
        jobject obj,
202                jint width, jint height);
203  JNIEXPORT void JNICALL Java_com_gotottheboard_gls12_NativeGLLib_step(JNIEnv * env, ↵
        jobject obj );
204  JNIEXPORT void JNICALL Java_com_gotottheboard_gls12_NativeGLLib_setupShaders(JNIEnv * env↵
        ,
205                jstring vshader,
206                jstring fshader);
207 */
208
209 JNIEXPORT void JNICALL Java_com_gotottheboard_nativegl_NativeGLLib_init(JNIEnv * env, ↵
        jobject obj,
210                jint width, jint height)
211 {
212     LOGI("Setting up native graphics code.");
213     setupGraphics( width, height );
214     LOGI("Finished setting up native graphics code.");
215 }
216
217 JNIEXPORT void JNICALL Java_com_gotottheboard_nativegl_NativeGLLib_step(JNIEnv * env, ↵
        jobject obj )
218 {
219     LOGI("Step...\n");
220     renderFrame( );
221 }
222
223 JNIEXPORT void JNICALL Java_com_gotottheboard_nativegl_NativeGLLib_setupShaders(JNIEnv * ↵
        env, jobject obj,
224                jobject assetManager,
225                jstring vshader,
226                jstring fshader)
227 {
228     int vid, fid;
229     off_t start, length;
230
231     LOGI("Starting to read shaders...");
232
233     const jbyte * utf_vshader = (*env)->GetStringUTFChars( env, vshader, NULL );
234     const jbyte * utf_fshader = (*env)->GetStringUTFChars( env, fshader, NULL );
235
236     LOGI("Created UTF strings..");
237
238     // create an asset manager
239     AAssetManager * mgr = AAssetManager_fromJava(env, assetManager);
240     assert(NULL != mgr );
241     if ( mgr == NULL )
242         LOGI("Unable to create asset manager");
243     else
244         LOGI("Created asset manager");
245
246     // read in vertex fragment
247     LOGI("Going to read %s", (char*)utf_vshader);
248     LOGI("Going to read %s", (char*)utf_fshader);
249     AAsset * vertex_asset = AAssetManager_open( mgr, (const char*) utf_vshader, ↵
        AASSET_MODE_UNKNOWN );
250     AAsset * frag_asset = AAssetManager_open( mgr, (const char*) utf_fshader, ↵
        AASSET_MODE_UNKNOWN );
251     LOGI("Created asset objects");
252     if ( vertex_asset == NULL )

```

```

253     LOGI("Vertex asset could not be opened.");
254     if ( frag_asset == NULL )
255         LOGI("Frag asset could not be opened.");
256
257     // release the Java string and UTF
258     (*env)->ReleaseStringUTFChars( env, vshader, utf_vshader );
259     (*env)->ReleaseStringUTFChars( env, fshader, utf_fshader );
260     LOGI("Released UTF8 strings");
261
262     // get file descriptor -- failing here with the vid = -1
263     vid = AAsset_openFileDescriptor(vertex_asset, &start, &length );
264     LOGI("vid = %d",vid);
265     assert( 0 <= vid );
266     gVertexShader = (char*) malloc( sizeof(char) * length + 1 );
267     memset(gVertexShader, '\0', length+1);
268     AAsset_read( vertex_asset, gVertexShader, length );
269     AAsset_close(vertex_asset);
270     LOGI("Read in vertex shader:\n %s\n",gVertexShader);
271
272     fid = AAsset_openFileDescriptor( frag_asset, &start, &length );
273     assert( 0 <= fid );
274     gFragmentShader = (char*) malloc( sizeof(char) * length + 1 );
275     memset(gFragmentShader, '\0', length+1);
276     AAsset_read( frag_asset, gFragmentShader, length );
277     AAsset_close(frag_asset);
278     LOGI("Read in frag shader:\n %s\n",gFragmentShader);
279
280 }

```

To use the eigen library, a similar effort to leverage the NEON processor, we must include the C++ libraries, which is fairly trivial but does require an *Application.mk* file

Listing 18: To Move into Vectorized C++ for the NEON Processor

```
1 APP_STL:=stlport_static
```

Along with the inclusion of the Eigen header files (Eigen is entirely made up of header files) allows us to better leverage vectorized calculations.

Listing 19: "Cuda Example Program"

```

1 #include <cuda.h>
2 #include <cuda_runtime.h>
3 #include <driver_types.h>
4 #include <stdio.h>
5 /*
6  nvcc square.cu -o square
7
8  results in:
9
10 0 0.000000
11 1 1.000000
12 2 4.000000
13 3 9.000000
14 4 16.000000
15 5 25.000000
16 6 36.000000
17 7 49.000000
18 8 64.000000
19 9 81.000000
20
21 */
22

```

```

23 __global__ void
24 square_array(float *a, int N) {
25     int idx = blockIdx.x * blockDim.x + threadIdx.x;
26     if ( idx < N ) a[idx] = a[idx] * a[idx];
27 }
28
29 int main( void ) {
30     float *a_h, *a_d;
31     int i;
32     const int N = 10;
33     size_t size = N * sizeof(float);
34     a_h = (float*) malloc( size );
35     cudaMalloc( (void**) &a_d, size );
36
37     // initialize and send to device
38     for (i=0; i<N; i++)
39         a_h[i] = (float) i;
40     cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
41
42     // do the device calculation
43     int block_size = 4;
44     int n_blocks = N / block_size + ( N % block_size == 0 ? 0 : 1 );
45     square_array<<<n_blocks, block_size>>>(a_d,N);
46
47     // get result
48     cudaMemcpy(a_h,a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
49
50     // print the results
51     for (int i=0; i<N; i++)
52         printf("%d %f\n",i,a_h[i]);
53
54     free(a_h);
55     cudaFree(a_d);
56
57     return 0;
58 }

```

OpenCL is part of the developer's environment for Mac OS X; and is part of the XCode download.

Listing 20: "OpenCL Example"

```

1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <math.h>
6  #include <unistd.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <OpenCL/opencl.h>
10
11 /*
12  * minimal OpenCL program... to build (on Mac OS X (lion)
13  * g++ example.cpp -o example -framework OpenCL
14  *
15  * Execution results in squaring of the input array [0,1,2,3...]
16  * Becomes [0,1,4,9,...]
17  */
18
19 // simple example, just use static data size
20 #define DATA_SIZE 1024
21
22 // simple kernel that computes the square of an input array
23 const char *KernelSource = "\n" \
24     "__kernel void square(

```

```

25     "    __global float* input,                \n" \
26     "    __global float* output,              \n" \
27     "    const unsigned int count)            \n" \
28     "{                                         \n" \
29     "    int i = get_global_id(0);              \n" \
30     "    if ( i < count )                      \n" \
31     "        output[i] = input[i]*input[i];    \n" \
32     "}"                                         \n" \
33     "\n";
34
35 // main
36
37 int main(int argc, char *argv[] )
38 {
39     int err; // error code
40
41     float data[DATA_SIZE];
42     float results[DATA_SIZE];
43     unsigned int correct;
44
45     size_t global;
46     size_t local;
47
48     cl_device_id device_id;
49     cl_context context;
50     cl_command_queue queue;
51     cl_program program;
52     cl_kernel kernel;
53
54     cl_mem input;
55     cl_mem output;
56
57     // get data to operate on...
58     int i=0;
59     unsigned int count = DATA_SIZE;
60     for (i=0; i<count; i++)
61         data[i] = i;
62
63     // get an id for the device
64     int gpu = 1;
65     err = clGetDeviceIDs(NULL, gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU,
66         1, &device_id, NULL );
67
68     if ( err != CL_SUCCESS )
69     {
70         printf("Unable to obtain device id\n");
71         return -1;
72     }
73
74     // create device context
75     context = clCreateContext(0,1,&device_id,NULL,NULL,&err);
76     if ( !context )
77     {
78         printf("Unable to create device context\n");
79         return -2;
80     }
81
82     // create queueing context
83     queue = clCreateCommandQueue(context, device_id, 0, &err );
84     if ( !queue )
85     {
86         printf("Unable to create CL queue\n");
87         return -3;
88     }
89

```

```

90 // create program
91 program = clCreateProgramWithSource(context, 1,
92     (const char**) &KernelSource,
93     NULL, &err);
94
95 // build the program executable
96 err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
97 if (err != CL_SUCCESS)
98 {
99     size_t len;
100     char buffer[1024];
101
102     printf("Unable to build CL program\n");
103
104     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
105         sizeof(buffer), buffer, &len);
106     printf("%s\n", buffer);
107
108     return -4;
109 }
110
111 // create the kernel
112 kernel = clCreateKernel(program, "square", &err);
113 if (!kernel || err != CL_SUCCESS)
114 {
115     printf("Unable to create kernel\n");
116     return -5;
117 }
118
119 // create input and output arrays in device memory..
120 input = clCreateBuffer(context, CL_MEM_READ_ONLY,
121     sizeof(float) * count,
122     NULL, NULL);
123 output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
124     sizeof(float) * count,
125     NULL, NULL);
126
127 if (!input || !output)
128 {
129     printf("Failed to create input or output streams\n");
130     return -6;
131 }
132
133 // write our inputs into device memory...
134 err = clEnqueueWriteBuffer(queue, input, CL_TRUE, 0,
135     sizeof(float) * count,
136     data, 0, NULL, NULL);
137 if (err != CL_SUCCESS)
138 {
139     printf("Unable to write inputs to device\n");
140     return -7;
141 }
142
143 // set arguments
144 err = 0;
145 err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
146 err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
147 err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
148 if (err != CL_SUCCESS)
149 {
150     printf("Unable to set kernel arguments\n");
151     return -8;
152 }
153
154

```

```

155 // get maximum work-group size...
156 err = clGetKernelWorkGroupInfo( kernel, device_id,
157     CL_KERNEL_WORK_GROUP_SIZE,
158     sizeof(size_t), &local, NULL);
159 if ( err != CL_SUCCESS )
160 {
161     printf("Unable to obtain maximum work group size\n");
162     return -9;
163 }
164
165 // execute kernel over range of data set!
166 global = count;
167 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
168     &global, &local, 0, NULL, NULL);
169
170
171 // wait for the command queue to get serviced before reading
172 // back results
173 clFinish(queue);
174
175
176 // read results from the device...
177 err = clEnqueueReadBuffer(queue, output, CL_TRUE, 0,
178     sizeof(float) *count, results,
179     0, NULL, NULL );
180 if ( err != CL_SUCCESS )
181 {
182     printf("Unable to read results back from device\n");
183     return -10;
184 }
185
186 printf("results[1]      = %f\n",results[1]);
187 printf("results[10]     = %f\n",results[10]);
188 printf("results[11]     = %f\n",results[11]);
189
190 // shut down and clean up...
191 clReleaseMemObject(input);
192 clReleaseMemObject(output);
193 clReleaseProgram(program);
194 clReleaseKernel(kernel);
195 clReleaseCommandQueue(queue);
196 clReleaseContext(context);
197
198 return 0;
199
200 }

```

Listing 21: "OpenCL via Python"

```

1 #!/bin/env python
2 import pyopencl as cl
3 import numpy
4 import numpy.linalg as la
5
6 a = numpy.random.rand(50000).astype(numpy.float32)
7 b = numpy.random.rand(50000).astype(numpy.float32)
8
9 ctx = cl.create_some_context()
10 queue = cl.CommandQueue(ctx)
11
12 mf = cl.mem_flags
13 a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
14 b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
15 dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

```



```
16
17 prg = cl.Program(ctx, """
18     __kernel void sum(__global const float * a,
19     __global const float *b, __global float *c)
20     {
21         int gid = get_global_id(0);
22         c[gid] = a[gid] + b[gid];
23     }
24     """).build()
25
26 prg.sum( queue, a.shape, None, a_buf, b_buf, dest_buf )
27
28 a_plus_b = numpy.empty_like(a)
29 cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
30 print la.norm(a_plus_b - (a+b))
```