GPUs for Malware and More on Mobile Devices

Jared Carlson

CONTENTS

J. Carlson is a Senior Engineer with viaForensics.
E-mail: jared.carlson23@gmail.com
Manuscript developed as part of DARPA CFT grant

List of Figures

List of Tables

Listings

Abstract

This paper, produced as part of a DARPA CFT grant, discusses how to use the GPU and other vectorized processors for cybers 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 [2]. 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.

D. *Encryption*

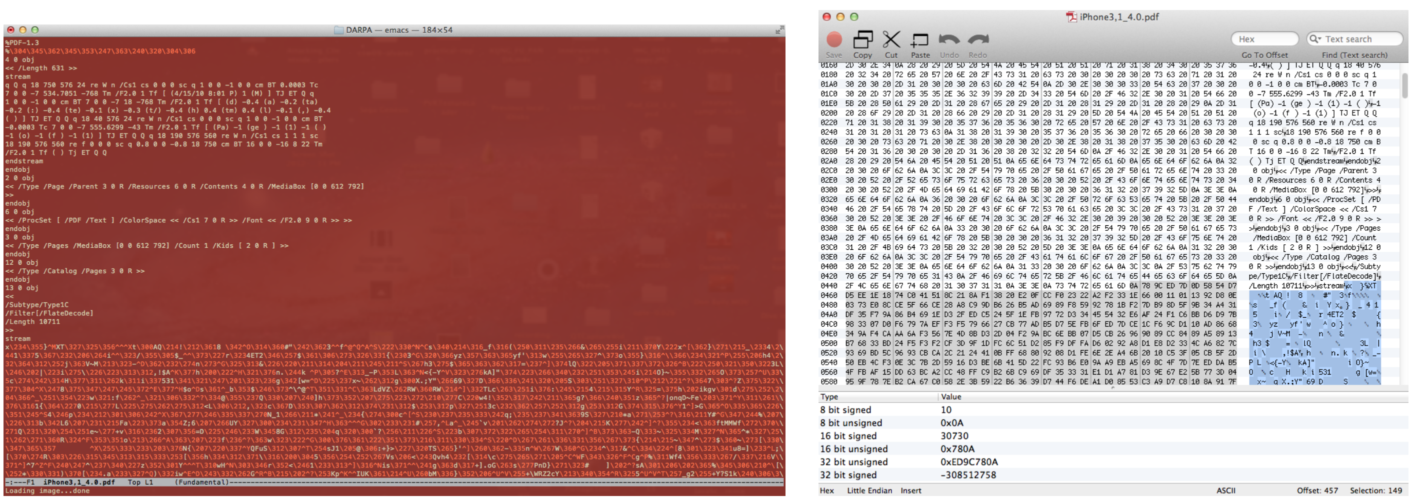
Malware decryption, the unpacking of a virus by the GPU, was one of the first noted uses for the GPU within the hacking community. However, here we explore encrypting live memory objects, data blobs and other sensitive parameters to obfuscate the code and help mitigate an attack.

III. Signature Checking

Signature checking is a staple of malware detection schemes often used in commercial anti-virus software. In our work, we investigate how can use the GPU to augment our ability to find and detect malicious signatures with a payload or other binary fragment.

It's worth mentioning a significant advantage of using a shader is our ability to dynamically compile the shader and create a GPU program to perform the signature check. In other words, even within a third party application we can replace text or insert text into a database. This text, can be used in the form of a shader and would then be compiled when the application next launches, giving the application a mechanism for Over-The-Air updates or other less intrusive ways to modify an existing defense.

As an example of recognizing a malicious signature, let's look at the star jailbreak, developed by *@comex*. While the vulnerability has been fixed, we can still use this an example for signature checking. If we look at the payload using emacs and a hex editor (in this case, 0xED), we can quickly see the binary payload to trigger the exploit.



((a)) Emacs View of Star Exploit ((b)) Binary View of Star Exploit

Fig. 1: Looking at an Attack Signature

The basic approach can be thought of as a sliding mask where we compare the result of texture operations. With this in mind, it's a relatively straightforward task to see if we can detect the binary signature of the exploit using a shader. The procedure can be outlined as:

1) Bind data, in the form of a payload or raw memory, to scan to texture
2) Bind signatures to masking texture atlas for testing
3) Use a shader to sweep our mask over the payload, using texture operations to signal for a match
4) If a suspicious entry, then inform/act

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

```
// load a signature
- (GLuint) loadSignatureFromFile:(NSString*)file
{
    NSString * pathToFile   = [[NSBundle mainBundle] pathForResource:file ofType:@"bin"];
    NSData * contentsOfFile = [[NSData alloc] initWithContentsOfFile:pathToFile];
    NSUInteger numBytes = [contentsOfFile length];

    NSUInteger pow = [self fitPowerOf2:(4 * numBytes / 3 )] * 2;

    if ( pow > (dimension*dimension) )
    {
        NSLog(@"File contents doesn't fit inside texture...");
        return 0;
    }

    unsigned char * data  = (unsigned char*) calloc( sizeof(unsigned char) * dimension*↩
        dimension*4, 1 );
    unsigned char * buffer = (unsigned char*) [contentsOfFile bytes];
    NSUInteger index = 0;

    for (NSUInteger y = 0; y < dimension; y++ ) {
        for (NSUInteger x = 0; x < dimension; x++) {
            int byteIndex = (dimension*4 * y) + x * 4;

            if ( index > numBytes ) goto endloop;
            memcpy(&data[byteIndex], &buffer[index], 3);
            data[byteIndex+3] = 0xff;
            index += 3;
        }
    }

endloop:

    GLuint texName;
    glGenTextures(1, &texName);
    glBindTexture(GL_TEXTURE_2D, texName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, dimension, dimension, 0, GL_RGBA, ↩
        GL_UNSIGNED_BYTE, data);

    return texName;
}
```

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

```

// relevant part of frag shader
// offset is a uniform we control from the CPU, allowing us to "sweep"
 gl_FragColor =  texture2D( Texture, TexCoordOut ) - texture2D( Mask, TexCoordOut + offset↩
     );

// placing breakpoint immediately after:
glReadPixels(0, 0, dimension, dimension, GL_RGBA, GL_UNSIGNED_BYTE, bytes);
//
// offset (0,0)
```

```
(gdb) x/20 bytes
0x6f33000:  0xff000000  0xffac420f  0xff002047  0xff000000
0x6f33010:  0xff090000  0xff5b1a00  0xff000000  0xff1c002c
0x6f33020:  0xff761900  0xff080900  0xff000001  0xff250f00
0x6f33030:  0xff000e00  0xff0d0013  0xff432500  0xff00005e
0x6f33040:  0xff1d8c1e  0xffc23646  0xffc36bb4  0xffbdc2b5

// offset(x,0)
(gdb) x/20 bytes
0x6f33000:  0xff070503  0xff000000  0xff000000  0xff000000
0x6f33010:  0xff000000  0xff000000  0xff000000  0xff000000
0x6f33020:  0xff000000  0xff000000  0xff000000  0xff000000
0x6f33030:  0xff000000  0xff000000  0xff000000  0xff000000
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

```
texture ← payload-to-check
Create atlas of various signatures
second-texture ← atlas-of-signatures
draw-call initiates GPU calculation
for each texel do
    The FOR loop is implicit as shader takes care of this
    for each atlas region do
        Frag Operation with i-th atlas region
    end for
end for
```

An illustration serves best...

IV. Memory Analysis

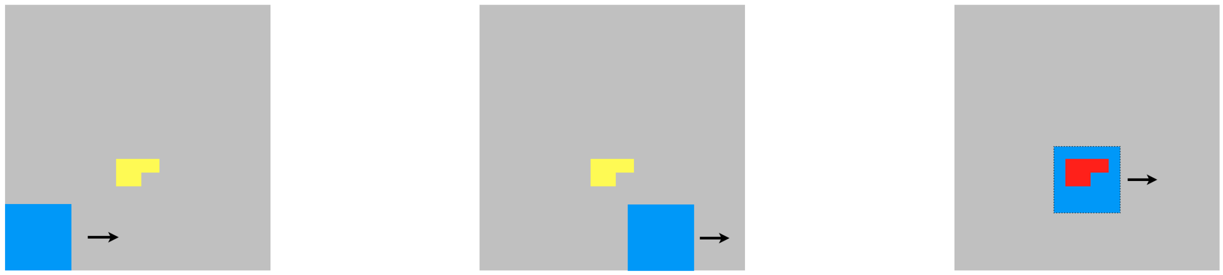
A common tactic for exploitation is a heap spray or some other data corruption. With access to dynamic memory we can investigate structures and other objects that may be compromised due to a bug, heap spray, and eventually lead to exploitation due to a NULL dereferencing or similar bug. This strategy 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

```

class MemoryObject
{
```



((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

```
    unsigned char leading[8];
    vector<string> objects;
    // other objects
    unsigned char trailing[8];


    // static methods
    static void Generator(unsigned char ptr[8]) {
        static unsigned char start = 0;
        const unsigned char interval = 0x02;
        start += interval;
        for (int i=0; i<8; i++)
            ptr[i] = start;

    }

public:

    // constructor
    MemoryObject(void) { Generator(leading); objects.push_back("Testing"); Generator(trailing);  }

    // destructor
    ~MemoryObject(void){
        memset(leading, '\0', 8);
        memset(trailing,'\0', 8);

    }

};
```

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

Listing 4: Watching the Heap

```

(gdb) list 295
290
291 // populate the heap..
292 - (void) populateHeap
293 {
294     // current pointer to dynamic memory, here, I'll actually just use the first object I want to track.
295     heapPtr = (void*) malloc( 0 );
296
297     // create a couple of objects following our pointer so that we can track them...
298     first  = new MemoryObject;
299     second = new MemoryObject;

(gdb) x/20 first
0x6d40330:  0x02020202  0x02020202  0x06d40350  0x06d40354
0x6d40340:  0x06d40354  0x04040404  0x04040404  0x00000000
0x6d40350:  0x06d3fc5c  0x00000000  0x00000000  0x00030000
0x6d40360:  0x07b91364  0x07b91364  0x07b91000  0x07b91378
0x6d40370:  0x07b91378  0x07b91378  0x07b91378  0x07b91378


(gdb) x/20 second
0x6d3fc70:  0x06060606  0x06060606  0x06d40460  0x06d40464
0x6d3fc80:  0x06d40464  0x08080808  0x08080808  0x00020000
0x6d3fc90:  0x012adb70  0x010012c0  0x01000009  0x00000002
0x6d3fca0:  0x00000000  0x00040000  0x012b0b40  0x06d3fcc0
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,

```
(gdb) x/20 first
0x6d40330: 0x02020202  0x02020202  0x06d40350  0x06d40354
0x6d40340: 0x06d40354  0x04040404  0x04040404  0x00000000
0x6d40350: 0x06d3fc5c  0x00000000  0x00000000  0x00030000
0x6d40360: 0x07b91364  0x07b91364  0x07b91000  0x07b91378
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.

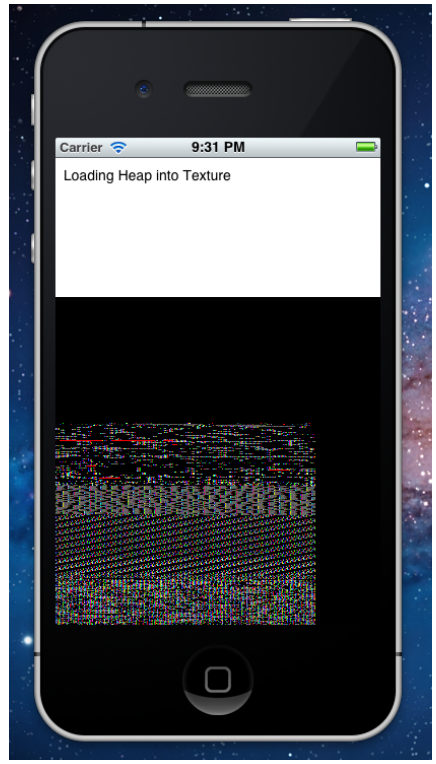


Fig. 3: Sample Screenshot of Memory as a Texture

Listing 5: GL code to read in our GPU results

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

Listing 6: Simple GDB Capture of GPU results

```
(gdb) x/10 bytes
0xb7fb000: 0xff020202  0xff020202  0xff204000  0xff06de5b
0xb7fb010: 0xffde5b24  0xff5b2406  0xff0406de  0xff040404
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

```
varying lowp vec4 DestinationColor;

varying lowp vec2 TexCoordOut;
uniform sampler2D Texture;
uniform sampler2D Mask;
uniform lowp vec2 offset;

// assume we have our signature in four bytes or less...
bool isValidSignature( lowp vec4 pixel )
{
    lowp float norm;
    norm = dot(pixel.rgb , pixel.rgb );
    if ( norm > 0.0 )
    {
      // for now we just see if the all RGB channels match, this means
      if ( pixel.r == pixel.g && pixel.g == pixel.b  )
          return true;
    }
    return false;
}

// basic shader to operate
// on multiple texture operations..
void main(void) {

    if ( isValidSignature( texture2D(Texture,TexCoordOut) ) )
    {
        gl_FragColor    = vec4( 1, 0, 0, 1 );
    }
    else
    {
        gl_FragColor    = vec4( 0, 0, 0, 1 ) * texture2D( Texture, TexCoordOut);
    }

}
```

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

```
(gdb) x/100 bytes
0x6f99000:  0xff0000ff  0xff0000ff  0xff000000  0xff000000
0x6f99010:  0xff000000  0xff000000  0xff000000  0xff0000ff
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 [3], 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

```
#!/bin/bash

# test to make sure the instructions directory is here..
if [ ! -d "instruction-contents" ]; then
    echo "[+] Creating instruction contents directory"
    mkdir "instruction-contents"
fi


# otool -t simple | cut -d " " -f 2-4
for i in `ls`
do
    type=`file $i`
    needle='executable arm'
    if [[ "$type" == *"$needle"* ]]; then
        echo "Found arm executable -> $type"
        # parse out the file name we'll write to...
        fileprefix=`echo $type | cut -d':' -f 1`
        filename="${fileprefix}.instr"
        `otool -t $fileprefix | cut -d " " -f 2-5 > "instruction-contents/${filename}"`
    fi
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 diassasembler predict the correct instruction.

Listing 10: Sample Disassembly

```
$ python simulate_dynamic_dissembly.py
Read  44  instructions from the file  sim.instructions
replacing 0x3c
 with 0x00

replacing 0xaf
 with 0xaf

replacing 0x00
 with 0x00

replacing 0x28
 with 0x28

replacing 0x58
 with 0x58

replacing 0x08
 with 0x08

replacing 0x00
 with 0x00

* * *  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

```
#!usr/env python

import subprocess, random, simplejson
# to install simplejson: pip install simplejson

# First we read in our instruction file that we want to use
filename = 'sim.instructions'
instructions = open(filename).readlines()
print "Read ",len(instructions)," instructions from the file ",filename

# now let's get rid of ~8 instructions (roughly 20% from our given file)
n = 8
random.seed(24) # for reproducability, if needed
samples = list(instructions)
for xx in xrange(0,n):
    index = random.randint(0,len(instructions)-1)
    samples[index] = 'xxx'

# Now we have a few unknown ops, let's use our C-code to on-the-fly generate
# disassembly using probability tables
last_instruction = False
application = './optimization'
args = ''
digraph = 'simple.digraph'
recreated = []
index = 0
for instr in samples:
    if ( instr == 'xxx' ):
        # call out...
        try:
            prob_instruction = subprocess.check_output([application,last_instruction,digraph])
            instr = prob_instruction
            print "replacing %s with %s " % (instructions[index],instr)
```

```
        except:
            print "Error in our C-code, time to debug..."
    # append...
    recreated.append(instr)
    last_instruction = instr
    index+=1

# Now we test our disassembly to see how much we got right...
errors = 0
for i in xrange(0,len(instructions)):
    if recreated[i] != instructions[i]:
        errors+=1

print "* * * ",errors," Errors in ",n," unknown operations * * *"
```

Listing 12: Main for creating disassembly tables

```
#include <iostream>
#include <algorithm>
#include <fstream>
#include <unistd.h>
#include <getopt.h>
#include "trie.h"
#include "util.h"
#include "controller.h"
#include "matrix.h"


using std::fstream;
using std::cout;
using std::endl;
using std::string;

int main( int argc, char *argv[] )
{

  /*
   * Argument error check
   */
  if ( argc< 2 )
    {
      cout << "Usage: " << argv[0] << " <infile> [additional file or file patterns]" << endl;
      exit(0);
    }

  /*
   * Parse argument
   */
  int option_index, c;
  bool dump_opcode   = false;
  bool combine_files = false;
  bool gen_digraph   = false;
  string outFileName = "temp.txt";
  string treeName    = "";
  bool dump_all      = false;

  static struct option options[] = {
    {"opcodes",0,0,0},
    {"outfile",1,0,0},
    {"combine",0,0,0},
    {"digraph",0,0,0},
    {"all",1,0,0},
    {NULL,0,NULL,0},
```

```
};

while( (c = getopt_long( argc, argv, "a:cgo:x", options, &option_index) ) != -1 )
  {
    switch ( c )
{
case 'a':
  cout << "Dumping all instructions.." << endl;
  dump_all = true;
  treeName = optarg;
  break;
case 'x':
  cout << "opcode output..." << endl;
  dump_opcode = true;
  break;
case 'o':
  cout << "outfile named..." << endl;
  outFileName = optarg;
  break;
case 'c':
  cout << "combine files..." << endl;
  combine_files = true;
  break;
case 'g':
  cout << "generating digraph..." << endl;
  gen_digraph = true;
  break;
default:
  cout << "unknown " << c << endl;
  break;

}

  }


vector<string> files_to_process = parse_args( argc, argv );

Controller * jobController = new Controller;
Trie * tree;

  for (auto fstring = files_to_process.begin();
 fstring != files_to_process.end(); ++fstring)
    {
if (!jobController->ProcessFile( (*fstring) ) )
  cout << "Error processing file " << (*fstring) << endl;
else {
  cout << "Processed file " << (*fstring) << endl;
  tree = jobController->TreeForFile( (*fstring) );
  cout << "Processed " << tree->NumberOfInstructions() << " instructions" << endl;
  cout << "Processed " << tree->NumberOfUniqueInstructions() << " unique instructions" << endl;
}

    }


  /*
   * look across our tries
   */
  size_t ninstructions = jobController->NumberOfUniqueInstructionAcrossTrees( );
  cout << "Batch: " << ninstructions << " unique instructions" << endl;
  cout << "Writing to file: " << outFileName << endl;

  if ( dump_all )
```

```
        {
  if ( dump_opcode ) jobController->WriteOpcodes();
  jobController->ListAllInstructionsToFile( treeName, outFileName );
        }
      else if ( dump_opcode && !gen_digraph )
        {
  jobController->WriteOpcodes();
  cout << "Dumping opcodes - mode: " << jobController->Mode() << endl;
  jobController->ListInstructionsToFile( outFileName );
        }
      else if ( gen_digraph )
        {
  cout << "Writing DiGraph..." << endl;
  if ( !jobController->WriteDiGraphToFile( outFileName ) )
    cout << "Unable to write digraph to " << outFileName << endl;
  else
    cout << "Written..." << endl;
        }
      else
        {
  jobController->ListInstructionsToFile( outFileName );
        }
      cout << "\n";

  /*
   * Cleanup
   */

  delete jobController;

  return 0;
}
```

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

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

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, cycript 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

INPUT: Uniforms as keys, etc; Shaders as encryption methods
Bind data to be encrypted to texture
Link frag shader as a ”block” encryption call
Draw call, renders encrypted data to off screen texture
if insufficient rounds **then**
    use newly rendered texture as input to above
    repeat
end if
if want to change block method **then**
    link new frag shader
    repeat using new shader program
end if
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. 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.

VIII. Optimizations and Other Tricks

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.

IX. 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 [4].

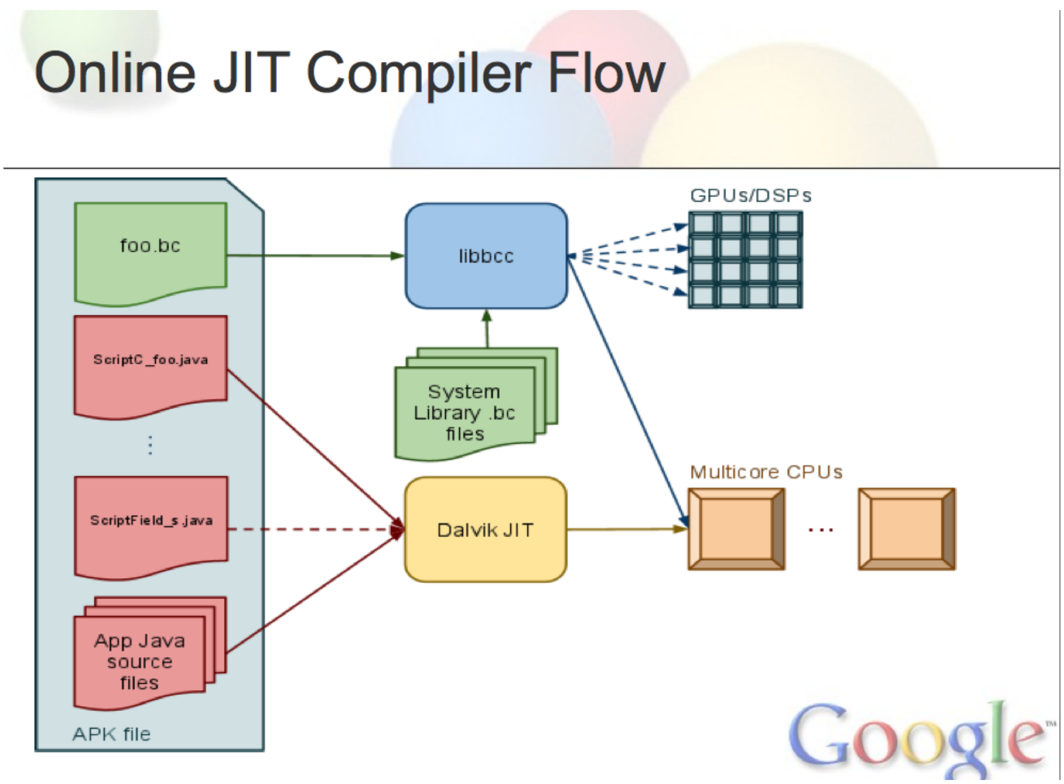


Fig. 4: Flow diagram of online JIT

For best results the ideal placement would be allow for ”on-the-fly” alteration of the bitcode (this would necissitate a recache of the newly compiled machine code for ICS as this a feature to save on-load time)

[1]For 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

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, propogation 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 4, which is a slide from Liao Shih-wei's presentation at *Linux Foundation: Collab-Summit*[4], and outlines the interplay of bitcode and reflected Java for interplay.

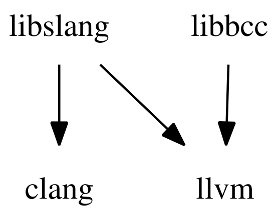


Fig. 5: Relationships between LLVM and Renderscript Core Libs

llvm-rs-cc is a driver on top of libslang 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 privilages. The architecture of libslang and libbcc is depicted in figure 5.

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

```

bool rsdProgramVertexInit(const Context *rsc, const ProgramVertex *pv,
                          const char* shader, uint32_t shaderLen) {
    RsdShader *drv = new RsdShader(pv, GL_VERTEX_SHADER, shader, shaderLen);
    pv->mHal.drv = drv;

    return drv->createShader();
}
```

These frameworks are non-existant 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.

X. 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).

XI. 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.

XII. 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.

XIII. 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.

XIV. Conclusion

A. Next Phase

The project will now move into an implementation phase, developing the ideas suggested in the minimal algorithm discussion and using OpenCL on the desktop as a method to prototype vectorized kernels and both integrate with and generate proper data sets.

XV. 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] G. Vasiliadis, M. Polychronakis, and S. Ioannidis *GPU-Assisted Malware* http://dcs.ics.forth.gr/Activities/papers/gpumalware.malware10.pdf
[3] Shah, Abhishek, *Approximate Diassembly using Dynamic Programming*, 2010, Master's Projects. Paper 8. http://scholarworks.sjsu.edu/etd_projects/8
[4] 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 OpenGLES framework.

Listing 15: Render Method for iOS project

```
- (void)render:(CADisplayLink*)displayLink
{
    glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_BLEND);

    glClearColor(0, 104.0/255.0, 55.0/255.0, 1.0);
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glEnable(GL_DEPTH_TEST);

    float top = 320.0f;
    float bottom = 0.0f;
    float left = 0.0f;
    float right = 320.0f;
    float aspect = 1.0f; //(float) self.frame.size.width / (float) self.frame.size.height;
    float projection[16] = {
        2.0/(right-left), 0.0f, 0.0f, 0.0f,
        0.0f, 2.0f/((top-bottom)*aspect), 0.0f, 0.0f,
        0.0f, 0.0f, -2.0f/1000.0f, -3.0f,
        0.0, 0.0, 0.0, 1.0f
    };
    float modelview[16]  = {
      1.0f, 0.0f, 0.0f, 0.0f,
      0.0f, 1.0f, 0.0f, 0.0f,
      0.0f, 0.0f, 1.0f, 0.0f,
      0.0f, 0.0f, 0.0f, 1.0f
    };

    GetGLError();


    glUniformMatrix4fv(_projectionUniform, 1, GL_FALSE,
                       projection);
    glUniformMatrix4fv(_modelViewUniform, 1, GL_FALSE,
                       modelview);

    GetGLError();

    glViewport(0, 0, self.frame.size.width, self.frame.size.width);
    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);

    GetGLError();

    glVertexAttribPointer(_positionSlot, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
    glVertexAttribPointer(_colorSlot, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (GLvoid*)(↩
        sizeof(float)*2) );
    glVertexAttribPointer(_texSlot, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                          (GLvoid*)(sizeof(float)*6) );

    GetGLError();

```

```
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);
    glUniform1i(sampler, 0);

    GetGLError();


    glDrawElements(GL_TRIANGLES, 6,
                   GL_UNSIGNED_BYTE, 0);


    // capture the data for analysis...

    glReadPixels(0, 0, 320, 480, GL_RGBA, GL_UNSIGNED_BYTE, bytes);

    // analyze our results
    [self analyzePixelData:bytes];

    [_context presentRenderbuffer:GL_RENDERBUFFER];

}
```

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

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <Accelerate/Accelerate.h>

typedef struct {
  char **opcodes;
  unsigned int numberOpcodes;
  float * elements;
  unsigned int numberElements;
} matrix;

matrix *read_matrix_from_file( const char * filename );
void print_matrix( FILE * file, matrix * m );
int destroy_matrix( matrix * m );
void populate_vector( float * vector, int nels, char ** opcodes, char * code );
void print_vector(float * result, int nels);
void print_result(float * result, int nels, char **opcodes );
char *choose_opcode( float * result, int nels, char **opcodes );

int main( int argc, char * argv[] )
{

  /*
   * BLAS documentation
   * https://developer.apple.com/library/ios/#DOCUMENTATION/Accelerate/Reference/BLAS_Ref/↩
     Reference/reference.html
   */

  /*
   * Here's a really good source...
```

```
 * http://www.prism.gatech.edu/~ndantam3/cblas-doc/doc/html/cblas_8h.html
 */

/*
 * doc's for LAPACK functions can be found at:
 * http://www.netlib.org/lapack/lug/node147.html#22228
 */
int solve_for_instruction = 0;
char * matrixfile = NULL;
matrix * m; // = read_matrix_from_file( "matrix.txt" );
//  print_matrix( stdout, m );
// destroy_matrix( m );


if ( argc == 3 ) {
  solve_for_instruction = 1;
  matrixfile = argv[2];
} else if ( argc == 1 ) {
  matrixfile = NULL;
} else {
  printf("Usage %s [previous opcode] [matrix file]\n",argv[0]);
  exit(1);
}

__CLPK_integer info;
float alpha = 1.0f;
float beta  = 1.0f;

if ( solve_for_instruction )
{
  m = read_matrix_from_file( matrixfile );

  __CLPK_integer mn = (__CLPK_integer) m->numberOpcodes;
  float * vec, * result;
  vec = (float*) malloc( sizeof(float) * m->numberOpcodes );
  result=(float*)malloc( sizeof(float) * m->numberOpcodes );

  // populate the vector...
  populate_vector( vec, m->numberOpcodes, m->opcodes, argv[1] );

  cblas_sgemv( CblasColMajor, CblasNoTrans, mn, mn, alpha, m->elements,
   mn, vec, 1, beta, result, 1 );

  char * chosen_opcode = choose_opcode( result, m->numberOpcodes, m->opcodes );
  printf("%s\n",chosen_opcode);
  // print_vector( result, m->numberOpcodes );

  free( vec );
  free( result );
  destroy_matrix( m );
  return 0;
}

  matrix * a = read_matrix_from_file( "matrix.txt" );
  __CLPK_integer dim = (__CLPK_integer) a->numberOpcodes;
  float * testA      = a->elements;

  __CLPK_integer n = 3;

  float A[9] = { 2.0f, 0.0f, 0.0f, // first column
                 0.0f, -1.0f, 0.0f,  // second column
                 0.0f, 0.0f, 2.0f };

  __CLPK_integer ipiv[3];
```

```
    sgetrf_(&n, &n, A, &n, ipiv, &info);
    printf(" dim = %d,  n = %d\n",dim,n);
    if ( info != 0 ) {
        printf("sgetrf failed with error code %d\n",(int)info);
        return 0;
    }

    __CLPK_integer inc = 1;

    float c[3] = {1.0f, 1.0f, 1.0f };
    float b[3] = {2.0f, -3.0f, 4.0f };

    char transpose = 'N';
    __CLPK_integer nrhs = 1;

    sgetrs_(&transpose, &n, &nrhs, A, &n, ipiv, b, &n, &info);
    printf(" n = %d\n",n);
    if ( info != 0 ) {
        printf("sgetrs failed with error code %d\n",(int)info);
        return 0;
    }

    printf("b = [ %f %f %f ]\n",b[0],b[1],b[2]);
    printf("c = [ %f %f %f ]\n",c[0],c[1],c[2]);
    printf("\n");

    printf(" testA * b \n");
    cblas_sgemv( CblasColMajor, CblasTrans, n, n, alpha, testA, n, b, 1, beta, c, 1);
    printf("x = [ %f %f %f ]\n",c[0],c[1],c[2]);

    printf(" b \n");
    printf("x = [ %f %f %f ]\n",b[0],b[1],b[2]);

    printf(" scaling b by 1.5 \n");
    cblas_sscal( n, 1.5f, b, 1 );
    printf("x = [ %f %f %f ]\n",b[0],b[1],b[2]);

    printf(" A * b \n");
    cblas_sgemv( CblasColMajor, CblasNoTrans, n, n, alpha, A, n, b, 1, beta, c, 1 );
    printf("x = [ %f %f %f ]\n",c[0],c[1],c[2]);
    destroy_matrix( a );
    return 0;
}


matrix *read_matrix_from_file( const char * filename )
{
  FILE * fhandle = fopen( filename, "r" );
  if ( fhandle == NULL )
    return NULL;

  matrix * pmtrx = (matrix*) malloc( sizeof(matrix) );
  float num;

  // read in the first line, containing the opcodes...
  unsigned int commas = 0, linelength = 0;
  int c;

  while ( (c = fgetc(fhandle)) != EOF ) {
    linelength++;
    if ( c == ',' ) commas++;
    if ( c == '\n') break;
  }

  // rewind the file...
```

```
  rewind(fhandle);
  pmtrx->numberOpcodes = commas+1;
  pmtrx->numberElements= pmtrx->numberOpcodes * pmtrx->numberOpcodes;
  pmtrx->opcodes = (char**) malloc( sizeof(char*) * pmtrx->numberOpcodes );
  pmtrx->elements= (float*) malloc( sizeof(float) * pmtrx->numberElements );

  char * cptr, * line = (char*) malloc( sizeof(char) * linelength );
  fgets( line, linelength, fhandle );

  // read in our opcdoes...
  cptr = strtok( line, "," );
  c = 0;
  while ( cptr != NULL )
    {
      pmtrx->opcodes[c++] = strdup( cptr );
      cptr = strtok(NULL,",");
    }

  free( line );

  c = 0;
  while ( fscanf(fhandle," %f ",&num) )
    {
      pmtrx->elements[c++] = num;
      if ( c == pmtrx->numberElements )
  break;
    }

  fclose(fhandle);
  // done...

  return pmtrx;

}

void print_matrix( FILE * file, matrix * m )
{
  int i,j;
  int num = (int) sqrtf( m->numberElements );
  for (i=0; i<m->numberOpcodes; i++)
    fprintf(file," %8s ", m->opcodes[i] );
  fprintf(file,"\n");

  for (i=0; i<num; i++)
    {
      for (j=0; j<num; j++)
  fprintf(file," %8f ",m->elements[i*num+j]);
      fprintf(file,"\n");
    }
}

int destroy_matrix( matrix * m )
{
  int i;
  for (i=0; i<m->numberOpcodes; i++)
    free( m->opcodes[i] );

  free( m->elements );
  free( m->opcodes );

  m->numberOpcodes = 0;
  m->numberElements= 0;

  m = NULL;

```

```c
  return 0;
}

void populate_vector( float * vector,
            int nels,
            char ** opcodes,
            char * code )
{
  int i;
  for (i=0; i<nels; i++) {
    if ( strncmp(opcodes[i],code,4) == 0 )
      vector[i] = 1.0f;
    else
      vector[i] = 0.0f;
  }
  // end of function
}


void print_vector(float * result, int nels)
{
  int i;
  for (i=0; i<nels; i++)
    printf(" %8f ",result[i]);
}

void print_result(float * result, int nels, char **opcodes )
{
  int i;
  // print out json result, can import into python easily...
  printf("{");
  for ( i=0; i<nels; i++ )
    {
      if ( result[i] > 0.0f )
  printf("'%s':%f,",opcodes[i],result[i]);
    }
  printf("}");
}


char *choose_opcode( float * result, int nels, char **opcodes )
{
  // get a rand num [0,1]
  float num = ( (float) rand() / (float) RAND_MAX );
  float sum = 0.0f;
  int i;
  for (i=0; i<nels; i++)
    {
      sum += result[i];
      if ( num < sum )
  return opcodes[i];
    }
  return NULL;
}
```

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

```c
#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>

#include <jni.h>
#include <android/log.h>
#include <GLES2/gl2.h>
#include <GLES2/gl2ext.h>

#include <sys/types.h>
#include <android/asset_manager.h>
#include <android/asset_manager_jni.h>
#include <assert.h>

#define LOG_TAG   "libglsl2"
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO,LOG_TAG,__VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR,LOG_TAG,__VA_ARGS__)


static void printGLString( const char *name, GLenum s )
{
  const char *v = (const char*) glGetString(s);
  LOGI("GL %s = %s\n",name,v);
}

static void checkGlError( const char * op )
{
  GLint error;
  for (error = glGetError(); error; error = glGetError() ) {
    LOGI("after %s() glError (0x%x)\n", op, error );
  }
}

int screenWidth;
int screenHeight;

int analyzeFrameBuffer( unsigned char * data )
{
  int i,j;
  FILE * tmpfile = fopen("/sdcard/tmp/results.txt","w");
  if ( tmpfile == NULL )
    return 1;

  for (i=0; i<screenHeight; i++)
    {
      for (j=0; j<screenWidth; j++)
  {
    fprintf(tmpfile," %c ",data[i*screenWidth+j]);
  }
      fprintf(tmpfile,"\n");
    }
  LOGI("WROTE GPU buffer to file");
  fclose( tmpfile );

  return 0;
}

GLuint loadShader(GLenum shaderType, const char* pSource) {
  GLuint shader = glCreateShader(shaderType);
  if ( shader ) {
    glShaderSource(shader, 1, &pSource, NULL);
    glCompileShader( shader );
    GLint compiled = 0;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
    if ( !compiled ) {
      GLint infoLen = 0;
```

```
      glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);
      if ( infoLen ) {
  char * buf = (char*) malloc( infoLen );
  if ( buf ) {
    glGetShaderInfoLog(shader,infoLen,NULL,buf);
    LOGE("Could not compile shader %d:\n%s\n",shaderType,buf);
    free(buf);
  }
  glDeleteShader(shader);
  shader = 0;
      }
    }
  }
  return shader;
}

GLuint createProgram( const char * pVertexSource, const char * pFragmentSource )
{
  GLuint vertexShader = loadShader( GL_VERTEX_SHADER, pVertexSource );
  if ( !vertexShader ) {
    return 0;
  }

  GLuint pixelShader = loadShader( GL_FRAGMENT_SHADER, pFragmentSource );
  if ( !pixelShader ) {
    return 0;
  }

  GLuint program = glCreateProgram();
  if ( program ) {
    glAttachShader(program, vertexShader);
    checkGlError("glAttachShader");
    glAttachShader(program, pixelShader );
    checkGlError("glAttachShader");
    glLinkProgram(program);
    GLint linkStatus = GL_FALSE;
    glGetProgramiv(program, GL_LINK_STATUS, &linkStatus);
    if ( linkStatus != GL_TRUE ) {
      GLint bufLength = 0;
      glGetProgramiv(program, GL_INFO_LOG_LENGTH, &bufLength);
      if ( bufLength ) {
  char * buf = (char*) malloc( bufLength );
  if ( buf ) {
    glGetProgramInfoLog(program, bufLength, NULL, buf);
    LOGE("Could not ink program:\n%s\n",buf);
    free(buf);
  }
      }
      glDeleteProgram(program);
      program = 0;
    }
  }
  return program;

}

GLuint gProgram;
GLuint gvPositionHandle;

char *readShader( const char *filename )
{

}

char * gVertexShader;
```

```
char * gFragmentShader;

int setupGraphics(int w, int h) {

  printGLString("Version", GL_VERSION);
  printGLString("Vendor",GL_VENDOR);
  printGLString("Renderer",GL_RENDERER);
  printGLString("Extensions",GL_EXTENSIONS);

  screenWidth = w;
  screenHeight= h;

  LOGI("setupGraphics(%d,%d)",w,h);
  gProgram = createProgram( gVertexShader, gFragmentShader );
  if ( !gProgram ) {
    LOGE("Could not create program.");
    return 0;
  }

  gvPositionHandle = glGetAttribLocation(gProgram, "vPosition");
  checkGlError("glGetAttribLocation");
  LOGI("glGetAttribLocation(\"vPosition\") = %d\n",gvPositionHandle);

  glViewport(0, 0, w, h );
  checkGlError("glViewport");
  return 1;

}

const GLfloat gVertices[] = {0.0f, 0.5f, -0.5f, -0.5f, 0.5f, -0.5f };

void renderFrame( ) {

  unsigned char * bufferdata;

  glClearColor(0,0,0,1.0f);
  checkGlError("glClearColor");
  glClear( GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT );
  checkGlError("glColor");

  glUseProgram(gProgram);
  checkGlError("gluseProgram");

  glVertexAttribPointer( gvPositionHandle, 2, GL_FLOAT, GL_FALSE, 0, gVertices );
  checkGlError("glVertexAttribPointer");
  glEnableVertexAttribArray(gvPositionHandle);
  checkGlError("glEnableVertexAttribArray");
  glDrawArrays( GL_TRIANGLES, 0, 3 );
  checkGlError("glDrawArrays");

  LOGI("Called draw functions");
  LOGI("Screen width,height are %d,%d",screenWidth,screenHeight);
  // post draw call, we call read pixels to obtain our result
  // note that we need depth of 4...
  bufferdata = (unsigned char*) malloc( sizeof(unsigned char) * screenWidth * screenHeight
      * 4 );
  glReadPixels( 0, 0, screenWidth, screenHeight, GL_RGBA, GL_UNSIGNED_BYTE, bufferdata );

  LOGI("Read Pixels");

  // pass this buffer data for analysis...
  analyzeFrameBuffer( bufferdata );
  LOGI("Analyzed...");

  // clean up
```

```
  free( bufferdata );

}

/*
  JNIEXPORT void JNICALL Java_com_gototheboard_glsl2_NativeGLLib_init(JNIEnv * env, jobject obj,
                      jint width, jint height);
  JNIEXPORT void JNICALL Java_com_gototheboard_glsl2_NativeGLLib_step(JNIEnv * env, jobject obj );
  JNIEXPORT void JNICALL Java_com_gototheboard_glsl2_NativeGLLib_setupShaders(JNIEnv * env,
                        jstring vshader,
                        jstring fshader);
*/

JNIEXPORT void JNICALL Java_com_gototheboard_nativegl_NativeGLLib_init(JNIEnv * env, jobject obj,
                      jint width, jint height)
{
  LOGI("Setting up native graphics code.");
  setupGraphics( width, height );
  LOGI("Finished setting up native graphics code.");
}

JNIEXPORT void JNICALL Java_com_gototheboard_nativegl_NativeGLLib_step(JNIEnv * env, jobject obj )
{
  LOGI("Step...\n");
  renderFrame( );
}

JNIEXPORT void JNICALL Java_com_gototheboard_nativegl_NativeGLLib_setupShaders(JNIEnv * env, jobject obj,
                      jobject assetManager,
                      jstring vshader,
                      jstring fshader)
{
  int vid, fid;
  off_t start, length;

  LOGI("Starting to read shaders...");

  const jbyte * utf_vshader = (*env)->GetStringUTFChars( env, vshader, NULL );
  const jbyte * utf_fshader = (*env)->GetStringUTFChars( env, fshader, NULL );

  LOGI("Created UTF strings..");

  // create an asset manager
  AAssetManager * mgr = AAssetManager_fromJava(env, assetManager);
  assert(NULL != mgr );
  if ( mgr == NULL )
    LOGI("Unable to create asset manager");
  else
    LOGI("Created asset manager");

  // read in vertex fragment
  LOGI("Going to read %s",(char*)utf_vshader);
  LOGI("Going to read %s",(char*)utf_fshader);
  AAsset * vertex_asset = AAssetManager_open( mgr, (const char*) utf_vshader, AASSET_MODE_UNKNOWN );
  AAsset * frag_asset   = AAssetManager_open( mgr, (const char*) utf_fshader, AASSET_MODE_UNKNOWN );
  LOGI("Created asset objects");
  if ( vertex_asset == NULL )
```

```
    LOGI("Vertex asset could not be opened.");
  if ( frag_asset   == NULL )
    LOGI("Frag asset could not be opened.");

  // release the Java string and UTF
  (*env)->ReleaseStringUTFChars( env, vshader, utf_vshader );
  (*env)->ReleaseStringUTFChars( env, fshader, utf_fshader );
  LOGI("Released UTF8 strings");

  // get file descriptor -- failing here with the vid = -1
  vid = AAsset_openFileDescriptor(vertex_asset, &start, &length );
  LOGI("vid = %d",vid);
  assert( 0 <= vid );
  gVertexShader = (char*) malloc( sizeof(char) * length + 1 );
  memset(gVertexShader,'\0',length+1);
  AAsset_read( vertex_asset, gVertexShader, length );
  AAsset_close(vertex_asset);
  LOGI("Read in vertex shader:\n %s\n",gVertexShader);

  fid = AAsset_openFileDescriptor( frag_asset, &start, &length );
  assert( 0 <= fid );
  gFragmentShader = (char*) malloc( sizeof(char) * length + 1 );
  memset(gFragmentShader,'\0',length+1);
  AAsset_read( frag_asset, gFragmentShader, length );
  AAsset_close(frag_asset);
  LOGI("Read in frag shader:\n %s\n",gFragmentShader);

}
```

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

```
APP_STL:=stlport_static
```

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