An Evaluation of Valgrind Performance on Commercially Available Maker Devices

Tzu-Wei Chuang Greg Cusack Kunal Patel Nathan Pilbrough Zhengshuang Ren
University of California, Los Angeles
{twc5, gregcusack, kunalpatel, nathanpilbrough, zhengshuang} @g.ucla.edu

ABSTRACT

As an increasing number of low-power, embedded devices are connected to the internet, it is important to ensure that these devices, with limited resources, manage their memory allocation properly. Failure to properly deallocate memory leads to memory leaks, which can then lead to system failure. Valgrind, a software analysis program, has the `memcheck` tool, which is an effective, yet memory intensive, program for identifying memory leaks in code. The EEclipse team not only evaluates `memcheck`'s ability to detect leaks on common, low-resource, maker devices, but also investigates memory leaks in popular open source software. We wrote test cases and compared the execution time and leak detection accuracy of `memcheck` across a BeagleBone Black, an Intel Edison, an Ubuntu virtual machine, and a MacBook Pro. We find that `memcheck`, while memory intensive, is able to identify memory leaks within a reasonable time frame on two maker devices. However, `memcheck` is not able to consistently and accurately detect injected memory leaks in the open source libraries, FANN, and Guetzli. In the end, we conclude that `memcheck` is an accurate and viable option for detecting memory leaks on maker devices in most cases, yet fails to properly identify memory leaks in a few.

Keywords

Software performances, Valgrind, Memcheck, Evaluation, Embedded systems, Memory management

1. INTRODUCTION

With the current and impending influx of internet-connected, embedded devices, individual device resources need to be allocated properly in order to perform efficiently with limited resources. While competent programmers aim to develop the best possible code, the lack of coupling between memory allocations and deallocations, means bugs are inevitably introduced into the system. Many of these bugs result from memory mismanagement and utilizing more memory than required. Due to the limited resources and speed requirements of many IoT devices, their code bases are largely developed in C/C++. Unfortunately, unlike Java, C/C++ do not have garbage collectors, so resources and objects allocated and not freed use up valuable memory in the processor. As a result, it is common for software developers to forget to free up program memory and return it back to the operating system.

Memory leaks have been comprehensively studied. Programs dynamically request memory from the system at execution. Large scale applications frequently allocate memory, so running out of memory tends to have a chain reaction effect on the rest of the system. Developers are responsible for programming applications to release dynamically allocated memory. Memory leaks occur when memory is not properly deallocated and is rendered unusable, typically associated with the misuse of pointers, i.e. not deallocating memory after usage completion. Memory leaks are common bugs that are often overlooked since programs can run successfully for a short period without the leaks accumulating enough to crash the system. Some memory leaks can be detected easily through a static analysis tool by simply looking for a corresponding deallocation to every memory allocation. After all available memory is consumed, the operating system sends virtual memory to the hard drive, which bogs down operation and eventually crashes the application when the limit of computer memory is reached [1].

Developers from the open source community have designed Valgrind, an open-source framework for building C/C++ source code analysis tools. Valgrind consists of tools such as `memcheck`, a widely used tool which identifies memory management issues such as memory leaks. While Valgrind has been used to debug many different projects including topics in graphics, IoT operating systems, and statistical computing, it is a relatively memory and resource intensive tool. However, to the best of our knowledge, there are no detailed and quantitative analyses on Valgrind performance across multiple platforms with varying resources. With the impending boom of IoT applications on maker devices, software is being developed specifically for maker devices and their respective architectures. The EEclipse team has ported Valgrind to two common maker devices in order to quantitatively analyze the performance of Valgrind and to prove a popular, complex memory leak checker can be used by developers to ensure better performing applications for maker devices.

Static analysis of code cannot guarantee that memory leaks will not occur. Increasing code complexity through polymorphism, external dependencies, and error handling can cause a static analysis tool to report false negatives. For example, exception handling can cause a branch in the code, which can cause a memory deallocation step to be skipped. Valgrind's `memcheck` is a dynamic analysis tool, which will not only pick up a static error such as lack of a delete after a corresponding memory allocation, but it will also identify memory errors not prevalent until runtime. Take the following code snippet in Figure 1, for example:

```
1  void createLeak()
2  {
3      int *memptr;
4      memptr = new int[2];
5      try {
6          if(winnerScore <= loserScore) {
7              throw 1;
8          }
9      }
10     catch(int x) {
11         cout << "Winner score <= Loser score.  ERROR: " << x << endl;
12         return;
13     }
14     memptr[0] = winnerScore;
15     memptr[1] = loserScore;
16     //do whatever you want here
17     delete [] memptr;
18 }
```

Figure 1: Code snippet requiring dynamic memleak analysis.

The function *createLeak()* has an exception handling block in order to ensure that a winner's score (winnerScore) is always greater than a loser's score (loserScore) (lines 5-9). Now, if the code was to be analyzed by a static analysis tool, the tool would report no memory errors because there is a corresponding delete[] to the dynamically allocated integer array. However, in the case that winnerScore is less than loserScore, the try block will throw an exception, which will cause the program to enter the catch block, report an error, and then return to the calling program. In this case, the dynamically allocated integer array is never deleted because line 17 is never reached. This is just one example where a dynamic memory leak detection tool is required for proper analysis.

We gather a stronger understanding of Valgrind's performance on maker devices compared to a standard PCs and VMs by designing nine memory leak test cases, analyzing FANN and Guetzli open source libraries, and testing an embedded, life critical application. Our performance metrics include Valgrind's runtime overhead and ability to detect prevalent memory leaks. The research question we ultimately answer is "Is Valgrind scalable on maker devices?" It is important to understand the scalability of Valgrind on these systems because they can enlighten future developers on performance limitations in common maker devices. In the end, we find that Valgrind is scalable and is able to provide meaningful results in some cases in a respectable amount of time on both the BeagleBone Black and the Intel Edison.

The paper is organized as follows. Section II describes Valgrind and introduces various types of memory leaks. Section III discusses key aspects of the approach and the experimental designs. Results from the experiment are presented and analyzed in Section IV. Section V and VI are devoted to case studies on real open source applications FANN and Guetzli, respectively. Section VII considers an application for IoT devices, for which we tested with Valgrind. Threats to validity are discussed in Section VIII. Section IX compares our approach to some of the related works. Finally, Section X concludes the paper and briefly presents future work.

2. VALGRIND OVERVIEW

2.1- Valgrind Overview

Valgrind is an instrumentation framework for building dynamic analysis tools that analyze programs at runtime at the level of machine code. It runs on multiple platforms, 32-bit and 64-bit: x86/Linux, AMD64/Linux, and PPC{32, 64} [15] and works directly with existing executables. While there are many tools included with Valgrind such as thread error detectors and profilers, `memcheck` is the most commonly used tool and was used to evaluate the performance of Valgrind in this paper.

2.2 - Valgrind Memcheck Tool

The Valgrind `memcheck` tool works in the following steps. Valgrind first takes control of the program before it starts and reads debugging information from the executable and libraries so that errors can be reported in terms of source code locations. `Memcheck` then adds code to check every memory access and every value computed, making it 5-100 times slower than native execution. `Memcheck` then simulates every instruction in the executable and looks for memory leaks throughout a user's code. When referring to memory leaks, we are referring to memory in a system which has been allocated to a specific program, but is never properly freed and returned back to the operating system. There are two broad categories of memory leaks, lost memory and forgotten memory [2]. Fundamentally, a lost memory leak occurs when a pointer to an allocated object is either deleted or set to point to another object. This memory taken up by the object can no longer be accessed and cannot be used by any other process running on the device. A forgotten memory leak, on the other hand, occurs when the memory block is still reachable but is not deallocated or accessed for the rest of the code execution. Hence, memory leaks become a large problem when systems are constantly running and servicing requests. Memory is allocated and never deallocated; therefore, a system's memory resources are quickly consumed to exhaustion. `Memcheck` tracks each memory allocation (malloc(), calloc(), new) and deallocation (free(), delete, delete []). Once the user program completes, Valgrind reports four different types of memory leaks in an intuitive manner. The output of Valgrind's `memcheck` tool and description of the four reported memory leaks is summarized below [15].

(1) Definitely Lost Memory: There exists no pointers in the program to the allocated memory such that the program has no way of deallocating it.
(2) Indirectly Lost Memory: Occurs in situations such as binary trees where the pointer referring to the root node is lost. As a result all subtrees are also lost. This is memory that we consider to be completely lost as it can no longer be accessed. If Definitely Lost errors are resolved, these Indirectly lost errors are also resolved.
(3) Possibly Lost Memory: Applies in situations such as at the program exit time, the pointer to the allocated memory isn't completely lost. According to the developers, possibly lost means that the program is almost certainly leaking memory, unless the code does strange actions that result in pointers pointing the center of an allocated block of memory

(4) Still Reachable Memory: Memory that was not freed, but a pointer to it still exists at the program's exit time.

And by default, Valgrind reports errors in all programs or libraries on users' GNU/ Linux system on which the application is dependent. The suppression mechanism is useful if there's a memory error in library code that the user cannot change. Therefore it hides those errors users defined by reading the .supp file.

3. APPROACH AND EXPERIMENTS

3.1 - Valgrind on Maker Devices

Valgrind performance is measured on two popular maker devices: BeagleBone Black (32-Bit, 1GHz ARM Cortex-A8, 512MB DDR3 RAM), and the Intel Edison (32-Bit, 500MHz Dual-core Atom, 1GB DDR3 RAM). Note that Valgrind requires an OS to run; therefore, Valgrind will not run on the widely-used Arduino, which is why it is left out of this project. Another maker device which continues to gain in popularity due to its ease of use and low cost is the Raspberry Pi model 2/3. Unfortunately, due to the system requirements of Valgrind, there were some compatability issues that prevented Valgrind's `memcheck` tool from running on the Raspberry Pi. Please refer to Appendix C for the output results for the Raspberry Pi.

3.2 - Valgrind on Virtual Machine and Macbook (El Capitan)

Valgrind was also run on an Ubuntu VirtualBox VM running on a laptop containing a 2.5GHz Intel Core i7 and on MacOS El Capitan in order to provide a performance reference frame. A virtual machine was used as a test case in order to simulate environments which require higher levels of security. Virtual machines are useful in the sense that if the virtual machine gets a virus, the host computer is left unaffected. As the Internet of Things rapidly expands, the security of these IoT networks lags behind. Running a virtual machine on a maker device could help protect a network of embedded devices from viruses. The MacOS system was used to test Valgrind as a performance baseline in which to compare the results of the other systems due to its relatively large memory capacity.

3.3 - Valgrind Performance Matrix

In this paper, performance of Valgrind refers to runtime, scalability, and memory leak identification success rate. Therefore, for each device, Valgrind performance was measured on programs of various length and complexity. One of the main challenges of measuring performance of Valgrind was designing proper test cases themselves. The EEclipse team developed test cases which accurately evaluated Valgrind performance across multiple embedded devices and will be discussed in detailed in the next section.

In order to evaluate Valgrind performance, we standardized the `memcheck` tool to see if it detects different kinds of memory leaks. We first installed Valgrind and the appropriate libraries on each platform, then developed nine test cases adapted from[3] in an attempt to study the most common memory leak cases.

3.4 - Common Memory Bugs

Since it is almost impossible to generate test cases to cover all types of memory-related bugs, we listed the bug types that are most common. Memory related bugs are caused by improper handling of memory objects, and can be further classified into (1) Missing deallocation: whereby the programmer simply forgot to release the allocated memory. (2) Pointer reallocation: a pointer to memory is assigned to a new memory block without freeing the original block. (3) Reachability issue: in this case the memory deallocator is not present on all paths through a program and the memory is not freed in some cases. During this standardization we collected execution time and leaks identified. Execution time was measured using the C++11 library Chrono, which measured leak creation and detection in nanoseconds. The native execution time was measured and compared to the execution time using Valgrind in order to get an idea of overall slowdown of the system.

3.5 - Test Cases Generation

Since we wrote these nine test cases in our test suite, we know the total expected memory lost that should be reported by Valgrind across all four systems. The description of each test case is as follows[1]:

1. Test Case 1: implements the simplest form of memory leak whereby memory is allocated to a pointer and never released.
2. Test Case 2: memory is allocated to a pointer and then the pointer is redirected to a new block of memory before the original block is deallocated
3. Test Case 3: tests the case where a pointer to a pointer array is used and the memory is not released
4. Test Case 4: combines test case 2 and 3, whereby a pointer to pointer array is allocated memory and then redirected to a new block of memory before the original block is deallocated
5. Test Case 5: This test case deals with runtime exceptions, which causes the memory deallocation step in a program to be skipped. A code snippet from this test case was referenced earlier in the paper in Figure 1. This test case helps prove that Valgrind's `memcheck` is able to correctly identify runtime memory leaks.
6. Test Case 6, 7, 8: memory is allocated to a static pointer and then redirected without deallocation of the original memory block. The difference between the tests is the scope of the static variable. Case 6: static class pointer, case 7: static global pointer, case 8 static local pointer - this case is the same as case 1 with the addition of the static variable attribute.
7. Test Case 9: simulates an error that can occur due to polymorphism whereby a base class pointer is assigned to a derived class. The base class destructor is not declared virtual and as a result the derived class destructors are not called and the allocated memory is not freed.

[1] Code for all nine test cases can be found on the gitHub repository available at https://github.com/vivi51123/CS230-Software-Engineering.

We discuss our findings from these test cases in detail in Section IV. Furthermore, all of the test scripts are available in our GitHub repository. Follow the instructions in Appendix E to run those test cases.

3.6 - Memory Leak Injection

Memory leaks are one of the primary causes of software aging, which is described as the tendency for software to fail over time. Due to limited time constraints, studies into software aging use various strategies to implement accelerated aging in order to study its effects over a short time frame. Zhao et al [4] injected memory leaks into a program to simulate the accumulation of memory leaks over time. This strategy can be categorized as an active implementation since unnecessary memory allocations were maliciously generated. Tsai et al [5] implemented a passive memory leak injection technique whereby they overloaded the memory deallocation function such that memory blocks were not released over the lifetime of the program. Matias et al [6] implemented a pseudo passive memory injection technique whereby they prevented the deallocation of some memory blocks based on a random variable and a leakage rate threshold.

As pointed out in [7] the activation conditions of the memory leaks implemented by Zhao et al in [4] was not complex. However, despite there being many ways in which a memory leak can be generated, the effect of that memory leak on the performance of a program is the same. Therefore, since this investigation was focused on the performance of Valgrind and its ability to identify memory leaks rather how memory leaks come about, the use of a simple injection method was justified.

The experimentation procedure was limited due to time constraints. Memory injection requires access to the source code and subsequently rebuilding project binaries can take hours on maker devices. Many open source libraries failed to build on the Beaglebone and/or Edison as mentioned in the Appendix B and were disregarded. The libraries that did compile were FANN and Guetzli which are mentioned later in the investigation. Passive memory leak injections were implemented in these two libraries whereby all the calls to memory deallocation functions in a particular source file were removed. Active memory leaks were then injected at strategic points such that memory leaks would accumulate over time.

4. RESULTS

From the test cases developed and executed, we were able to gather a much better idea of how well Valgrind performs on multiple devices. The results from our tests are outlined in Appendix A: Tables A.1-A.4. Table A.1 shows the expected number of bytes lost due to memory leaks for each test case. Table A.2 shows the reported number of bytes lost by Valgrind's `memcheck` tool on all four platforms. It can be seen from the Table A.2 that the BeagleBone Black and Intel Edison have comparable reports that match up directly with the expected memory lost listed in Table A.1. At first glance, the VM and the Laptop running OS X El Capitan seem to overestimate the number of bytes "definitely lost" by double the expected amount for test cases 3 and 4. However, this discrepancy is due to the difference between the 32-bit and 64-bit architectures of the devices. For the nine test cases run, the BeagleBone, Edison, and VM all correctly reported zero "possibly lost" bytes. However, El Capitan reported a consistent 2064 bytes "possibly lost" across all nine test cases. The extra reported memory leaks were unexpected, especially on a well-resourced machine. However, after doing some research, we found that El Capitan has a history of reporting false positives [8], which ends up lining up with our conducted research. After looking at memory leak detection ability, we turned our focus to runtime overhead of Valgrind.

Table A.3 shows the true runtime of each test case. The true runtime refers to the amount of time the program took to execute by itself, independent of Valgrind. While looking at Table A.3, it is clear that there is some variability in runtime performance across platforms. Not surprisingly, the VM and El Capitan execute the test cases faster than the two embedded devices. Table A.4 shows the runtime performance of each test case run with `memcheck`, while Table 1 shows the resulting slowdown of each program when Valgrind is used. The overall execution slowdown was found using Equation 1.

$$slowdown = \frac{runtime\ with\ Valgrind\ (ms)}{true\ runtime\ (ms)} \qquad (1)$$

From Table 1, it can be seen that both maker devices (for leaks 1-9) experience a slowdown within the expected 5-100x factor reported by Valgrind [9]. The larger slowdown factor experienced in Leak 5 is likely due to the inherent overhead experienced by the try-catch exception handling blocks in Leak 5. On average, the Intel Edison experiences a larger slowdown due to Valgrind than the BeagleBone at ~91.9x, while the BeagleBone handles the Valgrind overhead the best with an average slowdown of only ~45.5x. To our surprise, both the VM and El Capitan experience massive slowdown factors on the order of 600-700x, much larger than the expected slowdown. However, it is important to notice that the standard deviation of the runtime ratios is greater than the average in the case of the Edison and close to the average slowdown for the other three devices. As a result, it is hard to use the average slowdown as a definitive estimate of the slowdown factor. Furthermore, runtime varies between the VM and El Capitan. El Capitan outperforms the VM in 6 out of the 9 test cases, which is expected due to the test cases running natively on the El Capitan laptop. Relatively comparing slowdown factors across all cases, we find that the maker devices overall performed within a reasonable timeframe and performed better than the other devices we tested.

5. CASE STUDY 1 - FANN

In order to examine the accuracy and consistency of Valgrind performance as we scale to larger open source systems, we decided to test some examples using the Fast Artificial Neural Network (FANN) Library [10]. FANN is a neural network library that implements multilayer artificial neural

Table 1: Valgrind Slowdown Factor

Runtime Ratio (slowdown)	BeagleBone Black	Intel Edison	VM	El Capitan
Leak 1	19.140	21.848	603.957	652.822
Leak 2	61.247	41.920	708.442	452.618
Leak 3	20.139	45.937	585.789	1597.518
Leak 4	49.483	83.463	464.353	472.108
Leak 5	153.585	475.570	1665.545	1547.819
Leak 6	47.706	43.769	504.247	324.822
Leak 7	29.626	45.891	1058.257	967.930
Leak 8	15.602	24.075	435.207	31.149
Leak 9	13.165	44.931	413.573	162.660
Average Slowdown	45.521	91.934	715.485	689.938
Standard Deviation	43.989	144.927	407.249	568.707

networks in C. We chose FANN as our testing program mainly because it is easy to use, well documented, and lightweight. However, from our experience, these programs tend to have little memory leaks issues, and the test results show that no memory leaks were detected. As a result, we injected passive memory leaks into three FANN test cases by removing deallocating segments in the FANN libraries so that memory cannot be properly freed. Valgrind was able to pick up these memory leaks across all four devices as shown below in Table 2. It is important to note that no memory leaks were detected (except the standard 2064 bytes by El Capitan) before we injected memory leaks into these three programs. Once again, El Capitan reports the most number of memory leaks across all FANN test cases. The Edison and the BeagleBone produced similar memory leak results, especially in the "definitely lost" domain. The BeagleBone does, however, report more "indirectly lost" bytes than the Edison, while the Edison reports more "possibly lost" bytes.

One interesting bit of information gathered from the FANN test cases is that the BeagleBone reports its leaks in two out of the three test cases as "definitely" or "indirectly lost" but the Edison reports some lost bytes as "possibly lost." However, the total memory leaks reported by each device are nearly the same for every test case, as see in the last row of Table 2. This implies that while the Edison and BeagleBone interpret memory leaks differently, they still report the same total number of leaks. Furthermore, the VM and El Capitan report a similar number of total leaks even though they do not report them in the same manner. The difference between the total lost bytes from VM and El Capitan test cases is 6192 bytes. Similar to the nine generated test cases in

Table 2: Passive Memory Leak Injection Test Case Results for FANN

Test Case, Lost Memory		Beaglebone Black	Intel Edison	VM	El Capitan
Momentums	Definitely Lost (bytes)	1996	1996	2856	2856
	Indirectly Lost (bytes)	654868	540820	802640	802640
	Possibly Lost (bytes)	0	114048	0	2064
Mushroom	Definitely Lost (bytes)	340	340	504	504
	Indirectly Lost (bytes)	2247572	2247572	2328980	4359980
	Possibly Lost (bytes)	2031000	2031000	2031000	2064
Robot	Definitely Lost (bytes)	340	340	504	504
	Indirectly Lost (bytes)	269452	155404	297200	297200
	Possibly Lost (bytes)	0	114048	0	2064
	Total Lost (bytes)	5205568	5205568	5463684	5469876

Section IV, the trend of El Capitan reporting 2064 "possibly lost" bytes is seen in each of the FANN test cases above.

Note that 2064 bytes * 3 test cases equals 6192 bytes. As it turns out, if we remove the inherent 2064 bytes reported by El Capitan for each test case, the VM and El Capitan report the same number of total lost bytes.

Since the tests above rely on differential testing on the output of Valgrind, we implemented active memory leaks in the three above test cases in order to provide a more accurate baseline for testing the FANN test files. Since we know how much memory should leak, we can test the validity of Valgrind's output. These memory leaks are created using C's malloc() function for memory allocation. The results of the active memory leak tests can be seen in Table 3.

The BeagleBone, Edison, and VM all report the expected number of bytes lost. However, El Capitan is unable to detect any of the injected leaks. El Capitan's behavior is interesting since El Capitan has already been proven to detect

Table 3: Active Memory Leak Injection Results

Test Case, Lost Memory		Beaglebone Black	Intel Edison	VM	El Capitan
Momentums	Definitely Lost (bytes)	280	280	280	0
	Indirectly Lost (bytes)	0	0	0	0
	Possibly Lost (bytes)	0	0	0	2064
Mushroom	Definitely Lost (bytes)	162480	162480	162480	0
	Indirectly Lost (bytes)	0	0	0	0
	Possibly Lost (bytes)	0	0	0	2064
Robot	Definitely Lost (bytes)	23760	23760	23760	0
	Indirectly Lost (bytes)	0	0	0	0
	Possibly Lost (bytes)	0	0	0	2064
	Total Lost (bytes)	186520	186520	186520	6192

memory leaks created by C's malloc() function in the passive memory leak case above.

Another interesting issue uncovered is Valgrind's inability to detect injected, active memory leaks in the FANN test cases. In all three test cases, across the BeagleBone, Edison, and VM[2], memory leaks injected into the test cases are either reported or ignored depending on the location of the injected leak. Figure 2 shows a code snippet from the robot.c test case. The memory leak injected on lines 56-57 is detected and properly identified by `memcheck`. However, the memory leak on lines 60-61 is not detected. Furthermore, if we remove lines 56-58 and place them right before fann_reset_MSE(ann), or we place the lines right before line 55, `memcheck` will detect all injected memory leaks. However, Valgrind is unable to detect this simple memory leak at any place in the code after the for loop covering lines 53-59. Why `memcheck` is unable to detect memory leaks after line 59 is still unknown and out of the scope of the project. We leave the reason for `memcheck`'s shortcomings to future work and consider it a call to action for valgrind developers.

The initial results from the FANN test cases implied that FANN is a memory leak free neural network-based machine learning library for embedded devices. However, after further investigation, due to the lack of bug reporting in the specific cases mentioned immediately above and when using El Capitan, we cannot conclude that FANN is memory leak free. These results also raise concern for `memcheck`'s ability

[2] We exclude El Capitan here since it was unable to detect any leaks for the active memory leaks in FANN

```
52      fann_reset_MSE(ann);
53      for(i = 0; i < fann_length_train_data(test_data); i++)
54      {
55          fann_test(ann, test_data->input[i], test_data->output[i]);
56          int *p = (int *)malloc(sizeof(int)*10);
57          p[0] = 10;
58          printf("Value of 0th element: %d", p[0]);
59      }
60      int *p = (int *)malloc(sizeof(int)*10);
61      p[0] = 10;
62      printf("Value of 0th element: %d", p[0]);
```

Figure 2: Code Snippet From robot.c Test Case Utilizing FANN Libraries

to detect errors accurately on other systems. With that being said, the passive memory leak detection across all devices is consistent, implying that passive memory leaks may be more accurately detected than active ones. The FANN tests continue to support the findings that the BeagleBone and the Edison are able to handle Valgrind's memory intensive nature.

6. CASE STUDY 2 - GUETZLI

Guetzli is an open-source JPEG encoder developed by Google with the goal of achieving high compression density while maintaining high visual quality. Guetzli performs closed loop optimization with feedback provided by a human vision model, Butteraugli, to ultimately have uniform image quality while generating smaller JPEG images [11].

Guetzli is not an application that you would typically have on a maker device. We chose to include this case study because this is a memory intensive application that can emulate running intensive maker applications. There are some potential systems that use maker device nodes for distributed systems; therefore, there is potential of building on the current research. The researchers of Guetzli find that "it is questionable whether the savings at transfer time are worth the slowdown at decoding time," a process that can be tasked to connected maker devices for efficient processing power allocation. This paper is fairly recent, being published in March 2017; therefore, there are many possibilities on where the research can go in the future. An active memory leak was injected by inserting lines 3, 4 and 6 into the guetzli.cc file as shown in Figure 3.

```
1   case 3: {
2       // RGB
3       int *leak_ptr;
4       fprintf(stderr, "Reached RGB area\n");
5       for (int y = 0; y < *ysize; ++y) {
6         leak_ptr = new int[10];
7         const uint8_t* row_in = row_pointers[y];
8         uint8_t* row_out = &(*rgb)[3 * y * (*xsize)];
9         memcpy(row_out, row_in, 3 * (*xsize));
10      }
11      break;
12    }
```

Figure 3: Active Memory Leak Injection into Guetzli Code Snippet

No corresponding memory deallocator was called to free the memory after this section. Passive memory leak injections were implemented by commenting out four instances of the png_destroy_read_struct() function which was the only function responsible for memory deallocation in the

guetzli.cc file. The results of the active and passive tests are shown in Table 4 below.

Table 4: Guetzli Passive and Active Memory Injection Results

Guetzli		Beaglebone Black	Intel Edison	VM	El Capitan
Passive	Definitely Lost (bytes)	1272	1168	1680	3256
	Indirectly Lost (bytes)	395525	382236	396593	396968
	Possibly Lost (bytes)	0	13320	0	2064
Active	Definitely Lost (bytes)	10320	10320	10320	0
	Indirectly Lost (bytes)	0	0	0	0
	Possibly Lost (bytes)	0	0	0	2064

The passive test cases were implemented uniformly across all devices yet resulted in marginally different memory leak values. This was unlike the FANN passive test case where the results were the same on the maker devices and the laptops respectively. Whilst perplexing at first, this discrepancy can be attributed to different versions of the Libpng dependency which implements the png_create_info_struct() dissimilarly across the devices. Deeper knowledge of Libpng would be needed in order to fully explain this difference and is beyond the scope of the investigation. The fact that valgrind was able to detect memory leaks on the devices within a reasonable range is still a significant result. The active test output across all platforms matched the expected output except on the Mac running El Capitan due to debugging flag errors as explained in Appendix D.

7. CASE STUDY 3 - SMARTBED

In order to further investigate Valgrind's performance on maker devices, the EEclipse team measured memory leaks on an Intel Edison-based networked system. Networked systems, which are constantly running, are prone to system crashes due to memory leaks accumulating over time and exhausting the system's resources. As a result, we ran Valgrind on a UCLA student-developed Smart Hospital Bed project. Unexpected hospital bed exits result in many injuries to patients around the world, resulting in extended hospital stays and increasing medical bills. The SmartBed alerts doctors and nurses if patients attempt to get out of their bed when unsupervised. The SmartBed utilizes a machine learning, sensor network to track a patient's location, orientation, and movements. The SmartBed uses five Intel Edisons, four of which are attached to their own 9-degrees-of-freedom (9DOF) sensor. The four Edisons are attached to the underside of the hospital bed to measure bed movements, while the fifth Edison is utilized as a server. The server collects data sent from the four client Edisons, and runs the FANN algorithm in order to determine the fall risk of a patient. The system architecture can be seen in Figure 4.

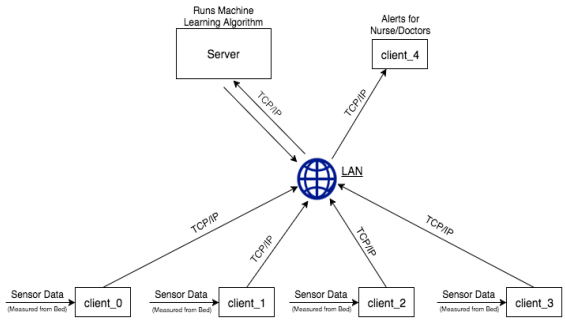

Figure 4: SmartBed High-level System Architecture

The SmartBed system is a networked system that requires operation in the "always on" state. As mentioned earlier, a system that is constantly running 24-hours a day is highly prone to system crashes due to memory leak accumulation. In a healthcare application such as this one, a system crash could result in serious injury to a patient or even death. Therefore, healthcare systems developed on low-resource devices, such as the Edison, must be memory leak free. As a result, we have taken the SmartBed system and run Valgrind's `memcheck` program in order to test for memory leaks. The SmartBed system was also ported over to the BeagleBone Black for a leak check comparison. The SmartBed server program was found to contain a vast number of memory leaks within its source code, while the client source code was memory leak free. After conducting this examination, a SmartBed team member was notified and he was able to quickly correct many of the errors and ensure greater stability in the system.

In terms of Valgrind's `memcheck` comparison performed on the SmartBed application between the Edison and the BeagleBone, the results were the same for both before and after some memory leak fixes. The results can be seen in the Appendix under Table A.5. While, at the time of writing, not all of the leaks in the SmartBed application have been fixed, it is clear that the Valgrind `memcheck` tool performed well enough on both the Edison and the BeagleBone in order to provide useful memory leak analysis. The SmartBed team is now able to fix the mistakes that were missed during initial development. However, due to the findings from the FANN Case Study, we cannot guarantee that if all memory leaks reported by Valgrind are fixed that the SmartBed application will be memory leak free.

8. THREATS TO VALIDITY

8.1 - External Validity

While we acknowledge that we only tested two maker devices (BeagleBone Black and Intel Edison) and only evaluated memory leaks on two open source C/C++ projects (FANN and Guetzli), this paper itself serves as an external validity to test Valgrind's performance as we compared it

among different platforms. We weren't able to incorporate as many maker devices as possibles due to limited resources available to the team. But we tried to evaluate memory leaks on some of the more popular open source libraries. FANN and Guetzli are both memory intensive and FANN works well with IoT devices. And we believe that understanding Valgrind performance on large scale open source applications can be valuable to other programs.

8.2 - Internal Validity

When testing the number of memory leaks in FANN, Guetzli, and SmartBed, the leaks recorded are assuming that Valgrind identified all memory leaks. However, as proven in the FANN case study, Valgrind was not completely accurate in reporting memory leaks. As a result, there are factors within Valgrind that we did not examine that prevent us from guaranteeing all memory leaks were found. Furthermore, the accuracy of Valgrind on the maker devices cannot be fully identified in this report due to this found Valgrind issue.

8.3 - Construction Validity

Our goal was to measure the performance of Valgrind's `memcheck` tool on common maker devices. In order to do so, we identified different types of memory leaks that occur in code. We then took those common leaks and implemented them into the nine test cases mentioned in Section IV. However, while we were able to identify "possibly lost" bytes in the two case studies, we were unable to replicate "possibly lost" bytes in our test cases in Section IV. Furthermore, there is also a chance that other types of memory leaks exist that we did not cover. In the end, we were able to develop strong and extensive test cases; however, we cannot guarantee the test cases exhaust all types of memory leaks.

9. RELATED WORKS

9.1 - Valgrind Performance Evaluation

Lu et al. built a benchmark suite called *BugBench* that systematically evaluate software bug detection tools. *BugBench* provides a good general guidelines on the criteria for selecting representative bug benchmarks as well as the metrics in evaluating bug detection tools such as Valgrind [12]. Although the focus of the *BugBench* paper was to evaluate the benchmark suite it proposed, it also evaluated the benchmark on CCured [13], Purify [14], and Valgrind. Eight memory-related applications from open source community were used as test cases as summarized in Figure 5 below. Valgrind misses a stack buffer overflow and a global buffer overflow. One conclusion can be drawn from this study that Valgrind handles heap objects much better than it handles stack and global objects. One interesting observation on Valgrind was that if the buffer is not overflowed significantly, Valgrind will miss it. With moderate overflow, Valgrind catches the bug after a long path of error propagation, not the root cause. Only with significant overflow can Valgrind detect the root cause. However Valgrind is the easiest to use among the three and requires no re-compilation. Another related work is included in the Valgrind source code [15]. Valgrind developers provide a few performance test cases. These test cases are meant for developers who add new tools to the Valgrind framework and want to evaluate their tool's performance. Two of the notable performance test cases include running memcheck on code which compresses and decompresses data and on a small and fast C compiler. However, the output of these performance tests is solely execution time. While *BugBench* did validate the selection of their benchmarks on Valgrind, it was solely on one machine and across various software defects including buffer overflows, stack smashing, double frees, uninitialized reads, memory leaks, etc. In this paper we evaluated Valgrind's performance in terms of memory leak identification ability and execution time across multiple platforms.

	Catch Bug?			False Positive			Pinpoint The Root Cause (Detection Latency(KInst)[1])			Overhead			Easy to Use		
	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured	Valgrind	Purify	CCured
NCOM	No	No	Yes	0	0	0	N/A	N/A	Yes	6.44X	13.5X	18.5%	Easiest	Easy	Moderate
POLY	Vary[2]	Yes	Yes	0	0	0	No(9040K)[2]	Yes	Yes	11.0X	27.5%	4.03%	Easiest	Easy	Moderate
GZIP	Yes	Yes	Yes	0	0	0	No(15K)	Yes	Yes	20.5X	46.1X	3.71X	Easiest	Easy	Moderate
MAN	Yes	Yes	Yes	0	0	0	No(29500K)	Yes	Yes	115.6X	7.36X	68.7%	Easiest	Easy	Hard
GO	No	Yes	Yes	0	0	0	N/A	Yes	Yes	87.5X	36.7X	1.69X	Easiest	Easy	Moderate
COMP	No	No	Yes	0	0	0	N/A	N/A	Yes	29.2X	40.6X	1.73X	Easiest	Easy	Moderate
BC	Yes	Yes	Yes	0	0	0	Yes	Yes	Yes	119X	76.0X	1.35X	Easiest	Easy	Hardest
SQUD	Yes	Yes	N/A[3]	0	0	N/A[3]	Yes	Yes	N/A[3]	24.21X	18.26X	N/A[3]	Easiest	Easy	Hardest

Figure 5: Evaluation of Valgrind, Purify, and CCured using *BugBench*.

9.2 - Valgrind vs Other Debugging Tools

Nethercote et al. evaluated Valgrind's ability to shadow every register and memory value with another value that describes it, which is a design feature that distinguishes Valgrind from other frameworks. [15] and [9] also compared Valgrind to several other tools such as Purify and LIFT [16]. LIFT is built with StarDBT and is much faster than `memcheck` partly because it does a simpler analysis [15]. And although Purify is a memory-checking tool similar to `memcheck`, Nethercote mentions that Purify doesn't track "definedness" of values through registers, and as a result, Purify detects undefined value errors less accurately than `memcheck`. Developers have also found that Purify is much more likely to change the behavior of users' program, even causing it to produce incorrect results when the program is correct. Clause et al. implemented a prototype tool called *Leakpoint* to address memory leakage issues [4]. The system was built on top of Valgrind such that it was able to track pointers to dynamically allocated memory and thereby detect memory leaks. As an extension to the Valgrind functionality, the system also directed the developer to the locations where the leakage could be fixed. The authors found that the system was able to detect at least as many errors as the Valgrind system as well as provide an effective means of fixing the leaks. After evaluation it was found that the cost of the added functionality was a runtime overhead of 100-300 times normal operation. This is in comparison to the reported 30 times overhead as a result of running Valgrind independently. *Sleigh* is another memory leak detection tool created by Bond et al [17]. The system tracks the time since last use to find memory leaks. It is also more lightweight than Valgrind and was designed for use during performance runs as opposed to test runs. As a result, *Sleigh* does not pick up as many errors as Valgrind, but only adds 11 - 29% overhead execution time. Sui et al. implemented a static memory leakage detector, Saber [18]. Saber differs from the above mentioned detectors through its use of sparse value flow graphs which enable the system to detect memory leaks.

The system was able to detect leaks with a false positive rate of 18.7%, however runtime performance tests were not conducted. In our paper, we aimed for depth instead of breadth. We found that implementing memory leak tools to be time intensive, and install several debugging tools would not have left enough time for testing our test suite. Therefore we focused on an in depth analysis of Valgrind due to its popularity within the computer science community and leave investigations of other debugging tools to related work.

10. CONCLUSION AND FUTURE WORK

In this paper, we have presented an evaluation of Valgrind performance on commercially available maker devices. We developed test cases that look at memory leaks specifically and have found that maker devices accurately identify memory leaks in our standardization cases and outperform the OS X system. Open source platforms were used to investigate memory leaks on real, large-scale applications. A case study on FANN has shown that as we begin to scale up to larger systems of code we observe performance variations across our test devices. However, the maker devices provide similar results, while the VM and El Capitan output similar memory leaks. Furthermore, Valgrind is found to have false negatives when active memory leaks are injected into certain spots in the FANN test cases. In a case study of Guetzli, the performance of Valgrind across all test cases vary for the passive leak test cases, while the active memory leak test cases report consistent leaks across the BeagleBone, Edison, and VM. Once again, El Capitan fails to perform as expected and reports no memory leaks in the Guetzli case study. Furthermore, the EEclipse team is able to help prove the Valgrind's viability in a real-world application on both the BeagleBone and the Edison. The tests run on the SmartBed application have provided vital information to the SmartBed team, so that they can remove memory leaks in their system. While we acknowledge that we only tested two maker devices and evaluated memory leaks on three open source C/C++ projects, this paper provides a first take on evaluating Valgrind on various maker platforms.

Current investigations are focused on FANN and Guetzli. However, with more time and resources available to us, this evaluation can be extended to other large scale applications. We plan to investigate additional maker devices to further improve the comparison results. We also will like to know why Valgrind can't detect some injected active memory leaks in FANN test cases. The investigation into the Raspberry Pi, OpenCV needs to be reopened in the future. We think that understanding the performance degradation with increasing scale code across a variety of platforms can aid developers in their time writing software for the rapidly growing IoT domain.

The detailed implementation and our benchmark test script is available at https://github.com/vivi51123/CS230-Software-Engineering. The website features documentation on how to install and use Valgrind. As the website evolves, more documentation and features on the comparison of Valgrind across different embedded maker platforms will be available. We hope this comparison study can provide some insights for developers in the IoT domain.

11. REFERENCES

[1] "MSDN: Microsoft Development, MSDN Subscriptions, Resources." URL: https://msdn.microsoft.com/en-us/library/ms859408.aspx

[2] J. Clause and A. Orso, "LEAKPOINT: Pinpointing the Causes of Memory Leaks," *ICSE*, pp. 515–524, 2010.

[3] "The nine types of memory leaks," *Software Verify*, 2011. URL: http://blog.softwareverify.com/the-nine-types-of-memory-leak/

[4] J. Zhao, Y. Jin, and R. M. Jr, "Injecting Memory Leaks to Accelerate Software Failures," *IEEE International Symposium on Software Reliability Engineering*, pp. 260–269, 2011.

[5] T. Tsai, K. Vaidyanathan, and K. Gross, "Low-Overhead Run-Time Memory Leak Detection and Recovery," *Pacific Rim International Symposium on Dependable Computing*, 2006.

[6] R. Matias, A. Andrzejak, F. Machida, D. Elias, and K. Trivedi, "A Systematic Differential Analysis for Fast and Robust Detection of Software Aging," *IEEE 33rd Int. Symp. Reliab. Distrib. Syst.*, 2014.

[7] A. Bovenzi, "On the Aging Effects due to Concurrency Bugs : a Case Study on MySQL," *IEEE 23rd Int. Symp. Softw. Reliab. Eng.*, pp. 211–220, 2012.

[8] F. Qin, S. Lu, and Y. Zhou, "SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs," International Symposium on High-Performance Computer Architecture, 2005.

[9] J. Seward. Valgrind. "Question #4: Other Tools?" URL: http://www.Valgrind.org

[10] S. Nissen, "Implementation of a Fast Artificial Neural Network Library," *Department of Computer Science University of Copenhagen*, 2003.

[11] Alakuijala, J., Obryk, R., Stoliarchuk, O., Szabadka, Z., Vandevenne, L., and Wassenberg, J. "Guetzli: Perceptually Guided JPEG Encoder." *CoRR, abs/1703.04421*, 2017.

[12] S. Lu and Z. Li, "*BugBench*: Benchmarks for Evaluating Bug Detection Tools", *UIUC*, 2005.

[13] G. C. Necula, S. McPeak, and W. Weimer. "CCured: Type-safe retrofitting of legacy code." *POPL*, 2002.

[14] R. Hastings and B. Joyce. “Purify: Fast detection of memory leaks and access errors.” *Usenix Winter Technical Conference*, 1992.

[15] N. Nethercote, J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM SIGPLAN*, v.42 n.6, 2007.

[16] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. “Lift: A low-overhead practical information flow tracking system for detecting security attacks.” *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[17] D. Bond and K. McKinley, “Bell: Bit-Encoding Online Memory Leak Detection,” *ASPLOS*, pp. 61–72, 2006.

[18] Y. Sui, D. Ye, and J. Xue, “Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis,” *ISSTA*, pp. 254–264, 2012.

[19] Itseez, Open Source Computer Vision Library, “OpenCV DevZone,” 2014. URL: http://code.opencv.org/projects/opencv/issues

12. APPENDIX

Appendix A: Test Case Tables

Table A.1: Expected Number of Bytes Lost For Each Test Case

Case	Expected Memory Lost (Bytes)
Leak 1	100
Leak 2	8
Leak 3	def: 8, indirect: 124
Leak 4	def: 8, indirect: 124
Leak 5	8
Leak 6	18
Leak 7	18
Leak 8	100
Leak 9	8

Table A.2: Leak Results Across all Platforms From Written Test Cases in C++

Leak #, Bytes Lost		BeagleBone Black	Intel Edison	VM	El Capitan
Leak 1	Definitely Lost	100	100	100	100
	Indirectly Lost	0	0	0	0
Leak 2	Definitely Lost	8	8	8	8
	Indirectly Lost	0	0	0	0
Leak 3	Definitely Lost	8	8	16	16
	Indirectly Lost	124	124	124	124
Leak 4	Definitely Lost	8	8	16	16
	Indirectly Lost	124	124	124	124
Leak 5	Definitely Lost	8	8	8	8
	Indirectly Lost	0	0	0	0
Leak 6	Definitely Lost	18	18	18	18
	Indirectly Lost	0	0	0	0
Leak 7	Definitely Lost	18	18	18	18
	Indirectly Lost	0	0	0	0
Leak 8	Definitely Lost	100	100	100	100
	Indirectly Lost	0	0	0	0
Leak 9	Definitely Lost	8	8	8	8
	Indirectly Lost	0	0	0	0

Table A.3: True Runtime of Test Cases

True Runtime	BeagleBone Black	Intel Edison	VM	El Capitan
Leak 1 (ms)	0.464	0.54552	0.00112	0.00094
Leak 2 (ms)	0.484	0.64104	0.00242	0.00302
Leak 3 (ms)	0.467	0.30804	0.00131	0.00113
Leak 4 (ms)	0.495	0.36759	0.00371	0.00170
Leak 5 (ms)	1.806	1.02149	0.03352	0.03084
Leak 6 (ms)	0.490	0.63075	0.00313	0.00309
Leak 7 (ms)	0.531	0.54267	0.00230	0.00103
Leak 8 (ms)	0.525	0.53025	0.00173	0.02080
Leak 9 (ms)	2.511	0.87408	0.00589	0.02167

Table A.4: Runtime of Test Cases with Valgrind

Valgrind Runtime	BeagleBone Black	Intel Edison	VM	El Capitan
Leak 1 (ms)	8.880	11.919	0.678	0.613
Leak 2 (ms)	29.618	26.872	1.713	1.366
Leak 3 (ms)	9.409	14.150	0.767	1.802
Leak 4 (ms)	24.507	30.680	1.723	0.804
Leak 5 (ms)	277.323	485.789	55.827	47.727
Leak 6 (ms)	23.359	27.607	1.576	1.005
Leak 7 (ms)	15.718	24.903	2.434	0.996
Leak 8 (ms)	8.192	12.766	0.753	0.648
Leak 9 (ms)	33.053	39.273	2.438	3.525

Table A.5: SmartBed Application Memory Leak Results

SmartBed		Intel Edison	BeagleBone Black
Memory Leaks on Original SmartBed System	Definitely Lost (bytes)	2360	2360
	Indirectly Lost (bytes)	5280	5280
	Possibly Lost (bytes)	0	0
Memory Leaks on SmartBed System after Valgrind Test and Initial Leak Fixes	Definitely Lost (bytes)	1384	1384
	Indirectly Lost (bytes)	3492	3492
	Possibly Lost (bytes)	0	0

Appendix B: Attempted Open Source Libraries

During the process of evaluating Valgrind performance, our team encountered several issues that could not be resolved by the end of the quarter. We present our findings here and will investigate and further study these platforms and maker devices in our future work.

Besides FANN and Guetzli, we also tried to test OpenCV as it is one of the most popular libraries used by programmers in the world and has an estimated number of downloads exceeding 14 million with a fairly complicated library. We looked at some of the memory bugs from OpenCV DevZone [19] such as bug#3847, bug#3961, bug#4489, and bug#4250. These issues were reported by developers and were fixed and

merged to the newest version of OpenCV on the gitHub repository. The memory leaks were happening inside the library code. We came up with a few main functions that calls the methods in the OpenCV API, wanting to see if Valgrind was able to detect the original bug mentioned by the bug log. We also tried to mimic the bug by commenting out some of the lines in the source code. However, it seems like Valgrind didn't pick up anything when the API is modified. This might be an issue that the API isn't being compiled everything we run the main program. Unfortunately recompiling the API typically takes around three hours on our maker devices, so with the time we had, we decided that it wasn't feasible.

We also looked into TensorFlow, which is a popular open source framework developed by Google for deep learning. Deep learning applications allocate large amount of memory for computation and memory leak bugs can cause severe issues during the runtime of the applications. However, TensorFlow is not a good fit for embedded devices because it is designed for devices with much richer computational resources and its applications usually run on multiple GPUs. TensorFlow is also designed for 64-bit systems and has less support for 32-bit systems such as our targets Intel Edison and Beaglebone, so we decide not to go deeper in TensorFlow.

Another candidate we considered is tiny-DNN, which is an open source C++14 implementation of deep learning. It is suitable for deep learning on limited computational resource, embedded systems and IoT devices. It has advantage such as reasonably fast without GPU, portable and header-only, easy to integrate with real applications and simply implemented. However, it has to be built from source and we encountered some build/compile issues when building with maker devices and they cannot be resolved in a short time so we decided that it wasn't feasible for this study. Other candidates such as Delib and macchina.io are also in similar situations, but they are definitely worthwhile to research on in our future work.

Appendix C: Attempted Maker Devices (Raspberry Pi)
Below, we present several error outputs from testing Raspberry Pi. Specifically, we looked at three different Raspberry Pi OS versions: Pi 3 Ubuntu MATE, Pi 2 Raspbian Wheezy, and Pi 3 Raspbian Jessie.We were able to accomplish leak tests on the most trivial cases. When we attempted to run leak 1 through 9 on the Raspberry Pi 3 Ubuntu MATE, for example, we get error messages such as this:

```
root@kunal-
desktop:/home/kunal/Desktop/Cpp_MemLeak_Tests#
valgrind --leak-check=yes ./leak1
==25901== Memcheck, a memory error detector
==25901== Copyright (C) 2002-2015, and GNU GPL'd, by
Julian Seward et al.
==25901== Using Valgrind-3.12.0 and LibVEX; rerun with -
h for copyright info
==25901== Command: ./leak1
==25901==
vex: priv/guest_arm_toIR.c:13352 (decode_V8_instruction):
Assertion `szBlg2 <= 3' failed.
vex storage: T total 131609816 bytes allocated
vex storage: P total 0 bytes allocated
valgrind: the 'impossible' happened:
  LibVEX called failure_exit().
```

Errors such as this were experienced across all three different Pi systems for every nontrivial leak test case. We concluded that the Pi is incapable of running full Linux operating systems, instead relying on special flavors of Ubuntu and Debian, and that causes Valgrind to fail for complex cases. Therefore, we ultimately decided after much tinkering that it was best not to pursue the Raspberry Pi as a testing platform.

Appendix D: Valgrind on El Capitan failed to detect injected active memory leak
As mentioned in Case Study 1 & 2, El Capitan is the only platform where Valgrind failed to detect the injected active memory leak. We found that when running valgrind on executables compiled with these two libraries (FANN and Guetzli), a warning message was reported as warning: no debug symbols in executable (-arch x86_64). We tested the same injected active memory leak on a separate file with standardized compilation flags and valgrind was able to detect the memory leak. At this point, we can conclude that due to less support on MacOS, Valgrind failed to extract debug symbols from the libraries so that it cannot detect active memory leaks in such situations. In the future work, we will dig deeper into this problem and find out more detailed explanation for this failure.

Appendix E: Instructions on running test cases
The steps to run the the leak examples that were developed by the team are shown below.
For example: To install Valgrind and running the test program "leak 1.cpp", do the following.
MAC
1.) brew install valgrind
2.) g++ -g -Wall -o leak1 leak1.cpp
3.) valgrind --leak-check=yes ./leak1

Ubuntu
1.) sudo apt-get install valgrind
2.) g++ leak1.cpp -o leak1
3.) valgrind --leak-check=full ./leak1

Appendix F: Individual Contributions
Finally the individual contributions to this project are the following.

Vivi Tzu-Wei Chuang
- Researched OpenCV bug log and composed test cases.
  - bug#_main.cpp and bug#_source.cpp
- Researched on other open source libraries version history to decide on the appropriate application to be tested. (Notepad++)
- Raspberry Pi 2 Wheezy implementation.
- Research on related work.

Greg Cusack
- Test Case 5 (try/catch) Development.

- Implementation and execution of Valgrind on Intel Edison and VM.
- FANN case study development.
    - Passive/active leak injections.
    - FANN false negative identification (w/ Nathan).
- SmartBed configuration and analysis.
- Raspberry Pi 3 Model Implementation.

Kunal Patel

- Defined the framework and direction of this project.
    - I.e. what we want to test, the scope of this project, what we want to test and measure, etc.
- Invested many hours into implementing Valgrind on the Raspberry Pi 3.
    - Tried various versions of Valgrind, building from source code in an attempt to find a compatible version.
    - Installed latest compatible version and performed extensive research into Valgrind testing ability. Ultimately found that the Pi was only capable of leak checking very trivial cases, and that the system could not be used for this study.
- Heavily involved in the research of this topic. This is an evaluation paper and required a good amount of study into Valgrind mechanism.
- Research and design of open source project case studies and searched bug logs for source code memory leaks that were fixed by developers.
- Contributed greatly to the writing of the paper.
- Contributed greatly to the writing of the powerpoint presentation.

Nathan Pilbrough

- Test case development
    - Sourcing various forms of memory leak occurrences.
- Sourcing open source projects for use in the evaluation.
- Beaglebone wrangling (configuration, setup and testing).
- Researched how to implement memory leak injections.
- Guetzli case study development.
    - Passive/active leak injection.
    - Related Work contribution.

Zhengshuang Ren

- Implementation of Valgrind on MacOS El Capitan and research on limitations of running Valgrind on MacOS.
- Developed TensorFlow and Tiny-DNN.
    - Build and run valgrind on them.
    - Either found memory leaks or injected memory leaks.
- Testing on El Capitan (test cases, case studies).
- Experiment on the causes of failures on El Capitan.