Graphics Card Based Fuzzing

Ryan Mower, Ben Bernard, Jeremy Straub
Department of Computer Science
North Dakota State University
Fargo, ND, USA
ryan.mower@ndsu.edu, ben.bernard@ndsu.edu, jeremy.straub@ndsu.edu

***Abstract*—Fuzzing is the art of creating data and using that generated data as input for a target program. The goal behind this is to crash the program in a manner that can be analyzed and exploited. Software developers are able to benefit from fuzzers, as they can patch the discovered vulnerabilities before an attacker exploits them. Programs are becoming larger and require improved fuzzers to keep up with the increased attack surface. Most innovations in fuzzer development are software related and provide better path coverage or data generation.**

This paper proposes creating a fuzzer that is designed to utilize a dedicated graphics card's graphics processing unit (GPU) instead of the standard processor. Much of the code within the fuzzer is parallelizable, meaning the graphics card could potentially process it in a much more efficient manner. The effectiveness of GPU fuzzing is assessed herein.

I. Introduction

Along with the increase in the development of computer applications has come an increase in the number of vulnerabilities that these programs contain. These new vulnerabilities open doors for attackers to exploit and gain access. Vulnerable computer applications can leak sensitive information such as names, credit cards, bank information, social security numbers and much more. This information is a target for criminals. After an attacker compromises a target, they can cause severe financial and reputational damage. A study by Livshits and Lam [1] discussed the value of this information and how many business financial losses range into the millions of dollars, due to its sensitivity. In order to prevent criminals from gaining access to the sensitive data within these programs, organizations will need to allocate resources towards cyber defense.

When a zero-day, an exploitable vulnerability that has not been released to the public yet, is discovered it can prospectively be sold to criminals, companies, or nation states for amounts ranging between $5,000 and $250,000 [2]. These organizations are willing to spend money for these zero-days attacks, because they are able to patch vulnerabilities before attackers are able to exploit and compromise them – or, in the case of state offensive actors and criminals, use them before they are patched. Organizations who successfully patch vulnerabilities and prevent criminals from stealing information save money, prevent lawsuits, and increase their reputation. In order to successfully discover new vulnerabilities, before a prospective attacker does, companies must innovate and move beyond the current methods of discovering new vulnerabilities.

One valuable cyber defense tactic is automated vulnerability discovery, which allows organizations to discover flaws within their programs and applications before attackers do. Many of these autonomous vulnerability discovery programs are open source, meaning attackers also have access to such tools.

A common dynamic analysis discovery technique is called fuzzing. Fuzzing is the art of generating data that is used as input toward a specific program to test different control paths and make sure the program continues to run properly. The American Fuzzy Lop (AFL) is a popular fuzzer with a unique data generation technique. AFL generates data based upon a test case that is provided as original input for the fuzzer. As this data makes it further throughout the target's program parser, AFL keeps track of such progress and continues to modify it accordingly. An example that highlights AFL's data generation is when the input string, "Hello" was used for a JPEG parser. Through the cycles, the input made it further through the parser and eventually became a valid JPEG image [3].

In addition to software innovations, hardware enhancement can facilitate increased performance of many automated vulnerability discovery tools. This paper considers the use of a graphics card's GPUs to accelerate the parallelizable processes while fuzzing. This may improve the performance of the automated vulnerability discovery tool, increasing the number of discovered vulnerabilities in a period of time and the speed of vulnerability discovery.

II. Background

Automated vulnerability discovery can be broken down into two main categories: static and dynamic analysis. Each is now discussed. Then, fuzzing algorithms and CPU versus GPU performance are considered.

A. Static Analysis

Static analysis is used when the tester can obtain and

evaluate the source code of the program. Using this, the code can be analyzed and flaws and vulnerabilities within the program, that could be exploited later, can be discovered.

Static analysis extracts information from the program being tested to direct the fuzzing process without actually running the program. There are different techniques and programs that can automate this process. Pixy, for example, is a program that uses "flow sensitive, interprocedural and context-sensitive dataflow analysis to discover vulnerable points in a program" [4]. There are numerous other static vulnerability discovery tools, such as one created by Stanford University. Their static analysis tool "is the first practical static security analysis that utilizes fully context-sensitive pointer analysis result" and has the added benefit of having a simple graphical user interface [1].

Static analysis is not a complete solution as many vulnerabilities can be missed because the static analysis tools are not analyzing the program during runtime. This makes automated static analysis a semi-effective vulnerability discovery technique, due to techniques' low accuracy and high rate of false positives [5].

B. Dynamic Analysis - Fuzzing

Alternatively, there are dynamic analysis techniques such as fuzzing. Fuzzing, which is the most commonly used dynamic analysis vulnerability discovery process, can be implemented with black-box, white-box, or grey-box testing approaches.

Black-box testing is an uninformed testing strategy that generates both valid and invalid input for target programs and does not require any program analysis [6]. The black-box fuzzer is unaware of the internals of the program it is testing; it only observes the input and output behavior of the program. The code within the fuzzer submits numerous forms of generated data into the target program as standard input in an attempt to crash the program. Once the generated data has been submitted, there is an algorithm within the fuzzer that mutates and alters the previous data to create a new data set.

White-box fuzzing analyzes the internals of the program being tested and the information gathered when executing that program. As a result, white-box fuzzing utilizes more system resources to complete than black-box fuzzing [7]. Symbolic execution tools are used to expose the control flow of the tested software and treat inputs as variables and branch conditions as variable constraints. Unfortunately, even average programs can have thousands of challenging branch conditions which consumes a significant amount of time [3].

Grey-box fuzzers obtain some limited static and dynamic information internal to the tested program and use this approximated information to speed up the processing of testing more inputs, as comparted to a pure white-box approach [7]. AFL is an example of a grey-box fuzzer. The increased performance comes with the cost that the approximated information can be wasted on redundant or malformed inputs [3].

C. Fuzzing Algorithms

Fuzzers use a variety of algorithms to perform their testing tasks. These algorithms are integral to the success of the fuzzer because they generate the actual data that is used to test programs' integrity. Symbolic and concrete execution are both traditional methods for test data generations.

Symbolic execution remembers which code triggers certain code paths, increasing the amount of code covered. On the other hand, concrete execution uses actual data. Concolic testing is a hybrid between the two (concrete and symbolic). It combines both techniques with the two parts feeding data to each other, improving the overall data set [8]. Concolic testing is also known as dynamic symbolic execution.

Once the data set is generated, the fuzzer repeats the data submission process and cycles back to the beginning unless a user terminates the program or a stop option is configured and triggered within the program [9]. Concolic execution is a unique and different approach to fuzzing. Instead of randomly generating and inputting data, it will map the control flow of the target program. When doing this, concolic execution remembers which values trigger certain parts of the code, allowing the data to test a wider range of control paths throughout the program [8].

When a crash is occurs, in many cases it may be a false positive. Because of this, programs and people have to go through every crash instance to confirm that a real vulnerability is present. Concolic testing is slower than many other techniques. One approach to compensate for this is to narrow the scope of the testing by letting the user specify parts of the software to test [7].

Many fuzzers contain techniques that include code coverage analysis, property checking, checksums, accelerated fuzzed execution, instruction emulation, and concolic execution [9], [10].

Fuzzing is a way to prospectively discover zero-day vulnerabilities. For defenders to stay ahead of attackers, they will need to use tools such as efficient fuzzers that employ innovative techniques. Improving the efficiency and accuracy of fuzzing is vital because they currently produce significant error [11]. Selecting which fuzzer to use can be a difficult process because there are so many. A few examples are AFL, Cert Bff Fuzz, Cluster Fuzz, and OSS-Fuzz. There are also a number of closed-source fuzzers [7].

D. CPU vs GPU Performance

Hardware improvements can aid fuzzer performance. Processor vendors are constantly refining their central processing units (CPU) to add more cores and achieve greater

speeds. Processor capability improvement enhances overall fuzzer performance.

An alternate approach is to use a dedicated graphics card for fuzzing. These cards are designed with numerous cores and the ability to run many operations in parallel. When cracking passwords, the utility of this approach is demonstrated by a tool called Hashcat (which is found in Kali Linux). This program can operate with a CPU, however, those using a graphics card achieve much higher performance level. This is due to the ability of modern GPUs to perform hundreds of billions of floating point operations per second [12].

GPU cards have been instrumental in allowing complex video games to run at reasonable speeds. Video games are an excellent example of parallelization because when rendering all of the shapes, each area is independent of all the other areas. Emphasizing this point even further is that the pixel itself is oblivious towards the bigger picture that it is constructing.

The concept of using a graphics card for fuzzing would appear to be a prospectively excellent way to increase the performance of the fuzzer, allowing for an increase of vulnerabilities discovered. Bohme, Pham and Roychoudhury [6] discuss how "converge-based greybox fuzzing is highly parallelizable because the retained seeds represent the only internal state". If they are able to discover more vulnerabilities, companies will be able to patch more of their system flaws before they are exploited. This will further enhance the difficulty of attempting to compromise the system.

III. Implementation

In order to adequately compare the performance of the GPU to the CPU for fuzzing, the inner workings of fuzzers must be well understood. To begin the physical testing, the CPU based version of AFL was executed first. Testing AFL was essential because a baseline for comparison among the two different fuzzers had to be established. AFL was the selected fuzzer due to its popularity and high ratings throughout the fuzzing community. Binutils was the first program fuzzed with AFL due to its numerous tutorials and walkthroughs online. Due to the well written documentation and tutorials online, running an instance of AFL while targeting Binutils was quite trivial. Successfully implementing the GPU component required the code to be parallelizable.

Before starting on the fuzzer development, a simple proof of concept was performed. This was a piece of software that adds the elements within a vector together via the CPU and then does the same calculation on a GPU. Both of these functions were timed to see which processing unit could complete the task quicker.

The next step within the project was to create a pseudorandom data generator that utilized the GPUs. NumPy and cuPy were two similar libraries that were designed for Python. Both libraries generate pseudorandom numbers, however, numPy is designed for the CPU while cuPy utilizes the CUDA Toolkit to harness the power of the GPU. To create the data generator, both numPy and cuPy would select a random number and these random numbers would correspond to the length of the word along with which characters to append to the line of data. This random data was printed out to the console (and could even potentially be fed into a program to see if it would cause the program to crash).

Within the two different code libraries, timers that triggered at the beginning of the data generation and terminated once the random data was created were created. After testing the two programs, cuPy seemed to be trailing numPy despite the fact that cuPy was utilizing the power of the GPUs. After further investigation, this seemed to be because numPy is an incredibly efficient C program. Due to numPy outperforming cuPy, additional testing had to be implemented. This approach included running a local executable compiled with NVCC (a CUDA compiler) to generate the random data, but it still was not fast enough to beat the CPU based generator with numPy.

When analyzing the programs, there are several explanations as to why the GPU based fuzzer fails at beating the CPU one. One of the reasons is that the graphics card takes time to get up to full speed and is bottlenecked by all the data transfers between it and the CPU. Also, numPy is an extremely well written C program, so it is already incredibly fast. In order to compete with numPy, lots of data along with well written GPU code will be necessary.

IV. Results

After selecting the AFL fuzzer, the chosen fuzzing target was the Readelf file located within the Binutils program. Binutils was the program of choice for testing due to the numerous tutorials and walkthroughs online. These tutorials were excellent step-by-step guides that assisted with the success of the AFL Fuzzer setup. However, once AFL was up and and fuzzing, no crashes were discovered. This could be a result because of its setup being so trivial, everybody knew how to fuzz the Readelf file from Binutils. From this, many vulnerabilities were discovered quickly and the creators of Binutils were able to patch such vulnerabilities quickly, making it incredibly difficult and time consuming to discover a vulnerability located within the Binutils program. Figure 1 is a snapshot of the AFL running.

The snapshot of the American Fuzzy Lop fuzzer, presented in Figure 1, illustrates a few key important details. The first key detail that is located in the top left box with the title

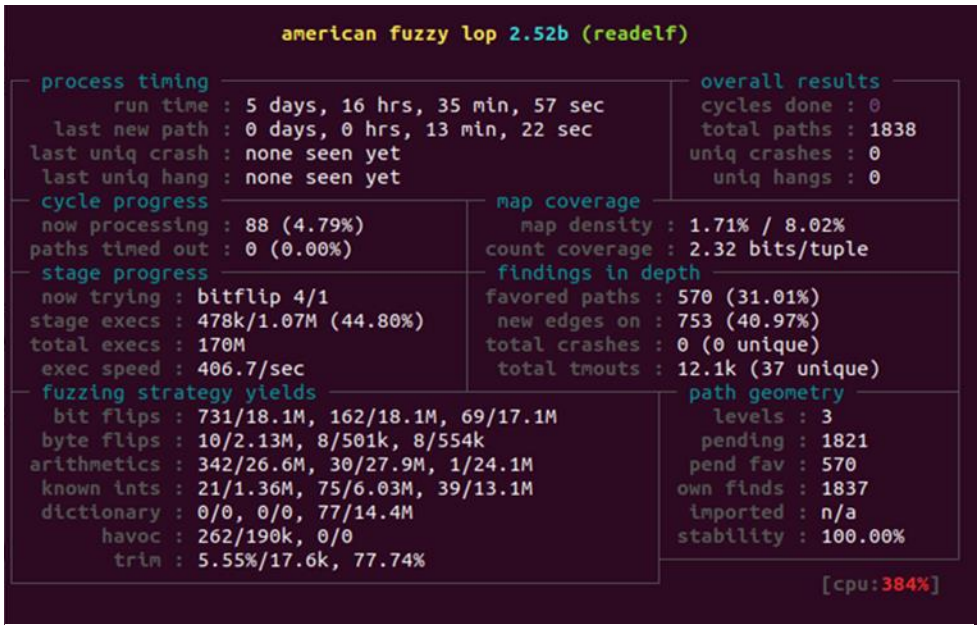

Fig. 1. American Lop Fuzzer progress display

"Process Timing" is that it has been running for five days, ten hours, thirty-five minutes and fifty-seven seconds, which is quite a long time. One line down within that same box it displays that it last discovered a new code path thirteen minutes and twenty-two seconds ago. Further down the vertical column, to the box titled "Stage Progress", the next important piece of information is that it is running at 406.7 executions per second, which is quite fast for most AFL instances.

Beneath the "Stage Progress" box is a box titled "Fuzzing Strategy Yields". Within this box, it reports how many bit and byte flips occurred along with a few other fields of data. Arguably the most important box, "Overall Results" is located at the top right of the screen shot. Inside this box, it displays the amount of cycles done (zero), the number of total paths (1821), the number of unique crashes (zero), and the number of unique hangs (zero).

A cycle is complete and the indicator is incremented when all of the generated data has been executed and the fuzzer has to mutate the data set for another cycle of fuzzing. It is important to note that one full cycle was not yet completed in Figure 1. Many testers will run their fuzzer instances for ten or more cycles to insure they have quality results. For this project, a complete cycle was not achieved due to time constraints.

The next goal was to develop a proof of concept system to evaluate the prospective superiority of the GPU-based fuzzer. To this end, a simple program that adds 1,000,000 elements in a vector with the CPU and then repeats the same process with a GPU was created and executed. After executing the program, the CPU function finished with a time of 32.11 seconds while the GPU function took 1.23 seconds. Next, two identical data generators were created, one for the CPU and the other for the GPU. The CPU generator was written with the numPy library and the other generator was written with the cuPy library. These libraries are quite similar as their syntax is the same with the exception that one is identified with cuPy and the other is identified by numPy.

When testing the two programs to generate 5000 lines of pseudorandom random data, the CPU based program took 0.0599 seconds until completion, while the GPU version took 1.4127 seconds. This seemed to be because the GPU version was bottlenecked by all the reading and writing of the lines of data. Additionally, numPy is an incredibly fast program written in C. To further the project, finding a GPU program that generates data faster than the numPy program is essential for success.

A GPU-based number generator was discovered and executed. This number generated 10,027 random numbers between one and zero. Next, a python script takes these numbers and turns them into random data in a similar fashion to the previous programs. The next steps consisted of modifying the CPU based fuzzer to write all of its random data to a file similar to how the GPU version was implementing to keep the two programs as similar as possible. Even with this version, the CPU data generator sill defeated the GPU version. The CPU fuzzer took 0.0319 seconds to generate 380 lines of data while the GPU version took 0.2968 seconds to generate the same amount of data.

V. Conclusions and Future Work

The work this far would tend to suggest that the current model of CPU-based fuzzing is preferable. Despite the discouraging results for GPU use attained so far, it is not entirely clear that the CPU based approach is superior and GPU-based fuzzers still prospectively have the potential to improve fuzzer performance. This is because the type of process that generates the random data can be parallelized, therefore making it an excellent candidate for GPU processing. If GPU fuzzing issues can be resolved and use becomes common, it is possible that there will be an increase in executions per second, vulnerabilities discovered per unit of time, and an overall increase in program security.

The next step in this work is to develop directly comparable fuzzers and fuzzer components from the ground up. Beyond

this, the testing protocol used above should remain relatively similar. A program will be written in C, utilizing the CUDA Toolkit that will generate pseudorandom data. When writing the new programs, efficiency and (for the GPU-based version) parallelization are key. The goal will be to compare a best-of-breed CPU version to a best-of-breed GPU version.

A larger testing data set will also be desirable. Fuzzing runs with similar inputs may complete at different speeds and each fuzzing run may produce different results due to the use of randomness by some of the fuzzing algorithms [13].

In the GPU version, intensive processes, such as data generation, could be performed on the GPU allowing the CPU to focus on other tasks. Once the GPU is able to successfully and efficiently generate data, the data process could be refined so that it never creates duplicate data. Another useful feature would keep track of which types of data trigger certain code paths so it could specialize the data generation to target certain locations within the program itself. Overall, if the GPU was able to handle most of the fuzzing process, it would improve the speeds of the fuzzer. If all the data could be generated by the graphics card, the CPU could focus on other main tasks, significantly improving the fuzzer's performance. Ideally testing will be performed comparing the CPU and GPU methods using a benchmark suite with meaningful configuration parameters and sufficient time to gather appropriate data [13]. Different GPU configurations could also be tested.

Fundamentally, computer programs provide value due to their ability to handle many tasks efficiently. However, this value cannot be achieved if the systems are breached and data is stolen. Prevention methods for impairing attacker activities are quite common, but they must continue to improve, to stay ahead of the modern threats. Many different security strategies exist such as automated vulnerability discovery. Both static and dynamic analysis can be used to this end.

Modification and development of new fuzzers must continue to occur to stay ahead of attackers. There are many different ways to improve the overall performance of a fuzzer, but the main approach is to enhance the software itself. Prospective improvements can include increased code coverage, better data generation, and other means of enhancing the fuzzer to produce more crashes and better document the crashes that occur.

Innovating with regards to the hardware platform that the fuzzer runs on is also option for improving the overall performance of fuzzers. Fuzzing data consists of many independent tasks including the generation along with testing the data. These tasks are parallelizable, meaning a GPU could prospectively improve the speed at which they are completed. If the GPU approach is successfully implemented, the overall fuzzer's performance could increase significantly facilitating greater levels of defect and bug detection. Discovering more vulnerabilities leads to corporate savings as well as allowing consumers and businesses to be more confident in the security of their personal and corporate data.

ACKNOWLEDGMENT

This work was supported by the U.S. National Science Foundation (award # 1757659) and the NDSU Institute for Cyber Security Education and Research.

REFERENCES

[1] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis."

[2] L. Bilge and T. Dumitras, *Before We Knew It*. .

[3] S. Karamcheti, G. Mann, and D. Rosenberg, "Improving Grey-Box Fuzzing by Modeling Program Behavior."

[4] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report)."

[5] Y. Wang, P. Jia, L. Liu, and J. Liu, "A systematic review of fuzzing based on machine learning techniques."

[6] M. Bohme, V. T. Pham, and A. Roychoudhury, "Coverage-Based Greybox Fuzzing as Markov Chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.

[7] V. J. M. Manès *et al.*, "The Art, Science, and Engineering of Fuzzing: A Survey," 2019.

[8] K. Sen, *Concolic Testing*. 2007.

[9] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "FUZZIFICATION: Anti-Fuzzing Techniques."

[10] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings - IEEE Symposium on Security and Privacy*, 2010, pp. 497–512.

[11] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward Large-Scale Vulnerability Discovery using Machine Learning," 2016.

[12] E. Niewiadomska-Szynkiewicz, M. Marks, J. Jantura, M. Podbielski, and P. Strzelczyk, "Comparative Study of Massively Parallel Cryptalysis and Cryptography on CPU-GPU Cluster."

[13] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," p. 16, 2018.