Increasing Execution Speed with In-Memory Persistence Mode Fuzzing

Brandon Lara*
blara4@lsu.edu
Louisiana State University
Baton Rouge, Louisiana, USA

ABSTRACT

Fuzzing has remained an important aspect of computer security. Testing Programs is a difficult task. It is impossible for a program to know if it has tested all possible states. Humans must bridge the gap and determine if all possible cases have been tested. In many cases programs are so complicated that humans cannot be sure if all states have been tested. The problem is that the possible state space of a sufficiently complex program is near infinitely large. This reality has ushered in a generation of research covering the topic of automated software testing. This process, named fuzzing, has proven to far out stripe a humans ability to test a programs possible states. In this paper, we aim to improve the performance of fuzzer's by altering their implementation. Google reported that their Open Source Project Fuzzer OSS-Fuzz found 28,000 bugs [goo 2023]. Given the effectiveness of fuzzers as opposed to human auditing, improving fuzzer design and implementation directly reduces bugs and security vulnerabilities in software products. We have found that fuzzer's cause many storage IO requests by inducing paging. This was improved by persistence mode fuzzing, but we have altered the implementation. We have managed to keep the induced paging low, but also doubled the performance of persistence mode fuzzers.

KEYWORDS

Software Testing, Software Verification, Software Security, Cyber Security, Fuzzing

*A simplified list of authors.

Author's address: Brandon Lara, blara4@lsu.edu, Louisiana State University, P.O. Box 1212, Baton Rouge, Louisiana, USA, 70820.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
0730-0301/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Brandon Lara. 2018. Increasing Execution Speed with In-Memory Persistence Mode Fuzzing. *ACM Trans. Graph.* 37, 4, Article 111 (August 2018), 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The expansive and complex nature of software architecture requires automated testing, as human auditing is insufficient to cover millions of lines of code. For this reason, fuzzing has become a staple tool for security professionals. Fuzzing is automated software testing. A fuzzer typically consists of two separate components. The first is the controller. The controller oversees the Program-Under-Test (PUT). The Controller runs the PUT, supplies the PUT inputs, analyzes the results of the test, and handles cases where the PUT crashes during runtime. The second component is an input generation engine. Input generation is a complicated subject for which many methods exist [Wang et al. 2020]. This subject will largely be beyond the scope of this paper and our methods.

The nature of these components will depend on the Program-Under-Test. Programs can accept inputs from various sources. At a low level, this list includes: operating system signals, network traffic, files, stdin [Wang et al. 2020; Zhu et al. 2022]. At a high level, this situation becomes much more complicated. Files could be JSON files or txt files, operating system signal implementation is unique to each operating system, and network traffic could refer to TCP or UDP requests. Fuzzers can even treat random number generation as inputs, even though the calculation is internal to the program. Many research hours have been spent on optimizing fuzzer performance. To this end, fuzzers are typically domain specific [Wang et al. 2020; Zhu et al. 2022]. Fuzzer implementation focuses on one subsection of potential inputs, whether it be HTTP requests, SQL queries, or JSON files that constitute "User Input" for the PUT [Wang et al. 2020; Zhu et al. 2022]. This allows fuzzers to spend their runtime focused on the inputs specific to the PUT.

The programs considered in this work accept arbitrary strings of bytes through stdin. The input generation engine would then generate arbitrary arrays of bytes of random legnths. This simple method of input generation is sufficient

to understand this paper; however, a real fuzzing effort would elect to carefully craft inputs that achieve desirable effects in the execution of the PUT. Further, binary programs can accept inputs from other sources. Binary programs could read from a file or accept input from a socket connection. This would complicate the engineering of the system, but would not significantly change the design of the system. The current fuzzer can easily be extended to accept diverse inputs without changing the system design.

2 FUZZER PERFORMANCE

Fuzzing performance is deep field of research. In order to exhaustively test a program, all arbitrary possible states much be reached. This means exploring every possible (unique) sequence of instructions. This problem is undecidable. It is equivalent to asking "Is it possible to reach a state with 'x' condition?" Due to the Halting Theorem, and it's derivative: Rice's Theorem, it is impossible to state definitively if an arbitrary program will enter a specific state [Kozen 1977].

Since definitively searching the full state space of a program is infeasible, and in some cases impossible, fuzzers try to exhaustively search as much of the state space as possible. Increased fuzzing performance directly translates to more of the state space being tested. Fuzzers can increase performance in several ways.

By selecting smart inputs, the fuzzer can avoid repeat executions. Consider the program below:

```
int x = getUserInput()

if x < 100 {

printf("x<100");

} else {

....

}
```

In the case that x is 20, the if branch executes. This is also true in the case that x is 21, 22, 23, 24, etc. In this shallow case, x = 0...100 all cause the if branch to execute. In order to explore more of the state space, the fuzzer should generate numbers such that x <100 and x>= 100. This allows the fuzzer to explore the if branch and the else branch. Exploring the if branch multiple times with different values for x does not generate a meaningful difference in execution. Programs are rarely this shallow in their use of inputs, but the idea of avoiding repeat execution can be used to great effect [Böhme et al. 2017; Wang et al. 2020; Zhu et al. 2022].

Another way to increase fuzzer performance is to limit the state space that the fuzzer is concerned with. Most program bugs are only located in small, complicated sections of code [Böhme et al. 2017]. Coverage guided fuzzing attempts to generate inputs that will cause most of the program to execute over the course of the multiple tests. Directed fuzzing is an improvement over coverage guided fuzzing where the fuzzer will not try to test all of the code in the program, but rather only some complicated subsection of code. Any input that does not cause the desired subsection of code to be executed will be discarded in favor of inputs that do cause the desired subsection of code to be executed.

Directed fuzzing can be further extended to constraint based fuzzing. Constraint variant fuzzers do not just require that a certain block of code was executed. They also require that certain state conditions are met. This could include socket creation, a memory write command, opening a file, or a malloc happening before a free. The fuzzer throws away any input that causes an execution path that does not meet the preset requirements. This has been used to great effect when searching binary programs for memory corruption errors [Lee et al. 2021].

In this work, we are concerned with optimizing the implementation of the fuzzer. The above optimizations work by changing the behavior of the fuzzer to waste less time on undesirable executions. We aim to improve the general implementation in a way that could be applied to any of the above fuzzing models.

2.1 The Fork/Exec Problem

Fuzzer implementation can vary wildly based on the fuzzer. For fuzzers concerned with path exploration, both coverage guided and directed, the typical implementation involves a compiler and a debugger [AFLplusplus [n. d.]; Google [n. d.]]. The fuzzing target is compiled with a program that puts a breakpoint at each branch of execution. The debugger catches this breakpoint and records the pathing. This allows the debugger to get data about the execution of the program. This is how a fuzzer can determine if target portions of code are executed.

Once the Program-Under-Test has been compiled with breakpoints, the fuzzer must run the program and supply it inputs. Fuzzer's like AFL++ and Honggfuzz would fork a child process and exec the PUT [AFLplusplus [n. d.]; Google [n. d.]]. This fork/exec implementation would cause a large amount of disk IO. Every time fork/exec was executed the storage system would be requested for the entire contents of the PUT file. This phenomenon is demonstrated in the evaluation section. This IO significantly slows down the operation of the fuzzer. Increased fuzzing performance leads to more testing, and thus more secure software. This IO is a significant bottleneck in early fuzzers. Modern fuzzers, like

AFL++ and HonggFuzz, have added an additional "persistence mode" as a means to cut down on the amount of IO performed by the fuzzer [AFLplusplus [n. d.]; Google [n. d.]]. Modern fuzzers like AFL++ and Honggfuzz, allow users to use fork/exec mode or presistence mode [AFLplusplus [n. d.]; Google [n. d.]].

Under persistence mode, fuzzers will only fork the PUT once. The active fuzzer will use debug tools (PTRACE on Linux) to stop the PUT right before a call to exit. The PUT's registers and virtual memory (it's entire state) is reset to the same values as when the program started main() [AFLplusplus [n. d.]; Google [n. d.]]. This allows the program to run again without needing to fork a new process. This was shown to have a 20x performance increase in the AFL++ project [AFLplusplus [n. d.]].

In this work, we present a modified persistence mode that keeps IO requests low compared to fork/exec fuzzing, but runs twice as fast as current persistence mode implementations. Our work reduces the amount of linux system calls required to administer the fuzzer, and simplifies the process of resetting the programs state in between runs.

3 SYSTEM DESIGN

The fork/exec problem is a problem because forking and exec'ing causes an out sized amount of IO to storage. This causes the fuzzer performance to be bottle necked by the speed of requests to the disk. Forking and exec'ing also causes a high volume of requests to the operating system. Each fork and exec call cause a trap to the operating system. The operating system then has to perform lengthy internal computations to service the requests. Exec requires that the operating system page the contents of the PUT into memory to build the virtual memory of the PUT. This generates a significant latency for both waiting for the storage to respond page in the contents of the PUT but also for the Operating System internal management.

To solve this problem, persistence mode was introduced in fuzzers like AFL++ and honggFuzz [AFLplusplus [n. d.]; Google [n. d.]]. This persistence mode only calls fork and exec once, and just restarts the program by resetting the registers, including the instruction pointer, back to their starting values. It is, however, not enough to reset the registers. If anything about the virtual memory changed during the runtime of the program, then those changes must be reverted as well. This can be costly as it involved writing to all the changed values which introduces more paging for sufficiently large Programs-Under-Test. Further, persistence mode fuzzing works by compiling breakpoints into the PUT. The fuzzer then acts as a debugger and catches the breakpoints thrown by the PUT. This requires significant back and forth between the Operating System and the CPU Hardware. These interrupts cause the program to give control to either the CPU or the Operating system depending on the implementation. This transfer process causes significant latency.

We introduce In-Memory Persistence Mode Fuzzing. Instead of forking a child process and exec'ing the PUT, In-Memory Persistence Mode Fuzzing works by pulling the assembly instructions of the PUT into the virtual memory of the fuzzer. The assembly instructions of the PUT are pulled into a function inside the virtual memory of the fuzzer, and the fuzzer is re-linked to accommodate this new function and the new code from the PUT. This is akin to creating a user space virtual machine to execute the PUT. This involves writing the PUT's assembly instructions and .data sections to the virtual memory of the fuzzer. After this, the fuzzer can continuously rerun the PUT's assembly instructions without requesting the assembly instructions be paged into memory again. This reduces the amount of IO the fuzzer does compared to fork/exec fuzzing and is comparable to current persistence mode fuzzing implementations. In the evaluations section, we detail a test showing the reduced storage IO requests. Instead of breakpoints, the PUT code is recompiled to have periodic function calls that transfer control back to the fuzzer. This gives the fuzzer opportunities to perform analysis and prepare inputs. This is a standard operation for all modern fuzzers, but it is usually accomplished by breakpoints. Having both the fuzzer and the PUT code inside the same virtual address space means that breakpoints are not required to trade execution back and forth between the fuzzer and the PUT. Function calls are sufficient to pass control back and forth between the fuzzer and the PUT.

We have glossed over several implementation details that need to be ironed out in order for this system to be realizable. To start, the fuzzer must be sufficiently re-linked to accommodate the new assembly code from the PUT. When the PUT was originally compiled, the compiler's linking process ensured that all functions calls and function locations match. Copying this pre-linked code into the virtual memory space of the fuzzer does not guarantee that the function calls and functions locations match. The fuzzer will need to act as a linker by modifying the assembly instructions. This would allow the fuzzer to avoid trapping to the operating system by calling breakpoints. In order for the fuzzer to rerun the program multiple times, we must be able to guarantee that the PUT state will be the same each time the PUT is restarted. In order to accomplish this, we maintain a list of address that have been changed and restore their original values at the end of each execution of the PUT. We allow the PUT to malloc freely without requiring splitting the heap arenas between the PUT and the fuzzer. The fuzzer frees each piece of malloc'ed memory before passing control back to

the PUT. We speculate on potential copy-on-write semantics that could further reduce the storage system IO requests induced by the fuzzer, but recognize the immense software engineering challenges associated with realizing such an idea. We hope that future work will tackle the difficult problem of reducing the IO of the state resetting problem

3.1 Re-compiling

In order for a fuzzer to operate, the fuzzer must take back control at key points in the execution. The PUT must yield control back to the fuzzer right before input is expected, so that the fuzzer can prepare input. The PUT must also yield control back to the fuzzer when branching paths are selected. This is crucial because it allows the fuzzer to determine the path that the execution is talking. This information is required for the fuzzer to perform analysis regarding if the execution was desirable, reached the target sections of code, or met the criteria the practitioner set. In a typical fuzzer, control is usually given back to the fuzzer via a breakpoint. Since the fuzzer is acting as a debugger, the breakpoint yields control back to the fuzzer.

In In-Memory Persistence Mode fuzzing, the fuzzer is not acting as a debugger. Both the PUT and the fuzzer exist in the same virtual memory space. Breakpoints will not yield control back to the fuzzer. These breakpoints are placed at branching locations. In high level terms, this means that the breakpoints are placed at function calls, if statements, and loops. In x86 assembly, this means before each cmp, jmp, and call instruction. Traditional fuzzers will compile these breakpoints into the PUT's binary [AFLplusplus [n. d.]; Google [n. d.]]. Our fuzzer will do the same. We propose using a modified version of the AFL++ compiler that replaced the breakpoints with function calls to the fuzzer's entry point [AFLplusplus [n. d.]]. This allows the PUT to yield control back to the fuzzer.

The PUT code was not meant to be a function sharing virtual memory with the fuzzer. When the PUT is recompiled, a context switch must be compiled into the code as well. Before the PUT code is called, all registers must be saved, and after the PUT finishes all registers must be restored before control is returned to the fuzzer. This allows the PUT and fuzzer to have their own register values. The entry point to the PUT (this is main()), will have a pusha command, and all jmps to the fuzzer will have a popa command. These x86 instructions push and pop all of the registers. In the case of 64 bit environments, pusha and popa do not exist. Instead the fuzzer will add an individual push command for each register and an individual pop command for each register.

In a traditional fuzzer, right before the PUT triggers a breakpoint and transfers control back to the fuzzer, the PUT will store a unique identifier into the RAX register. This allows the fuzzer to read the RAX register and know which breakpoint was triggered. This allows the fuzzer to track the execution path of the PUT. Since the PUT and the fuzzer share the same registers, this is not viable. If the PUT stored a value in RAX, then the fuzzer's own RAX value would be mangled. To get around this issue, we assign a neutral memory location where the PUT will write the unique identifier. The fuzzer can then read the memory location to receive the unique identifier.

3.2 Re-Linking

During the compiling process, a linker must create symbolic links between external function calls and the external functions themselves. When a program calls library code, functions like printf() defined in the stdio.h file for C/C++ programs, the linker must find the code for this external function and copy it into the compiled object file or the runtime memory of the program, depending on implementation. Once this code is placed into memory, all references to this code must be updated to point to the relevant location. This situation is much more complicated under PIE (Position Independent Executable) and ASLR (Address Space Layout Randomization). As the independent sections of the executable are loaded into virtual memory, it is common for Operating Systems to randomize the location that the sections are loaded into. This is a security tactic. This means that the the addresses of the function calls must be updated again. In addition to function calls, which have been the motivating example thus far, any reference to an address must be changed to match the new linked/loaded address. This process ensures that assembly code can run correctly by calling valid functions, reading valid data, and writing to valid locations. In order for the PUT code that has been copied into the fuzzer's virtual memory to run correctly, this process must be performed.

We propose a special re-linking subroutine. This routine would scan the code before it is copied into memory and change all address values to appropriate values. Readers familiar with linking and loading will be curious why we choose to re-implement a linker when linkers are readily available. We believe that several benefits come from the effort of re implementing a linking routine. First, our work is primarily interested in reducing storage IO. Implementing the Re-Linking process allows us to compact the linked code. Second, by implementing the linker, we can forgo the ASLR and PIE accommodations. Our fuzzer will get to decide where code gets loaded, and testing the PUT is not a security risk, meaning that ASLR provides no benefit. Additionally, we want to allow practitioners to fuzz targets with varied Standard Library versions. This allows the fuzzer to

test the program under multiple Standard Library versions seamlessly; because, the fuzzer can select which Library to pull code from.

Our system first works by calculating the size of the PUT code + PUT data + Library Functions. The PUT code and PUT data sizes are easy to obtain. The .text and various .data sections are located at the header of the a.out file. The size of the library functions can be calculated by scanning the Standard Library file for the function in question and reading the length. The fuzzer will allocate virtual memory space that is the size of the PUT code + PUT data + Library Functions. Then the code, data, and library functions will be copied into that space. This causes the allocated section to be packed as tightly as possible.

Once the code and data has been pulled into the virtual memory, it must be re-linked. The fuzzer will scan the PUT code and modify any memory access address to the new address of the data or function. This typically means changing addresses associated with jmp's, call's, or mov's that access memory. The fuzzer will maintain a dictionary where the index is the memory address originally compiled into the PUT, and the value is the new address after copying the PUT code, data, and library functions into virtual memory. The fuzzer can reference this dictionary to finish the re-linking process..

This creates a compact space that maintains all of the data required to run the PUT code.

3.3 Preventing Blocking

A traditional fuzzer would normally act as a debugger. When the fuzzer forks a child process, it has access to that child's stdin and stdout buffers. The fuzzer can supply inputs to the program via stdin. When the PUT prepares to accept input, the program will block and go into a waiting state. The PUT will no longer be scheduled by the operating system scheduler until the blocking state ends. This means that the PUT will block until an appropriate input is supplied via stdin. Normally, the fuzzer would end the blocking state by supplying the input to the PUT. In In-Memory Persistence Mode fuzzing, the PUT is inside the same process as the fuzzer. If the PUT blocks then the fuzzer will also be blocked. Neither will be scheduled for execution because the fuzzer and the PUT are the same process.

In order to prevent blocking, we use the C function freopen(). Freopen() directs a file to stdin. Once the PUT triggers a blocking command to read from stdin, the Linux Kernel will redirect the command to the file provided in the freopen call. This prevents the fuzzer from blocking indefinitely when requesting input.

Other forms of inputs require other solutions. Networking traffic input would need to be redirected in a similar manner to prevent blocking and waiting for network latency, as a motivating example. Our work only focuses on programs that accept input via stdin. We believe similar low latency options exist to prevent blocking for other forms of IO, but we leave this as future work.

3.4 Resetting State and Reducing Storage IO

One of the major problems with traditional Persistence Mode fuzzing is that any alteration to the state of the the program must be undone. All heap objects must be unmade, and any write to memory must be changed back to its original value. Registers must also be intentionally set to the correct values. Once this has been done then the PUT can be run again without any side effects of the previous run affecting further executions. This process can involve a significant amount of IO as writing all of the virtual memory back to it's original values could potentially induce paging for every write. Despite this, Persistence Mode fuzzing has been a great improvement over fork/exec fuzzing, pulling a 20x performance increase over fork/exec fuzzing. Persistence mode fuzzing still experience slow down from this state resetting.

Our fuzzer must also reset the state of the PUT. This process is very similar to the case of traditional Persistence Mode fuzzing. In-Memory Persistence Fuzzing resets the state of the program using 3 tactics. First, the fuzzer must reset the values on the stack. Second, the fuzzer must free all of the heap values. Third, the fuzzer must reset any modifications made the .text or .data sections of the PUT.

Resetting the stack of the program is trivial. Moving the stack pointer back to to all the stack base3 pointer above all of the saved stack value will cause them to be overwritten on the next execution. It is important that we are not overwriting the stack of the fuzzer. To ensure that the stacks do not overwrite each other, we malloc a large portion of memory on the heap and point the PUT stack base pointer to the bottom of the malloc'ed memory location. It is important to remember that all of the registers are saved by pushing and popping all of the registers before and after returning control to and from the fuzzer. This means that the PUT can have it's own stack by maintaining a different stack pointer and stack base pointer value.

Our fuzzer must free all of the values on the heap. Each time to fuzzer takes control, it reads the address of malloc'ed data. It can do this by taking control after each malloc, calloc, or alloc call. This means that after any heap function call, the PUT must save all of the registers and return control back to the fuzzer. This allows the fuzzer to read and record the address of the malloc'ed call. In between runs, the fuzzer will free each of the recorded addresses. This allows the fuzzer to reset the status of the heap in between runs.

Finally, the fuzzer must reset any changes to values inside the .text and .data sections. Changing these values is rare, and typically occurs when global values are stored and changed. During the compiling phase, all writes to values inside the .text and .data section will have a call to the fuzzer placed above the write. The fuzzer will maintain a list of all memory addresses that have been written to and their original values. In between runs of the PUT, the fuzzer will rewrite the original value to the memory address.

This incurs potential paging for each address that must be rewritten. Our work does not improve the situation over traditional persistence mode fuzzing. We believe that it is possible to further reduce the amount of paging by utilizing the fact that both the PUT and the fuzzer are in the same virtual memory space.

First, we speculate that it would be possible to utilize copy on write semantics to copy the data before it is written, and redirect the write to the copy. This would successfully reduce the amount of paging required when resetting the state of the PUT. All of the copy-on-write pages created could be discarded. This is difficult to orchestrate. This would require writing a kernel level driver that would monitor the writes to memory and induce a copy on write of the PUT memory. This is possible, as the Linux kernel readily supports copy on write semantics. We leave this possibility open as a further researcher into the subject and the method. We believe the potential benefits are worth the large development effort of changing the Linux functionality.

Second, we speculate that it would be possible to utilize delta encoding to compress the changes to the state to a small area. This would reduce the size of writes to virtual memory, and receives the same benefits of copy on write semantics as above. The drawback is that reading data becomes more computationally intensive as the delta encoding must be resolved to generate the current values. We believe this could be a viable route to reduce the amount of storage IO in In-Memory Persistence Mode Fuzzing, but it is not clear how much overhead is generated by utilizing a delta encoding scheme for writes. We believe that the benefit of delta encoding out weights the computational overhead. Further testing is required. We leave this as further work for future research.

4 EXPERIMENTAL SETUP

During the course of our work, we were unable fully implement the systems described above. To motivate the software engineering effort required to actually realize the fuzzer described in the paper we decided to develop a toy example that could show the performance benefits of running the fuzzer and the PUT code inside the same virtual memory process. Naturally, the example provided does not capture all of the functionality described, and thus cannot be used as an actual fuzzer. We will discuss these limitations and how we believe the toy example still accurately captures the benefits of our design. First, we outline what we believe a minimum toy example should achieve to show the performance benefits associated with In-Memory Persistence Mode Fuzzing. Second, we detail the our experimental setup, toy examples, and how they meet the requirements listed. Third, we discuss the limitations of our setup and layout a road map to further testing.

Our experiments were run on the most recent Kali Linux VM, on a Intel 12th gen i7 CPU machine running the Windows 10 operating system. The Kali VM was run on a Samsung SSD with 2 TB of storage and no internal DRAM cache. We will be testing the amount of paging occurring during the run-time of the fuzzer, and we will also be measuring the time it takes for the fuzzer to complete it's fuzzing task.

4.1 Ideal Toy Example

We now outline what we believe a toy example of In-Memory Persistence Mode Fuzzing should capture and include to be motivating for the intense software engineering effort required to realize the real fuzzer.

A toy example should meet two main requirements. The toy example should run the PUT code in memory. It is not required that the code be pulled into memory from another file, or that the re-linking or recompiling process take place. We use a compiler to directly compile the hello world code into the fuzzers binary. This is very simple to accomplish. We treat main as the fuzzer and create a hello world function, containing the printf command. This compiles the code and correctly links the hello world code into the binary. Testing arbitrary programs would require the complicated re-linking and recompiling process. We do not believe this adds meaningful overhead because re-linking and recompiling only happen once during the fuzzing process.

A toy example should also reset the changed state of the PUT. This is a very difficult engineering task. We did not implement resetting the state. We only tested the hello world example, which does not include mutable state. The state resetting algorithm adds additional overhead, but our design is identical to traditional Persistence Mode fuzzers. We believe the overhead would be similar, or less than that of the state resetting associated with persistence mode fuzzers. The In-Memory Persistence Mode Fuzzer can directly write to memory addresses while the traditional Persistence Mode Fuzzer must send write signals through the operating system using the PTRACE function. We are confident that this process is slower than the In-Memory variant we have presented. We leave the implementation, testing, and verification of this hypothesis to further research and future researchers.

4.2 Our Minimum Toy Example

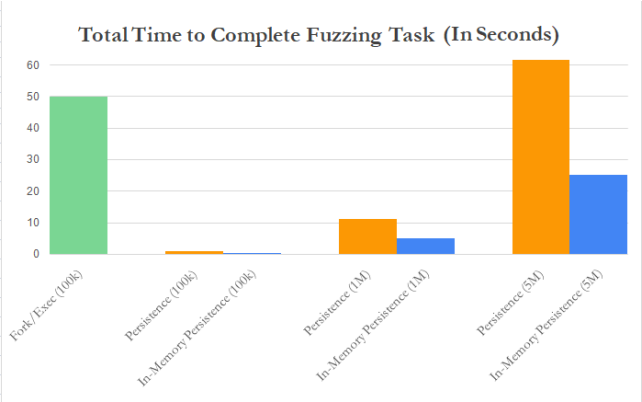

Figure 1: Time to complete the fuzzing task for each kind of fuzzer. Includes tests for 100K, 1 million and 5 million iterations.

Our toy example includes a recreation of a fork/exec fuzzer, a traditional persistence mode fuzzer, and a model of In-Memory Persistence Mode Fuzzing. Each fuzzer runs a hello world C program. The program consists of only the command:

```
printf("Hello World");
```

The fork/exec fuzzer works by forking a child process and exec'ing the hello world program. The traditional persistence mode fuzzer works by forking a child process and exec'ing the hello world program. The child process makes a call to PTRACE with the PTRACE ME flag set. This allows the process to be debugged by the parent process (the fuzzer). The fuzzer inserts breakpoints at the beginning and end of the program. When the second breakpoint triggers, the fuzzer resets all of the registers values (including the instruction pointer) with a call to PTRACE with the SET REGS flag set. This allows the program to be restarted. The hello world program being fuzzed does not have any mutable state, and therefore the state does not need to be reset in between runs, only the registers.

The In-Memory Persistence Mode Fuzzer treats the fuzzer as main() and calls a function containing the hello world program code. We compile this source file using gcc. It is not required to pull the hello world program into memory and perform the re-compiling and re-linking process. Re-compiling and re-linking only happens once, making it's overhead negligible. Directly compiling the hello world code into the binary as a part of the original source file simulates this process without requiring the difficult engineering to realize the process.

Each fuzzer runs the hello world code in a loop. Each will be tested at 100,000 iterations and traditional persistence mode and In-Memory Persistence Mode will be tested with 1 million and 5 million iterations.

Table 1: Paging Operations with 100K iterations

Fork/Exec	42272
Persistence Mode	188
In-Memory Persistence	9244

Table 2: Paging Operations with 1 Million iterations

Persistence Mode	239016
In-Memory Persistence	270980

Table 3: Paging Operations with 5 Million iterations

Persistence Mode	1464656
In-Memory Persistence	1548396

5 EVALUATION

We cover the evaluation of our results in two sections. First, we show that our method does increase fuzzing performance. Second, we speculate on why the data about how many paging operations occur during fuzzing does not match our hypothesis.

5.1 Fuzzing Speed

Based on our experimentation, In-Memory Persistence Mode resulted in doubling the execution speed of the fuzzing task over regular Persistence Mode. Figure 1 shows the results of our speed tests. fork/exec fuzzing performed much worse than both versions of persistence mode. For this reason, we did not continue to test the fork/exec fuzzer in subsequent tests. With both 1 million and 5 million, the In-Memory Persistence Mode completed the fuzzing task in half the time of traditional Persistence Mode. We believe this can be explained by two factors. First, regular persistence mode must perform an OS level context switch between the fuzzer and the PUT. Second, traditional persistence mode fuzzing must trap via a breakpoint to the Operating System. Both of these operations require requests to the Operating System which is known to be a bottleneck for program activity.

5.2 Induced Paging

Additionally to measuring execution speed, we measured the amount of paging that occurred while the fuzzer was running using the Linux tool vmstat. We found that the In-Memory Persistence Mode fuzzer had comparable paging to the traditional Persistence Mode fuzzer. As stated in the Experimental Setup Section, our tests do not include the

cost of resetting the state of the program after each run. Table 1 shows our results for the first test, running the PUT with 100K iterations. Fork/exec fuzzing causes far more IO than both Persistence Mode implementations. This confirms the claims that fork/exec fuzzing is bottle necked by the storage IO induced by the implementation. Table 2 shows our results for the second test, running the PUT with 1 million iterations. Traditional Persistence Mode induced less paging during our testing. This violates our assumptions. We believed that In-Memory Persistence Mode would have a comparable amount of paging to traditional persistence mode. This result is shown to be consistent in the Table 3. Table 3 shows our results for the third test, running the PUT with 5 million iterations.

This disparity in paging could be a by-product of debug printing of the In-Memory Persistence Mode Fuzzer. It is difficult to track down what actions specifically are inducing fuzzing. We believe the next step is to use kernel level tools to track which actions are inducing paging. The vmstat tool only records the amount of paging, but does not record what process, or functionality is responsible for the paging.

We discuss further testing and implementations in the Future Work Section.

6 FUTURE WORK

We have left many aspects of the fuzzer as future work. The core of our work outlines a system that we believe would improve fuzzing performance. We generate a toy example that shows a 2x performance improvement. We believe this toy example motivates continuing the work on fully implementing the fuzzer. In order to finish implementing the fuzzer, the re-compiling process and re-linking process need to be implemented, and the state resetting algorithm must be implemented. We would purpose starting with the re-linking and re-compiling process, as these are likely the hardest part of implementing the current design.

Additionally, it is not clear why the In-Memory Persistence Mode Fuzzer generated more paging requests than the traditional persistence mode fuzzer in Tables 1, 2, and 3. We would purpose that future research aim to utilize kernel level functionality to try and find the specific operations that are inducing paging. We believe that the amount of paging should be comparable. The results we showed in this work do not reflect that hypothesis.

Finally, In-Memory Persistence Mode Fuzzing offers several avenues for further reducing the storage IO requests. These avenues were discussed in Section 3. We plan to further investigate algorithms that could further decrease the storage bottleneck fuzzing faces.

7 CONCLUSION

In this work we presented a new implementation of persistence mode that drastically increases performance of fuzzers. This implementation keeps the storage IO low, but gains other performance benefits by keeping the number of system calls and break points lower than traditional persistence mode fuzzing. We suspected that our method would also reduce storage IO requests, or at least induce a similar amount of paging as traditional persistence mode. Our testing showed this hypothesis to not be true. Even though we greatly improved performance, we believe that the performance could be even further improved in future research by finding the cause of the extra paging. We leave this as future research.

REFERENCES

2023. Taking the next step: Oss-Fuzz in 2023. https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html

AFLplusplus. [n. d.]. AFLplusplus. https://github.com/AFLplusplus/AFLplusplus

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox fuzzing. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017). https://doi.org/10.1145/3133956.3134020

Google. [n. d.]. Google/AFL: American fuzzy lop - a security-oriented fuzzer. https://github.com/google/AFL

Dexter C. Kozen. 1977. Rice's theorem. *Automata and Computability* (1977), 245–248. https://doi.org/10.1007/978-3-642-85706-5_42

Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3559–3576. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu

Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. 2020. The progress, challenges, and perspectives of directed greybox fuzzing. *arXiv preprint arXiv:2005.11907* (2020).

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A survey for roadmap. *Comput. Surveys* 54, 11s (2022), 1–36. https://doi.org/10.1145/3512345