Forkfuzz: Leveraging the Fork-Awareness in Coverage-Guided Fuzzing

Marcello Maugeri[1] [0000−0002−6585−549], Cristian Daniele[2] [0000−0001−7435−4176], and Giampaolo Bella[1] [0000−0002−7615−8643]

[1] University of Catania, Catania, Italy
`marcello.maugeri@phd.unict.it, giamp@dmi.unict.it`
[2] Radboud University, Nijmegen, Netherlands `cristian.daniele@ru.nl`

Abstract. Fuzzing is a widely adopted technique for automated vulnerability testing due to its effectiveness and applicability throughout the Software Development Life Cycle. Nevertheless, applying fuzzing "out of the box" to any system can prove to be a challenging endeavour. Consequently, the demand for target-specific solutions necessitates a substantial amount of manual intervention, which diverges from the automated nature typically associated with fuzzing. For example, prior research identified the lack of a solution for testing multi-process systems effectively. The problem is that coverage-guided fuzzers do not consider the possibility of having a system with more than one process. In this paper, we present *Forkfuzz*, a "fork-aware" fuzzer able to deal with multi-process systems. To the best of our knowledge, *Forkfuzz* is the first *fork-aware* fuzzer. It is built on top of Honggfuzz, one of the most popular and effective coverage-guided fuzzers, as reported by the *Fuzzbench* benchmark. To show its effectiveness, we tested our fuzzer over two classical programming problems: the *Dining Philosophers Problem* and a version of the *Producer-Consumer Problem* where the consumer (the child) process crashes for specific inputs. Furthermore, we evaluated *Forkfuzz* against a real and more complex scenario involving an HTTP server that handles multiple connections through multiple processes. The results of our evaluation demonstrate the effectiveness of *Forkfuzz* in identifying crashes and timeouts. Finally, we discuss possible improvements and challenges for the development and application of *fork-aware* fuzzing techniques.

Keywords: fuzzing · automated vulnerability testing · multi-process system · security testing · concurrent programming

1 Introduction

Fuzzing is currently considered a standard technique, within the larger security testing process, to achieve software correctness. The concept is quite straightforward: it involves the repetitive execution of a *SUT (System-Under-Test)* with various inputs, including malformed ones, in order to uncover bugs [14]. In particular, fuzzing excels at identifying critical bugs such as crashes and timeouts, which play a pivotal role in vulnerability analysis. These observable anomalies serve as indicators of potential weaknesses in software. Given that bugs are an intrinsic aspect of program code, the significance of fuzzing in the software development landscape is undeniable. Its substantial potential lies in its ability to enhance software reliability and security for the future [24]. However, various challenges lie at the horizon of fuzzing. Notably, tailoring fuzzing to multi-process systems is still daunting at present, and our research stands on the observation that modern coverage-guided fuzzers, albeit generally powerful, may not succeed in capturing misbehaviour in child processes. This limitation is known as the "fork-awareness" problem [15], where the fuzzer may not properly account for the behaviour of child processes hence for the potentially undiscovered bugs in the overall system.

In this study, we present *Forkfuzz*, a new fuzzer to tackle the problem outlined above. Based on *Honggfuzz*[3], it seems fair to argue that *Forkfuzz* is the first *fork-aware* coverage-guided fuzzer. *Forkfuzz* leverages the *ptrace* system call[4] to monitor child processes and maintains a set of process identifiers to keep track of them. *Forkfuzz* is first demonstrated on two classical concurrency problems: the *Dining Philosophers Problem (DPP)* and the *Producer-Consumer Problem (PCP)*. As a practical test case, we then execute *Forkfuzz* on a distributed, open-source project delivering a web server that handles multiple connections through the use of multiple processes.

Overall, our experiments demonstrate the ability of our fuzzer to effectively identify vulnerabilities in complex, multi-process systems. It is worth noting that these experiments are reproducible as both the code of our fuzzer and the experiment code are available as open source [5]. Intentionally, all of our case studies feature bugs of varying nature, so that our tool can be widely evaluated, ultimately offering a solid baseline for further developments and applications.

The article is structured as follows. Section 2 presents the background, Section 3 describes notable and related works in the state-of-the-art. Section 4 provides an overview of the overall scenario and explains the *fork-awareness* property at the foundation of this study. Section 5 presents the first *fork-aware* fuzzer, *Forkfuzz*, evaluated on three different case studies described in Section 6. Finally, Section 7 discusses limitations and future directions while the last section summarises the key findings.

2 Background

In Computer Science, a *process* is an instance of a computer program executed by the Central Processing Unit (CPU) [25]. Essentially, every running program is a process. Each process is associated with the address space, a list of memory locations that contain the executable program, its stack and data. In addition, every process has registers, open files, and signals. Furthermore, each process

[3] https://github.com/google/honggfuzz
[4] https://man7.org/linux/man-pages/man2/ptrace.2.html
[5] https://github.com/marcellomaugeri/forkfuzz

has its unique identifier, called Process ID or *pid*, which helps to distinguish between different processes.

Sometimes, multiple processes may need to work together to achieve a common goal. Henceforth, this work will refer to such a scenario as a *system*. For example, a web server may spawn multiple processes to handle incoming client requests [9]. Each process would be responsible for serving a subset of the requests, and they would communicate with each other to ensure that all requests are handled efficiently.

In a similar scenario, the web server must be capable of spawning processes as needed. In UNIX systems, new processes can be created by invoking the *fork* system call. This system call creates a new process, called the child process, which is an exact copy of the process that made the call, also known as the parent process. The primary distinction between the two is their *pid*. After the fork, both processes go their separate ways and accomplish their respective tasks, possibly fork again, e.g. when the server receives a new request.

It should be noted that parent and child are still associated with each other and form a *process hierarchy*. In UNIX, a process and its descendants form a *process group*. The primary use of a process group is to facilitate the management of multiple processes simultaneously. For example, a process could simultaneously send signals to all processes within its group. This applies when all processes in a web server are notified when the configuration file is updated. Another example regards a process within the group that encounters an error. In such cases, the affected process could send a signal to other processes in the group to notify them of the issue. Then, the other processes can take appropriate measures to mitigate the problem.

When mitigation is not accomplished correctly, or worse, is not even considered, the bug causes the program to behave unexpectedly or incorrectly. Consequently, the bug can result in a vulnerability that a malicious attacker can exploit to gain unauthorised access, steal data or disrupt the system. Therefore, it is essential to identify and mitigate bugs using secure coding practices, debugging and testing. In particular, debugging involves identifying and fixing errors and anomalies in the system. On the other hand, testing is validating the functional requirements, performance, and security of a system.

One way to debug a process is to use the *ptrace* system call. *ptrace* allows a process to trace the execution of another process. With *ptrace*, a process called "tracer" can inspect and modify the memory, registers, and system calls of the "tracee" process. This feature is handy for debugging since it allows developers to monitor the behaviour of a process and identify issues. As a result, it can be used for building up sophisticated tools for analysing and testing software, such as debuggers, dynamic analysers, and fuzzers. Additionally, *ptrace* can inject code into a process or modify its behaviour, making it a powerful tool for vulnerability research and exploitation. For instance, *ptrace* is used in *Honggfuzz* as the interface to monitor processes during the fuzzing campaign under *Linux* and *NetBSD*.

Honggfuzz is a popular fuzzer developed by *Robert Święcki* that uses coverage-guided fuzzing to perform automated software testing. In common with many coverage-guided fuzzers, it repeatedly executes a *SUT* with various inputs, known as test cases, to trigger unexpected or erroneous behaviour. Specifically, test cases are generated by applying random mutations on valid inputs provided by the tester, known as seeds. While the program runs, *Honggfuzz* monitors code coverage using instrumentation and generates new test cases based on the feedback it receives. This process continues until a bug is found or a predetermined number of iterations is reached [12].

3 Related work

The field of fuzzing has seen a lot of developments in recent years, with a wide range of fuzzing tools available for various purposes. One major approach to fuzzing is *coverage-guided fuzzing*, which leverages the code coverage reached during the execution to steer the generation of new inputs to explore deeper areas of the code.

Among the coverage-guided fuzzers, *American Fuzzy Lop (AFL)*[6] and its successor *AFL++* [7] have emerged as an effective fuzzer for finding vulnerabilities and have been the basis for the development of other fuzzers [8,13,21,17]. Unfortunately, the mechanism that *AFL++* uses to deal with the *SUT* makes it difficult to handle multiple processes. Making *AFL++ fork-aware* would mean modifying the core of the fuzzer allowing it to capture the process creation and termination. In particular, *AFL++* uses control pipes to communicate with the parent process of the *SUT* ignoring the existence of other processes.

On the contrary, *Honggfuzz* [23] uses the *ptrace* system call to monitor all the processes, making it manageable to integrate mechanisms to monitor timeouts and crashes also in the child processes. The *ptrace* system call has inspired the creation of several debugging tools [11] such as *gdb* and *ltrace*, as well as more complex dynamic analysis tools such as *DroidTrace* [27], which leverage its capabilities to monitor and control the execution of an Android app.

Another example is *strace*, which is used to monitor system calls made by a process and can also be used to monitor child processes. Actually, it has inspired the development of *MoonShine* [20] framework which leverages *strace* to collect execution traces of an application and then applies a trace distillation algorithm to identify the most promising seeds for a fuzzing campaign. The most promising seeds are then fed to *Syzkaller*[7], a state-of-the-art evolutionary fuzzer for the Linux kernel, to conduct a thorough fuzzing campaign.

Intercepting library calls is an approach to modify application behaviour, enabling runtime instrumentation and monitoring. Dynamic linking enables intercepting functions in shared libraries before application calls. This technique is employed by various tools, including the *Preeny* project, offering dynamically linkable libraries to modify the *SUT* behaviour. For instance, the *defork* module

[6] https://github.com/google/AFL
[7] https://github.com/google/syzkaller

intercepts the *fork* calls, making them ineffective. While the use of the *defork* module can disable the functionality of *fork()* and prevent the creation of child processes, it can also disrupt the parallelism of processes in the system, leading to a linear flow of execution. This may not be desirable in scenarios where concurrency is required for the system to function efficiently and correctly.

To address specific requirements as the aforementioned, testers often resort to adopting a specialised component known as a *harness*. The *harness* serves as a custom-coded solution that is meticulously crafted according to the specific *SUT*. Its primary purpose is to preserve the intended functionalities and validate for potential errors. However, it is important to note that the development of a *harness* entails considerable manual effort. This includes not only the creation of the *harness* itself but also the design process that requires a comprehensive understanding of the inner workings of the *SUT*. This stands in contrast to the ongoing research in the field of fuzzing, which aims to enhance the automation and user-friendliness of fuzzing techniques across various scenarios [2].

In particular, when dealing with concurrent software, preserving the ability to run multiple processes or threads in parallel is crucial to maintain the functionalities of a system. To address this challenge, researchers have developed fuzzers specifically tailored for testing concurrent systems. Two notable examples are *ConFuzz* [19] and *CONZZER* [10]. *ConFuzz* uses assertions in concurrent *OCaml* programs to detect new program schedules and paths, leading to assertion failures. In contrast, *CONZZER* provides a more general solution by exploring thread interleavings and detecting hard-to-find data races. Furthermore, *Muzz* [3] represents another notable contribution to the field of *thread-awareness*. This fuzzer takes into account various thread interleavings resulting from the scheduler, enabling the detection of concurrency vulnerabilities and bugs. However, these fuzzers are suitable to detect concurrency bugs in multi-threaded systems only and are not designed for detecting bugs in multi-process systems.

4 Motivational Scenario

The study introduces a scenario that involves a software system S. At the outset of S, its first process P_0 begins its execution. At a particular stage, it could initiate the *fork* system call. As a result, the call creates an identical copy of P_0, named P_1. Following this, both P_0 and P_1 run independently. After, both P_0 or P_1 could possibly *fork* again, resulting in a new process P_2, P_3 or, more generally, P_i.

To exemplify this scenario, Figure 1 illustrates a web server as an example. In this setup, the primary server process, denoted as P_0, plays the role of receiving incoming requests. When a client sends a request, P_0 invokes the fork system call to create a new worker process, represented as P_i. P_i is then responsible for handling the request from the client. When fuzzing such a sophisticated system, the implications are various.

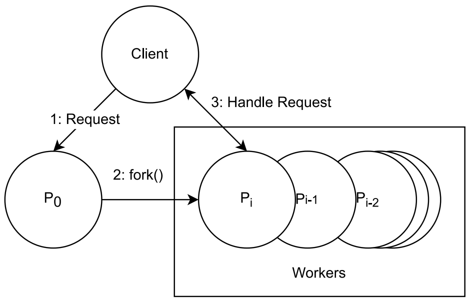

Fig. 1. Example of a forking server

First, P_i is prone to encountering bugs while handling the request. For example, suppose the server expects a request with a specific format or content, but a malicious client sends a request with a completely different format or with invalid data. If the server is not designed to handle the request and does not have proper input validation and error-handling mechanisms, it may crash or behave unpredictably. Consequently, to effectively fuzz-test a system, it is crucial to use a fuzzer that detects bugs regardless of the process involved, as depicted in Figure 2.

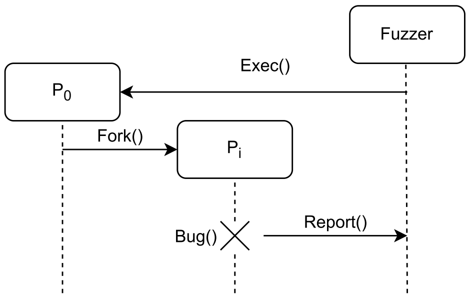

Fig. 2. Fuzzer capturing bugs from a child process

Second, capturing the cumulative code coverage of every process P_i plays a pivotal role in effectively fuzzing S. This well-established mechanism enables the fuzzer to generate new inputs that can uncover uncharted parts of the code, thereby enhancing the likelihood of uncovering previously undetected vulnerabilities.

Thirdly, it is essential to address the potential impact of a maliciously crafted request, which has the capability to instigate a denial-of-service scenario. When such an attack occurs, the performance of the server may be severely affected, leading to unresponsiveness or a significant slowdown [26]. Consequently, the execution time of processes can be prolonged beyond the expected limits, potentially triggering a timeout condition. This detrimental outcome not only diminishes the overall user experience but also poses a significant risk of revenue loss for the affected organisation.

These three considerations described can be applied to any process in the system, resulting in the definition of the “fork-awareness” property. This property refers to the ability of a fuzzer to test each process of an entire system in the same manner. In other words, a “fork-aware” fuzzer should be able to: detect bugs, timeouts and code coverage from all the processes under test [15]. Consequently, such a fuzzer would thoroughly and accurately test the system for potential bugs and vulnerabilities.

The impact of this work lies in the importance of the fork system call as a widely used pattern in many software systems. The ability to spawn new processes and run them independently is crucial for the efficiency and scalability of many applications, such as servers and operating systems. Moreover, in operating systems, the fork system call is essential for creating new processes and managing resources, enabling the system to run multiple applications concurrently.

Another example of the use of *fork* is for creating daemon processes, background processes that continuously run and perform tasks, usually by forking a new process and letting the parent process exit.

Overall, the use of *fork* is a well-established pattern in systems programming and provides an efficient way to create new processes that run independently of the parent process. As an illustration, a simple search for *fork()* on GitHub Search[8] returns over 500,000 C or C++ files, indicating the prevalence and importance of the *fork* system call in modern software development.

Understanding the implications of *fork*, particularly in the context of fuzz testing, is crucial for developing effective testing strategies that can detect bugs and vulnerabilities in complex systems. Building upon this concept, we have developed *Forkfuzz* and its corresponding workflow, which will be presented in the next section.

5 Forkfuzz

The workflow of *Forkfuzz* follows quite the same as *Honggfuzz*. To simplify the description, it can be dissected into three steps: setup, execution and termination as shown in Figures 3, 4, 5 and 6. In the figures, the novel contribution is highlighted in blue.

5.1 Setup step

First, *Forkfuzz* parses command line arguments and opens necessary files such as a dictionary of keywords or other interesting byte sequences, as well as a

[8] https://github.com/search?q=fork%28%29+%28language%3AC+OR+language%3AC%2B%2B%29&type=code

file containing a set of valid inputs for the *SUT*. Additionally, it prepares the signal handler for managing the fuzzing threads and constructs the required data structures, such as the coverage map, to manage the fuzzing campaign.

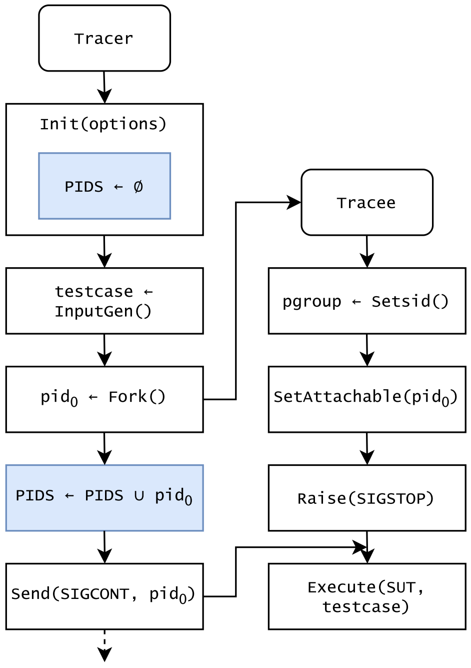

Fig. 3. Setup step

To keep track of all the processes active on the system during fuzzing, it is necessary to store their identifiers. To achieve this, we defined a novel set data structure $PIDS$. The $PIDS$ set contains an array of type pid_t and its length. The pid_t type is a built-in data type in C that represents a process identifier. After the fuzzer initialisation, the target initialisation begins.

First of all, the fuzzer generates the test case mutating the seeds. This test case is then passed to the System Under Test (SUT) as an argument, either through standard input or via a socket using the Netdriver module[9]. The SUT execution will happen in a process called *Tracee*, which is created by forking the main process of *Forkfuzz*, named *Tracer*. Note that *Tracer* and *Tracee* naming follow the *ptrace* nomenclature.

Before the actual execution of the SUT, the *Tracee* performs two important operations:

– it sets up a new process group using the *setsid()* function;

[9] https://github.com/google/honggfuzz/tree/master/libhfnetdriver

– sets itself as traceable by setting the *PR_SET_DUMPABLE* flag.

By instantiating a new process group, it is possible to keep track of all descendant processes at once. Moreover, by enabling the *PR_SET_DUMPABLE* flag, the process becomes attachable by the *ptrace* system call. After that, the *Tracee* blocks by raising a *SIGSTOP* signal in order to wait for also the *Tracer* to be ready to start.

Meanwhile, the *Tracer* records the identifier pid_0 of the new process in the *PIDS* set and starts tracing it with *ptrace*. Next, the *Tracer* sends a *SIGCONT* signal, allowing the *Tracee* to resume execution. The next step for *Tracee* is to call the *exec* function on the *SUT* with the test case generated earlier. At this point, the *Tracee* image is replaced with the *SUT* and the actual test begins.

5.2 Execution step

With the help of *ptrace*, the fuzzer is able to trace and monitor the events that occur while the *SUT* is running. The *Tracer* continuously performs the *waitpid* function over the process group previously set to capture events. When one of the *SUT* processes stops, it means that something occurred, hence, an event happened. Then, the fuzzer analyses the event to determine its nature.

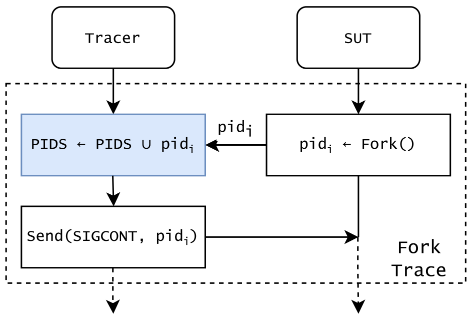

Fig. 4. Fork event tracing

Forkfuzz can handle two events: the invocation of two system calls - *fork* and *exit*. The *fork()* system call is captured using the *PTRACE_EVENT_FORK* option, allowing the *Tracer* to capture the identifier pid_i of the new process, add it to the *PIDS* set, and resume the*SUT* execution by sending a *SIGCONT* signal. Notably, the option also allows the fuzzer to start tracing the newly forked process automatically and looks for events that occur within it.

Additionally, if a process calls *exit()*, the pid_i of the calling process is removed from the *PIDS* set. If pid_i has terminated its execution in an unexpected manner, it raised a signal, which will be reported as a crash. In either case, the execution of the SUT resumes by sending a SIGCONT signal. These two steps are crucial for tracking all processes throughout their entire life cycle.

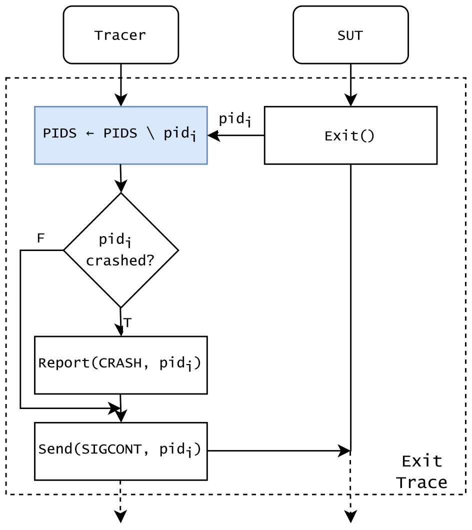

Fig. 5. Exit event tracing

5.3 Termination step

After tracing all the descendant processes, *Forkfuzz* waits for all of them to finish executing. If all the processes conclude the execution within the specified time limit, i.e. the $PIDS$ set empties before the timeout expires, the run ends successfully. However, if at least one process remains active at the end of the time slot, *Forkfuzz* reports the run as a timeout and kills all pending processes.

In addition, it succeeds to notify the tester which process went into a timeout exactly, since it is still inside $PIDS$. This operation ensures that *Forkfuzz* can detect and report all bugs and vulnerabilities, even in complex systems with multiple processes. Without this operation, *Forkfuzz* would risk leaving some pending processes running indefinitely, potentially missing critical bugs or causing system instability, as $AFL{+}{+}$ and other fuzzers do. Therefore, the ability of *Forkfuzz* to track and handle process termination is the core of the fuzzer since it can effectively fuzz multi-process systems.

To test the capabilities of the fuzzer, we carried out a series of evaluations using different case studies. The findings from these experiments are presented in the next section, along with a detailed analysis of the results.

6 Evaluation

We evaluated the effectiveness of *Forkfuzz* through a series of experiments on:

1. The *Dining Philosophers Problem (DPP)*
2. A bug-injected version of the *Producer-Consumer Problem (PCP)* in which the consumer, executed in a child process, crashes for specific inputs.

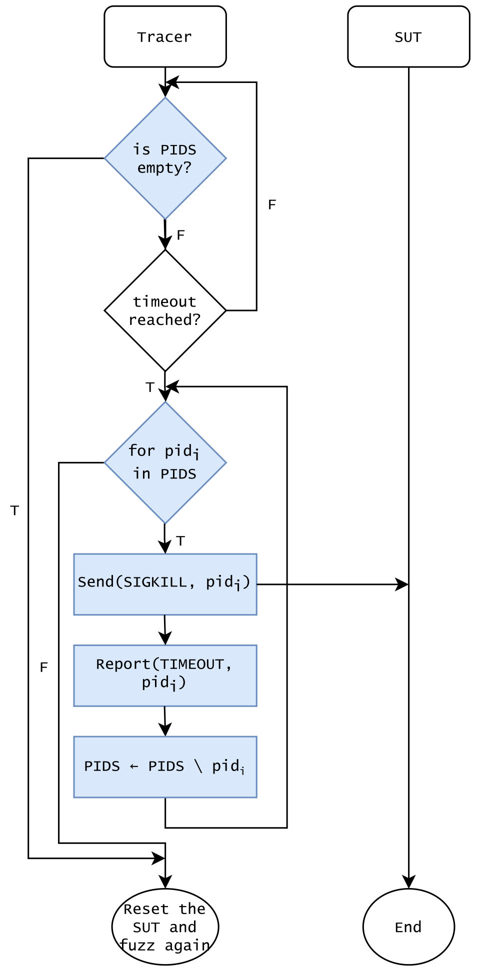

Fig. 6. Termination step

3. A Web Server that employs a Fork-Based Process Model to handle multiple connections. In this scenario, the server invokes the fork() system call to create a new process for each incoming connection, which presents a more complex and realistic use case for *Forkfuzz*.

The idea is to demonstrate the effectiveness of *Forkfuzz* on classical and realistic problems. In fact, the two classical problems presented have been chosen to evidence two possible and simple scenarios: one in which a child process hangs, while a child process experiences a bug in the other. In both classical problems, the parent process terminates its execution correctly. To apply such possible issues in a realistic case study, both scenarios have been transposed to a web server, as described throughout this work as an example. More details will be provided in Sections 6.1, 6.2 and 6.3.

6.1 Dining Philosophers Problem

The *DPP* is a classic problem in computer science that *Edsger Dijkstra* described in 1971 [5]. It is an example of a synchronisation problem, which arises when multiple processes or threads access a shared resource. In the *DPP*, philosophers are seated at a round table with a fork between them. To eat, a philosopher must have both the fork to his left and the fork to his right. However, only one philosopher can hold each fork at any given time, which can lead to a deadlock in which all the philosophers are waiting for a fork to become available. As a result, each process (representing a philosopher) of the system stays in a hung state indefinitely.

This problem is useful to reproduce a plausible pattern where forked processes stay in a hung state indefinitely without the possibility of exiting. To avoid a process staying in a hung state and occupying resources, most fuzzers incorporate a timeout feature, which terminates a process if its execution time exceeds a specified limit. However, common fuzzers prevent only the parent process from getting stuck, leaving all child processes active and potentially filling the host memory, as is the case with *AFL++*.

Forkfuzz succeeds in detecting this kind of issue since it waits for all processes from the group to terminate by detecting the exit event. Consequently, if one or more processes remain active after the time slot expires, *Forkfuzz* deduces that they are probably stuck in a deadlock or infinite loop, reports them as a timeout, and terminates them. The tester can adjust the timeout by estimating the plausible duration of normal execution.

6.2 Producer-Consumer Problem

The *PCP* [6] is another classic computer science synchronisation problem involving two types of processes: producers and consumers. Producers generate data items and place them into a shared buffer, while the consumers remove the data items from the buffer and process them. The problem relies on ensuring that producers do not try to add data items to the buffer when it is full, causing a buffer overflow. Additionally, consumers should not try to remove them from the buffer when it is empty. In the setup proposed, the access to the shared buffer is handled correctly and the two processes do not interfere with each other.

The parent process is designated as the producer, and, as such, it reads strings from the standard input and sends them to the shared message queue. Once all input strings have been sent to the queue, the parent process sends the message *quit* to the queue and then terminates. The child process is designated as the consumer and is responsible for reading messages from a shared message queue. Once a message is received, the consumer reverses the string and checks if it is a palindrome. However, if the condition is verified, the program crashes abnormally. This behaviour is simulated using the *abort* function. Otherwise, the child process terminates correctly if the message is *quit*.

This case study outlines a situation in which the parent and child processes are loosely coupled. In fact, after the *fork* call, both processes go in their separate ways. The producer sends messages without notice if the consumer is reading. Ultimately, it terminates its execution without waiting for the child process to finish. Meanwhile, the consumer processes messages and possibly experiences a bug.

However, *AFL*-based fuzzers in their default configuration wait for the *SUT* to terminate and receive its exit status through a control pipe, which allows them to detect only one exit code per run. This limitation can be problematic when multiple processes are involved, such as in the *PCP* scenario. In contrast, *Honggfuzz* and *Forkfuzz* detect signals on a low level, which allows them to inspect multiple signals and identify which process crashes. This capability makes *Forkfuzz* a suitable fuzzer for testing scenarios that involve multiple processes, such as the *PCP* scenario, where crashes may occur in the consumer (child) process.

6.3 Web Server

The third case study focuses on a web server, which presents different challenges in terms of testing due to the complexity and variability of web applications. Unlike command-line programs, web servers take input through a network socket, so the fuzzer needs to send input over the network. To achieve this, we leveraged the *Netdriver* module of *Honggfuzz*, which waits for the target to be available on a predetermined port before injecting input[10].

Another challenge when fuzzing a web server is caused by the underlying protocol. Although HTTP is a stateless protocol, the server can still maintain an internal state. For example, certain operations may require a user to be logged in and have specific privileges. In this case study, we will not address the stateful nature of web servers, but there is extensive literature on related works [4].

In our case study, the complexity of the web server fuzzing task increased due to the use of a forking model for request handling. In this model, the main process accepts incoming connections and then delegates the handling of those connections to child processes created by *fork*. As a result, each child process operates independently of the others, introducing a level of concurrency and potential race conditions. This presents a challenge for the fuzzer, which must be able to track and monitor the behaviour of these child processes. Failure to do so could result in missed bugs and other issues that arise from the concurrent execution of multiple processes.

Our setup is based on an open-source implementation in C of the described web server model[11]. This server exposes different endpoints that respond to GET

[10] http://blog.swiecki.net/2018/01/fuzzing-tcp-servers.html
[11] https://github.com/foxweb/pico

and POST methods. In particular, when the GET method is invoked on a non-existent path, the server attempts to retrieve the corresponding file from the public directory. However, the code has a character limit on the path length. As a result, if a user sends a request that exceeds this limit, the server becomes vulnerable to a buffer overflow attack.

In addition to the buffer overflow vulnerability, we have added a dangerous request that can make the server susceptible to a Denial-of-Service (DoS) attack. In particular, the "is_prime" POST request takes a number as input and applies a primality test: the simple trial division algorithm [1]. This algorithm has a complexity of $\mathcal{O}(\sqrt{n})$ and, for big numbers could take several seconds or even minutes to execute, depending on the machine it is running on. Therefore, the role of this algorithm is to simulate a scenario in which an attacker could send multiple requests with large inputs, which would keep the server occupied and disrupt the service.

As expected, *Forkfuzz* was able to detect both issues, particularly the second, which is challenging to identify with standard fuzzers. While timeouts are typically included in most fuzzers, they may not be sufficient to identify issues with forked child processes. Overall, our findings highlight the effectiveness of *Forkfuzz* in detecting both crash and timeout issues both in classical and realistic scenarios. Additional discussion follows in Section 7.

6.4 Performance

The performance of *Forkfuzz* was evaluated in multiple case studies, and the results were promising. Forkfuzz is built on top of *Honggfuzz*, one of the most effective coverage-guided fuzzers as reported by Fuzzbench benchmark[16]. The fundamental difference between Forkfuzz and Honggfuzz is the management of multiple processes.

"Fork-awareness" is upheld through a set of Process IDs ($PIDS$), which is updated upon new process creation and process termination. Adding entries to $PIDS$ is a swift $O(1)$ operation, while removing them, with an $O(n)$ complexity, marginally impacts overhead during both addition and deletion.

Note that overhead may rise in systems with numerous processes during a single run, though this is uncommon. In summary, *Forkfuzz* performs comparably to *Honggfuzz*, with negligible overhead on typical systems.

7 Discussion

7.1 Limitations

In many cases, the *fork* system call is used with the *wait* and *waitpid* functions, forming the *fork-join* mechanism [18]. This mechanism involves the parent process creating a child process for a separate task and then waiting for the child process to complete.

Consequently, Forkfuzz does not exhibit a substantial distinction from other fuzzers in terms of handling timeouts, since they all wait for the parent process to conclude. As a result, if the child processes become unresponsive, the parent process will persist in waiting, ultimately resulting in a timeout. It is worth noting that the AFL family of fuzzers does not inherently identify bugs within the child process. Therefore, if the parent process does not detect misbehaviour in the child process, AFL fuzzers will never spot the bugs.

Forkfuzz excels when testing systems with loosely-coupled processes, where each process operates independently and follows its own distinct path. In such scenarios, where processes operate independently without strong dependencies or synchronisation requirements, Forkfuzz excels in automatically detecting bugs and timeouts that may remain undetected by the individual processes within the system. In other words, if the processes within the system are already capable of notifying errors themselves, *Forkfuzz* may not provide significant additional benefits.

An additional limitation arises from the absence of the persistent mode. Persistent mode[7] is a fuzzing strategy that improves performance by running the SUT within the same process instead of creating a new one for each test case. Enabling this mode needs more precise management of the $PIDS$ set, which remains an area for future improvement.

7.2 Aggregated coverage

Forkfuzz currently employs aggregate code coverage without distinguishing individual process contributions. Future work may investigate the benefits of separate coverage maps per process, offering insights into their coverage and enhancing fuzzing accuracy.

Furthermore, this technique would be especially valuable for processes using the *fork-exec* paradigm, where the child process is replaced with another program [22]. Capturing separate coverage maps for multiple programs in a single fuzzing run would allow independent coverage assessment.

7.3 Areas of improvement

Distributed Fuzzing The concept underlying this research is to conduct parallel fuzzing of concurrent processes. Building upon this concept, it is possible to explore concurrent processes running on different machines, thereby forming a distributed system. As a future direction, this idea can be extended to encompass concurrent and distributed software. The objective is to investigate whether simultaneously fuzzing the entire system can yield improvements in terms of performance, code coverage, and the effectiveness of bug detection.

Real-World Benchmark Forkfuzz has been evaluated through a series of simple, yet realistic, case studies. The objective was to provide a clear explanation of the approach while demonstrating its effectiveness. However, it is important to note that future work will involve testing the approach on real-world systems.

Extending Support to Non-Linux Systems Expanding *Forkfuzz* to support software on non-Linux systems is another crucial direction. While *Forkfuzz* currently focuses on fuzzing software using the *fork* system call in Linux, there is a need to address other process creation mechanisms such as the Windows *CreateProcess* function and their equivalents in various operating systems. We plan to enhance the applicability of *Forkfuzz*, enabling the detection of bugs in a broader spectrum of software systems.

Concurrency bugs One of the major challenges in concurrent software testing is the presence of concurrency bugs. These bugs occur due to the interleaving of processes running concurrently, where different schedules can produce different results. *Muzz* [3] addresses this challenge by adjusting thread priorities and manipulating execution orders to uncover potential concurrency bugs. This approach helps in systematically exploring different execution paths and identifying vulnerabilities that may only manifest under specific interleavings. As future work, further enhancements can be made to systematically execute processes in different orders, effectively expanding the exploration space and increasing the chances of discovering subtle concurrency bugs.

8 Concluding remarks

Although state-of-the-art fuzzers have proven very effective in finding bugs and timeouts in the parent process of the target system, we found that they are somewhat limited over child processes. This paper presented *Forkfuzz*, a new fuzzer based on *Honggfuzz* capable of inspecting child processes. *Forkfuzz* was verified over three case studies, and the findings are:

- In the *Dining Philosophers Problem* case study, *Forkfuzz* identifies a timeout;
- In the *Producer Consumer Problem* case study, *Forkfuzz* detects an artificial bug occurring in the child process;
- In the web server case study, *Forkfuzz* finds a buffer overflow vulnerability and a denial of service (DOS) attack.

It is evident from our experiments that *Forkfuzz* serves as a valid fuzzer for identifying bugs and timeouts in child processes. Also, it sheds light on the importance of fuzzer awareness of the multi-process nature of the *SUT*.

Furthermore, we discussed limitations in current approaches, including creating harnesses and sequential process flattening using methods such as *defork*. These discussions emphasise the ongoing research focus on automating fuzzing techniques, where *Forkfuzz* aligns with the goal of enhancing the automation and effectiveness of fuzzing.

Beyond our experiments, we outlined future directions addressing concurrency bugs and distributed system testing challenges. By systematically exploring different execution orders and scheduling patterns, future work can further enhance *Forkfuzz* to detect concurrency-related vulnerabilities effectively. By incorporating these advancements, *Forkfuzz* strives to enhance its capabilities and contribute to the overall progress of automated fuzz testing methodologies.

Acknowledgment

We would like to express our gratitude to the **Consortium GARR** for providing us with the necessary cloud services to conduct our experiments and evaluation.

References

1. Barnes, C.: Integer factorization algorithms. Oregon State University (2004)
2. Böhme, M., Cadar, C., Roychoudhury, A.: Fuzzing: Challenges and reflections. IEEE Software **38**(3), 79–86 (2020)
3. Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H., Liu, Y.: MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2325–2342. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu
4. Daniele, C., Andarzian, S.B., Poll, E.: Fuzzers for stateful systems: Survey and research directions. arXiv preprint arXiv:2301.02490 (2023)
5. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta informatica **1**, 115–138 (1971)
6. Dijkstra, E.: Co-operating sequential processes. In: Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys. pp. 43–112. Academic Press Inc., United States (1968)
7. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: Afl++ combining incremental steps of fuzzing research. In: Proceedings of the 14th USENIX Conference on Offensive Technologies. pp. 10–10 (2020)
8. Fioraldi, A., Mantovani, A., Maier, D., Balzarotti, D.: Dissecting american fuzzy lop–a fuzzbench evaluation. ACM Transactions on Software Engineering and Methodology (2023)
9. Halsall, F.: Computer Networking and the Internet. Pearson Education (2006), https://books.google.it/books?id=RvW-6t-uwaYC
10. Jiang, Z.M., Bai, J.J., Lu, K., Hu, S.M.: Context-sensitive and directional concurrency fuzzing for data-race detection. In: Proceedings of the 29th Network and Distributed System Security Symposium (NDSS) (2022)
11. Keniston, J., Mavinakayanahalli, A., Panchamukhi, P., Prasad, V.: Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In: Proceedings of the 2007 Linux symposium. pp. 215–224 (2007)
12. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1), 1–13 (2018)
13. Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.H., Song, Y., Beyah, R.: Mopt: Optimized mutation scheduling for fuzzers. In: USENIX Security Symposium. pp. 1949–1966 (2019)
14. Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2019)
15. Maugeri., M., Daniele., C., Bella., G., Poll., E.: Evaluating the fork-awareness of coverage-guided fuzzers. In: Proceedings of the 9th International Conference on Information Systems Security and Privacy - ICISSP,. pp. 424–429. INSTICC, SciTePress (2023). https://doi.org/10.5220/0011648600003405

16. Metzman, J., Szekeres, L., Simon, L., Sprabery, R., Arya, A.: Fuzzbench: an open fuzzer benchmarking platform and service. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. pp. 1393–1403 (2021)
17. Natella, R.: Stateafl: Greybox fuzzing for stateful network servers. Empirical Software Engineering **27**(7), 191 (2022)
18. Nyman, L., Laakso, M.: Notes on the history of fork and join. IEEE Annals of the History of Computing **38**(3), 84–87 (2016). https://doi.org/10.1109/MAHC.2016.34
19. Padhiyar, S., Sivaramakrishnan, K.: Confuzz: Coverage-guided property fuzzing for event-driven programs. In: Practical Aspects of Declarative Languages: 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18-19, 2021, Proceedings 23. pp. 127–144. Springer (2021)
20. Pailoor, S., Aday, A., Jana, S.: Moonshine: Optimizing os fuzzer seed selection with trace distillation. In: USENIX Security Symposium. pp. 729–743 (2018)
21. Pham, V.T., Böhme, M., Roychoudhury, A.: Aflnet: a greybox fuzzer for network protocols. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 460–465. IEEE (2020)
22. Stevens, W.R., Rago, S.A., Ritchie, D.M.: Advanced programming in the UNIX environment, vol. 4. Addison-Wesley New York. (1992)
23. Swiecki, R.: Honggfuzz. Available online at: http://code. google. com/p/honggfuzz (2016)
24. Takanen, A., Demott, J.D., Miller, C., Kettunen, A.: Fuzzing for software security testing and quality assurance. Artech House (2018)
25. Tanenbaum, A.S., Bos, H.: Modern Operating Systems. Prentice Hall Press, USA, 4th edn. (2014)
26. Tripathi, N., Hubballi, N., Singh, Y.: How secure are web servers? an empirical study of slow http dos attacks and detection. In: 2016 11th International Conference on Availability, Reliability and Security (ARES). pp. 454–463 (2016). https://doi.org/10.1109/ARES.2016.20
27. Zheng, M., Sun, M., Lui, J.C.: Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In: 2014 international wireless communications and mobile computing conference (IWCMC). pp. 128–133. IEEE (2014)