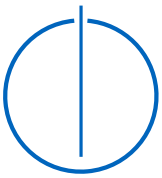
DEPARTMENT OF INFORMATICS
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Forkever: A Framework for testing and exploiting programs

Jasper Karl Ottomar Rühl



DEPARTMENT OF INFORMATICS
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Forkever: Ein Rahmenprogramm zum Testen von Programmen und zur Ausnutzung derer Schwachstellen

Forkever: A Framework for testing and exploiting programs

Author:	Jasper Karl Ottomar Rühl
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Clemens Jonischkeit
Submission Date:	15th of October 2020

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum

Jasper Karl Ottomar Rühl

Acknowledgements

I want to thank my advisor Clemens Jonischkeit for contributing with helpful tips and introducing Forkever to his binary exploitation course.

I want to thank my amazing girlfriend for taking great care of me when I was recovering from my knee injury, helping me through some of the worst days of my life.

I want to thank my parents for enabling my privilege to be a full-time student, following my passion at the TU Munich. In addition, I want to thank my father for visiting me after my injury.

I want to thank my friends Leander, Clemens H. and Simon for proofreading this thesis and providing me valuable feedback.

I want to thank my supervisor Professor Eckert for giving me the opportunity to write this thesis at her chair and answering the questions I had in earlier semesters when I was unsure about which subjects to elect.

Abstract

In cybersecurity it is fundamental to know your enemy. As any compromisation of a cyber system finds its root in an exploit of a vulnerable of software, their mitigation requires a thorough understanding.
When studying offensive security a lot of time is naturally spent on debugging programs to understand the different classes of exploits, such as code injection-, code reuse- or data only-attacks.
This process can be tedious: A lot of times probing for exploits does not yield the desired effect. The program crashes, has to be restarted and all steps need to be repeated to restore the possibly vulnerable state. To speed this task up, we present a tool that simplifies this by providing easy interaction with memory, insertion of function calls at arbitrary points in time and the ability to fork the inspected program to avoid restarting upon crash.
We achieve this by memory mapping an additional page of memory to which we write instructions responsible for the added functionality, which are then executed by setting instruction pointer to said instructions using `ptrace`.
Additionally, we demonstrate how this framework can be used for evaluating inputs for fuzzing and compare the performance of our alternative approach with two commonly used ones to find that forking programs at specific points in control flow is only applicable in niche contexts.

Contents

1 Introduction

In this section we first motivate the introduction of Forkever as a new tool for interactive program analysis to then list our contributions.

1.1 Motivation

When analysing a program's behavior, one typically uses a debugger such as GDB (GNU Debugger). Since exploiting of a program often involves complicated input/output actions, one uses scripts interacting with the program's standard input and output. Then, the researcher steps through the program to see how it processes the given input. Many times the exhibited behavior is neither the expected nor the desired one: The program might crash due to a segmentation fault, flawed input, a function call with malformed parameters (e.g. `free`ing an address that has not been `malloc`ed) or violation of a security mechanism (e.g. change of stack canary).

All these cases lead to the inevitable restarting of both the debugger and the script. This is a repetitive and tedious process, often including stepping through multiple breakpoints again to get to the point of interest. If there is some random element to the exploit, this process might have to be repeated multiple times just to restore the state in which the researcher wants the program to be.

Another issue when debugging programs is inspecting and altering of memory. While inspecting memory is possible in GDB, it is neither very intuitive nor interactive. When using the *examine* function, the terminal is cluttered and one needs to repeat the process to spot possible changes.

1.2 Contributions

We provide a framework that introduces a new mechanism aiming to eliminate the redundant procedure of restarting the debugged program. Further, we use that framework in a tool that helps researchers in understanding and exploiting programs. It offers easy viewing and editing of the program's memory, function call insertion and of course the ability to save its state by forking it, which allows for careless experimenting with the program.

2 Background

Before jumping into the design of Forkever, we want to give a broad introduction on some of the technologies Forkever builds upon.

2.1 ptrace

ptrace is a system call that was first introduced in Version 6 Unix and has made its way into other Unixlike operating systems like Linux or MacOs since[10]. It can be used to control and alter another process in many different ways. ptrace's functionality includes:

- reading and setting registers
- reading and writing to arbitrary locations in memory
- intercepting all signals that are sent to the traced process
- sending any signal to the traced process
- stopping execution
- continuing execution, optionally for just one instruction

These powerful features are the backbone of many program analysis tools such as GDB or Strace. To trace another process and be able to manipulate it, a program needs to execute the ptrace system call and indicate which process it would like to trace, attaching onto it upon success. Because manipulating a process is a very powerful ability, the kernel decides if the calling process is allowed to attach to it. This depends on a number of factors, which can be configured by the user. On most Linux systems the default rule is that processes can only be traced by their parents to prevent malicious programs from attaching to arbitrary processes[3].

Since ptrace is a system call, any interactions with it are made by calling it with different parameters. The kernel takes care of the actual writing, signal sending et cetera.

2.2 Fork

The `fork` system call is used to create an "exact duplicate" of the process invoking it (called the parent process). The newly created process is called the child process. At time

of creation, the content of the processes' memory is equal while they are also running in separate memory spaces. Open file descriptors are inherited as well, meaning that parent and child initially share the same standard input (STDIN), output (STDOUT), and error (STDERR) streams.

To differentiate between the parent and child, some exceptions to this equality have to be made. Some of these are

- the process IDs
- the child's parent process ID is equal to the parent's process ID
- the child's set of pending signals is initially empty

In Linux, `fork` is implemented using a copy-on-write mechanism. This means that while the processes have their own virtual memory space, these spaces translate to the same physical memory. A physical frame is only copied once the page is modified by either the parent or the child. Thanks to this mechanism, forking induces just a small penalty, namely the "time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child"[8]. This makes duplicating processes several times for analysis purposes very inexpensive.

The `fork` system call is also used to create processes running in the background, also called daemons. The most prominent creator of these is the "init" process that also goes by the name "PID 1": On Unix systems, this process is the first process that is started by the kernel during boot. It forks itself many times to subsequently create other processes necessary for the operating system.

In earlier days, forking a process was the exclusive way to create new processes in Unix, leading to the "init" process being the oldest ancestor to all processes running on a system[16]. Nowadays, other mechanisms of creating new processes have been introduced, namely `posixspawn`. The rationale behind this alternative solution lies in the complex requirements of `fork`, which includes a component for address translation that is necessary to have different virtual memory point to the same physical memory. Therefore, the specification of `posixspawn` explicitly states that it does not try to be a perfect replacement, but a substitute for "at least 50% of typical executions of fork" aimed for "systems that have difficuilty with `fork()`" and that it "must be without an MMU or unusual hardware"[9].

2.3 Pipes

In Unix, pipelines, abbreviated as pipes, are unidirectional streams for inter-process communication that were introduced as early as 1973 in Version 3 Unix[14]. They can be created by the `pipe` system call, which returns two file descriptors: One to write data to and one to read this data from. Any of these can be passed to a second process, either by inheriting it or via Unix sockets, to then send or receive data. The data producing process and the data consuming process may do so at different speeds. Consequently, buffering is employed, which means that bytes written to the pipe are stored by the kernel until consumed. Since memory is finite, one cannot indefinitely write data to a pipe if nothing is read from the corresponding end. This leads to the write and read system calls blocking if the buffer is full. When trying to write to a full pipe or read from an empty

one, the respective system calls block until enough space or some data becomes available (respectively). In modern Linux systems, the size of this buffer is 64 KiB by default.

2.4 Procfs

The procfs is a special filesystem presenting data of individual processes as regular files. Its introduction was motivated by debuggers solely relying on ptrace, which did support reading and writing to another processes memory, albeit at a very slow speed as it required "two context switches per word of data transferred"[12].

At its initial proposal, one could, given she had the permission, access the memory of process pid by opening the file /proc/pid. However, the Linux kernel has extended this approach and added other useful files, resulting in that the processes memory can now be accessed via /proc/pid/mem. One of these useful files, namely "maps", depicts the processes' virtual address mappings, e.g. the location of the stack, the processes' base image or shared libraries used.

2.5 ASLR

Address Space Layout Randomisation aims to make code reuse attacks harder by randomising the placement of a program's different memory segments in virtual memory, preventing malicious actors from reliably reusing code at certain addresses. It has been introduced in 2002 and found its way into Linux three years later[17].

On Linux, the smallest amount of memory distributed by the kernel is one memory page of the size 4096 (0x1000) bytes which means that the random addresses will always be aligned to a multiple of 0x1000. In a Linux 64 Bit system, the largest possible address in userspace is 0x7fffffffffffff with a length of 55 set bits. Due to the aforementioned page alignment and being specified to internally only use 48 Bit physical addresses, we end up with 28-32 Bits of entropy, depending on configuration. It goes without mentioning that the earlier descriped /proc/pid/maps file is a great help for analysing and debugging any sort of process. Further, note that ASLR does not randomise the address of the base image if it is not a position independent executable (PIE).

2.6 PIE

A position independent executable is a binary image that does not rely on being placed at a fixed addresses to execute properly, allowing its placement in memory to be randomised by ASLR. To achieve this independence, addressing data is often done relative to the instruction pointer. Further, so-called "function stubs" are used to tackle the challenge of shared libraries which addresses are unknown at compile time. These function stubs are collected in the so called Procedure Linkage Table (PLT) of the binary. They simply forward the function call to the respective function in the shared library.

As just mentioned, the address of this shared library is not known, so the stub cannot directly call the actual function. Initially, the stubs instead redirect the call to a function address that has been placed in its memory by the loader when the program was loaded and started. This function will then calculate the address of the initially desired function, save the address in the program's memory and finally call it. Subsequent calls to the stub

will now immediately forward to the desired function as its address has already been resolved. Note that this procedure has to be done with any program utilizing shared libraries.

2.7 Interrupts

```
119b <busy>:
...
11bc:   5d                      pop    rbp
11bd:   c3                      ret
11be <is_in_alphabet>:
11be:   55                      push   rbp
11bf:   48 89 e5                mov    rbp,rsp
```

Listing 1: *Example where a 2 bytes long instruction creates problems*

Interrupts are generated by the CPU if it encounters an exceptional condition and requires assistance from the kernel, such as a division by zero or an illegal instruction. If the kernel is able to, it will tend to the condition and resume the execution of the code that generated the interrupt afterwards. If it cannot, a signal is sent to the responsible process, where it is either handled by a signal handler or processed in a default way that is described by the according POSIX standard[11].

In x86 assembly, an instruction exists that enables programmers to generate these interrupts without meeting the necessary conditions for the CPU to raise the interrupt, namely the `INT` instruction. It is 2 bytes wide, one byte to indicate the instruction and one byte to indicate the value of the interrupt that is to be raised. Due to a one byte wide operand, we end up with 256 different interrupt numbers we can generate.

One particularly interesting and often used instruction is the `INT3` instruction, intended for calling the debug exception handler. While the instruction could be written in the same 2 bytes wide manner, there is a special one byte long opcode doing the same. This property comes in very useful when debugging, as it allows the insertion of breakpoints while overwriting just one instruction [4].

See Listing 1 as an example: If one wanted to set a breakpoint at the very end of the `busy` function, this could not be done with a 2 byte wide instruction without unintendedly changing part of the adjacent function. Thus, any calls to `is_in_alphabet` prior to reaching the breakpoint would end in a disarrayed execution instead.

2.8 Signals

Signals are very similar to Interrupts, except that they are generated by the kernel and sent to a process. In some cases interrupts raised by the CPU are forwarded to the process in form of the SIGINT signal, in others a more specific signal is sent. For example, the aforementioned interrupt caused by `INT3` leads to a SIGTRAP signal on Linux[11]. The corresponding default action would be to exit and dump the core file. With a debugger attached through ptrace to the process, however, this signal is intercepted and the debugger deduces an `INT3` instruction, and thus most likely a breakpoint, has been reached.

Programs can install signal handlers that will replace the default behavior. The SIGWINCH signal, for example, is sent to a process if the terminal in which it is executed changes its'

size. While the default behavior for this signal is to ignore it, some interactive programs continuously displaying information on the terminal handle this signal by recomputing their output.

Note that "the signals SIGSTOP and SIGKILL cannot be caught, blocked or ignored"[11]. Upon receiving a SIGKILL signal, a process exits immediately.

3 Design

The core functionality of Forkever is implemented in Python and can be controlled over the command line interface. For easy reading and altering of memory, one can spawn a hexeditor displaying a user-selected memory segment. Over a Unix socket, these two programs update each other on changes in memory made by the user or the program itself, respectively.

Forkever is made up by several Python Classes. Fig. 1 depicts it's architecture (omitting some small helper classes). Fig. 2 depicts the architecture of the hexeditor.

3.1 Forkever

The Forkever class is used to start the program and is responsible for parsing the arguments that were given to launch it, such as the name of the program we want to execute or the optional argument to enable randomisation. After the parsing is done, the "inputloop" function of the InputHandler is called.

3.2 InputHandler

The de-facto main function of Forkever resides in InputHandler.py, the controlling class. In an infinite loop, it awaits input from the terminal, Hyx or the debugged program and calls the appropriate function handling said input. It also is responsible for invoking Hyx.

3.3 InputReader

As there are multiple sources for user-input, a seperate thread is introduced that simply waits for and reads input given over the terminal. This is done so that Forkever's main thread will not wait for input over the command line when other action triggers (such as input via Hyx) might occur.

3.4 Pipe

To keep tampering with the forked processes at a minimum, all artificially created children use the same file descriptors as their parent (the original process). Writing something to

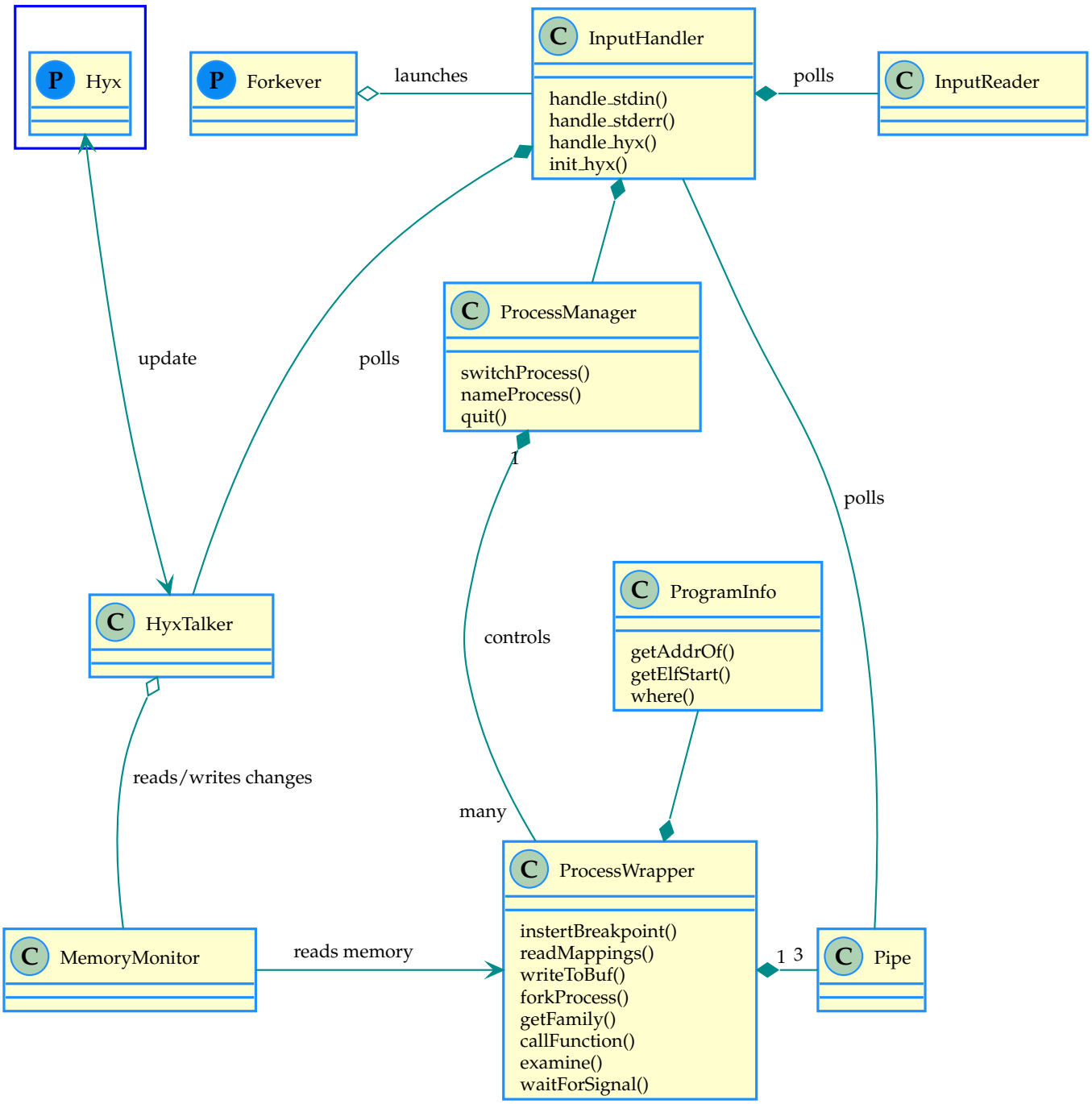


Figure 1: *Class Diagram of Forkever*

STDIN is not directly written to STDIN but instead internally cached and written once a read syscall requests data from STDIN.

3.5 HyxTalker

HyxTalker.py handles all the communication with Hyx over a Unix Socket. It is responsible for launching Hyx with the appropriate parameters, sending and receiving changes in memory and sending a whole memory segment if the user switches processes. Detected changes are sent interval wise using a simple selfmade protocol.

3.6 ProcessManager

To keep track of the various processes that are created by injected or naturally occurring fork instructions, ProcessManager was introduced. It allows for renaming and switching to processes and holds the debugger class implemented by the Python-ptrace library. Any non-ordinary events such as the execution of a new process through ”execve” are also handled by the ProcessManager.

3.7 ProcessWrapper

The ProcessWrapper is the core of Forkever as it takes care of the forking, breaking, continuing, examining etc. The insertions of forks or function calls is done via assembly code on a separate page of memory that was memory mapped there by temporarily injected code at the very start of the process. This is only done once and will save us writing to the process's memory in the future.

Several features of Forkever rely on injected instructions. To minimize writing accesses to the memory, these are written to a custom memory page, which has been mmaped by an injected syscall. Later on, when a feature requires some special instructions, the instruction pointer is set to the injected code on that page.

Fork To fork the debugged program the `fork` system call has to be called. This is done by either manipulating the register indicating the desired syscall (if the program was about to perform a syscall), or by setting the instruction pointer to a syscall instruction. After the syscall was performed, the states of the two processes are set back.

Function calls Forkever enables us to call any function at an arbitrary point of execution (except when the process is entering a syscall). The address of a function will be calculated and moved to a register while the instruction pointer will be set to point to an instruction calling the address stored in this register.

Breakpoints Breakpoints are implemented using the standard `INT3` instruction. Whenever a SIGTRAP signal is received the instructionpointer is matched with one of the saved breakpoints.

3.8 ProgramInfo

The ProgramInfo class is a small helper class responsible for translating virtual addresses into symbolnames (plus an offsets) and vice versa. For example, the user can enter `x libc:free_hook+1` to inspect the word starting 1 byte after the symbol `__free_hook` of the libc library.

```
./Forkever.py demo/vuln
type ? for help
b main+0x17
c
hit breakpoint at 0x5555555557c6
fork
switched to 22375
RIP = main + 0x17
x libc:free_hoo+1
__free_hook+0x1   00000000
si
RIP = setbuf + 0x0
<Enter key>
RIP = setbuf + 0x6
```

Listing 2: *Example demonstrating the effect of ProgramInfo*

If an address is inspected and printed out, the nearest smaller symbol and the according offset is printed. Further, when singlestepping or the user switches processes, the function in which the execution is currently halted is given. See listing 2 for an example.

3.9 MemoryMonitor

This class interacts with the memory segment the user wants to monitor or manipulate. Any changes by the user are written to memory by this class. Further, changes in memory by the program itself are detected by hashing the whole segment and comparing it to the previously calculated hash. A copy of the inspected memory is kept to then pinpoint the bytes that changed.

The below mentioned feature of slicing for interesting memory pages is also implemented here.

3.10 Hyx

Hyx is a hexeditor, which means that it is a small program that displays files as the raw byte fileformat. It offers VIM like navigation and allows for easy change of each single byte. Changes made in Hyx are propagated to the process's actual memory. Further, commands for Forkever can be issued directly in Hyx the way one would enter a console command for VIM.

When launching Hyx, the user may slice the desired memory segment with Python similar syntax. She may also give "i" (for interesting) as an index to limit the scope of the segment only to memory pages that contain a non-zero byte at the current time.

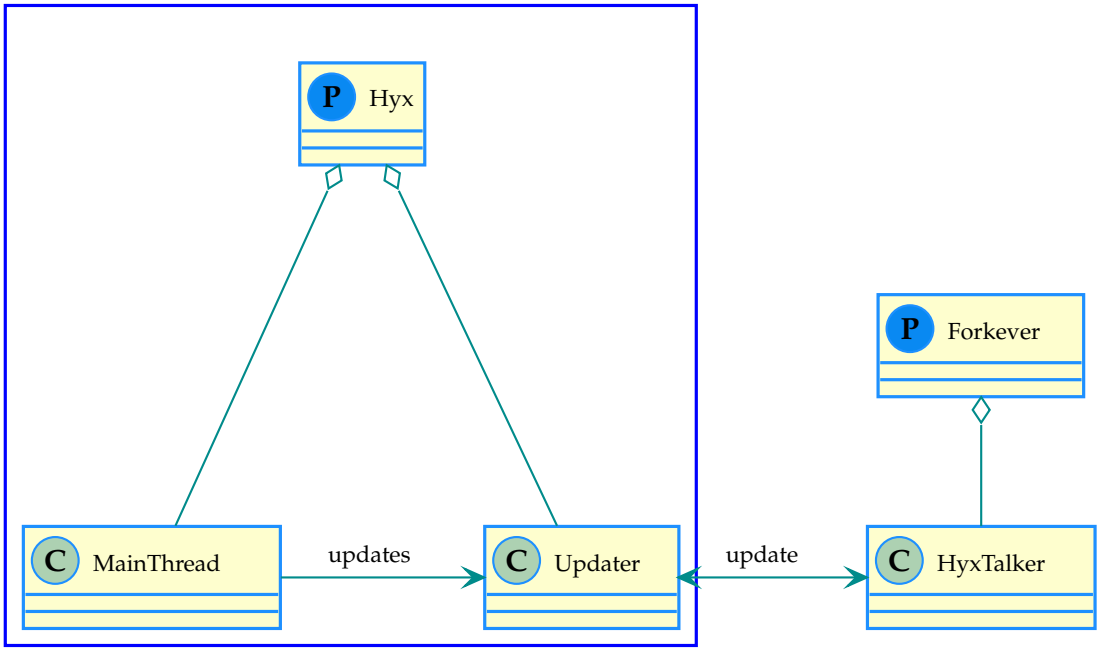


Figure 2: *Class Diagram of Hyx*

4 Implementation

This section will go over the components of Forkever, describing how they work on a detailed technical level.

4.1 InputHandler

The InputHandler class can be described as the main class that controls the different components. Using `poll`, it continuously monitors four sockets for events and acts accordingly. These 4 sockets monitor:

1. commands input via Forkever
2. commands and writes via Hyx
3. the program's STDOUT
4. the program's STDERR

In case the program has written something to its STDOUT or STDERR, the InputHandler prints it out and sends it to Hyx, if a Hyx Instance is running.

The UserInput socket receives commands that the user sends over the InputReader. The InputHandler executes the command and prints the result. Most commands by the user are forwarded to the ProcessManager, the Hyx Invokation however is done by the InputHandler and the HyxTalker class.

The Hyx socket receives both commands for Forkever and updates to memory. The InputHandler decides if the incoming data is a user command or a memory update. If it is the former, the command is executed, and the result sent back. If it is the latter, the appropriate HyxTalker function is called to handle the request.

While testing Forkever, a bug occurred when sending the "fork" command via Hyx to Forkever. After sending a command to Forkever, Hyx expects a 0x100 byte long response, which should be the result of the command. However, Forkever would successfully fork the program as the user requested, and switch to it. Whenever Forkever switches the process, the HyxTalker class sends the whole heap of the process to Hyx. This lead to Hyx crashing, as it was still waiting for the response of the command. Avoiding this edge case was done by checking if the user was requesting to fork and sending a response prior

forking. Further, a new function was introduced in Hyx so that it could receive messages from Forkever at any time to display the message generated by the fork. This mechanism is not really elegant and would certainly have to be improved in case more commands triggering this condition were introduced.

4.2 InputReader

The InputReader is a subclass of the Python Thread class and reads any data the user enters over the terminal. For the sake of extendability, the commands are sent over a `Queue`. As the InputHandler uses `poll`, a new class encapsulating the `Queue` was created that would send a single byte over a socket whenever something was placed in it's `Queue`.

4.3 Pipe

When launching the program that is to be debugged, Forkever passes the endpoint of a pipe as that program's STDIN. To communicate with it, we can now simply write data to the pipe.

While this is a simple solution, it bears one problem regarding sending data to STDIN: Since all processes share their file descriptors, they read input from the same pipe serving as STDIN. Suppose we had a process called $p1$. We continue execution of $p1$ up to the point where it is about to read some data from STDIN, stopping execution and writing something to STDOUT.

For some reason, we decide to fork $p1$, which yields the new process $p2$ that we now continue to execute. As $p2$ continues, it reads the data just written to STDIN. Shortly after, $p2$ crashes and Forkever switches back to $p1$. Even though we had written to $p1$'s STDIN, we need to write to it again, as the previously written data has already been consumed.

This problem was tackled by creating a second layer of pipes that cached the input and forwarded the correct amount of requested data to the debugged processes actual pipe when the process would request to read from it via the read system call. Forking a process copies the cached input-data to the new processes pipe cache.

4.4 HyxTalker

The HyxTalker class is responsible for making sure the states of the inspected memory segment and its representation in Hyx are consistent. It sends/receives updates in the specified memory segment to/from Hyx. The InputHandler calls this classes `updateHyx` method whenever it just handled some request from the socket. This method in turn then calls `MemoryMonitor.checkChange` to determine where the memory changed and sends any changes found to Hyx.

If Hyx transmits an external change, the update is received and forwarded to MemoryMonitor.

4.5 ProcessManager

This class is responsible for handling the different processes that were forked off of the initially launched process. It enables the user to switch between different processes and

selects the next process if the currently manipulated process terminates. As it keeps track of the various instances of the program, it also is responsible for killing them off when Forkever is quit.

Handling Events Certain events that can occur during the Process execution are handled by the ProcessManager, such as a new process being executed or the process exiting. When these occur in `ProcessWrapper.continue`, they are propagated to the ProcessManager using Pythons `raise` keyword. This allows to handle these events with a `try:except` mechanism and simply returning `ProcessWrapper.continue`'s result if none occur.

Ignoring signals The ProcessManager class allows the user to switch between different processes in a straightforward manner. However, one issue arose when one wanted to fork in a special constellation: Suppose the user forks $p1$ off of $p0$, continues execution on $p1$ and eventually produces a segmentation fault. $p1$ crashes and the ProcessManager automatically switches back to the parent process $p0$. Because $p1$ has terminated, the kernel sends a SIGCHLD to the parent process.

If the user now decides to continue execution on $p0$, the SIGCHLD signal is received by the tracer and forwarded to the executed program, which (in most cases) ignores it.

If she instead opts to fork $p0$, however, the SIGCHLD signal is received in the middle of the fork procedure. Because signal delivery is stopped until the process is allowed to continue execution, we need to intercept the signal without changing the process' state. To do so, we write an infinitely looping instruction to our special page, set the instruction pointer to said instruction and continue execution. This enables the signal delivery again, and the tracer receives the signal. After setting the registers back to their previous state, $p0$ can now be forked again. Although the need for this function stems from the ProcessManager, the actual implementation resides in the ProcessWrapper.

In an earlier draft, the instruction used to wait for the signal was not a self-looping instruction but a simple `NOP; INT3` instead. While this did work initially, it posed a problem during the evaluation, when numerous children were forked in a quick manner: The SIGCHLD signal could not be delivered quickly enough, leading to a SIGTRAP being generated and received (and ignored) instead.

4.6 ProcessWrapper

This class is the heart of Forkever where the Process manipulation through ptrace happens.

Extra code segment After process creation, a special page is memory mapped using the `MMAP` system call. This is achieved by injecting a system call instruction the same way it is done for forking the process. This is `MMAP`'s signature:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset)
```

addr specifies the location of our page in the virtual address space. If *addr* is set to 0, it is left up to the kernel to choose where to place the requested pages. In most Implementations, this will result in the pages being mapped right next to the stack and dynamically linked libraries (such as libc). This behaviour is undesirable as it might result in the page being

mapped between the stack and a library to be mapped in the future, which would change the distance from structures on the stack (exploiters might write to) to some data residing in the later mapped pages. As a result, exploiting a program with such an auxiliary page might go about differently. Nevertheless it is unavoidable to place the extra page somewhere, as can be seen below. We chose to place the extra page before the executables base image, with one page unmapped inbetween. E.g., if the image is loaded to address 0x9000 in memory, we set $addr = 0x7000$.

As the code we inject is of very little size, we set $length$ to the minimal suitable value 0x1000. $prot$ sets the permissions of the mapped page, we will set it so that the debugged program may execute code on it, but neither read nor write from it.

Our first design of this mechanism omitted the part where a custom page was mapped. Instead, the code was injected to the location of the current instruction pointer. The problem in this was the possibility of the called function using the injected code, which would have lead to an infinite recursion (limited by the stack).

Injecting function calls Once this page has been mapped, we write some assembly code there. The code simply is `CALL RAX; INT3`

RAX is set to contain the address of the function we want to call. The address of the function is determined by looking up the location of the respective symbol in the base image of interest and, if the ELF is a PIE executable, adding the images base address. If the user does not specify which ELF's symbol she wants to call, the ELF of the program itself is assumed. The `INT3` instruction serves to raise a SIGTRAP signal.

The part of the program handling the continuation of the process checks the instruction pointer whenever a SIGTRAP is raised by an `INT3` instruction. If it is equal to the location of this particular instruction, we can be certain that the inserted function has returned. We now check the registers for the return value and restore them to the state prior to the call injection.

Injecting fork system calls The key feature of Forkever is the ability to fork the debugged process at any time. The actual forking is of course not done by Forkever, but the kernel. Forkever simply instruments the binary to do a `fork` system call. If the program is about to enter a system call, we modify the syscall opcode to be the opcode of `fork`. Otherwise (in case the process would simply execute the next instruction), the instruction pointer is changed to point to a system call instruction that we have previously written on the aforementioned custom `MMAP`ed page.

The process then continues using `PTRACE_SYS`, leading it to halt execution upon entering (if it has not already), and completing the system call. As the process is set to trace forks, Forkever is notified about the fork. The newly created process is traced automatically by the underlying `Python-ptrace` library. Note that when implementing this procedure without the `Python-ptrace` library it could happen that the tracer receives a signal (`PTRACE_EVENT_STOP`) from the forked child before the SIGTRAP from the parent.

After that, the original registers are restored. Further, if the parent was entering a system call, `PTRACE_SYS` is done once so the processes enter that system call again.

Our initial draft for this feature did not rely on changing RIP but wrote the `SYSCALL` instruction to the current position of RIP. After forking, the original instruction was written back and RIP restored. The final method is significantly faster as it does not involve writing memory accesses. Details can be seen in the evaluation.

Breakpoints When analysing a program one naturally needs breakpoints. The breakpoints are created using the underlying `Python-ptrace` library, which writes the `INT3` instruction to the desired location. When the continue method hits a breakpoint, it is temporarily removed. Upon continuation, Forkever singlesteps over the current instruction at which the breakpoint is to be reinserted, writes the instruction again and finally continues execution. When a process child is created off of a Fork, seperate Python objects corresponding to its parents' breakpoint(s) have to be created.

Continue The logic for continuing the processes execution is split into two methods, `_getNextEvent` and `continue`. `_getNextEvent` handles the actual work of resuming using `PTRACE_SYSCALL` while `continue` calls the former to then differentiate on the various cases that occur when continuing.

Before resuming, `_getNextEvent` checks if there is a breakpoint to be reinserted or the process is requesting data from STDIN. Breakpoints are inserted as described above.

Using `PTRACE_SYS`, the process is continued and waited for. With the `PTRACE_SYSGOOD` flag enabled, we can differentiate between a SIGTRAP that was triggered by enterting/leaving a system call and a SIGTRAP triggered by reaching a breakpoint.

If a normal SIGTRAP is reached, `_getNextEvent` returns and `continue` decides what to do with it. However, if a system call is entered, the registers are read to determine the syscall. In case its a `READ` system call and the register indicating the file descriptor to be read from is 0 (STDIN), the appropriate amount of cached STDIN bytes are written to the processes STDIN. If no data is available in the cached STDIN, the function returns without resuming the process, as it would otherwise be waiting for the `READ` system call indefinitely.

Further, if the user defined the specific system call is of interest, `_getNextEvent` and `continue` return, printing out information about the syscall. The information are either the arguments of the system call or it's return value, respectively.

The various cases `continue` can encounter when calling `_getNextEvent` are:

1. Trap because Breakpoint was reached
2. Message about Syscall that was hit
3. Trap because an inserted function call returned
4. Signal was sent to the tracee
5. Other kind of event

A trap signal is delivered in two cases. Either when a breakpoint is reached or when an aforementioned inserted function call returns. In the former case, `continue` sets a variable to make sure `_getNextEvent` will reinsert the breakpoint, in the latter it calls a method that prints out the injected functions result and restores the registers.

Due to using ptrace, any signal that is to be delivered to the tracee is delivered to the tracer instead. If `_getNextEvent` returns a signal other than SIGTRAP, `continue` checks if this signal is in a user-defined list of signals to be filtered. Depending on whether it is to be sent or not execution is resumed, forwarding the signal if it is not in a user-defined list of signals. Note that at the current state, this list is empty.

If `_getNextEvent` yields another event (e.g. ProcessExited), the event is raised and handled by the ProcessManager.

4.7 ProgramInfo

The ProgramInfo class provides information about the ELFs loaded by the process. For construction, the name of the base image and the PID is required. When information about a specific virtual address is requested, /proc/pid/maps is read to determine which ELF resides at that address. This ELF is then loaded and cached using a function provided by the `pwntools` library. After adding the virtual address of the image to each symbol, the closest smallest symbol to the requested address and the distance to it is returned.

When the address of a symbol is requested, /proc/pid/maps is again opened and the respective ELF loaded, if it has not been cached already. Out of the available symbols, the shortest, for that the requested symbolname is an infix, is selected. Further, the selected symbolname's length can at most be 1.5 the length of the requested symbolname in order to avoid ambiguous findings.

4.8 MemoryMonitor

The MemoryMonitor is responsible for writing external updates to memory and determining changes in it. Upon construction of the class, it is given the ProcessWrapper and the desired boundaries in which the memory has to be monitored as arguments.

To write updates as the HyxTalker class requests, the MemoryMonitor simply calls the `writeBytes` method of its respective ProcessWrapper.

When initialized, the start and end of the segment that is to be monitored are passed as arguments. Optionally, one can specify to trim the scope for "interesting" pages, meaning pages containing at least one non-zero byte. If this is specified, the most outer page is read and bytewise summed up. If the sum is not zero, stop. Otherwise, the page is dismissed, the boundary adjusted, and the process repeated.

Detecting changes Upon initialisation, a copy of the memory segment is read in via /proc/pid/mem and hashed. Later on when the `checkChange` method is called, another copy is read, hashed, and the hashes are compared. If the hashes are not equal, the copies are compared bytewise to find the differences. There is some room for optimisation here: By hashing the segment more fine-grained, one could detect which of the subsegments changed and which did not, avoiding a significant part of the bytewise comparisons. The downside of this would be that it requires the multiple calling of a hash function and comparing of the resulting hashes.

4.9 Hyx

The ”frontend” for Forkever, Hyx, is a minimalistic Hexeditor originally written by Lorenz Panny. It has been modified to allow for communicating changes made to the memory, either by the user or the debugged program itself, to Forkever. To do so, a second thread was introduced, from here forward on called the ”updater”. The updaters routine consists of monitoring two sockets over which changes were transferred. One socket connects the updater to the main thread (socketpair), the other to Forkever. For invokation of Hyx, a few arguments are passed to it, namely the name of the unix socket, the name of the file serving as a snapshot of the memory segment as well as the virtual address of this segment.

If the user manipulates the file opened in Hyx, the main thread sends the address of the changed byte to the Updater via the other socket. The updater receives that address and looks up the new value of that byte before sending the change to Forkever, where the `HyxTalker` class takes care of writing the update into memory.

Memory updates by the process itself are recognized by Forkever and sent over the socket. Internal changes (made by the program itself) are directly processed by the updater with no involvement of the main thread. To prevent race conditions, a lock for accessing the array mirroring the memory is introduced. After acquiring this lock, the updater writes directly into the this array. If the main thread wants to write to the array (meaning the user is trying to change the memory) and the lock is currently held by the updater, the change is dismissed.

5 Evaluation

The following experiments were conducted on a notebook equipped with 8 GB of RAM and an Intel i5-6267U Processor (2 physical, 4 logical cores) running Debian Buster.

5.1 Speed of Forking

Fig. 3 shows the speed of forking a process multiple times. The forked process was halted at the start of `main` and repeatedly forked.

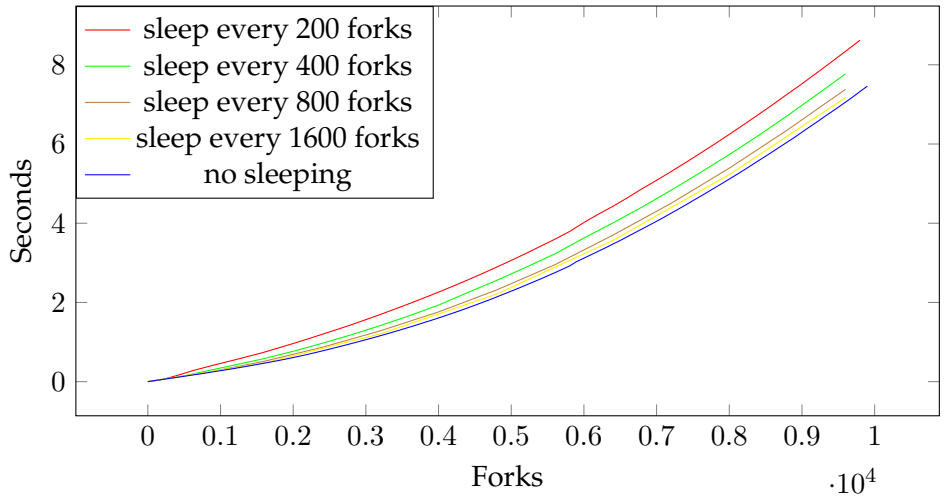


Figure 3: *Repeated forking of a single process over time*

As described in Chapter 4, the fork mechanism was changed to avoid several writing memory accesses. The above experiment was repeated with slight modifications, with these being the absence of delays or logging of intermediate timings. For both implementations, the experiment was conducted 20 times. Table 1 shows the empirical mean and empirical variances. Fig. 4 shows the result of a paired sided t-test. Note the very low p-value, indicating we should accept the alternative hypothesis that the MMAP implementation is faster on average.

	PtraceWrite	MMAPed
Mean	0.00086	0.00067
Variance	7.02e-10	2.68e-10

Table 1: *Empirical mean and variance for the two tested implementations, sample size = 40*

```
> t.test(ptrace,mmap, alternative="greater", paired=TRUE)

Paired t-test

data:  ptrace and mmap
t = 41.229, df = 39, p-value < 2.2e-16
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
0.0001787011          Inf
sample estimates:
mean of the differences
0.0001863151
```

Figure 4: *Paired TTest showing the improvement is statistically significant*

5.2 Forkever for fuzzing

Forkevers feature of forking a program at an arbitrary point in time is especially interesting for analyses of programs where the program is executed over and over again. A closely related example is Fuzzing, a technique often used to automatically discover flaws in software. In the following experiment, this approach was simplified, but still adheres to the same principle: Randomly generating various inputs and observing the program's reaction.

5.2.1 Fuzzed Program

The fuzzed program implements a small deterministic finite automaton, depicted in Fig. 5. Input is read bytewise from STDIN and each transition taken is counted. Once there is no more input available, the number of unique transitions made is printed out. This number gives a simple heuristic that aims to simulate the measurement of code coverage made by real fuzzers without the need of reinstrumenting the program.

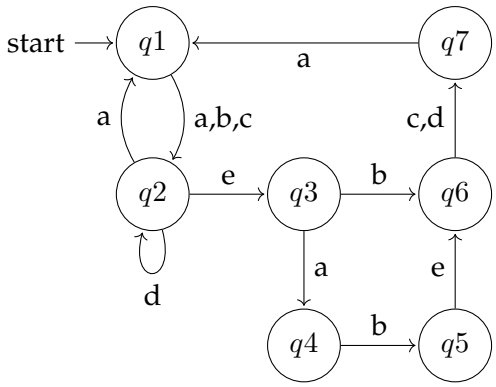


Figure 5: *Automaton for which a word covering all transitions was to be found*

5.2.2 Algorithm for generating samples

To generate the samples, an evolutionary algorithm was employed. It takes the N best scoring words and creates a mutation of each of these. The newly created mutations are then evaluated by the program. For each letter in the word that does not discover a new transition, 0.4 points are reduced to give an incentive for shorter words. Eventually, the algorithm should find a word of minimum length that traverses each transition of the automaton.

A word is manipulated by independently drawing n $\sim U_0^{length(word)}$ samples from a geometric distribution, where a sampled value of 1 corresponds with the highest index of the letter and 0 leads to the addition of a new letter. If a letter is chosen to be manipulated, a $\delta \sim U_0^{length(alphabet)}$ is added to it. Further, there is a probability p for every letter to simply be deleted. This choice was made to enable the possibility of words with non-optimal length to shrink.

It is worth noting that this algorithm was designed for this specific task and would not yield good results when applied on a generic binary, partly due to the reason that manipulation at the end of the input data is more likely than manipulating bytes at its beginning, which would lead to bugs related to that area being less likely to be found.

5.2.3 Different Fuzzers

To enable comparisons, the evaluation of samples was done in four different ways.

Naive The simplest and not Forkever requiring approach is to run a new process for each sample with which we wish to fuzz the program. This is easily implemented using Pythons subprocess module, namely the checkoutput function.

Naive with Forkever Initially we wanted to mirror the naive approach using Forkever instead of the subprocess module to evaluate the samples to measure the slowdown induced by ptrace. Unfortunately, this would have required rewriting large parts of the code as Forkever loads the ELF of the launched program at least once, which also requires it to open the relevant /proc/pid/maps file. This procedure leads to a major slowdown, making it unfeasible to be taken into the comparison. For the following implementations, this procedure is also executed, but just once instead of per-process.

```
1 void busy()
2 {
3     for (volatile int i = 0; i < BUSY_COUNT; i++) {};
4 }
```

Listing 3: *The slowing function called after reading a letter*

Forkserver The Forkserver closely resembled the earlier described technique by AFL. Since ptrace was used, it was not necessary to patch the server to call `waitpid`. Instead of forking once of the "server" process, we forked an additional time to then start fuzzing, to avoid having a multitude of SIGCHILD signals being sent to the "server" process. After being supplied with input the processes were detached.

As described by the creator of AFL in Section 6.3.2, this technique shines when dealing with programs consuming a lot of time at startup. Although this was not the case for our small DFA emulating program there still was a noticeable performance increase.

Forkfuzzer The Forkfuzzer utilized the feature of Forkever by forking the program after each read letter. A dictionary of input prefixes mapping to a ProcessWrapper is kept and extended. During evaluation, input is written to the process bytewise. After each letter, the process is forked, with one process being stored in a dictionary together with the input already being read as its respective key. When later evaluating another input, the process with the key that is the longest prefix available is selected from this dictionary and the procedure is repeated for the rest of the input string.

As Forkever uses `PTRACE_SYSCALL` to continue and this way stops at the read syscall, the ProcessWrapper was modified so that `PTRACE_CONT` was used instead to reduce the amount of slowing interactions with ptrace. As read syscalls could no longer be detected, the input was written directly to the processes STDIN without buffering.

After 120 generations, the ForkFuzzer would reach a limitation: It appeared that the Operating System imposed a limit of maximum processes that could be created, as it would always crash after about 10120 forked processes.

5.2.4 Slowing the program

The two fuzzers not using Forkever were orders of magnitude faster. To highlight the advantage of the Forkfuzzer, the automaton implementing program was artificially slowed down, entering a slowing loop after each character read from STDIN. After adding a slowing loop (see Listing 3) to the program, this approach was faster than the naive one.

This loop serves to simulate a computationally intensive step after the consumption of some data. A possible usecase might be an application stepwise rendering a GIF[1]. As the ForkFuzzer saves the program's state after each letter, the ForkFuzzer's advantage over the regular fuzzers increases with the duration of a transition.

The program was compiled without optimisations. Disassembling the binary shows that memory is accessed twice per iteration of the loop (Listing 4). This multitude of memory accesses is responsible for the major slowdown.

[1] https://awakened1712.github.io/hacking/hacking-whatsapp-gif-rce/

```
1 mov     eax, DWORD PTR [rbp-0x4]
2 add     eax, 0x1
3 mov     DWORD PTR [rbp-0x4], eax
4 mov     eax, DWORD PTR [rbp-0x4]
5 cmp     eax, 0x4ffff
6 jle     <busy+0xd>
```

Listing 4: *Disassembled iteration part of the slowing function*

5.2.5 Results

The following Figures depict the speed of the fuzzers. Each fuzzer was run once. The timestamps were taken in 5 Generation intervals.

Fig. 6 shows the performance of the fuzzers without any slowing instrumentations.

Fig. 7 shows the performance of the fuzzers with the variable `BUSY_COUNT` set to $0x50000$. The value was chosen so that it is the smallest multiple of $0x10000$ for that the forkfuzzer is faster than the naive fuzzer.

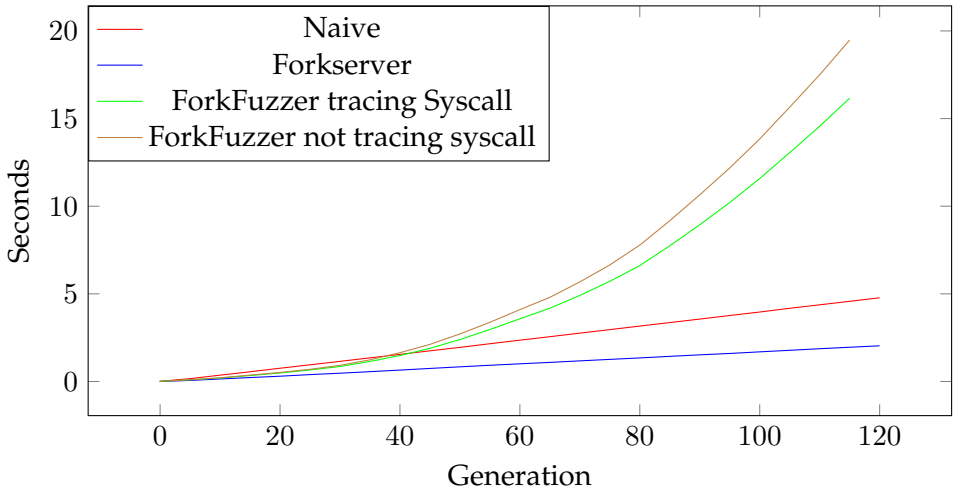


Figure 6: *Speed of sample evaluation methods with BUSYCOUNT = 0*

Fig. 8 compares the naive approach with the forkserver.

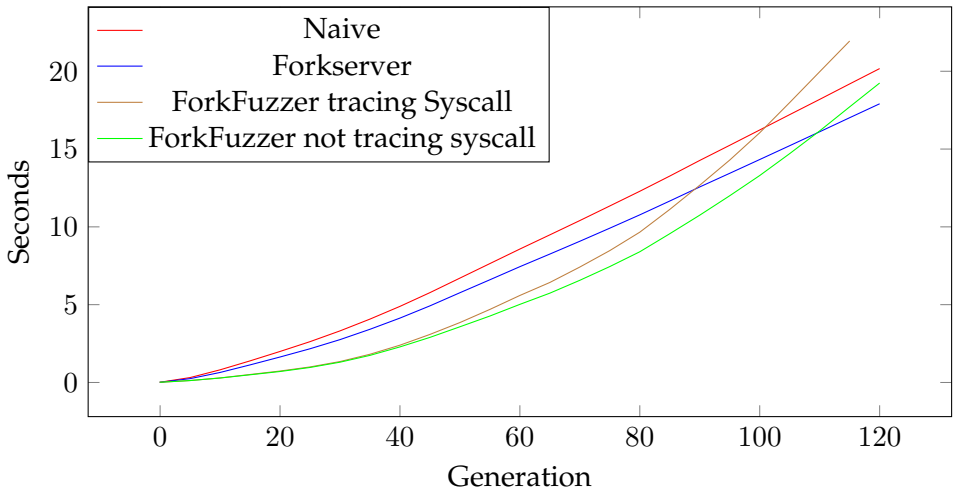


Figure 7: *Speed of sample evaluation methods with BUSYCOUNT = 0x5000*

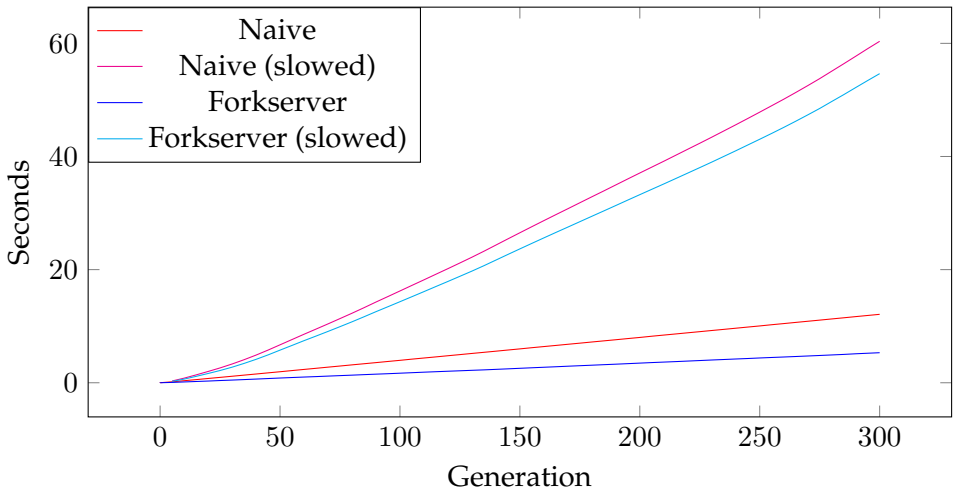


Figure 8: *Speed of Fuzzer with different sample evaluation*

6 Discussion

In this section we describe similar and related techniques as well as possible improvements to Forkever.

6.1 Forkever for exploitation

Forkever was already used in this semesters' *binary exploitation* course. The course covers basic concepts of exploiting software such as buffer overflows, code reuse attacks and printf exploits. The last topic taught is heap exploitation, for which Forkever was initially developed.

As the craft of heap exploitation requires a thorough understanding of the control structures on the heap, Forkevers frontend "Hyx" appears to be rather useful as it concisely displays a lot of memory. To further ease the understanding of the heap, the user can have the program call functions interacting with the heap such as `malloc` or `free`. Paired with the immediately displayed changes in Hyx, this certainly embodies a more natural experience than printing out the memory contents via GDB before and after the execution of an interesting code snippet and comparing the printed results. Further, one can easier spot changes in unapparent locations that she might not have considered initially. By forking the process before triggering an action, it is possible to jump between the prior and resulting state and explore both at need.

Although the students gave positive feedback during the demo sessions, only two of them were able to find some time for participating in a survey.

Forkever has also been used for solving a task of the *TSG CTF 2020*. The writeup[2] features a GIF of a heap displayed in Hyx, cycling through different states, showing how Forkever can be used for demonstration purposes.

6.2 Forkever for fuzzing

As seen in the Evaluation, the Forkfuzzer could only compete with more traditional approaches after heavily slowing down the automatons transitioning.

[2] https://hxp.io/blog/75/TSG-CTF-2020-Karte/

As mentioned in Section 5.2.3, the ForkFuzzer was modified so that it would use `PTRACE_CONT` instead of `PTRACE_SYSCALL` to reduce usage of ptrace.

To apply this technique on a binary, one requires knowledge about the program's consumption pattern to guarantee no data is being left in STDIN and then consumed by a child process it was not intended for. In the case of no slowing function added, the improved version was roughly 17% faster. With the slowing function, it decreased to being about 12% faster.

Fig. 7 depicts how the ForkFuzzer performed better then the naive fuzzer. This hints at that fuzzing using Forkever may possibly be useful if one was to fuzz a binary that is both computation heavy and allows for stepwise consumption of input. Another requirement for the ForkFuzzer requires precise knowledge about the consumption pattern so that the program is forked at the right point of execution. It is safe to say that only a small set of programs satisfy these properties, leading to the conclustion that the ForkFuzzer can only be applied in unique usecases.

As mentioned, the ForkFuzzer crashed due to creating too many processes. A script verifying this to be the cause is in the Appendix. Various tried solutions such as increasing the limit NPROC did surprisingly not resolve this issue. We believe that this boundary can be pushed with further investigation.

Another workaround worth exploring would be to prune the resulting tree of child processes regularly by killing the processes of whom the fewest children are forked.

6.3 Related Work

This section outlines the approaches of work that inspired Forkever and similar tools that aim to assist researchers in analysing a program.

6.3.1 Fuzzing

Fuzzing is a methodology involving the repeated running of a program with varying input to find security-critical bugs such as buffer overflows or crashes. As repeatedly running a program is computationally expensive, researchers hunting for bugs is computationally expensive, researchers hunting for bugs often find their computers running for days or even weeks. Fuzzing is dynamic program analysis and has shown to find many bugs missed by static analysis tools or manual inspection by developers.

Blackbox Blackbox fuzzing is the first and simplest form, no additional knowledge about the program is required. A given binary is supplied with different input and the result is recorded. Inputs leading to an extraordinary case are saved for later inspection by the user. Note that some fuzzers require the user to give a sample of a well-defined input. The name "Fuzzer" originates from the practice of generating inputs off of previous ones by applying small, random changes to them. One of the very first tools employing this technique is the 1990 developed crashme[3] that aimed to test an operating systems robustness: Continously trying to execute randomly generated byte sequences, the fault handling by the kernel would be stressed until failure[13].

[3] https://github.com/gjcarrette/crashme

Grammar-based Grammar-based testing is employed, the "user provides an input grammar specifying the input format of the application under test. Often, the user also specifies what input parts are to be fuzzed and how"[6]. Current examples are dharma[4] or boofuzz[5].

Some fuzzers greatly increase the probability of finding edge cases through symbolic execution, achieving levels of code coverage that are practically impossible for plain blackbox fuzzers. A notable tool implementing this technique, "SAGE", is being referred to by its creator as a whitebox fuzzer, although it performs the symbolic execution on the binary level, allowing it to be used "on any program regardless of its source language or build process"[7].

Whitebox Whitebox fuzzing is the most advanced and most information requiring form of fuzzing. As the name implies, whitebox fuzzers require source code of the program to be fuzzed. The source code is modified to contain functions that will assist in the process of fuzzing the later compiled binary. One of the best-known fuzzers, namely AFL, optionally allows for this, leading it to be referred as a "greybox fuzzer" by some.

There also exist fuzzers that are meant to fuzz in a client-server architecture. See [5] for an example on the game "Teeworlds" where the functions interacting with the socket for communication are hooked to intercept and manipulate the data sent.

6.3.2 AFL's fork server

American Fuzzy Lop is both the name of a type of rabbit and a fuzzer developed by Michał Zalewski. To date, the latter is among the most known fuzzers and adopts a variety of techniques.

To measure code coverage, AFL instruments the program with certain instructions at each branch. These then increment a counter that maps to the specific branch which lets the user later analyse which parts of the code have been executed several times, once or just not at all. If source code is not available the binaries are instrumented directly using a version of QEMUs "user emulation mode" [6] which also allows for executing code designated for a different architecture. The author states that this practice induces, combined with the below mentioned fork server, an overhead ranging from 200% to 500%[19].

One very notable practice AFL uses is its so called fork server: As initialising a program is very time expensive and can make up the biggest part of the fuzzing process if the library is considerably small, one is interested in minimizing this process. By instrumenting the fuzzed program to fork itself after reaching the main function, the loading, linking and library initialization routines are done just once. The originally started process will serve as the server, off which a new process is cloned. After sending the PID of the new child to the monitoring process, the server waits for the created child to finally send the exit code back to the monitoring process. This procedure is then repeated. Note that when AFL instruments the binary using QEMU, the fork server is extended so that a child communicates the translations done by QEMU to the parent[18].

[4] https://github.com/MozillaSecurity/dharma
[5] https://boofuzz.readthedocs.io/en/stable/index.html
[6] https://www.qemu.org

With the usage of injected fork calls, this is quite a similar approach to forkever. Due to AFL avoiding ptrace but instead instrumenting the binary at control flow branches, a fine grained managing of control flow allowing single stepping and therefore forking at any point in time is not possible. On the other hand it is to be expected that the use of ptrace would lead to a significant decrease in performance.

While different ways of evaluating the random samples were examined, the algorithm that yielded the samples was simple and not modified. In this algorithm, however, resides a lot of space for improvement. Researchers are coming up with various different approaches on how to derive input samples from the intermediate fuzzing results, adapting concepts from biology[1] or deep-learning[2].

6.3.3 Villoc

Another useful tool that arguably visualizes the control structures of the heap much better than Forkever is Villoc[7]. It relies on Strace to reconstruct the heap by tracing all calls to libc functions interacting with it. With the heap reconstructed, the allocated chunks are drawn via html. Since Strace only traces a binary's calls to external functions, Villoc cannot consider allocations made by other libraries such as libc itself.

6.3.4 GDB

As mentioned throughout the thesis, Forkever tries to mimic the very basic functionality of the GNU debugger, which was written by Richard Stallman in 1984. Besides featuring breakpoints, singlestepping and manipulation of memory like Forkever, it also allows debugging multiple threads and can match a binary with the source code of many languages such as C, Fortran or Rust.

A particularly interesting feature of GDB is the ability of "reverse-debugging", allowing the user to rewind execution step by step, as precise as instruction by instruction. Unfortunately, this is only supported for a selected range of architectures, namely

- i386-linux
- amd64-linux
- moxie-linux
- arm-linux

This reversing and replaying of process execution works by logging all executed machine instructions of the debugged program "together with each corresponding change in machine state" works by logging the execution of each machine instruction in the child process, together with each corresponding change in machine state (the values of memory and registers). By successively "undoing" each change in machine state, in reverse order, it is possible to revert the state of the program to an arbitrary point earlier in the execution. Then, by "redoing" the changes in the original order, the program state can be moved forward again.[15]

[7] https://github.com/wapiflapi/villoc/

6.4 Future Work

When using Forkever as an interactive hacking tool, a new ProgramInfo instance is created for each forked process, leading to a lot of information being redundantly loaded as the information on mapped binaries is initially equal to the parents information. The loading of a separate ProgramInfo should therefore be shifted to the point where the ProgramInfo of either the process or its parent's would change, which could be achieved by analysing the responsible syscalls.

To further increase performance, the creation of breakpoint objects should be done in a way such that it does not require the reinsertion of `INT3` instructions. This could be achieved by either changing the implementation of breakpoints in the `python-ptrace` library or keeping track of breakpoints inserted at a parent process.

If the analysed program performs a regular fork, e.g. to call another program with `popen`, Forkever does not trace the new process but instead immediately detaches from it for the sake of simplicity, which of course limits the ways in which the program can be analysed. We believe that this problem can be tackled with medium effort thanks to the functionality provided by the ProgramInfo class. Further, Forkever also has no support for multithreaded programs.

Another useful feature for analysis would be the detecting of a certain program state. This state should be definable in a concise syntax. Forkever could monitor if the user-defined condition is met and fork a copy for later use.

For now, Forkever has only been tested on X86 Linux. Some changes had to be made to make it universally UNIX compatible, see the hardcoded accessing of the `origrax` register for example.

Although the usage of ptrace-induced forks is unefficient, we believe that the technique of codemmaping another page of memory for special pre-inserted instructions could improve the performance of fuzzers. With the help of Forkever, a binary could be loaded and reinstrumented at control flow branches to jump to the inserted memory page, allowing for simple yet powerful instrumentation. Overwritten instructions would have to be replaced at the inserted page, of course. After doing arbitrary instrumentations, one could employ the demonstrated Forkserver technique. As a generic binary will not necessarily give out information via STDIN, the server has to call `waitpid` as described in the AFL approach.

6.5 Conclusion

In this thesis we developed a tool for intuitive memory manipulation and manual runtime analysis through the trial-and-error principle. Further, we emulated a fuzzing environment and tested how creating copies of a process at precisely chosen points in execution match up to the traditional approach when fuzzing. We saw that the ForkFuzzer approach can not compete in the generic use case but only when the processing of input happens interval wise and is computationally intensive. Further, we show that the Forkserver approach adopted by AFL leads to noticeable increases in performance even when applied to small binaries.

References

[1] Konstantin Böttinger. "Fuzzing binaries with Lévy flight swarms". In: *EURASIP journal on information security (2016)* 1 (2016), p. 10. URL: http://dx.doi.org/10.1186/s13635-016-0052-1.

[2] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. "Deep Reinforcement Fuzzing". In: *IEEE Symposium on Security and Privacy Workshops, SPW 2018*. San Francisco, CA, USA: IEEE, 2018. URL: https://ieeexplore.ieee.org/document/8424642.

[3] The kernel development community. *Yama*. https://www.kernel.org/doc/html/latest/admin-guide/LSM/Yama.html. visited 14.09.2020.

[4] Intel Corporation. "Intel 64 and IA-32 Architectures Software Developer's Manual Vol 2A". In: 2016, pp. 3–457.

[5] Michal Dardas. *Introducing SocketInjectingFuzzer and its first target - Teeworlds*. https://logicaltrust.net/blog/2020/07/socketfuzzer.html. visited 02.08.2020. 2020.

[6] Patrice Godefroid. *Fuzzing: Hack, Art, and Science*. https://cacm.acm.org/magazines/2020/2/242350-fuzzing/fulltext. visited 02.08.2020. 2020.

[7] Patrice Godefroid. *SAGE: Whitebox Fuzzing for Security Testing*. https://queue.acm.org/detail.cfm?id=2094081. visited 02.08.2020. 2012.

[8] IEEE and The Open Group. *Fork(2)*. https://www.man7.org/linux/man-pages/man2/fork.2.html. visited 7.09.2020.

[9] IEEE and The Open Group. *posix spawn*. https://pubs.opengroup.org/onlinepubs/9699919799/functions/posix_spawn.html. visited 23.07.2020. 2004.

[10] IEEE and The Open Group. *Ptrace(2)*. https://man7.org/linux/man-pages/man2/ptrace.2.html. visited 14.09.2020.

[11] IEEE and The Open Group. *Signal(7)*. https://man7.org/linux/man-pages/man7/signal.7.html. visited 23.07.2020.

[12] T. J. Kilian. "Processes as Files". In: *USENIX Summer Conference*. 1984.

[13] Tavis Ormandy. "An empirical study into the security exposure to host of hostile virtualized environments". In: (Jan. 2007).

[14] David Presotto. *Interprocess Communication in the Ninth Edition Unix System*. 1990.

[15] GNU Project. *Process Record and Replay*. https://sourceware.org/gdb/wiki/ProcessRecord. visited 23.07.2020, further reading: gnu.org/software/gdb/news/reversible.html.

[16] Dennis M. Ritchie. *The Evolution of the Unix Time-sharing System*. https://www.bell-labs.com/usr/dmr/www/hist.html, visited 23.07.2020. 1979.

[17] Hovav Shacham et al. *On the Effectiveness of Address-Space Randomization*.

[18] Michal Zalewski. *Fuzzing random programs without execve()*. https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html. visited 23.07.2020. 2014.

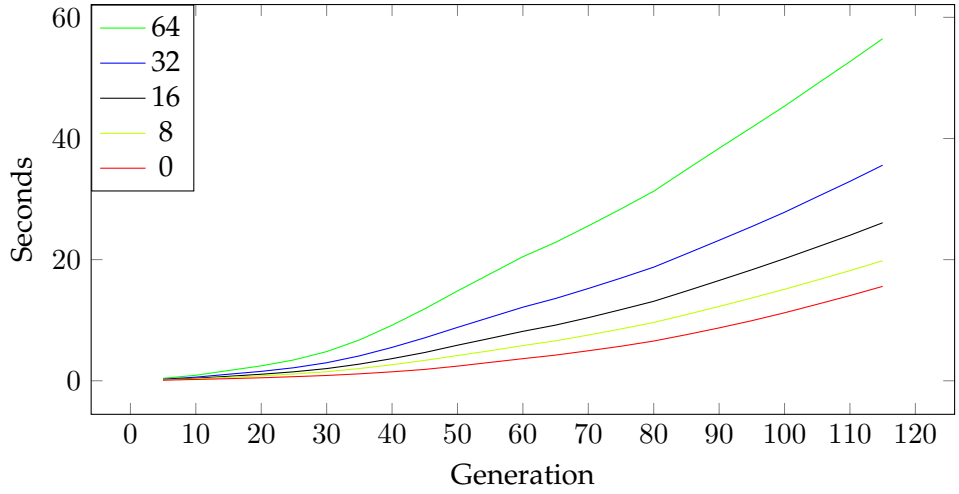
[19] Michal Zalewski. *Technical whitepaper for afl-fuzz*. https://lcamtuf.coredump.cx/afl/technical_details.txt. visited 23.07.2020.

Appendix

Supporting material that is too long for the main matter goes here.

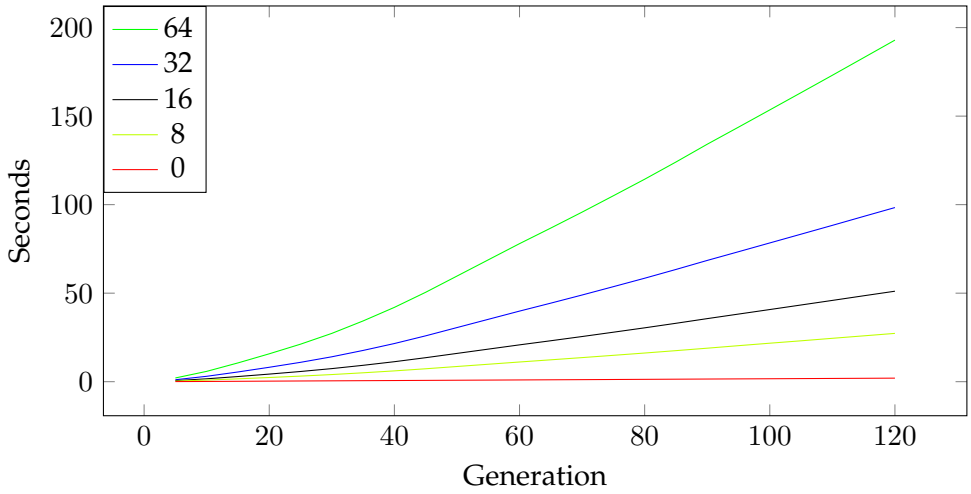
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define ALP_START 'a'
#define ALP_SIZE 6
#define STATES_SIZE 10
#define MAX_CHR (ALP_START + ALP_SIZE - 1)

int trans[][7]={
    {0,0,0,0,0,0,0,},
    {2,2,2,0,0,0,0,},
    {1,0,0,2,3,0,0,},
    {4,6,0,0,0,0,0,},
    {0,5,0,0,0,0,0,},
    {0,0,0,0,6,0,0,},
    {0,0,7,7,0,0,0,},
    {1,1,0,0,0,8,0,}

};

int trans_counter[STATES_SIZE * ALP_SIZE];
int count_edges()
{
    int result = 0;
    for (int i = 0; i < sizeof(trans_counter); i++) {
        if (trans_counter[i])
            result++;
    }
    return result;
}

#define BUSY_COUNT 0x50000
void busy()
{
    for (volatile int i = 0; i < BUSY_COUNT; i++) {};
}

int is_in_alphabet(char input_char)
{
    return (input_char >= ALP_START && input_char <= MAX_CHR);
}

int main()
{
    char input_char;
    int current_state = 1;

    while (1) {
        busy();
        if (1 > read(STDIN_FILENO, &input_char, 1))
            break;

        if (is_in_alphabet(input_char)) {
            int prev_state = current_state;
            current_state = trans[current_state][input_char - ALP_START];

            if (current_state) {
                trans_counter[prev_state * ALP_SIZE + (input_char - ALP_START)]++;
            }
        } else {
            break;
        }
    }

    printf(",%d.", count_edges());
    exit(0);
}
```

Figure 11: *The program implementing the DFA*



Legend denotes BUSYCOUNT in multiples of 0x10000

Figure 9: *Performance of ForkFuzzer with increasing BUSYCOUNT*



Legend denotes BUSYCOUNT in multiples of 0x10000

Figure 10: *Performance of Forkserver with increasing BUSYCOUNT*