PtraceObfuscator: Using the Linux Debug Interface for Hiding the Control Flow of x86-64 ELF Binaries

Ben Swierzy

***Abstract*—Reverse engineering aims at analyzing software to understand its functionality and behavior. Obfuscation techniques try to increase the complexity and resources required for successful reverse engineering by applying transformations to a program without modifying the behavior from the user's point of view. In this paper we introduce a novel obfuscation technique, which removes control flow instructions from a binary and restores the original behavior by using the debug interface of Linux. Furthermore, we conduct a case study to analyze the obfuscation strength as well as the effects on the resource consumption of the binary. The results show, that a reverse engineer needs to fully undo the obfuscation to perform meaningful analysis. However, this comes at the cost of an increased runtime by factors of several thousands.**

I. Introduction

With the increasing importance of IT security in the last decades, various essential methods in the field such as reverse engineering are also gaining more and more relevance. Software reverse engineering describes the process of deconstructing and analyzing a piece of software with the goal of understanding its components and functionality. This is often done without having additional documentation such as a manual or the source code beforehand [1]. In the security context, reverse engineering is often used for two tasks: On the one hand, vulnerability research tries to find exploitable vulnerabilities in existing programs, so they can be patched in the future. On the other hand malware analysts examine malicious software, trying to clarify its functionality and the effect on infected systems. This information can be used in further steps to develop countermeasures.

However, reverse engineering can also be used by malicious actors, e.g., trying to circumvent the license checking process of a given software. As countermeasures so-called obfuscation techniques can be used. Their task is to ideally transform a program into a virtual black box, which allows the user to execute the program without being able to look into it. As this property is not achievable in practice [2], meaning that every software can be reverse engineered given enough time, obfuscation tries to increase the resources required for reverse engineering so that is becomes infeasible to perform.

Obfuscation techniques can work on different levels, e.g., on a source code level for interpreted languages or on an instruction level for general executables. In this paper we present the PtraceObfuscator, an obfuscation technique for x86-64 Linux binaries, which uses the process tracing interface `ptrace` of the Linux kernel. While this is normally used by debuggers to get an insight into a running process, it can also be used for obfuscation.

As disassemblers rely on certain instructions to track the control flow, we can replace these instructions with breakpoints handled by an external runtime process. When these are reached during execution, the runtime is able to perform the original instruction itself to restore the original control flow.

In section II we outline the underlying foundations the PtraceObfuscator has been built upon. Next, we describe the logic behind the obfuscation technique in detail (section III), and conduct a case study in section IV. Finally, section V discusses the results of the case study with respect to the obfuscation strength and the practical use.

II. Preliminaries

This section introduces basic information required for understanding how the presented obfuscation technique is working.

A. x86-64

Microprocessors represent implementations of a so-called instruction set architecture (ISA). The ISA is essentially a specification of instructions, registers and many more aspects the processor needs to fulfill. As a broad classification, ISAs can be divided into RISC (reduced instruction set computer) and CISC (complex instruction set computer) architectures. While RISC architectures have only few and simple instructions trying to allow the construction of simple and efficient processors, CISC architectures move the complexity from software into hardware by providing more powerful and complex instructions. [3]

There have been numerous ISAs in the history of microprocessors, but only a few candidates are found in the majority of devices. On the RISC side, ARM is the most popular ISA used especially in System-on-Chips (SoCs) found in smartphones. While there are some further candidates, that still play a role in modern RISC architectures like MIPS or RISC-V, the CISC world is dominated by the x86 architecture, which was introduced by Intel in 1978 with a register size of 16 bit. Since then, many extensions and updates have been introduced to the architecture including increments to the register size over 32 bits to the current 64 bits. The latest iteration is also called x86-64. [4]

Microprocessor instructions can usually be split into three components: An (optional) prefix, an opcode and a list of operands. All components of the instructions defined in x86-64 have variable sizes. The lengths range from 1 byte to a maximum of 15 bytes. The ISA specifies many general-purpose

registers and some special-purpose registers such as the instruction pointer `rip` holding the address of the instruction, that should be executed next, or the stack pointer `rsp` used for the implementation of a stack in hardware. Furthermore, there exist possibilities to only address the lower 32/16/8 bits of these registers in instructions. For memory access a complex syntax consisting of five components has been realized: To the base address inside a register, a displacement value is added. Additionally the index value taken from another register is multiplied with a constant scale and added to the first value. As a last step, the highest-valued bits of the address can be determined from special segment registers, which however is not relevant for x86-64 anymore. [4]

B. ELF

The executable and linkable format (ELF) is a common format for executable files on Unix-based operating systems. Every ELF file starts with a header containing basic information required for execution, such as the ISA of the program code. Furthermore, it contains a program header table specifying multiple segments of the program. Each segment can contain multiple sections, which have specific tasks. The most prominent section is the `.text` section, which contains the program instructions. [5]

In Linux, there are multiple steps taken when executing an ELF file. First, the segments are mapped to the memory according the program header table. If the program is dynamically linked, the required external libraries are mapped to the memory by the loader. Afterwards, the execution starts at the offset specified by the entrypoint field in the ELF header.

C. ptrace

Linux and other Unix-based operating systems offer an interface for process tracing called `ptrace`. It allows a tracer process to attach to another tracee process to observe and manipulate it. Primary use cases are debugging using breakpoints and system call tracing. Programs can access the interface via the `ptrace` system call. [6]

Often, the usage of the interface starts with a tracer process forking to create the tracee as its child. However, before the control is handed over using an `execve` system call, the child can indicate, that is wants to be traced. This way, the execution of the tracee can be suspended as soon as the new program has been loaded into memory. The main features used for process tracing are

- reading and writing of registers,
- reading and writing of memory,
- getting informed about signals the tracee receives and
- continuing the execution of the program. [6]

The interface can be used effectively in connection with hardware breakpoints offered by the x86-64 architecture: The one-byte interrupt instruction with the hexadecimal opcode `cc` and mnemonic `int 3` informs the kernel to issue a `SIGTRAP` signal for the process. Additionally, the tracer will be informed about the signal, and is able to handle the breakpoint accordingly. Afterwards, the execution can be resumed by using another `ptrace` system kernel.

While it can be possible for any process to attach to another process even without it asking to be traced, the kernels of many Linux distributions are compiled with a security module, allowing only a parent to attach to its child (if the current user is unprivileged). It needs to be taken into consideration, that the security settings can also completely disable the possibility of process tracing. [6]

III. Methodology

The PtraceObfuscator can be split in three logical components: First of all, the obfuscating component takes an ELF file, removes a certain set of instructions and conserves them as external metadata. Next, a runtime component needs to start the execution of the ELF file and use the `ptrace` interface to restore the original execution on-the-fly. Finally, a packing component can bundle the runtime with the additional data to output a single, obfuscated ELF file.

All the software components have been implemented in Go. As a statically-typed, compiled language, Go offers good performance, while the integration of additional features like garbage collection and an extensive standard library speed up the development process. Furthermore, system interfaces can be accessed easily through the `syscall` package [7].

In addition to the standard library, two further libraries are used by the PtraceObfuscator: On the one hand, the `x86asm` package (which is also used by the official Go toolchain) implements decoding of x86 and x86-64 instructions [8]. Though, since the decoding of few instructions has shown to be problematic for the library, some workarounds needed to be implemented (see section III-A). On the other hand, a `ptrace` library published by Ethan Burns optimizes the Linux `ptrace` system call interface to be more elegant to use in Go [9]. Since it does not provide access to all features of the process tracing interface, we extend the library with some functions for reading and writing of registers and memory.

A. Obfuscator

The Obfuscator is the first element in the component chain of the PtraceObfuscator. The input for this component consists of an existing ELF file.

First, the Obfuscator reads the ELF header to locate and extract the contents of the `.text` section. Next, a linear disassembler decodes the instruction one-by-one. If the opcode of an instruction is considered obfuscatable, i.e., it is either a jump, a conditional jump or a call instruction, it remembers the respective offset and instructions as metadata.

Since x86-64 instructions have variable length, an undecodable instruction is problematic for the disassembler, as it does not know the exact offset where it should continue. Thus, in case the `x86asm` library returns an error, the Obfuscator cannot continue disassembling and only considers all instructions found up to this point. However, this proved to be problematic, as the library was not able to decode the bytes `f3 0f 1e fa` which correspond to the mnemonic

endbr64. This instruction (frequently used, e.g., by the GNU Compiler gcc) is a relatively recent addition to the ISA and a security feature against ROP attacks [10]. The Intel x86-64 reference states, that this instruction will be decoded to a nop (no operation) on older processors, however, this is clearly not the case for the library. Thus, we hardcoded the byte sequence to be ignored by the disassembler, which proved to be a successful workaround. There is one similar issue, where the library is not returning an error, but instead just decoding an instruction wrongly: The instruction e9 7e 59 ff ff, which corresponds to a relative jump, is decoded as a prefix with value e9 and no opcode, if the buffer with the program instructions ends afterwards. Since this issue only occurs at the end of the text section, it does not have a considerable impact on the obfuscation process and the PtraceObfuscator can simply stop the disassembler at this point.

Now, the Obfuscator can create a copy of the original ELF file and modify the .text section in the following way: Depending on the user's wish, all offsets in the metadata are either replaced with nop or with random bytes. Since this operation, does not alter the length of the text section, we can copy the other parts of the ELF file without further modifications. At the end, the Obfuscator outputs the new ELF file and the metadata serialized to JSON.

B. Runtime

The Runtime component is responsible for tracing the process of the obfuscated ELF file and restoring the program flow as needed. The required input matches the outputs of the Obfuscator.

After deserializing the metadata, the Runtime parses the ELF header to fetch the offset of the .text section in the corresponding segment, as it is required for matching the positions from the metadata correctly. Next, the child process is forked, inherits the arguments, the file descriptors and the environment, and directly indicates, that it wants to be traced by its parent, as described in subsection II-C. Thus, the newly created tracee is directly suspended as soon as it is loaded into memory. This is important, since the Runtime needs to perform two tasks before the execution can begin.

Most importantly, it needs to get access to the segment address, which contains the .text section. However, this is not possible by using the process tracing interface. At most, it is able to access the instruction pointer, which does not yet point to the entrypoint, but to a memory address in the loader library. Furthermore, we cannot make any assumptions on the address, if address space layout randomization (ASLR) is in use. Fortunately, Linux provides the virtual /proc filesystem, which contains a variety of information on a running process, including its memory layout [11]. Thus, the Runtime can parse the contents of the memory map from /proc/<PID>/maps and searches for the first segment marked as executable. While this segment is not necessarily required to contain the .text section, it is the case for all ELF files that we observed in practice. The computed segment base address can then directly be used to perform a second task before the execution starts, namely setting hardware breakpoints. For this, the Runtime alters the memory at every offset contained in the metadata by writing the corresponding one-byte instruction to the tracee's memory. Afterwards, the execution of the child can be started.

TABLE I
THE LIST OF CONDITIONS TO CHECK FOR EACH CONDITIONAL JUMP. FOR SOME INSTRUCTIONS THERE EXIST MULTIPLE MNEMONICS, WHICH HOWEVER CORRESPOND TO THE SAME OPCODE.

Instruction Mnemonic	Condition
JO	$OF = 1$
JNO	$OF = 0$
JP	$PF = 1$
JNP	$PF = 0$
JS	$SF = 1$
JNS	$SF = 0$
JZ, JE	$ZF = 1$
JNZ, JNE	$ZF = 0$
JC, JB, JNAE	$CF = 1$
JNC, JNB, JAE	$CF = 0$
JBE, JNA	$CF = 1 \vee ZF = 1$
JA, JNBE	$CF = 1 \wedge ZF = 1$
JL, JNGE	$SF \neq OF$
JGE, JNL	$SF = OF$
JLE, JNG	$ZF = 1 \vee SF \neq OF$
JG, JNLE	$ZF = 0 \wedge SF = OF$
JRCXZ	$RCX = 0$
JECXZ	$ECX = 0$
JCXZ	$CX = 0$

During this phase, the Runtime waits on the event, that the tracee has been suspended due to a SIGTRAP signal, which means that a breakpoint has been reached. Then, it can search the metadata for the offset corresponding to the current instruction pointer. Next, an action depending on the original instruction is taken:

Unconditional jumps manipulate the current instruction pointer based on the argument. However, we need to take care, that relative operands are added to the instruction pointer, when it points to the instruction after the unconditional jump. However, currently the instruction pointer points to the instruction following the breakpoint. As unconditional jumps usually have a length greater than one byte, we need to compensate this by adding the instruction length minus one before evaluating the argument. [12]

Conditional jumps behave similar to unconditional jumps, except that the jump is only performed, if a certain condition is fulfilled. There are 19 different conditions listed in the x86-64 reference. The mapping of instructions to their condition can be found in table I. Most of the conditions consider the rflags register, which is set for every calculation performed by the processor. While it contains many more flags, 5 are evaluated for the conditional jumps, i.e., the carry flag (CF), the overflow flag (OF), the sign flag (SF), the zero flag (ZF) and the parity flag (PF). Furthermore there are 3 conditions, which check if the (lower) 64/32/16 bits of the general-purpose register rcx are zero. If a condition is fulfilled, the instruction can be handled exactly like an unconditional jump. [12]

Function calls are an extension of unconditional jumps. While the program flow is impacted in the exact same way, an additional action is taken: The processor pushes the instruction pointer (which points to the instruction after the call) onto the stack, i.e., decreases the stack pointer by 8 and writes the 8 byte address to the memory the `rsp` points to. Thus, for computation of the return address the same aspects as for the computation of the jump/call targets need to be considered. [12]

All the obfuscated instructions support relative, register and memory operands available in the x86-64 ISA [12]. Only immediate operands are not available. The handling of register and relative operands requires only few steps: In the first case, we read the value of the operand register and write it to the instruction pointer. In the second case, we add the relative offset to the instruction pointer. Memory operands, however, are more complex: The Runtime first needs to read the base register, add the displacement, read the value of the index register and multiply it by the scale to compute the complete address. Then, it reads 8 bytes from the memory address to get the new value of the instruction pointer. In case the memory address is not valid, the execution is terminated with an error message.

C. Packer

While the Obfuscator with the Runtime in combination already implement the main functionality of the PtraceObfuscator, we would still need three files (Runtime, obfuscated ELF, and metadata) for running the program. The Packer component solves this issue by taking the three files and generating a single output file.

Since the contents of the files need to be compiled into the final binary, the Packer generates a Go source file for the obfuscated ELF and the metadata. Compilation of the Runtime will then include the corresponding buffers in the output. However, this required an adjustment in the Runtime, as it previously expected the obfuscated ELF to be available in the filesystem. To circumvent this, an in-memory file is created using the `memfd_create` system call, which is available in Linux kernels since 3.17 [13]. The contents of the file are then available through the process filesystem under `/proc/self/fd/<FD>`.

As an optional addition, the Packer also calls the tool `strip` from the GNU binutils [14] on the obfuscated ELF as well as the packed binary to increase the obfuscation strength by removing unnecessary (debug) information.

IV. Case Study

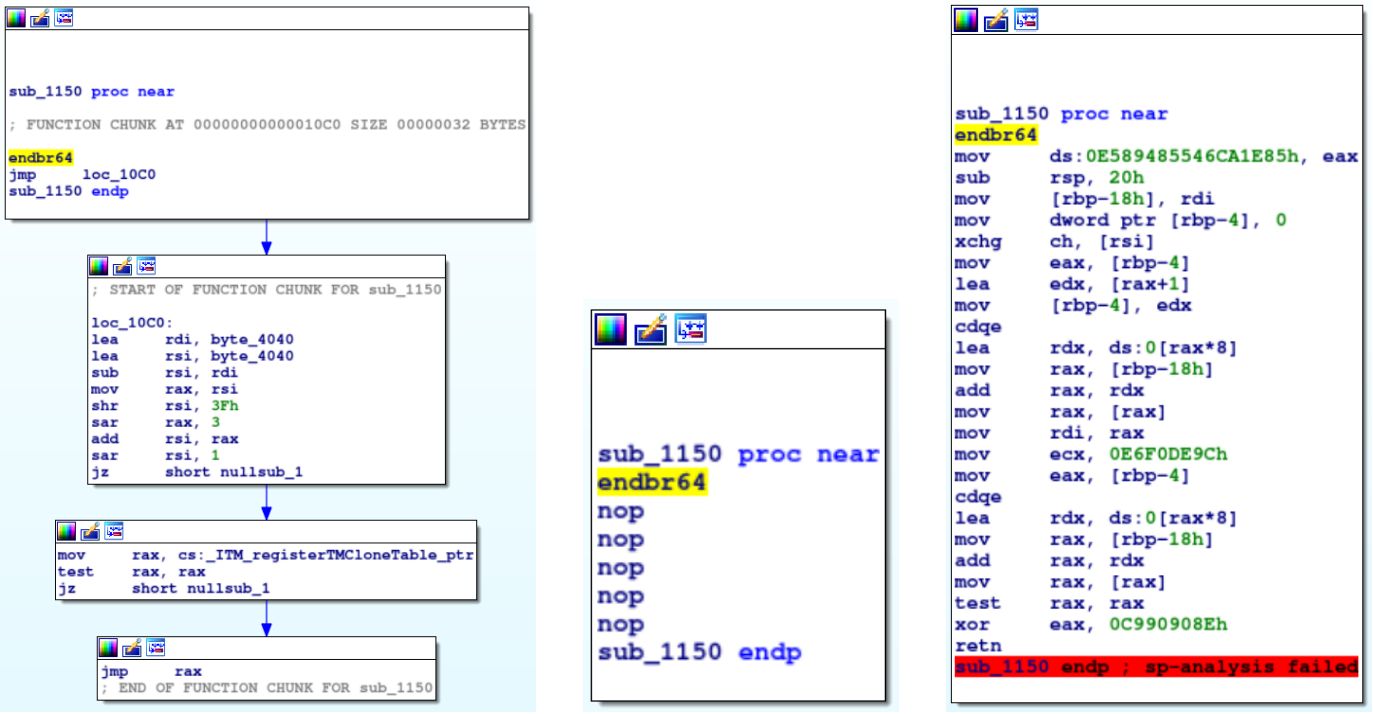
In the following section, we will analyze different properties of the PtraceObfuscator. First, we will look at the effectiveness against static and dynamic analysis. Then, we will consider side factors such as resource consumption, which are relevant when using the PtraceObfuscator in practice.

A. Resistance Against Static Analysis

Static Analysis is a reverse engineering technique, that focuses on features and instructions of a binary without executing it. In the following subsection, we will examine the effects of the PtraceObfuscator on static analysis methods by looking at a small example program which contains of a function, a loop and some library calls. The source code can be found in the PtraceObfuscator repository and has been compiled with the GNU Compiler Collection's C compiler version 10.2 [15].

There is a variety of methods which is applied by reverse engineers. When reverse engineers try to get an overview over the program, one of the first technique used is the listing of strings [16], e.g., using the tool `strings` from the GNU binutils [14]. This tool searches for all valid ASCII strings in the binary. Only strings with a certain length a printed to prevent too many false positive as binary data can also accidentally contain matches. In the minimal working example, there are more than 30,000 strings with the default minimum-length of 4 characters found. Even the harsh increase to a minimum of 10 characters yields around 5,000 matches. The reason for this is the use of Go: By default the compiler links the binary statically, i.e., integrates all libraries into it. This increases the portability of the binary at the cost of a larger size. However, from the obfuscation side this is not necessarily a disadvantage, as the reverse engineer has to look at more code and separate (often not too interesting) library code from the main code in the binary. In total, the majority of strings are package and function names from Go. However, one string is particular relevant: As the metadata is encoded in JSON, it is also contained in the list. An interesting, but harder-to-find string is the beginning of the obfuscated ELF, which can be identified by the letters `ELF` contained in the magic bytes of an ELF file and thus is also contained in the output for a minimum-length of 3.

For the following aspects, we will consider the case, when the reverse engineer has identified the integrated obfuscated ELF in the packed program. Then, they would probably isolate the program into a separate file to be able to use advanced static analysis tools such as IDA Pro [17] or Ghidra [18]. Figure 1 shows how IDA is able to visualize the control flow in the original ELF and the obfuscated variants. We consider the subroutine `sub_1150` as it is the only subroutine except for the process termination found by IDA for the random bytes chosen during obfuscation of our example. In the original ELF (fig. 1a) IDA is able to follow jumps and visualize a section of the control flow. When `nop`s are used for replacement (fig. 1b), all control flow information is lost and only one small code block is found. In the third case (fig. 1c) where random data is used as replacement, IDA is again not able to follow the control flow. Furthermore, it disassembles lots of instructions, which do not occur in the original binary. Additionally, we can compare the amount of subroutines discovered in the `.text` section. These amount to (a) 10, (b) 6 and (c) 2 detected subroutines, when the binary is (a) only stripped, (b) obfuscated with `nop` instructions and (c) obfuscated with random data. In the last case, IDA does not even offer the option to open the graph view on entrypoint. However, the exact result is clearly depending on the concrete random



(a) Unobfuscated Binary

(b) Using `nop` instructions as replacement

(c) Using random bytes as replacement

Fig. 1. The graph view in the disassembler IDA visualizes how the operation modes of the PtraceObfuscator increase the strength against static analysis. For all examples the subroutine `sub_1150` has been disassembled.

values chosen. Thus, static control flow analysis is not possible without applying suitable deobfuscation beforehand.

There are also static analysis techniques, which are not necessarily impacted by the PtraceObfuscator. An example is the consideration of API calls. They are still identified by the corresponding `syscall` instruction and preceding setting of register values. Although in some special cases, the obfuscation can also impact these by some amount. On the one hand, if the return value of a function is used as parameter for a system call, the reverse engineer is not able to see what function has been referenced. On the other hand, the replacement of jumps and calls with random data might break the disassembler, and thus might hide the `syscall` instruction.

B. Resistance Against Dynamic Analysis

Previous work has shown, that dynamic analysis is an important tool for reverse engineers to help understanding the behavior of a given program [16]. Dynamic analysis is often performed in debuggers (such as the GNU debugger `gdb` [19]), which also use the process tracing interface. Thus, they can not only set simple breakpoints as the PtraceObfuscator does, but instead have a variety of tools to assist the user.

However, using a debugger on obfuscated binaries as usual proves to be difficult. At first, a reverse engineer would load the packed binary into the debugger. While this is possible without complications, the reverse engineer is only able to examine the runtime, but not the obfuscated ELF itself as it is started in a child process. In this case, `gdb` will by default keep tracing the parent. However, it also offers different options. For instance, the user could configure it to stop tracing the parent, and switch to the child instead. However, this causes an error in the runtime, as it is not able to load the obfuscated binary, because it is not permitted to trace an already traced program. An alternative approach would be to start the runtime normally, and attach to the PID of the child during runtime. On most Linux distributions this requires a privileged user or a specific configuration of the Linux security module Yama [6]. But again, this method fails, since `gdb` cannot attach to the child, as it is already traced by the runtime. Thus, dynamic analysis is not possible on binaries created with the PtraceObfuscator without applying deobfuscation techniques beforehand.

C. Resource Consumption

While the available computing resources continue to grow, they are still an important aspect to consider, especially, since there exist orders of magnitudes between the resources of different device types, e.g., between a desktop computer and an embedded device. In general, the three major types of resources need to be considered: file size, memory consumption and time consumption. Measuring the file size is trivial. However, measuring the amount of memory consumed is not

TABLE II
COMPARISON OF THE RESOURCE CONSUMPTION BETWEEN OBFUSCATED AND UNOBFUSCATED BENCHMARK PROGRAMS.

Command	Original Binary		Obfuscated Binary	
	File Size	Real Time	File Size	Real Time
`factor 97543911563533004446771 9`	84 KB	0.006 s	2,364 KB	198.040 s
`tar -cf out.tar PtraceObfuscator`	530 KB	0.005 s	3,565 KB	8.858 s
`du -hs ~/Downloads`	174 KB	0.009 s	2,630 KB	38.752 s

as easy, as most tools only consider the amount of virtual memory allocated and not of physical memory used. Thus, we will consider memory consumption only theoretically in section V. Similarly, one needs to be careful, when measuring the execution time. Generally, there are 3 types of execution times: real time (or wall time), user time and system time. While the first one measures how long the user had to wait for the program to terminate, the others measure the CPU time of the process and the kernel, respectively. In our measurements, we will only compare the values for real time returned by the `times` system call with millisecond precision.

We measure the resource consumption in the remaining two categories for different benchmark command-line tools, which are installed on most Linux distributions by default.

1) `factor` is a tool from the GNU coreutils [20], which factors arbitrary large integers using Pollard's ρ-method. It is build upon the GNU Multiple Precision Arithmetic library [21]. We choose the input as a product of two primes, which allows us to adjust the runtime of the program by adjusting the sizes of the prime factors. However, since the underlying factoring algorithm is randomized, we need to take the average runtime of multiple runs as measurement. The speed of this program is heavily relying on the speed of the processor.
2) `tar` is a common archiving tool, which packs multiple files (possibly in a directory hierarchy) into a single file [22]. It is often used in conjunction with compressing tools (e.g. `gzip`). As a benchmark, we use `tar` on its own on the freshly cloned repository of the PtraceObfuscator. The resulting execution should use a mixture of I/O operations and computations.
3) `du` is a tool for estimating disk usage and part of the GNU coreutils [20]. For the benchmark, we call it on a download folder, which contains 3 GB of data consisting of few big and hundreds of smaller files. Thus, the speed of this program mainly relies on the performance of I/O operations.

The results of the measurements are found in table II. It can be observed that, while the original file sizes are different by factors of up to 6, this is not the case for the obfuscated binaries anymore. Thus, the main increase in the file sizes of the binaries is caused by a constant, additive component.

In contrast, the increase in runtime differs largely between the benchmark programs. The time needed for the computation intensive factoring is clearly increasing the most by a factor of about 33,000. The more I/O dependent benchmarks have an increased runtime of around 1,800 (`tar`) and 4,300 (`du`) times the original runtime. In total, we can observe a significant increase in runtime due to the obfuscation, which is smaller, if I/O operations are used.

V. DISCUSSION

In this section, we discuss the results of the case study.

A. Obfuscation Strength

Obfuscation strength can be considered from two different points of view. On the one hand, one can analyze the strength with respect to a reverse engineer, who encounters this technique for the first time. In this case, we will assume, that they are in a situation in which they have expertise in their field, but do not know any details on the obfuscation technique in use. On the other hand, there is a reverse engineer, which has multiple past experiences with the technique under consideration. Due to this, they do not only know most details, but also have the possibility to develop tools for deobfuscation.

First, we will consider a reverse engineer without detailed knowledge. Following the usual workflow [16], they start with an overview on the given program. Section IV-A already outlines, that static analysis provides several interesting results for the reverse engineer, e.g., system calls as well as a lot of strings. Especially the metadata string is an important find for the reverse engineer. However, we propose a more efficient storage format in section V-B, which would prevent the reverse engineer from finding it this early. Although, the first static analysis might be promising, the initial dynamic analysis certainly is not. The case study shows, that debuggers are able to attach to the runtime and recognize the forked child process. However, attaching to the child process is not possible.

As the workflow progresses, the reverse engineer will start the phase of subcomponent scanning [16]. First, the reverse engineer will focus on the runtime binary, since it is more visible, though this will be a tedious task, since it also contains a large amount of statically linked Go libraries needing to separated from the custom code. Assuming he will find the integrated obfuscated binary at some point in this process, he will need to perform subcomponent scanning again. However, this will prove to be difficult as all control flow information have been replaced with random data leading to wrong disassembly. With time, the reverse engineer will nonetheless be able to formulate some hypotheses about the runtime and possibly the obfuscated binary.

During the focused experimentation phase, the reverse engineer will use dynamic analysis and the help of a debugger

to validate, reject or refine his hypotheses [16]. This will work as usual on the runtime binary, but not on the obfuscated ELF as the debugger is not able to attach, which implies, that none of the techniques of this phase can be used to progress on the hypothesis. Thus, the reverse engineer will first need to fully analyze and understand the runtime before he can reassemble the original binary. Once this is done, the binary has been deobfuscated successfully and the PtraceObfuscator will not provide further protection. Thus, we can conclude, that a reverse engineer without prior knowledge of this technique will need to invest a significant amount of resources into understanding the runtime to be able to undo the obfuscation.

However, this looks different when we consider a reverse engineer with detailed knowledge on the PtraceObfuscator. In this case, they would first need to identify the obfuscation technique. This seems relatively easy, as the PtraceObfuscator leaves a unique footprint in form of a forked child process, which gets traced by the main process. Then, they only need to locate the metadata as well as the obfuscated binary to be able to restore the original program. Thus, the PtraceObfuscator does not protect well against reverse engineers knowing the obfuscation scheme. To some degree, we can work around this issue by applying other, complimentary obfuscation techniques to the binary. While there are some exceptions outlined in section V-C, this should work for most x86-64 ELF obfuscation techniques.

B. Resource Consumption

While resource consumption is not the most important aspect for an obfuscation technique, it is still a considerable factor concerning the practical use. In the best case, a technique does not have any effect on the resources used by the obfuscated program. Very few techniques are even able to optimize the resource usage, e.g., stripping a binary decreases the file size and does not affect performance. This leads to the use of obfuscation techniques in situations, where obfuscation is certainly not necessary. As an example, the open source command-line tools on most Linux distributions are delivered in a stripped format to minimize the required storage space.

Considering the file size, the case study shows, that the PtraceObfuscator significantly increases the size of the binaries. This is the result of three components:

1) the compiled size of the runtime,
2) the size of the obfuscated binary and
3) the size of the metadata.

In the case study, the compiled size of the runtime has been most impactful. As it is written in Go, the size of this component gets comparably large due to the statically linked libraries. However, it does not change for large binaries and is an additive component, with decreasing impact as the original binary grows in size. The size of the obfuscated binary matches exactly the size of the original one, as we only modify bytes in-place. The additional stripping might even slightly decrease the size. The amount of metadata scales linearly with the amount of obfuscated instructions, which often correlates with the size of the original binary. In the current implementation, the obfuscation of a 3 byte instruction leads to over 30 bytes of metadata. This is caused by using JSON as an easy-to-use, but inefficient storage format. A custom, optimized storage format would be able to use approximately 5 bytes (or slightly more, depending on the offset) for a 3 byte instruction. Furthermore, an implementation of the runtime in a language like C should be able to limit the total increase in size to a factor of 2 in most cases.

While the file size is certainly considerable, the increase in runtime is definitely more relevant. Although the nature of this obfuscation technique already implies, that the performance will get worse, factors of several thousands might seem a bit surprising. There are different factors contributing to this, for which we need to look at what exactly happens internally. Thus, consider an obfuscated instruction, which has been replaced by a breakpoint. As soon as it is reached by the control flow, the CPU triggers an interrupt, i.e., the execution is suspended, and the interrupt handler registered by the kernel will be loaded. After recognizing it as a hardware breakpoint, the kernel sends a signal to the attached process tracer, i.e., the runtime. Then, the runtime is able to recognize the situation, and first requests the set of registers from the kernel. Next, the offset of the current breakpoint is calculated and used to search the already decoded instruction in a hashmap. For memory operands and `call` instructions the runtime needs to not only update the registers, but also read (or write, respectively) the process' memory. Finally, it indicates the kernel to resume the execution of the child. As this procedure occurs for every jump in the program flow, there is a significant overhead especially for execution-wise recurring code sections such as loop bodies. Additionally, important processor features such as pipelining (i.e., the parallel execution of instructions), which are heavily used in modern CPUs [4], cannot accelerate the execution at all, since the pipeline needs to be flushed every time the execution is suspended. With this reasoning, we conclude, that the overhead in performance is growing linearly with the amount of breakpoints triggered in total.

As this heavily limits the practical use of the PtraceObfuscator, we will now consider possible performance optimizations. There are two categories of adjustments to decrease the execution time. On the one hand, we can improve the performance of the breakpoint handling. On the other hand, we can decrease the amount of breakpoints triggered. Adjustments from the first category are able to increase the overall performance without harming the obfuscation strength. An early implementation of the PtraceObfuscator used linear search to find the offset corresponding to the current instruction for simplicity reasons. Adjusting the data structure to a hashmap decreased the running time by 20%. However, the improvements in this area are limited. As the described hashmap is the only data structure in use, and the remaining code consists only of simple condition checking, there are not too many more optimizations possible. The only major improvement still left from our point of view, is the reimplementation of the runtime in a language such as C, where the programmer has more possibilities for manual optimizations. Adjustments from the second category allow us

to certainly speed up the code, at the cost of possibly reducing the obfuscation strength. There are multiple ways to achieve this:

1) We can modify the Obfuscator to only obfuscate a fraction of all jumps and calls, which we randomly choose. This might drastically increase the performance, if the right instructions are left unobfuscated, but also might have no effect, if only instructions are not obfuscated, which are rarely executed in practice. The static obfuscation strength is reduced by a non-negligible amount as disassemblers have more control flow information available and thus, should be able to identify more subroutines. However, this again depends on the randomness.
2) As an improvement, we can analyze how often each breakpoint occurs in a specific execution of the program. Then, we can exclude the instructions from obfuscation, which are executed the most. While this technique is able to specifically target the bottlenecks of the program, this might only hold for the specific control flow observed in a single run. In general, this might not represent the execution of the program with different inputs and requires manual selection or an automated decision process. However, for certain programs this approach can yield good results.
3) If we do not want to reduce the obfuscation strength against static analysis, we can adjust the dynamic behavior of the program. For this, we would choose a fixed n, which represents the number of times a breakpoint may be called at maximum. If this threshold is passed, we will replace the breakpoint with the original instruction. Thus, the program is partially reassembled during runtime, and only at performance relevant instructions. A reverse engineer trying to utilize this and read the instructions from memory, would need to detach the runtime from the child and attach his own debugger in time, before the child will get terminated.

C. Limitations

While the case study of the PtraceObfuscator shows a very promising obfuscation strength, there are also some limitations. First of all, the current implementation can only obfuscate programs using a single process and a single thread. The reason is a limitation in the process tracing interface as every thread and process needs its own tracer [6]. While this can be implemented by using special events triggered on `fork`, `vfork` and `clone` system calls, it requires complex logic for analyzing the parameters of the system call, whether a thread or a process has been created.

Furthermore, the PtraceObfuscator requires, that the `.text` section of the binary can be successfully disassembled by a linear disassembler, which is the case for most binaries produced by standard compilers. While a recursive disassembler has been implemented for comparison, we quickly realized, that this approach is not suited for the PtraceObfuscator as they require some degree of manual interaction. As an example for this requirement, we can consider binaries produced by `gcc`. These do not start with the `main` function, but with a small code block calling the function `__libc_start_main`, which takes a pointer to `main` as one of its arguments. However, a recursive disassembler cannot assume, that an address loaded into a register points to valid code, and thus already terminates after disassembling the first few instructions.

Jumps, conditional jumps and calls are the most common x86-64 instructions, which affect the instruction pointer. Additionally, there are more complex instructions which also modify the instruction pointer. An example would be the `loop` instruction, which decrements the register `rcx`, and then jumps if the zero flag is set. An improved version of the PtraceObfuscator could also obfuscate these advanced instructions. However, these do not provide performance advantages as they are split internally into multiple so-called microinstructions [4], most compilers do not use them. Thus, this would mainly increase the obfuscation strength on code not produced by compilers.

An essential condition for the produced binary to work is that the target system needs to have the process tracing interface enabled. Furthermore, the corresponding security setting may not be set to the most restrictive value. While this will probably not be the case on most Linux-based systems, this might be the case in highly-secured environments.

Last, the current PtraceObfuscator is only able to obfuscate binaries for the x86-64 architecture. Because of the similarities it is relatively simple to adjust the functionality for the 32-bit x86 ISA. Other ISAs, however, require more effort as the available instructions might work inherently different.

VI. Conclusion

The PtraceObfuscator is a working implementation of an obfuscation, which uses the process tracing interface to hide the control flow of an ELF binary. The case study is able to reveal the most important properties of this technique. It is able to offer a good protection against static analysis as well as dynamic analysis, essentially requiring all the obfuscation steps to be reversed. However, once the details of this technique are known, it is not too difficult to write an automatic deobfuscator to completely restore the original binary. The practical use is heavily limited by the performance impact, making it infeasible in many applications. Although, future work might improve this aspect by following some of the presented approaches.

The source code of the PtraceObfuscator can be found on GitHub: https://github.com/BlobbyBob/PtraceObfuscator.

References

[1] E. Eilam, "Reversing: Secrets of reverse engineering," Hoboken, N.J, 2005.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Annual international cryptology conference*. Springer, 2001, pp. 1–18.

[3] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[4] G. Blanchet and B. Dupouy, *Computer architecture*. Wiley Online Library, 2013.

[5] T. Committee *et al.*, "Tool interface standard (tis) executable and linking format (elf) specification version 1.2," 1995.

[6] *ptrace(2) Linux Programmer's Manual*, July 2020.

[7] Go Project, "Package syscall," Feb 2021. [Online]. Available: https://golang.org/pkg/syscall/

[8] ——, "Package x86asm," Feb 2021. [Online]. Available: https://golang.org/x/arch/x86/x86asm

[9] E. Burns, "A go library for the linux ptrace system call," Aug 2014. [Online]. Available: https://github.com/eaburns/ptrace

[10] I. Corporation, "Control-flow enforcement technology specification," May 2019.

[11] E. Burns, "The linux kernel documentation: The /proc filesystem," Feb 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/proc.html

[12] I. Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Nov 2020.

[13] *memfd_create(2) Linux Programmer's Manual*, Nov 2020.

[14] Free Software Foundation, "Gnu binutils," Feb 2021. [Online]. Available: https://www.gnu.org/software/binutils/

[15] ——, "Gcc, the gnu compiler collection," Jan 2021. [Online]. Available: https://gcc.gnu.org/

[16] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1875–1892.

[17] Hex-Rays, "Ida pro," 2020. [Online]. Available: https://www.hex-rays.com/products/ida/

[18] National Security Agency, "Ghidra," Feb 2021. [Online]. Available: https://ghidra-sre.org/

[19] Free Software Foundation, "Gdb: The gnu project debugger," Oct 2020. [Online]. Available: https://www.gnu.org/software/gdb/

[20] ——, "Coreutils - gnu core utilities," 2016. [Online]. Available: https://www.gnu.org/software/coreutils

[21] ——, "The gnu multiple precision arithmetic library," 2020. [Online]. Available: https://gmplib.org/

[22] ——, "Gnu tar," Jan 2021. [Online]. Available: https://www.gnu.org/software/tar/