LONDON'S GLOBAL UNIVERSITY

Multi-Variant Execution Environments

Deterministic protection against memory corruption exploitation in security-critical applications.

Michael Kozlov
MEng Computer Science

April 2023

Supervised by Prof. Lorenzo Cavallaro

ABSTRACT

Memory corruption vulnerabilities persist as a challenge in computer systems security, especially as long as memory-unsafe languages, including C/C++, continue to be used. Defence measures prompt adversaries to adapt and develop new attacks, engaging in a constant arms race with systems security engineers. This highlights the need for higher assurance defence methods in security-critical applications.

Firstly, this project seeks to understand memory corruption exploits, their countermeasures, the subsequent evolution of attacks, and an overview of other more comprehensive methods of defence.

Next, the project will explore Multi-Variant Execution Environments (MVEE) – a proposed technique for run-time detection of program exploitation being considered in research. It achieves this by transparently executing multiple program variants in lockstep, diversified in a way that guarantees a detectable divergence in behaviour if exploited. Existing research is analysed to find the strengths and weaknesses of their design and implementation choices, and to understand the current state-of-the-art.

Finally, this project presents Aurora, a proof-of-concept implementation of an MVEE building on insights from existing work, with benchmark data on several real-world applications. It utilises the `ptrace` Linux API with its recent improvements, as well as other new system calls. Different methods for data replication between variants are discussed, and a new novel approach for direct inter-variant data transfer is proposed.

Contents

Chapter 1

Introduction

C/C++ are powerful programming languages, originally designed with performance instead of memory safety in mind. Their lack of memory safety and complexity results in even the most experienced programmers being able to make errors which result in severe memory corruption vulnerabilities. Through the years, attack-specific countermeasures have been introduced into operating systems and compilers, forcing adversaries to adapt and find other methods of attack. However, these countermeasures only result in an arms race between attackers and defenders. Various more comprehensive solutions have been proposed that attempt to stop certain types of attacks entirely, with various costs, which will be discussed.

The focus of this project is on the concept of multi-variant execution environments (MVEE), a proposed strategy for deterministic detection of program exploitation at runtime. By diversifying two or more *variants* of the same program and running them synchronised in parallel with I/O replication, the system can detect program exploitation as the diversification guarantees these variants will diverge in behaviour when exploited. This can be a valuable high-assurance security mechanism for running programs in highly secured environments, and for running untrusted or legacy software.

There are several challenges to implementing an MVEE system which has acceptable performance overhead and can support different programs. The first is implementing a mechanism for monitoring and containing the behaviour of variants when interacting with the system outside of their process, which usually includes interposing system calls. Several methods have been used in existing research – loadable kernel modules, kernel patching, the `ptrace` debugging API provided by the Linux kernel, and hardware-assisted virtualisation utilizing dedicated CPU features. Other challenges include supporting multi-threading and shared memory inter-process communication in a way that maintains secure monitoring and prevents benign divergence of behaviour in variants.

This project was approached by first studying memory corruption attacks and Linux systems programming basics, then studying existing research on MVEEs and finally implementing a proof-of-concept using the knowledge gained.

1.1 Project Goals

The first goal of this project is to gain an understanding of memory corruption attacks, existing defences against them, and weaknesses of these defences which can warrant the use of a system like MVEE.

The second goal is to study existing research on multi-variant execution environments to perform an analysis of the strengths, weaknesses, and opportunities for improvement of their solutions.

The third goal is to implement and benchmark a working proof-of-concept MVEE, using insights from the analysis of existing research to inform design decisions.

learned: linux internals and programming interface, linux process lifetime, elf format, details of static and dynamic runtime linking

1.2 Report Structure

Chapter 2 reviews memory corruption errors and attacks, countermeasures and other more comprehensive defences.

Chapter 3 introduces the concept of multi-variant execution environments, the required components for such a system, and analyses existing research.

Chapter 4 presents Aurora, this project's proof-of-concept MVEE, and discusses its design choices and implementation details.

Chapter 5 evaluates Aurora benchmarks on two real-world applications and discusses the security of the solution.

Chapter 6 concludes with an evaluation of project achievements and discusses future work.

Chapter 2

Memory Corruption

2.1 Overview

C/C++ continue to be widely used programming languages for their high performance and developed well-developed toolchains. However, these languages are complex and not memory-safe. The misuse of features and developer mistakes can lead to memory corruption errors introducing vulnerabilities. Adversaries can exploit these vulnerabilities to affect program behaviour or, in the worst case, inject and execute malicious code, known as *shellcode*, to take over the system.

Memory errors can be categorised into violations of spatial safety, temporal safety, or type safety [14][21]. Spatial safety violation relates to accessing memory outside the bounds of an object. Temporal safety issues using objects before or after their intended lifetime −such as uninitialised variable use, use after free and double free. Type safety violations occur when memory is incorrectly assumed to be of a different type or when unsafe type-casting is performed.

Memory errors can corrupt code, code pointers, data, and data pointers. Each type of corruption, or a combination of them, can enable an adversary to achieve various goals, including hijacking control flow and arbitrary code execution.

This chapter presents a review of prominent memory corruption attacks and defences, both historical and current.

2.2 Memory Corruption Attacks

2.2.1 Stack Buffer Overflows

The program stack is a region of a process's virtual memory that functions as a first-in, first-out (FIFO) data structure for managing function calls. Within a function's *stack frame*, programs store temporary values and control information needed to restore the execution state of the calling function. This includes a *saved return address* pointer, which indicates the code address to which the program will transfer execution when returning from the current function.

When reading user-controlled input into a buffer on the stack, the use of unsafe library functions can result in writing past the end of the allocated buffer, which can modify values on the stack, including the saved return address. This is known as *stack smashing* [2].

For instance, the legacy `gets` function to read a string from standard input is inherently unsafe, as it continues writing to the destination buffer until a terminating null byte is reached in the input. Similar is the `strcpy` function for copying strings, which performs no bounds checking and continues writing to the destination buffer until a terminating null byte.

An attacker can achieve code injection by crafting an input which overflows the buffer and overwrites the saved return address with a pointer back to the buffer where they have stored shellcode. When the function returns, execution follows the modified saved return address pointer and begins executing shellcode in the buffer. With no mitigations in place, the memory address of the buffer can be estimated by the attacker with knowledge of the execution environment. To improve chances of success, a *NOP sled* is used – by placing a series of `nop` instructions [1] prior to the shellcode in overwritten memory. This way, if a slightly imprecise jump lands in memory before the shellcode, execution will eventually reach the shellcode start.

2.2.2 Heap Attacks

The heap is a region of process memory used for dynamically allocating space for data whose size is not known at compile time. Code requests space on the heap using the `malloc` family of library functions, and must free allocations with `free` when no longer needed. The allocation program manages the placement of allocated blocks in memory and the efficient reuse of freed blocks.

Similarly to stack-based attacks, buffers in the heap can be overflowed to overflow pointers and data. Such vulnerabilities are usually less useful to an attacker and highly

[1] or semantically equivalent instructions with no side effects, such as `xchg` twice.

dependent on program semantics.

More complex heap-based exploits target insecure allocation program designs by corrupting heap management metadata in adjacent dynamic memory blocks. For instance, in [4], an attack is presented that targets a vulnerable function involved in merging adjacent freed blocks of memory. It achieves a write-anything-anywhere primitive, which gives an attacker powerful capability.

One possible target for write-anything-anywhere vulnerabilities is the Global Offset Table (GOT), used to resolve addresses of functions of dynamically linked libraries. The GOT memory must be writable for normal operation; therefore, it is vulnerable to being modified by an attacker to instead point to malicious code.

2.2.3 Format String Vulnerabilities

Format string vulnerabilities [5] were discovered in the `printf` family of functions, which give write-anything-anywhere capability to an attacker. The `printf` function takes a format string, which can contain special format directives and zero or more arguments. When the string is parsed by the function, directives trigger actions such as to read and insert an argument from the stack [2]. When the format string is controlled by user input, an adversary can carefully construct a malicious string with format directives. The simplest attack is to specify more argument directives in the string than arguments actually provided on the stack, thus reading further data from the stack and resulting in information leakage. By using the `%n` directive, which triggers a write to the stack, a specific combination of directives can achieve write-anything-anywhere.

2.2.4 Code reuse attacks

With stack execution prevention (discussed further), adversaries cannot write executable shellcode onto the stack and must turn to code reuse attacks.

Return-to-lib(c) was an early code reuse attack that hijacked program execution to a library function at a known address. On 32-bit x86 architecture, arguments to functions are passed on the stack, making it possible to invoke a library function and provide arguments to it with a stack-based buffer overflow. For instance, an attacker could make a library call to replace the current process with a shell, by calling `execve` with argument string `"/bin/sh"`.

Return-oriented programming (ROP) is the general case for exploits similar to return-to-lib(c), which reuse any executable code in the program memory rather than just standard library functions. These code fragments, referred to as ”gadgets”, end in a `ret`

[2]This historical attack targeted x86 32-bit architecture, where arguments were passed in the stack.

instruction. With a carefully constructed stack of saved return addresses, gadgets can be chained together in a ”ROP chain” to perform more complex exploits. While function arguments are no longer passed on the stack in x86-64, gadgets can be used to set up registers with arguments such as by popping values from the stack – for example `pop rdi; ret;`.

2.3 Countermeasures

2.3.1 Stack Protection

First introduced by StackGuard [3], this type of protection aims to prevent stack-based buffer overflows, which target the saved return address. It is a compiler extension which detects buffer overflows by placing a random *canary* byte at the end of a function's stack frame, between the buffer and the saved return address. This canary is checked before a function exit; the program terminates if the value has changed. Buffer overflows that contiguously write on the stack until the return address is overwritten will also overwrite the canary and fail the check. Modern compilers such as GCC place canaries in functions that meet certain criteria, balancing security with performance.

This protection is vulnerable to a direct overwrite of the saved return address with write-anything-anywhere. Additionally, the canary may be leaked to an adversary through other vulnerabilities and then used as part of the buffer overflow payload to pass the check successfully.

In general, this class of stack defences attempts to ensure that control flow is returned to the actual calling function.

2.3.2 Stack Execution Prevention

This defence constrains memory sections not to be both writeable and executable simultaneously. Code sections in process memory are made non-writeable, and the rest of memory is set to not executable. This prevents code injection, as hijacking control flow to shellcode on the stack will cause a segmentation fault due to the stack being non-executable memory. Initially implemented in software, this defence now exists as a hardware feature in processors – a no-execute flag bit in memory page table entries.

As discussed previously, this defence changed attacks from injecting shellcode onto the stack to instead exploiting code reuse.

2.3.3 Address Space Layout Randomisation

Address Space Layout Randomisation (ASLR) randomises the base address of the code, thread stacks, and heap location in a process' virtual memory. For an attacker to redirect control flow to shellcode they need an address; merely guessing will most likely be incorrect and cause a segementation fault due to landing in unmapped memory, which can act as an indicator of exploitation. Therefore, attackers must rely on information leaks or side-channel attacks to find addresses necessary for an exploit.

Randomisation in a 32-bit address space does not provide enough entropy and has been shown to be vulnerable to brute-force attacks [6]; however, on 64-bit systems, the address space is large enough for effective randomisation.

Modern Linux kernels have full ASLR support. For effective ASLR, the code must be position independent (PIE for Position Independent Executable) with no absolute addresses. Binaries compiled to non-PIE or using shared libraries which are not position independent will reduce its effectiveness.

2.4 Towards Comprehensive Defences

In the previous section, attacks and attack-specific defences have been shown to be in a constant arms race. This section explores more comprehensive potential solutions to protect against certain classes of attacks.

2.4.1 Taint analysis

Taint analysis is designed to detect exploitation by treating sources of untrusted data, such as user-provided input, as *tainted*. It tracks the data during execution and propagates the taint, and detects insecure use of this data.

One implementation of this concept was TaintCheck [7]. This dynamic taint analysis system ensures that target addresses in control-flow transfers are values not derived from tainted data, which would indicate an exploit. TaintCheck is designed to work with already compiled binaries – transparent solutions like this are useful as source code is not always available, or integrating a solution into the source code or compiler may be expensive. This is done using the Valgrind system for dynamic analysis. At runtime, code is inserted in the Valgrind system to implement taint tracking and security checks. The large performance overhead made this implementation too slow for use in production.

In addition to performance overhead, a challenge with taint analysis is carefully defining taint policies that do not lead to high rates of false positives marking all data as tainted.

2.4.2 Software Fault Isolation and Sandboxing

Software Fault Isolation (SFI) is a security solution that runs untrusted or potentially vulnerable application code in a restricted environment known as a *Sandbox*. In the paper [1], it works by loading untrusted code into its fault domain, a separate area in process memory, and ensures that code does not write to memory outside of the sandbox or transfer control outside of the sandbox. SFI instruments application code with instructions that sandbox unsafe instructions at compile time or with a binary rewriter. Before running the sandboxed application, a verifier is run on the modified code to ensure safe instructions are safe and unsafe instructions are properly sandboxed.

In x86-64 binary code, there are two types of memory addresses – relative offset (from a register) and absolute addresses. Absolute address values can be verified to be safe statically, whereas the result of a relative address calculation can be impossible to tell – therefore sandboxing instructions must be inserted which ensure it does not produce an unsafe address. In this solution, several registers must be reserved for use by the sandboxing instructions, which introduced a performance overhead but with the increased register count in x86-64 this is less of an issue.

There are various implementations of sandboxing in use by modern software, such as JavaScript sandboxing in browsers, and application sandboxing in mobile operating systems. Weaknesses of this approach include implementation complexity, performance overhead, and risk of sandbox escape.

2.4.3 Control-Flow Integrity

Control-Flow Integrity (CFI) is a class of defence aiming to prevent control-flow hijacking by detecting control transfer to anomalous memory locations, such as done by ROP attacks.

One approach [10] is a run-time monitoring system designed to detect control-flow hijacking by limiting the allowed targets of control-flow transfers to a predefined set. This is done in two stages – first, using static analysis to produce a control-flow graph and a run-time enforcement mechanism. This prevents an attacker from redirecting control flow, such as to the location of shellcode or to a ROP gadget.

Code-Pointer Integrity (CPI) [20] protects code pointers by preventing modification by code that interacts with data. During compilation, data pointers and sensitive code pointers are identified, and code pointers are protected using memory safety checks. This prevents code pointer modification and subsequent control-flow hijacking.

Indirect jumps are a challenge for static analysis, as well as various programming language-specific abstractions.

2.4.4 Memory-Safe Programming Languages

There are many memory-safe programming languages which can be used for new software, such as Java, Python and Rust. Applications such as operating systems require high performance and low-level memory access as allowed by C/C++, making many languages unsuitable. For this, Rust is a capable language which is now supported [3] in the Linux kernel since 6.1.

Smaller programs could be rewritten in these languages more easily, but for larger code bases, this requires a significant investment of resources.

2.5 Summary

There is an arms race between system security engineers adopting new defences and attackers finding new ways to bypass them and exploit programs regardless. Attackers need only to find a single path of least resistance through security mechanisms, while defenders must provide solutions which provide broad security without significantly compromising on performance or other costs. Some countermeasures work well but only against specific attacks. Protections such as ASLR provide probabilistic protection at best and may depend on a *secret* to be not known by the attacker, such as the location of code under randomisation or the value of a stack canary.

More comprehensive defences attempt to stop certain attacks entirely or contain their damage. The following chapters of this project will explore multi-variant execution environments – a potential runtime defence for deterministic detection of large classes of attacks.

[3]`https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/`

Chapter 3

Multi-Variant Execution Environments

3.1 Concept

A Multi-Variant Execution Environment (MVEE) is a proposed technique for deterministic runtime detection of classes of attacks. This is achieved by executing multiple diversified *variants* of the same program in parallel, synchronised at certain rendezvous points. The variants are diversified in a way that does not affect their execution under normal circumstances but will cause a divergence of behaviour between variants if the program is exploited by malicious input. Which class of attacks is detected by the system depends on the diversification strategy adopted. The system replicates input to all variants and produces a single output, which is verified to be the same across all variants. This is a *secretless* design, immune to security mechanism compromise due to information leakage or side-channel attacks.

This concept of an N-variant execution system for security was first introduced by Cox et al. [8]. The architecture is shown in figure 3.1 – a *polygrapher* receives input and replicates it to variants, while the *monitor* controls synchronised execution of variants and ensures semantically equivalent output from the variants, producing one output. This way, I/O operations are performed only once to maintain transparency. In practice, most existing MVEEs combine the monitor and polygrapher into one. Excluding I/O, the variants all execute the rest of their code in parallel, which makes this method well-suited for modern high-core-count processors.

MVEEs could potentially be implemented to, in the event of divergence, determine the unaffected and correct variant state that should continue executing and recover the system to it.

Existing research has demonstrated that solutions requiring source code integration

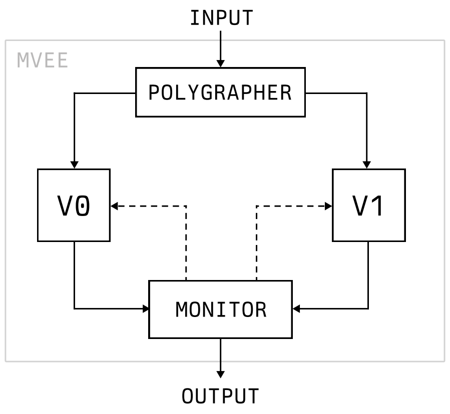


Figure 3.1: General MVEE architecture

or a modified compilation or linking process can allow more powerful system features. On the other hand, fully transparent solutions that can run pre-compiled binaries, potentially using static or dynamic code rewriting, could be more useful as the source code of software is not always available, especially for legacy applications.

This chapter reviews the components of an MVEE system and analyses existing research to gather the strengths and weaknesses of its design and implementation choices.

3.2 Diversification Strategies

The diversification strategy defines the class of exploits that will cause divergence in behaviour and consequently be detected by the monitor. It is a mostly separate component of the system and, in theory, could be swapped out without needing significant changes to the rest of the system.

3.2.1 Disjoint Address Spaces

Most memory corruption exploits depend on using absolute addresses in code pointer overwrites. By diversifying variant processes to use non-overlapping, or *disjoint*, sections of their virtual address spaces, exploits depending on completely overwriting a code pointer with an absolute address can succeed in at most one variant, as in the other variants it is not a validly mapped address and will cause a segmentation fault. This is illustrated in figure 3.2b.

Linux executable binaries can be statically linked or dynamically linked. Static linking involves all required code, including libraries, being linked into the binary. For dynamically linked binaries, after process creation, the kernel loads a dynamic linker and hands off execution to it, which then resolves the binary's specified dependencies in the system and loads them into process memory.

Several things must be done to create this diversification. Firstly, programs and shared libraries must be fully PIE so that they can be arbitrarily relocated in the address space. At the compilation link stage, an ld linker script can change the start address for the `text` section, and `data`, `bss` and heap after it. Next, the runtime dynamic linker must be modified to change where shared libraries are mapped. However, implementing fully disjoint address spaces requires a kernel modification to be able to change the location of the stack, and the location where the dynamic linker itself is loaded.

Disjoint address spaces alone do not cover protection against partial address overwrite attacks, which can be used to bypass security mechanisms. Cavallaro et al. [9] proposed an improvement to this – additionally shifting the address space of one variant [1] by k bytes. This results in partial overwrite attempts producing different relative offsets from the original address between the variants, therefore causing a divergence in execution. This is done by inserting junk data at the beginning of the `text` segment with an ld linker script.

A different diversification in MvArmor MVEE by Koning et al. [17] also provides protection against relative attacks in dynamically allocated memory. It uses two different heap memory allocators to change the distance between heap objects on one of the variants.

A weakness of this strategy is it cannot detect corruption of non-pointer data, as pointers are neither fully nor partially overwritten.

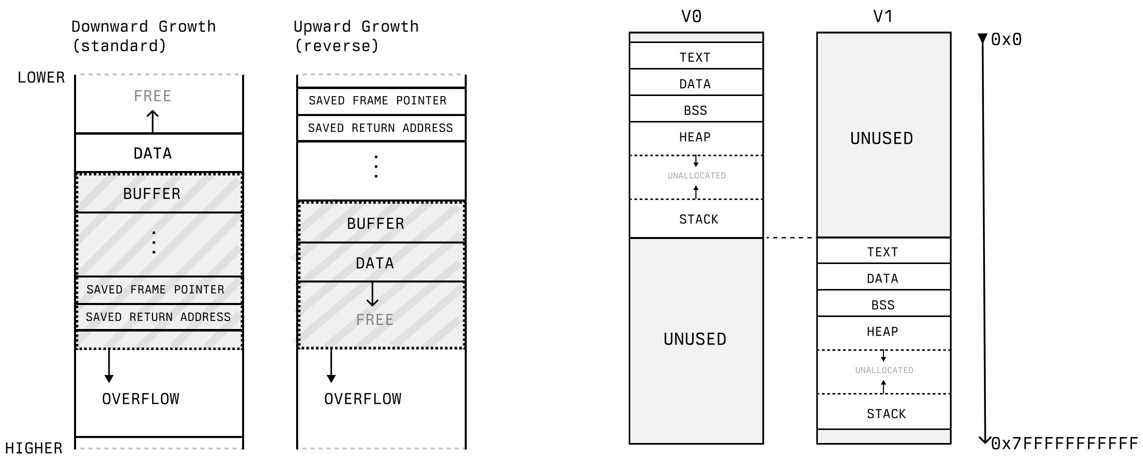
Disjoint address spaces are the predominant strategy adopted by existing MVEE systems, as it provides protection against a large class of memory error exploits.

3.2.2 Reverse Stack Growth

This is a compiler-based technique proposed by Salamat et al. [11] alongside Orchestra MVEE. Its advantage is that it requires no kernel modification compared to disjoint address spaces.

By reversing the direction in which the stack grows, it can thwart a buffer overflow as the saved return address is now on the other side of the direction in which a buffer overflow writes. This is illustrated in figure 3.2a. The implementation is complex, while only providing security against stack-based buffer overflows specifically. Potentially, it could be paired with a non-fully implemented disjoint address space randomisation, for increased protection without any kernel modifications.

[1] In a two-variant architecture.



(a) Reverse stack growth

(b) Disjoint address spaces

Figure 3.2: Variant diversification strategies for a two-variant architecture

3.3 Monitoring and Synchronisation Approaches

There are several proposed approaches for monitoring and synchronisation of variants in existing research, which are discussed in this section.

A logical location for rendezvous points between variants is at system call entry, as the context switch to the system call gives an opportunity for interposition with monitor logic. Additionally, the monitor needs to be able to inspect system calls to verify non-divergence of behaviour, modify them when needed to keep the system synchronised and transparent, and prevent the escape of malicious actions from the system via dangerous system calls.

In further detail, the requirements for the monitor are as follows.

The monitor must check that system calls and their arguments are semantically equivalent between variants. If they are not equal, this indicates a divergence of behaviour which should be treated as a result of malicious exploitation, and the system should halt and enter an alarm state.

The monitor must handle dangerous system calls that may break the containment of the system, including `mmap`, `fork` and `clone`, and otherwise handle system calls correctly to prevent benign divergence.

The monitor may allow execution of the system call for both the master and replica or the master only. Additionally, the monitor may need to:

(a) Modify arguments of the replica's system call prior to execution.

(b) Replicate the return value from the master's call into the replica after execution.

(c) Replicate more complex effects on the replica, such as copying data newly read data from a master buffer into the corresponding replica buffer.

3.3.1 Loadable Kernel Modules

A Loadable Kernel Module (LKM) allows extending the functionality of the Linux kernel. This is a powerful approach; however, it increases the size of the trusted computing base (TCB) of the system, which is undesirable. Vulnerabilities can potentially be introduced by the monitor itself, which if exploited in the LKM would grant an attacker the highest possible privilege level. Compared to user-space programs, developing LKMs is more complex, harder to debug and the produced code is less portable.

Cox et al. [8] use an LKM in their MVEE to interpose system calls by wrapping them in additional code – which detects if the caller is running inside the MVEE, and if so carries out the necessary logic. Koning et al. [17] observed that this approach of system call interposition produces the lowest run-time overhead compared to other approaches, which are discussed in this section.

3.3.2 Ptrace

`Ptrace` is a system call allowing a process to trace and debug another process. It provides capabilities such as controlling execution, tracing system calls, reading and writing registers and process memory.

Using the `PTRACE_SYSCALL` method, the monitor can make the process execute until a system call entry, or if already at an entry – until system call exit. When both traced variants reach a system call entry, the monitor can inspect the call number and arguments to perform the semantic equivalence check, and modify the call if needed. When both traced variants reach the system call exit, the monitor can perform data replication between the master and replica variants.

A user-space monitor using `ptrace` benefits from Linux process security isolation from the variants and privileged code in the kernel, and is simpler to develop and debug. A disadvantage is the overhead introduced by this approach due to the context switching between user-space and kernel for `ptrace` operations. Additionally, the level of access the monitor has to the variants is constrained by the capability `ptrace` methods and other available system calls.

The existing MVEEs using a `ptrace`-based approach are: MVEE by Cavallaro et al. [9], Orchestra by Salamat et al. [11], and GHUMVEE/ReMon by Volckaert et al. [15, 18, 22].

3.3.3 Binary Rewriting

VARAN by Hosek and Cadar [16] is software-reliability-oriented MVEE that uses selective binary rewriting to replace system calls with jump instructions to a handler within an *agent* – code injected into process memory. This agent communicates with agents in other variants for synchronisation. For a security-oriented MVEE, however, this is unsuitable as the agent has no isolation from the rest of the process code. An attacker may subvert the agent and still trap into a system call to cause damage, such as with a ROP chain that abuses x86-64's non-requirement of instruction alignment by jumping into the middle of an instruction containing the bytes for a `SYSCALL`.

3.3.4 Hardware-Assisted Virtualisation

MvArmor MVEE by Koning et al.[17] uses hardware-assisted process virtualisation for its monitor. It builds upon Dune [13] – a hypervisor using Intel VT-x extensions, allowing processes to run in ring 0 and use privileged x86-64 CPU instructions. Dune switches between running the normal Linux Kernel and running MvArmor monitor. There is complete isolation of the monitor from the rest of the system, thereby not increasing TCB with the monitor.

All `syscall` instructions trap into the monitor instead of the kernel. For each variant, there is a *variant manager* communicating with the *detector* component, which inspects behaviour for divergence. Certain system calls such as `getpid` are emulated by MvArmor for a performance improvement, otherwise a `vmcall` is made to the kernel to execute the system call. MvArmor's *namespace manager* ensures variants receive the same information, such as process IDs and file descriptors, as their partner variant to prevent benign behavior divergence.

Microbenchmarks in the paper report the system call overhead for Dune being an order of magnitude smaller than for `ptrace`, albeit only tested with `getpid` system call. The authors note that the limitation of this system is that extending MvArmor to support multi-threaded applications would require the underlying Dune software to be made thread-safe, which it is currently not.

3.4 Performance Considerations

At the time of Orchestra, its system had to work with the limitation of the `ptrace`'s `PTRACE_PEEKATA` and `PTRACE_POKEDATA` methods for reading and writing a traced process' memory returning only four bytes at a time. This meant comparing the memory of variants required many calls to `ptrace` which incurs a high latency cost due to context switches to the kernel. They improved this by creating a shared memory block between each variant and the monitor, and injecting code into the variant memory, which writes to this buffer when activated by the monitor. Since then, new system calls `process_vm_readv` and `process_vm_writev` allow arbitrarily large reads and writes from multiple regions of memory with just one system call, and transfer data directly between processes without copying to the kernel first.

VARAN is an MVEE targeting software reliability instead of security but nonetheless has an interesting *event streaming* architecture which can be learned from for designing better-performing security MVEEs. As described previously, it uses selective binary rewriting to redirect system calls to an agent injected into variant memory. Agents in different variants use a shared ring buffer for communication, which allows lock-free concurrent access. The *leader* variant records all events (system calls and signals) into the shared ring buffer, which is later read by *follower* variants that mimic the leader's behaviour. This architecture results in no bottleneck from a central monitor waiting on multiple variants. Benchmarks in the paper showed a 2.4% overhead introduced versus Orchestra's, which uses `ptrace`, reported overhead of 50%.

MvArmor adopted the ring buffer design for communication between its variant managers, with one variant executing all system calls, and the rest consuming their recorded result. Additionally, this allowed variants to run non-security-sensitive system calls asynchronously, while execution for security-sensitive system calls remains in lockstep.

3.5 Multi-Process Monitoring

An MVEE system must handle the creation of child processes by a variant to maintain security, setting up a new multi-variant system by creating a replica variant for the child process and running the pair under the control of a monitor. A design may choose to synchronise all children and their variants from one monitor process; however, this would be complex for the monitor to manage and be a performance bottleneck. The better alternative is to create a new monitor thread or process for each multi-variant system, and retain minimal communication with a central monitor.

In Orchestra, which uses `ptrace` and sets up a multi-variant system for each variant child process, the authors noted that a parent monitor detaching from a variant would mean a period of execution without system call monitoring before the new monitor attaches. In their workaround, they let parent monitor let them execute until their first system call, which is saved and replaced with `pause`, allowing a safe handoff to the new monitor. The new monitor then restores original system call, and signals to resume. This is no longer a problem with new `ptrace` handling of forking and cloning, where it immediately stops their execution.

VARAN uses a *zygote* process, which acts as a template for a variant monitor process. The Zygote creates a `socketpair` with the coordinator (essentially a central monitor), which is inherited by forked children, enabling the efficient setup of a communication channel.

3.6 Benign Behaviour Divergence

Differences between variant processes must be handled to prevent benign divergence. Information retrieved via system calls, such as process ID, can be easily modified.

The virtual dynamic shared object (vDSO) is a shared library mapped into process memory by the kernel, which handles a set of system calls without a normal kernel trap, such as `__vdso_clock_gettime` . This presents a problem for `ptrace`-based monitors, as these system calls cannot be traced. VARAN handles this by replacing the entry point of vDSO functions with a jump to its own handler within the process, but as discussed this strategy is not suitable for a security-based MVEE. In MvArmor, vDSO calls are intercepted with a mechanism provided by Dune.

One source of time available to programs are raw processor time stamp counters via the `RDTSC` instruction. GHUMVEE and MvArmor solve this by setting x86 control registers to cause this instruction to trap.

When used with a disjoint address space diversification, pointer-value-based program logic such as hash tables can cause non-determinism. GHUMVEE attempts to solve this by interposing common hash functions using a glibc patch, and requires developers to integrate this interposition in source code.

This section discusses other Linux process behaviours which can cause benign divergence.

3.6.1 Shared Memory

Memory mapping in the Linux system allows mapping files, shared memory for inter-process communication, and devices into a process' virtual memory. This can be an issue as reads and writes to memory do not go through the system call interface; therefore, the monitor cannot inspect and control variant operations or do I/O replication.

GHUMVEE's workaround for memory-mapped files is to switch `MAP_SHARED` mappings to `MAP_PRIVATE` type. The monitor then tracks manipulated blocks and performs a manual write to the file.

Vinck et al. [22] proposed a solution for shared memory with an extension to GHUMVEE. The initial solution involves trapping all accesses to shared memory pages, with the monitor then performing the operations on behalf of the variants, which was also proposed by Cavallaro et al. [9]. Next, a more efficient solution is presented – utilising dynamic analysis and an in-process agent to identify and instrument instructions accessing shared memory ahead of time, allowing user-space data replication bypassing the monitor. This solution enables MVEEs to support inter-process communication through shared memory, which has been an obstacle to using MVEE systems for more complex applications until this most recent publication.

3.6.2 Multi-threading

Multi-threading is a challenge due to thread scheduling being non-deterministic. This results in the monitor observing system calls from variants in different orders, and falsely flagging this as a divergence due to malicious input. Threading can be achieved with kernel-supported threads or user-space threads. It is difficult to modify the behaviour of user-space threading libraries.

Partial support for multi-threading is implemented in MvArmor. For non-security-sensitive system calls that are allowed to execute asynchronously, the followers are forced to adhere to the order of system calls of the leader.

Volckaert et al. [18] build on GHUMVEE to provide a solution for thread synchronisation. Due to the high frequency of synchronisation events, latency added by context switches of `ptrace` and `process_vm_readv/writev` system calls make them not feasible for use in the replication of events between variants. Instead, this paper proposes in-process agents which communicate with each other using a ring buffer. A modified glibc with a *replication agent* is loaded into variant processes. After a configuration stage, agents in variants communicate only with each other and not the monitor, avoiding the latency overhead of system calls. The master variant captures the order of synchronisation events into a ring buffer, which is read by replica/follower variants, whose agents replicate the synchronisation behaviour of the master variant.

As noted with VARAN, an in-process agent can be insecure. In this system, the ring buffer must be protected from potentially malicious code. A more secure agent design, using a *hidden buffer array*, is then proposed by the authors. An array containing pointers to a hidden buffer is mapped into the memory of a variant, and its location attempts to be hidden from user-space code. The `gs` and `fs` segment registers are used to store a base address for use with relative memory access. While user-space code is not allowed to access these registers directly, the replication agent can use relative memory access with gs/fs registers as the base, which produces the address to the hidden buffer array. This system is claimed to guarantee that the pointer will never leak into process memory.

3.6.3 Signal Delivery

Asynchronous signal delivery can cause divergence of variant behaviour. For example, if a signal affects system call S_n and only one of the variants has received the signal before trapping into S_n, the variant system calls will be different at the rendezvous point.

The monitor can can intercept signals and deliver signals at the same time; however, this does not guarantee that variants are at an identical state in execution. Therefore, the monitor can hold back signals and deliver them to variants at the same point of execution

– at the rendezvous point. Delayed signal delivery is likely to have a performance impact.

3.7 Summary

This chapter has introduced the concept of multi-variant execution environments, a technique for deterministic run-time detection of software exploitation through executing multiple diversified variants in parallel. Various methods for monitoring and synchronisation in MVEEs have been proposed in existing research, as well as methods to handle benign divergence. These choices entail finding a balance between security, performance, solution complexity, and transparency to applications.

Chapter 4

Aurora MVEE

The goal of Aurora is to apply insights from the analysis of existing MVEEs to implement a prototype targeting a subset of features a production-ready solution would need. A fully featured, state-of-the-art MVEE is a substantial undertaking, with teams of researchers still investigating potential solutions [19].

Aurora's contributions include a `ptrace`-based implementation in C for x86-64 on a modern Linux kernel[1], utilising new system calls `process_vm_readv/writev` and new `ptrace` features. Next, the project proposes a novel design for an inter-variant data replication method with no kernel extensions necessary. Finally, benchmark results on several real-world applications are presented.

4.1 Design and Implementation

Aurora is a user-space monitor using `ptrace`, running unprivileged with no increase to the trusted computing base. This approach was additionally chosen as it is the easiest to get started with and to debug, but nonetheless, previous literature has show it is possible to base a feature-rich and high performance design [18][22]. It uses a two-variant architecture with a *master* variant and *replica* variant, suitable for a disjoint address spaces diversification strategy, and currently only supports single-threaded, single-process applications.

A modular system call execution system allows to conveniently develop support for new system calls, currently implementing for two applications: GNU Cat and GNU Gzip. Aurora is envisioned to run binaries with no source code modification or recompilation required, achieving so for these applications due to their relative simplicity.

The monitor sets up variants by forking, attaching a `ptrace` trace back to the parent, and then executing the target program with specified arguments and environment

[1]Ubuntu 22.04.2, kernel 5.19.0-38-generic

variables. The variants are traced with `PTRACE_SYSCALL` option, which stops the variants upon a system call entry or exit. The main loop of the program is a state machine where variants switch between the state *running* and state *waiting* when they are the first of the two to reach a system call entry and are waiting for the other to reach the rendezvous point.

4.1.1 System Call Handling

When both variants reach the rendezvous point of system call entry, if their system call codes match, the handler for that system call is invoked. The handler ensures semantic equivalence of the call by inspecting arguments and reading into the variants' process memory to perform comparisons where required. A large part of the implementation effort was understanding what each system call does, how to check its equivalence, and how execution should proceed.

A new method of `ptrace` is used – it allows requesting system call information into a struct `ptrace_syscall_info`, containing data such as arguments on system call entry and return value on system call exit. For memory reads to and from variants, the `process_vm_readv` system calls are utilised, which are more efficient than older methods, as previously discussed.

In the set of system calls implemented to allow the chosen applications to run, their execution handling can be categorised as follows:

(a) **Execute in both.**
For system calls that affect each process themselves and are needed for correct execution, such as `mmap`, or calls returning some data where allowing the replica to do the call is faster instead of copying result from the master, as long as it has no side effects.

(b) **Execute in master only, replicate result.**
Including I/O related system calls, for instance – `read` where the data is read by master, then copied from the master buffer into the replica buffer. As ptrace on x86-64 does not support blocking a system calls, and variants will always execute the system call they have just entered and paused on, instead the `RAX` register in the replica variant can be modified to perform `getPid` which has no side effects. The return value is then set.

(c) **Execute in both, replicate master return.** Some system calls need to be executed in both but will need return value copied to the replica due it failing. One such example was `openat` in order to reserve the same file descriptor, but which can fail in the replica variant due to master system call already have affected outside state (creating file for writing). This specific case could have been handled differently by changing replica's system call to successfully reserve a file descriptor another way, but number of `ptrace` calls to Kernel would not improve.

4.1.2 Memory Mapping

Memory mapping is allowed to be write-able only if it has flag `MAP_PRIVATE`, meaning it will have no effect outside of the variant process. For a file, this flag makes the mapping a copy-on-write copy, with no changes written back to the file.

A security manager component was implemented to dictate security policies such as this – upon a `mmap` or `mprotect`, the system call handler checks with the security manager to see if the *protection* and *flags* arguments represent an allowed action.

Mappings are recorded in the monitor to keep track of their flags so that security policy can be maintained when protection is updated with `mprotect`.

4.1.3 Diversification

As the main focus of this project was on the MVEE monitor rather than the implementation of the diversification, the final provided protection is probabilistic as it relies on ASLR for diversity. However, a linker script was created for the GNU Binutils' ld linker, which changes the entry point of the `text` section, and subsequently `data`, `bss`, and heap, to further diversify the replica variant. This improves protection but is still not deterministic, as fully disjoint address spaces would require a kernel modification, as previously discussed. Due to this and challenges with re-linking the applications used for benchmarking, the script was not used.

4.1.4 Direct Inter-Variant Memory Copy

This subsection proposes a potential novel approach of transferring memory between variants directly, which would reduce execution time for I/O replication as copies would not have to be done via the monitor. In GHUMVEE, Salamat et al. implemented a kernel-based extension to `ptrace` to achieve direct tracee-to-tracee memory copy – however, it may be possible to achieve this with existing system calls.

The idea is to allow the replica variant to use `process_vm_read` to read from the master's buffer into its own, for instance, during a `read` system call. Figure 4.1 illustrates three methods for data replication purely in user-space code – the old approach through `ptrace`, `process_vm_readv/writev` through the monitor, and the new proposed direct `process_vm_readv` invoked by the replica.

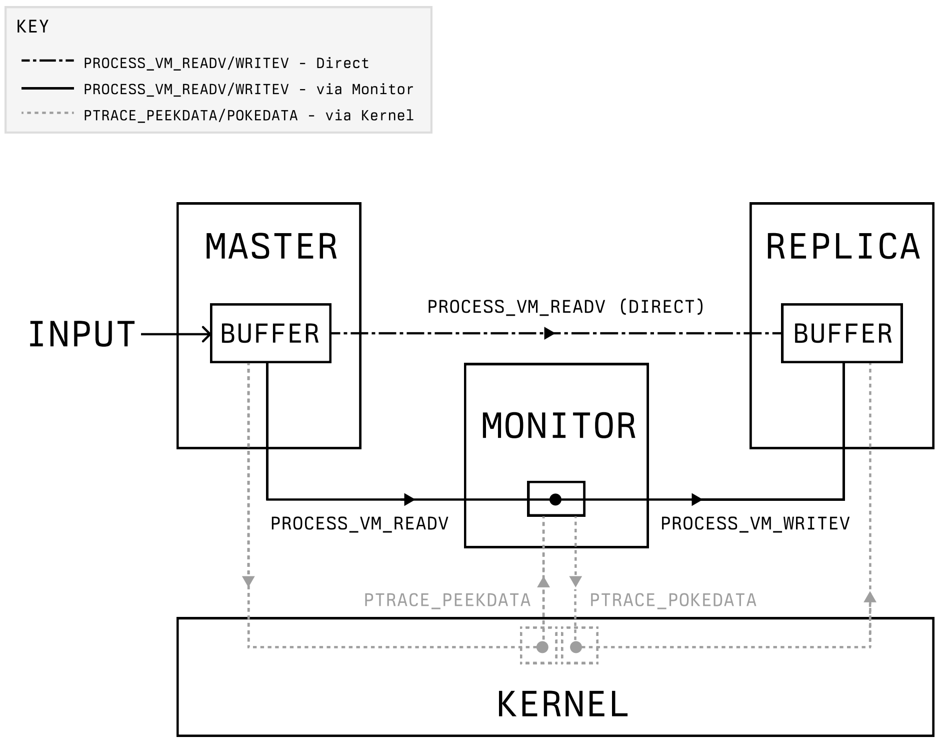

KEY
PROCESS_VM_READV/WRITEV - Direct
PROCESS_VM_READV/WRITEV - via Monitor
PTRACE_PEEKDATA/POKEDATA - via Kernel
MASTER
REPLICA
INPUT
BUFFER
PROCESS_VM_READV (DIRECT)
BUFFER
MONITOR
PROCESS_VM_READV
PROCESS_VM_WRITEV
PTRACE_PEEKDATA
PTRACE_POKEDATA
KERNEL

Figure 4.1: Illustration of data flow in different transfer methods: direct inter-variant transfer, transfer via monitor, and transfer via the monitor through the kernel. A solid dot represents data passing through an intermediate buffer.

To implement this – firstly, some prerequisites:

1. **System call replacement.**

   Using `ptrace`, the system call a variant is about to execute is replaced by modifying its registers, and arguments are replaced as well if needed. To pass more complex argument structs, the monitor will borrow unused memory from the variant's stack to place these structs, or create a dedicated memory mapping for this use in the process memory using the same system call replacement method.

   Of course, the original system call may still need to be executed, or the monitor may want to make the variant execute multiple calls. To achieve this, the current instruction register, `RIP`, is adjusted so that upon the variant resuming, it will trap into the kernel with a system call again.

2. **Linux Capabilities**

   Privileged processes running under effective user ID 0, or *root*, bypass all permission checks from the kernel for actions which unprivileged processes cannot do. Capabilities enable granting specific permissions to unprivileged processes as needed, in accordance with the formal security principle of least privilege.

> Performing `process_vm_readv` on another process requires the `CAP_SYS_PTRACE` capability, so the goal is to grant the replica this capability to allow it to read from the master, and then revoke the capability in a controlled way to guarantee security.
>
> There are multiple capability sets - effective, inheritable, permitted, bounding, and ambient. The *permitted* set is the set of capabilities a process is allowed to assume by bringing it into the *effective* set. The *ambient* set is the capabilities preserved across a `execve` for a non-privileged process. To raise a capability into the ambient set, the current process must have the capability in its permitted and inheritable sets.

The monitor must first have the `CAP_SYS_PTRACE` capability in its inheritable and permitted set. This allows the replica, created with `fork()` from the monitor, to raise the capability into its ambient set. Consequently, after its `execve`, the replica retains the capability in its permitted and effective sets. The replica should then drop it from its effective set until needed for data replication. To achieve this, upon tracing of the first system call, the monitor must save the current system call, replace it with a `capset` system call to drop the capability from its effective set, then re-run the original system call with the aforementioned method. During execution, the monitor will check system calls to ensure the process itself does not bring the capability into its effective set, leaving no opportunity to escalate privileges.

During handling of system calls which require replication of data from the master to the replica, the monitor would behave as follows:

1. Replica variant's system call is replaced with `capset` to bring `CAP_SYS_PTRACE` capability into the effective set.
2. Monitor places `process_vm_read` argument struct into the replica's memory, specifying information about the source buffer in the master variant and the destination buffer in the replica variant.
3. Replica's `RIP` is rewound to trap into a system call immediately again.
4. Replica's registers are changed to request `process_vm_readv` system call, with pointers to arguments structs. The system call is run, and the kernel replicates data from the master's memory to the replica's memory directly.
5. Upon system call exit, the return value is replaced by the master's initial return value.

Given the increased latency overhead from a higher number of system call context switches required for setup, this method is best suited for situations where the volume of data to be replicated surpasses a threshold at which I/O operation execution time becomes predominant.

Chapter 5

Evaluation

5.1 Methodology

For GNU Cat and GNU Gzip, tests were run on a series of text files with random data, from 1K bytes up to 1B. Cat is purely I/O based, outputting the content of a file to standard output, while GZip is a compression utility with both a computational workload and I/O operations. Perf [1] was used to measure execution time, taking an average of 10 runs for each test file. Tests were conducted on a machine with Intel(R) Core(TM) i7-8750H 2.20GHz CPU with Hyper-Threading turned off, 16GB RAM, on live Ubuntu 22.04.2. The monitor and each variant were pinned to separate cores using `sched_setaffinity` system call.

5.2 Results

Size (Bytes)	Baseline (ms)	MVEE (ms)	Factor Increase
1K	0.35	15.14	43x
10K	0.35	16.20	46x
100K	0.39	15.09	39x
1M	0.47	16.82	36x
10M	1.24	24.13	19x
100M	9.54	94.46	10x
1B	90.43	805.59	9x

Table 5.1: GNU Cat - average execution time comparison

Results for both Cat (table 5.1) and GZip (table 5.2) show a large overhead, which is expected due to added latency from additional context switches to the Kernel when the

[1]https://perf.wiki.kernel.org/index.php/Main_Page

Size (Bytes)	Baseline (ms)	MVEE (ms)	Factor Increase
1K	0.46	17.76	39x
10K	0.66	17.88	27x
10K	3.46	24.19	7x
100K	31.07	81.71	3x
10M	308.55	651.09	2x
100M	2,959.00	6,244.73	2x
1B	29,589.03	62,113.28	2x

Table 5.2: GNU Gzip - average execution time comparison

monitor makes `ptrace` system calls, and the additional I/O operations for replication between variants and semantic equivalence checks of their system calls. The overhead due to additional system calls becomes less significant with increasing file size due to the added I/O operation time.

For GZip, the overhead becomes much smaller from 10K bytes and larger compared to Cat. This is due to increasing computation time to compress the data, which is unaffected by the MVEE system, thereby making the latencies less significant overall.

One thing about the code which could be optimised is reducing the number of system calls when doing deep equivalence checks into the variants' memory. Currently, where required, for each argument, the monitor will read memory from the variant separately. This can be changed to read all of the necessary noncontinuous memory sections from one variant in one system call by correctly constructing a call to `process_vm_readv`. However, in the current code, the monitor is still benefiting from this system call not having size limits and its implementation of copying data directly between processes to bypass the kernel.

5.3 Security Analysis

The monitor is well isolated from the variants by user-space process isolation and executes their system calls in lockstep after performing a semantic equivalence check which will detect divergence if present. All system calls which have not been safely implemented yet are blocked to prevent escape from the system. However, vDSO calls interception has not been implemented or blocked, which may cause divergence in behaviour but should not be a security issue as there is no opportunity for environment escape. Creation of shared memory has been disallowed by ensuring mappings are only writeable if private. As previously discussed, the protection provided by the system currently is probabilistic as ASLR is the only diversification between the variants.

Chapter 6

Conclusion

6.1 Summary

This project has successfully completed all three goals which were set. The first goal of studying memory corruption attacks and countermeasures, and the second goal of analysing existing research on MVEEs, were successful. Insights gained are documented in Chapters 1 and 2, respectively.

The third goal of implementing a proof-of-concept MVEE can also be considered successful – Aurora is a working MVEE which runs two real-life programs, with benchmark data collected. The code was written to be clean and expandable, with separation of different internal components, and clear logic to handle the state of the variants. This goal was left intentionally open-ended to decide the specific sub-goals after studying existing research and understanding what would be possible.

There were some sub-goals not achieved in time. System calls for a *netcat* benchmark were implemented, but there was a bug with the system not working with very large files, which was not resolved. Next, the project intended to support a web server, however, most utilise `fork` or `code` system calls and multi-process support is not yet implemented in the system. Therefore, system calls for *darkhttpd*, a simple web server that doesn't fork or multi-thread, were implemented. Here there was an unresolved bug with a system call, `pselect6`, which was not returning from the kernel upon a new connection as it should. To collect data comparing different data replication methods, a compile option to instead use the old `ptrace`-based methods was added but faced a bug during testing. Similarly, implementation for direct inter-variant memory transfer was mostly finished, but a final issue was not resolved in time.

6.2 Future Work

Future work on Aurora could expand system call support, implement the diversification component, further improve performance by aggregating usage of data transfer system calls as mentioned in the evaluation, or complete the direct inter-variant memory transfer approach.

Multi-variant execution environments are a promising concept to ensure the security of programs by deterministically detecting classes of exploitation at runtime. They can add a valuable layer of protection to high-security environments and allow running legacy software safely. Within existing research, sophisticated solutions exist which often require some integration with source code. Therefore, future work may explore ways to transparently run pre-compiled binaries in an MVEE while still supporting required features. There is potential to use hardware-assisted virtualisation for increased system access to achieve this.

Bibliography

[1] Robert Wahbe et al. "Efficient software-based fault isolation". In: *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 1993, pp. 203–216.

[2] Aleph One. "Smashing the stack for fun and profit". In: *Phrack magazine* 7.49 (1996), pp. 14–16.

[3] Crispan Cowan et al. "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." In: *USENIX security symposium*. Vol. 98. San Antonio, TX. 1998, pp. 63–78.

[4] Anonymous. "Once upon a free()". In: *Phrack Magazine* 57.9 (2001).

[5] Team Teso Scut. "Exploiting format string vulnerabilities". In: *March2001* (2001).

[6] Hovav Shacham et al. "On the effectiveness of address-space randomization". In: *Proceedings of the 11th ACM conference on Computer and communications security*. 2004, pp. 298–307.

[7] James Newsome and Dawn Xiaodong Song. "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." In: *NDSS*. Vol. 5. Citeseer. 2005, pp. 3–4.

[8] Benjamin Cox et al. "N-Variant Systems: A Secretless Framework for Security through Diversity." In: *USENIX Security Symposium*. 2006, pp. 105–120.

[9] Lorenzo Cavallaro. "Comprehensive memory error protection via diversity and taint-tracking". In: *Universita Degli Studi Di Milano* (2007).

[10] Martín Abadi et al. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[11] Babak Salamat et al. "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space". In: *Proceedings of the 4th ACM European conference on Computer systems*. 2009, pp. 33–46.

[12] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[13] Adam Belay et al. “Dune: Safe user-level access to privileged CPU features”. In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12).* 2012, pp. 335–348.

[14] Victor Van der Veen et al. “Memory errors: The past, the present, and the future”. In: *Research in Attacks, Intrusions, and Defenses: 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings 15.* Springer. 2012, pp. 86–106.

[15] Stijn Volckaert et al. “GHUMVEE: efficient, effective, and flexible replication”. In: *Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers 5.* Springer. 2013, pp. 261–277.

[16] Petr Hosek and Cristian Cadar. “Varan the unbelievable: An efficient n-version execution framework”. In: *ACM SIGARCH Computer Architecture News* 43.1 (2015), pp. 339–353.

[17] Koen Koning, Herbert Bos, and Cristiano Giuffrida. “MvArmor: Secure and efficient multi-variant execution using hardware-assisted process virtualization”. In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE. 2016, pp. 431–442.

[18] Stijn Volckaert et al. “Multi-Variant Execution of Parallel Programs”. In: *arXiv preprint arXiv:1607.07841* (2016).

[19] Stijn Volckaert et al. *ReMon: A Secure and Efficient Multi-Variant Execution Environment for x86 Linux Programs.* `https://github.com/ReMon-MVEE/ReMon`. GitHub repository. 2017.

[20] Volodymyr Kuznetzov et al. “Code-pointer integrity”. In: *The Continuing Arms Race: Code-Reuse Attacks and Defenses.* 2018, pp. 81–116.

[21] Mathias Payer. “How memory safety violations enable exploitation of programs”. In: *The Continuing Arms Race: Code-Reuse Attacks and Defenses.* 2018, pp. 1–23.

[22] Jonas Vinck et al. “Sharing is caring: Secure and efficient shared memory support for mvees”. In: *Proceedings of the Seventeenth European Conference on Computer Systems.* 2022, pp. 99–116.

Appendices

A System Manual

A.1 Code Overview

Overview of code structure:

- `monitor.c` : main program logic.
- `variant.c` : methods for creating, configuring and controlling variants.
- `mappings.c` : methods for management and tracking of memory mappings.
- `security_manager.c` : methods for security policy decisions.
- `sys/` : directory with system call executors.

The monitor creates and configures the variants, then loops, waiting on any child process until one of them terminates. When `wait` returns – control is passed to `processVariantEvent` method, which manages variants' state. When both variants are at a system call entry, `processSyscallSync` is called, which invokes the relevant executor.

Both variants are represented with a struct:

```
typedef struct {
    pid_t pid;
    int state; //VSTATE_RUNNING or VSTATE_WAIT
    int type; //VTYPE_master or VTYPE_replica
    unsigned long long syscallNum;
    struct ptrace_syscall_info syscallInfo;
} variant;
```

`syscallInfo` contains the latest system call information returned from `ptrace` call. `syscallNum` was added to potentially support the master variant executing ahead of the replica, out of lockstep, for select non-dangerous system calls.

Supporting new system calls involves creating a new executor, adding its header to the `syscall_executors.h` header, and invoking it in the switch statement in `monitor.c`. An executor can resume a system call at its entry (with `resumeVariantFromWait`) and return control to the main loop. Alternatively, if more complex handling is required, it can wait for the system call exit, then resume from exit (with `resumeVariantFromExit`) and now return control.

A.2 Building

Requirements: CMake 3.24, `libcap-dev` library

1. Set compile options: for data variant transfer method in `compile_options.h`; debug output in `debug.h`
2. Create and enter directory `cmake-build-debug`
3. Configure with `cmake ..`
4. Build with `make`

B User Manual

Usage of compiled program:

```
./MVEE <path to program> [args...]
```

For direct inter-variant memory transfer, must first set capabilities:

```
sudo setcap 'cap_sys_ptrace=eip' ./MVEE
```

C Project Plan and Interim Report

Project Plan

Project title

Multi-Variant Execution Environments

Supervisor's name

Lorenzo Cavallaro

External supervisor's name
N/A

Aims and objectives
Aim: To understand and compare existing multi-variant execution environments (MVEE) to detect memory corruption exploits and to implement a proof of concept.

Goals:

- Review of existing MVEE solutions and report on the comparison of the state-of-the-art design choices and implications.
- Design an MVEE solution informed by previous research, possibly building on existing work.
- Create, benchmark and document a working proof of concept implementation of the MVEE solution.

Expected outcomes/deliverables

- An implementation of an MVEE system.
- Benchmarks on real Linux programs.
- A final report describing design choices and implementation.

Work plan

Target	**Finish by**
• Background reading on memory exploitation techniques and Linux systems programming to inform: • Reading of core papers on MVEE	Start-Mid December
• Analysis of existing MVEE papers • Design of the project's MVEE solution	Mid Jan - Early Feb
• Implementation and benchmarks	April 12th
• Report and final submission	April 26th

Interim Report

Name

[redacted]

Project title as given in your October Project Plan (and Current project title)

Multi-Variant Execution Environments (MVEE)

Your internal supervisor's name

Prof Lorenzo Cavallaro

Progress made to date

I have done background reading in Linux systems programming and memory corruption vulnerabilities to gain knowledge to understand the core MVEE papers reading of this project.

For Linux systems programming, I read select chapters from the book *The Linux Programming Interface* by Michael Kerrisk. I will continue to review further parts of this book as needed throughout the project. For memory exploitation, I have read papers giving a current overview of vulnerabilities and mitigations, and read papers on specific vulnerabilities.

Reading of the core MVEE papers was completed.

While reading the core MVEE papers I have made detailed notes on each, beginning the analysis and comparison of techniques.

Remaining work to be done before the final report deadline

- Finish analysis of core MVEE papers,
- Review if there are any further papers relevant to the state-of-the-art design of MVEE systems that would be important to include,
- Create a proof-of-concept (POC) design of an MVEE system that is feasible to implement given my expertise and time constraints,
- Implement POC design and run benchmarks,
- Create report structure
- Document completed background knowledge, literature review, and analysis of core papers
- Complete report with explanation of POC design choices and analysis of benchmark results.

D Code Listings

D.1 Aurora MVEE

Header files (`*.h`) and system call executors have been intentionally omitted from this document for conciseness, with the exception of a selection of executors that may be of particular interest. If needed – please refer to the accompanying source code archive.

`mappings.c`

```c
#include "mappings.h"
#include "debug.h"
#include <malloc.h>


typedef struct node node;
struct node {
    variantMMap entry;
    node *next;
    node *prev;
};

node dummy = {{}, NULL, NULL};
node *mappings = &dummy;


void addMapping(void *addr, size_t len, int prot, int flags) {
    node *new = malloc(sizeof(node));
    new->entry.addr = addr;
    new->entry.len = len;
    new->entry.prot = prot;
    new->entry.flags = flags;
    new->next = NULL;

    node *current = mappings;
    while (current->next != NULL) current = current->next;

    current->next = new;
    new->prev = current;
}

int removeMapping(void *addr) {
    node *current = mappings->next;

    while (current != NULL) {

        if (current->entry.addr == addr) {
            current->prev->next = current->next;
            if (current->next != NULL) {
                current->next->prev = current->prev;
            }
            free(current);
            return 0;
        }

        current = current->next;
    }

    return -1;
}

```

```c
variantMMap *findMapping(void *targetAddr) {

    node *current = mappings->next;

    while (current != NULL) {

        if ((current->entry.addr == targetAddr && current->entry.len == 0) ||
            (current->entry.len > 0
             && targetAddr >= current->entry.addr
             && targetAddr < current->entry.addr + current->entry.len)) {

            return &current->entry;
        }

        current = current->next;
    }

    return NULL;
}

void printMappings() {
    node *current = mappings->next;

    int count = 0;
    debug_printf(stderr, "==== Mappings ===\n");
    while (current != NULL) {
        debug_printf(stderr, "[%d] addr: %p, len: %lu, prot: %d, flags: %d\n",
                     count++, current->entry.addr, current->entry.len,
                     current->entry.prot, current->entry.flags);
        current = current->next;
    }
```

security_manager.c

```c
#include "security_manager.h"
#include "debug.h"

#include <sys/mman.h>
#include <stdio.h>

int checkMMapPerms(int prot, int flags) {

    if ((prot & PROT_WRITE) && !(flags & MAP_PRIVATE)) {
        fprintf(stderr, "mmap: Shared writable mmap not supported\n");
        return -1;
    }

    return 0;
```

variant.c

```c
#include "variant.h"
#include "sys/syscall_util.h"
#include "debug.h"
#include "compile_options.h"

#include <sys/reg.h>
#include <sys/wait.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
```

```c
#include <sys/uio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <sched.h>
#include <sys/capability.h>
#include <linux/prctl.h>
#include <errno.h>
#include <assert.h>

/* General purpose buffers for variant memory operations */
#define BUF_SIZE 100000000
char bufA[BUF_SIZE];
char bufB[BUF_SIZE];

/*
 * Forks new variant and sets up ptrace.
 * PID of new variant is stored in *pid.
 */
void createVariant(pid_t *pid, char *argv[], long core) {

    if ((*pid = fork()) == 0) {

#ifdef USE_DIRECT_TRANSFER
        cap_t caps;
        cap_value_t cap_list[] = {CAP_SYS_PTRACE};
        if ((caps = cap_get_proc()) == NULL) {
            perror("cap_get_proc");
            exit(EXIT_FAILURE);
        }
        if (cap_set_flag(caps, CAP_PERMITTED, 1, cap_list, CAP_SET) == -1 ||
            cap_set_flag(caps, CAP_INHERITABLE, 1, cap_list, CAP_SET) == -1) {
            perror("cap_set_flag");
            exit(EXIT_FAILURE);
        }
        if (cap_set_proc(caps) == -1) {
            perror("cap_set_proc");
            exit(EXIT_FAILURE);
        }
        cap_free(caps);
        if (syscall(SYS_prctl, PR_CAP_AMBIENT, PR_CAP_AMBIENT_RAISE, CAP_SYS_PTRACE, 0, 0) == -1) {
            perror("prctl");
        }
#endif

        cpu_set_t mask;
        CPU_ZERO(&mask);
        CPU_SET(core, &mask);
        if (sched_setaffinity(getpid(), sizeof(cpu_set_t), &mask) == -1) {
            perror("sched_setaffinity");
            exit(EXIT_FAILURE);
        }

        ptrace(PTRACE_TRACEME, 0, 0, 0); /* Gives parent ptrace access */
        execvpe(argv[1], argv + 1, environ); /* argv + 1: skip first argument (path to MVEE program) */
        exit(EXIT_FAILURE);
    }
}

/*
 * Configures child process for ptrace.
 * PTRACE_O_EXITKILL kills child when parent dies.
 * PTRACE_O_TRACESYSGOOD for enhanced syscall tracing
 */
void configureChildProc(pid_t pid) {
    ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_EXITKILL);
    ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);
}

void printSyscallDebug(variant *v) {
    int isEntry = v->syscallInfo.op == PTRACE_SYSCALL_INFO_ENTRY;

    if (isEntry) {
        debug_printf(stderr, "%s Syscall: ENT \t%llu \t>>%s<<\n",
                     v->type == VTYPE_master ? "[M]" : "{R}",
                     v->syscallInfo.entry.nr,
                     getSyscallName(v->syscallInfo.entry.nr));
    } else {
        debug_printf(stderr, "%s Syscall: EXIT \t%lld\n",
                     v->type == VTYPE_master ? "[M]" : "{R}", v->syscallInfo.exit.rval);
    }
}

/*
 * Create and configure variant
 */
int initVariant(variant *v, int vtype, char *argv[], long core) {

    v->syscallNum = 0;
    v->type = vtype;
    createVariant(&v->pid, argv, core);
    waitpid(v->pid, NULL, __WALL); /* __WALL: waits for child regardless of its state */
    configureChildProc(v->pid);
    return 0;
}

/*
 * Wait for variant to reach syscall entry or exit
 * Returns 0 if successful, -1 if variant exited
 */
int waitVariant(variant *v) {
    int status;
    waitpid(v->pid, &status, __WALL);
    return checkVariantExit(v, &status);
}

/*
 * Check if variant exited -
 * returns 0 if variant is still running, -1 if variant exited.
 * If exited, prints exit status.
 */
int checkVariantExit(variant *v, const int *status) {

    if (WIFEXITED(*status)) {
        debug_printf(stderr, "%s exited with status %d\n",
                     v->type == VTYPE_master ? "master" : "replica", WEXITSTATUS(*status));
        return -1;
    }

    if (WIFSIGNALED(*status)) {
        debug_printf(stderr, "%s exited with signal %d\n",
                     v->type == VTYPE_master ? "master" : "replica", WTERMSIG(*status));
        return -1;
    }

    return 0;
}


```

```
/*
 * Resume variant and mark as running
 */
int resumeVariantFromWait(variant *v) {
    debug_printf(stderr, "\t %s resume from wait [%llu]\n",
                 v->type == VTYPE_master ? "[M]" : "{R}", v->syscallNum);
    ptrace(PTRACE_SYSCALL, v->pid, 0, 0);
    v->state = VSTATE_RUNNING;
    return 0;
}

/*
 * Resume variant and increment syscall number
 */
int resumeVariantFromExit(variant *v) {
    debug_printf(stderr, "\t %s resume from exit [%llu] = %lld\n",
                 v->type == VTYPE_master ? "[M]" : "{R}", v->syscallNum, v->syscallInfo.exit.rval);
    ptrace(PTRACE_SYSCALL, v->pid, 0, 0);
    v->syscallNum++;
    return 0;
}

/*
 * Runs master syscall from entry wait, runs replica through NOP syscall,
 * replicates master's return value to replica, and finally resumes both.
 */
int runMasterReplicateRet(variant *master, variant *replica) {

    resumeVariantFromWait(master);
    if (waitVariant(master) != 0) return -1;
    requestSyscallInfo(master);
    int64_t masterRVal = (int) master->syscallInfo.exit.rval;

    setNopSyscall(replica);
    resumeVariantFromWait(replica);
    if (waitVariant(replica) != 0) return -1;
    setSyscallReturnValue(replica, masterRVal);

    resumeVariantFromExit(master);
    resumeVariantFromExit(replica);

    return 0;
}

/*
 * Runs syscall in both variants and replicates master return value to replica.
 */
int runBothCopyMasterRet(variant *master, variant *replica) {

    resumeVariantFromWait(master);
    if (waitVariant(master) != 0) return -1;
    requestSyscallInfo(master);
    int64_t masterRVal = (int) master->syscallInfo.exit.rval;

    resumeVariantFromWait(replica);
    if (waitVariant(replica) != 0) return -1;
    setSyscallReturnValue(replica, masterRVal);

    resumeVariantFromExit(master);
    resumeVariantFromExit(replica);

    return 0;
}

```

```c

int setSyscall(variant *v, int call) {
    ptrace(PTRACE_POKEUSER, v->pid, 8 * ORIG_RAX, call);
}

int setNopSyscall(variant *v) {
    setSyscall(v, SYS_getpid);
    return 0;
}


int setSyscallReturnValue(variant *v, int64_t returnValue) {
    ptrace(PTRACE_POKEUSER, v->pid, 8 * RAX, returnValue);
    return 0;
}

/*
 * Read memory from remote variant buffer into local monitor buffer
 */
ssize_t readVariantMem(variant *v, void *localBuf, void *remoteBuf, size_t size) {

    if (size > BUF_SIZE) {
        debug_printf(stderr, "readVariantMem %zu size exceeds buffer size\n", size);
        return -1;
    }

#ifdef USE_PTRACE_TRANSFER
    size_t bytesRemaining = size;
    ssize_t bytesTransferred = 0;
    while (bytesRemaining > 0) {
        size_t bytesToTransfer = bytesRemaining > sizeof(long) ? sizeof(long) : bytesRemaining;

        ((long *) localBuf)[bytesTransferred / sizeof(long)] =
                (long) ptrace(PTRACE_PEEKDATA, v->pid, remoteBuf + bytesTransferred, 0);

        bytesRemaining -= bytesToTransfer;
        bytesTransferred += (ssize_t) bytesToTransfer;
    }
    return bytesTransferred;
#else
    struct iovec local_iov[1];
    local_iov[0].iov_base = localBuf;
    local_iov[0].iov_len = size;
    struct iovec remote_iov_master[1];
    remote_iov_master[0].iov_base = remoteBuf;
    remote_iov_master[0].iov_len = size;
    return process_vm_readv(v->pid, local_iov, 1, remote_iov_master, 1, 0);
#endif
}

/*
 * Write memory from local monitor buffer into remote variant buffer
 */
ssize_t writeVariantMem(variant *v, void *localBuf, void *remoteBuf, size_t size) {

#ifdef USE_PTRACE_TRANSFER
    ssize_t bytesRemaining = (ssize_t) size;
    ssize_t bytesTransferred = 0;
    if (size < sizeof(long)) {

        /* Smallest transfer size is 1 word, for writes smaller than this
         * Must get existing data beyond buffer to prevent POKE overwrite erasing it*/
        long data = (long) ptrace(PTRACE_PEEKDATA, v->pid, remoteBuf, 0);
        /* Apply effect of writing into buffer */
        memcpy(&data, localBuf, size);
        ptrace(PTRACE_POKEDATA, v->pid, remoteBuf, data);
        return (ssize_t) size;

    } else {
        while (bytesRemaining > 0) {

            if (bytesRemaining >= sizeof(long)) {  /* Write full word */
                ptrace(PTRACE_POKEDATA, v->pid, remoteBuf + bytesTransferred,
                       ((long *) localBuf)[bytesTransferred / sizeof(long)]);

                bytesRemaining -= sizeof(long);
                bytesTransferred += sizeof(long);

            } else { /* Write last word of data, counting from last byte */
                ptrace(PTRACE_POKEDATA, v->pid, remoteBuf + size - sizeof(long),
                       ((long *) localBuf)[size - sizeof(long)]);
                bytesTransferred += bytesRemaining;
                bytesRemaining = 0;
            }
        }
        return bytesTransferred;
    }
#endif
    struct iovec local_iov[1];
    local_iov[0].iov_base = localBuf;
    local_iov[0].iov_len = size;
    struct iovec remote_iov_master[1];
    remote_iov_master[0].iov_base = remoteBuf;
    remote_iov_master[0].iov_len = size;
    return process_vm_writev(v->pid, local_iov, 1, remote_iov_master, 1, 0);
}

int requestSyscallInfo(variant *v) {
    memset(&v->syscallInfo, 0, sizeof(struct ptrace_syscall_info));
    ptrace(PTRACE_GET_SYSCALL_INFO, v->pid, sizeof(struct ptrace_syscall_info), &v->syscallInfo);
    return 0;

}

/*
 * Transfer memory between variant buffers, through the monitor
 */
int transferVariantMem(variant *sourceV, void *sourceVBuf, variant *destV, void *destVBuf, size_t size)
↪ {

    if (size > BUF_SIZE) {
        debug_printf(stderr, "transferVariantMem %zu size exceeds buffer size\n", size);
        return -1;
    }

    ssize_t r = readVariantMem(sourceV, bufA, sourceVBuf, size);

    if (r != size) {
        debug_printf(stderr, "transferVariantMem read failed\n");
        return -1;
    }

    ssize_t w = writeVariantMem(destV, bufA, destVBuf, size);

    if (w != size) {
        debug_printf(stderr, "transferVariantMem write failed. Wrote %zd out of %zu\n", w, size);
        return -1;
    }
```

```c

    return 0;
}



/*
 * Read from remote variant buffers into local buffers and perform string compare
 * returns:
 *  -1 on error
 *  0 on equal
 *  1 on not equal
 */
int variantStrCmp(variant *v1, void *v1Buf, variant *v2, void *v2Buf) {

    memset(bufA, 0, BUF_SIZE);
    memset(bufB, 0, BUF_SIZE);

    size_t curSize = 100;
    size_t stringLen;

    while (curSize < BUF_SIZE) {

        readVariantMem(v1, bufA, v1Buf, curSize);
        stringLen = strnlen(bufA, curSize);

        if (stringLen == BUF_SIZE) {
            debug_printf(stderr, "variantStrCmp string exceeds buffer size\n");
            return -1;

        } else if (stringLen == curSize) {
            debug_printf(stderr, "variantStrCmp string too long, increasing...\n");
            curSize += 100;

        } else {
            debug_printf(stderr, "variantStrCmp max buffer size exceeded\n");
            return -1;
        }
    }

    readVariantMem(v2, bufB, v2Buf, curSize);
    int cmpRes = strncmp(bufA, bufB, stringLen);

    if (cmpRes != 0) {
        debug_printf(stderr, "variantStrCmp strings not equal: \n\t%s \n\t%s\n", bufA, bufB);
    }

    return cmpRes == 0 ? 0 : 1;
}

/*
 * Read from remote variant buffers into local buffers and perform memory compare
 */
int variantMemCmp(variant *v1, void *v1Buf, variant *v2, void *v2Buf, size_t size) {

    if (size > BUF_SIZE) {
        debug_printf(stderr, "variantMemCmp size exceeds buffer size\n");
        return -1;
    }

    readVariantMem(v1, bufA, v1Buf, size);
    readVariantMem(v2, bufB, v2Buf, size);

    return memcmp(bufA, bufB, size) == 0 ? 0 : 1;
```

```c
}


#ifdef USE_DIRECT_TRANSFER

int variantChangePtraceCap(variant *v, struct user_regs_struct *regs, int newState) {

    ptrace(PTRACE_GETREGS, v->pid, 0, regs);
    regs->rip -= 2;
    ptrace(PTRACE_SETREGS, v->pid, 0, regs);

    ptrace(PTRACE_SYSCALL, v->pid, 0, 0);
    if (waitVariant(v) != 0) return -1; // now at entry

    struct __user_cap_header_struct capheader;
    capheader.pid = v->pid;
    capheader.version = _LINUX_CAPABILITY_VERSION_3;
    struct __user_cap_data_struct capdata[2];
    if (capget(&capheader, capdata) == -1) {
        perror("capget");
        return -1;
    }

    /* Modify CAP_SYS_PTRACE in effective */
    if (newState == 0) {
        capdata[CAP_TO_INDEX(CAP_SYS_PTRACE)].effective &= ~CAP_TO_MASK(CAP_SYS_PTRACE);
    } else if (newState == 1) {
        capdata[CAP_TO_INDEX(CAP_SYS_PTRACE)].effective |= CAP_TO_MASK(CAP_SYS_PTRACE);
    } else {
        assert(0);
    }

    /* Transfer capheader and capdata to variant's stack after RSP */
    void* headerAddr = (void*) v->syscallInfo.stack_pointer - sizeof(struct __user_cap_header_struct);
    void* dataAddr = headerAddr - sizeof(struct __user_cap_data_struct) * 2;
    writeVariantMem(v, &capheader, headerAddr, sizeof(struct __user_cap_header_struct));
    writeVariantMem(v, &capdata, dataAddr, sizeof(struct __user_cap_data_struct) * 2);

    /* Setup syscall arguments
     * int syscall(SYS_capset, cap_user_header_t hdrp,
                   const cap_user_data_t datap);*/
    ptrace(PTRACE_GETREGS, v->pid, 0, regs);
    regs->orig_rax = SYS_capset;
    regs->rdi = (unsigned long) headerAddr;
    regs->rsi = (unsigned long) dataAddr;
    ptrace(PTRACE_SETREGS, v->pid, 0, regs);
//    setSyscall(v, SYS_capset);

    /* Execute syscall */
    ptrace(PTRACE_SYSCALL, v->pid, 0, 0);
    if (waitpid(v->pid, NULL, 0) == -1) {
        perror("CAPDROP: waitpid");
        return -1;
    }
    int retCapset = (int)ptrace(PTRACE_PEEKUSER, v->pid, sizeof(long) *RAX, 0);
    debug_printf(stderr, "variantChangePtraceCap: capset returned %d\n", retCapset);

    return 0;
}

int transferVariantMemDirect(variant *sourceV, void *sourceVBuf, variant *destV, void *destVBuf, size_t
↪ size) {

    struct user_regs_struct regs;
    if (variantChangePtraceCap(destV, &regs, 1) != 0) return -1;

    ptrace(PTRACE_GETREGS, destV->pid, 0, regs);
    regs.rip -= 2;
    ptrace(PTRACE_SETREGS, destV->pid, 0, regs);
    /* Get to syscall entry.*/
    ptrace(PTRACE_SYSCALL, destV->pid, 0, 0);
    if (waitVariant(destV) != 0) {
        return -1;
    }

    /* Create transfer details structs */
    struct iovec local_iov[1];
    local_iov[0].iov_base = destVBuf;
    local_iov[0].iov_len = size;

    struct iovec remote_iov_master[1];
    remote_iov_master[0].iov_base = sourceVBuf;
    remote_iov_master[0].iov_len = size;

    /* Transfer to dest variant stack */
    void* localAddr = (void*) destV->syscallInfo.stack_pointer - sizeof(struct iovec);
    void* remoteAddr = (void*) sourceV->syscallInfo.stack_pointer - sizeof(struct iovec) * 2;
    writeVariantMem(destV, &local_iov, localAddr, sizeof(struct iovec));
    writeVariantMem(destV, &remote_iov_master, remoteAddr, sizeof(struct iovec));

    /* Setup syscall arguments
     *        ssize_t process_vm_readv(pid_t pid,
                          const struct iovec *local_iov,
                          unsigned long liovcnt,
                          const struct iovec *remote_iov,
                          unsigned long riovcnt,
                          unsigned long flags);*/

    ptrace(PTRACE_GETREGS, destV->pid, 0, &regs);
    regs.orig_rax = SYS_process_vm_readv;
    regs.rdi = sourceV->pid;
    regs.rsi = (unsigned long) localAddr;
    regs.rdx = 1;
    regs.r10 = (unsigned long) remoteAddr;
    regs.r8 = 1;
    regs.r9 = 0;
    ptrace(PTRACE_SETREGS, destV->pid, 0, &regs);


    /* Execute syscall */
    ptrace(PTRACE_SYSCALL, destV->pid, 0, 0);
    if (waitpid(destV->pid, NULL, 0) == -1) {
        perror("transferVariantMemDirect: waitpid");
        return -1;
    }

    requestSyscallInfo(destV);
    if (destV->syscallInfo.exit.rval == -1) {
        perror("transferVariantMemDirect: process_vm_readv");
        return -1;
    } else if (destV->syscallInfo.exit.rval != size) {
        debug_printf(stderr, "transferVariantMemDirect: process_vm_readv read size mismatch\n");
        return -1;
    }

    if (variantChangePtraceCap(destV, &regs, 0) !=0) return -1;

    return 0;
```

```
}
```

monitor.c

```
#include "monitor.h"

#include "variant.h"
#include "sys/syscall_executors.h"
#include "debug.h"
#include "compile_options.h"

#include <sys/syscall.h>
#include <stdio.h>
#include <sys/ptrace.h>
#include <linux/ptrace.h>
#include <sys/wait.h>
#include <assert.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>
#include <sched.h>
#include <stdlib.h>
#include <sys/capability.h>
#include <sys/user.h>
#include <sys/reg.h>



#ifdef USE_DIRECT_TRANSFER

int firstCall = 1;


int dropVariantsCap(variant *v) {
    int originalCall = (int) v->syscallInfo.entry.nr;
    if (originalCall != SYS_brk) {
        fprintf(stderr, "CAPDROP full cover not implemented: initial syscall not 'brk' \n");
        return -1;
    }

    ptrace(PTRACE_SYSCALL, v->pid, 0, 0);
    if (waitVariant(v) != 0) return -1; // now at exit
    requestSyscallInfo(v);
    int ret = (int) v->syscallInfo.exit.rval;

    struct user_regs_struct regs;
    variantChangePtraceCap(v, &regs, 0);

    /* resume execution */
    setSyscallReturnValue(v, ret); // restore brk return value
    resumeVariantFromWait(v);

    return 0;

}

#endif


int processSyscallSync(variant *master, variant *replica) {

    if (master->syscallInfo.op != replica->syscallInfo.op) {
        debug_printf(stderr, "Syscall op mismatch\n");
```

```c
        return -1;
    }

    if (master->syscallInfo.entry.nr != replica->syscallInfo.entry.nr) {
        debug_printf(stderr, "Syscall numbers mismatch\n");
        return -1;
    }

#ifdef USE_DIRECT_TRANSFER
    if (firstCall) {
        firstCall = 0;
        dropVariantsCap(master);
        dropVariantsCap(replica);
        debug_printf(stderr, "Dropped capabilities\n");
        return 0;
    }
#endif

    int execResult = 0;

    switch (master->syscallInfo.entry.nr) {

        /* No arguments */
        case SYS_getuid:
        case SYS_geteuid:
        case SYS_getgid:
        case SYS_getegid:
            resumeVariantFromWait(master);
            resumeVariantFromWait(replica);
            break;

            /* Find syscall executor */
        case SYS_brk:
            execResult = execSyscall_brk(master, replica);
            break;
        case SYS_mmap:
            execResult = execSyscall_mmap(master, replica);
            break;
        case SYS_read:
            execResult = execSyscall_read(master, replica);
            break;
        case SYS_arch_prctl:
            execResult = execSyscall_arch_prctl(master, replica);
            break;
        case SYS_openat:
            execResult = execSyscall_openat(master, replica);
            break;
        case SYS_close:
            execResult = execSyscall_close(master, replica);
            break;
        case SYS_write:
            execResult = execSyscall_write(master, replica);
            break;
        case SYS_pread64:
            execResult = execSyscall_pread64(master, replica);
            break;
        case SYS_newfstatat:
            execResult = execSyscall_newfstatat(master, replica);
            break;
        case SYS_access:
            execResult = execSyscall_access(master, replica);
            break;
        case SYS_set_tid_address:
            execResult = execSyscall_setTidAddress(master, replica);
            break;
        case SYS_set_robust_list:
            execResult = execSyscall_setRobustList(master, replica);
            break;
        case SYS_mprotect:
            execResult = execSyscall_mprotect(master, replica);
            break;
        case SYS_munmap:
            execResult = execSyscall_munmap(master, replica);
            break;
        case SYS_rseq:
            execResult = execSyscall_rseq(master, replica);
            break;
        case SYS_fadvise64:
            execResult = execSyscall_fadvise64(master, replica);
            break;
        case SYS_getrandom:
            execResult = execSyscall_getrandom(master, replica);
            break;
        case SYS_exit_group:
            execResult = execSyscall_exit_group(master, replica);
            break;
        case SYS_prlimit64:
            execResult = execSyscall_prlimit64(master, replica);
            break;
        case SYS_rt_sigaction:
            execResult = execSyscall_rt_sigaction(master, replica);
            break;
        case SYS_rt_sigprocmask:
            execResult = execSyscall_rt_sigprocmask(master, replica);
            break;
        case SYS_unlink:
            execResult = execSyscall_unlink(master, replica);
            break;
        case SYS_utimensat:
            execResult = execSyscall_utimensat(master, replica);
            break;
        case SYS_fchown:
            execResult = execSyscall_fchown(master, replica);
            break;
        case SYS_fchmod:
            execResult = execSyscall_fchmod(master, replica);
            break;
        case SYS_lseek:
            execResult = execSyscall_lseek(master, replica);
            break;
        case SYS_getcwd:
            execResult = execSyscall_getcwd(master, replica);
            break;
        case SYS_getpid:
            execResult = execSyscall_getpid(master, replica);
            break;
        case SYS_chdir:
            execResult = execSyscall_chdir(master, replica);
            break;
        case SYS_socket:
            execResult = execSyscall_socket(master, replica);
            break;
        case SYS_setsockopt:
            execResult = execSyscall_setsockopt(master, replica);
            break;
        case SYS_bind:
            execResult = execSyscall_bind(master, replica);
            break;
        case SYS_listen:
            execResult = execSyscall_listen(master, replica);
            break;
        case SYS_accept4:
            execResult = execSyscall_accept4(master, replica);
            break;
        case SYS_poll:
            execResult = execSyscall_poll(master, replica);
            break;
        case SYS_shutdown:
            execResult = execSyscall_shutdown(master, replica);
            break;
        case SYS_getsockname:
            execResult = execSyscall_getsockname(master, replica);
            break;
        case SYS_pselect6:
            execResult = execSyscall_pselect6(master, replica);
            break;

        default:
            debug_printf(stderr, "##ERROR## Syscall %llu not implemented\n",
                         master->syscallInfo.entry.nr);

            return -1;
    }

    return execResult;
}


int processVariantEvent(variant *vSelf, variant *vOther) {

    requestSyscallInfo(vSelf);

    printSyscallDebug(vSelf);

    if (vSelf->syscallInfo.op == PTRACE_SYSCALL_INFO_ENTRY) {

        if (vSelf->syscallNum == vOther->syscallNum) {

            if (vOther->state == VSTATE_WAIT) {
                /* Partner has reached syscall entry and is waiting */

                variant *vmaster = vSelf->type == VTYPE_master ? vSelf : vOther;
                variant *vreplica = vSelf->type == VTYPE_master ? vOther : vSelf;

                int result;
                if ((result = processSyscallSync(vmaster, vreplica)) != 0) {
                    fprintf(stderr, "[MVEE] Syscall sync execution failed. Killing Variants.\n");
                    fprintf(stderr, "[MVEE]  strerror: %s\n", strerror(errno));
                    kill(vmaster->pid, SIGKILL);
                    kill(vreplica->pid, SIGKILL);
                    return result;
                }


            } else {
                /* Wait for other to reach syscall entry*/
                vSelf->state = VSTATE_WAIT;
            }

        } else if (vSelf->syscallNum > vOther->syscallNum) {
            /* Wait for other to fin prev syscall*/
            vSelf->state = VSTATE_WAIT;
            debug_printf(stderr, "\t %s waiting for partner\n",
                         vSelf->type == VTYPE_master ? "[M]" : "{R}");
        } else {
            fprintf(stderr, "[MVEE] Invalid State\n");
            exit(EXIT_FAILURE);
        }

    } else if (vSelf->syscallInfo.op == PTRACE_SYSCALL_INFO_EXIT) {
        /* Syscall exit (if not done by executor), resume */
        resumeVariantFromExit(vSelf);

    } else {
        fprintf(stderr, "[MVEE] Invalid State\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}


int main(int argc, char *argv[]) {


    if (argc < 1) {
        printf("Usage: ./MVEE <Program Path> <Arguments...>\n");
        return 1;
    }

    long systemCores = sysconf(_SC_NPROCESSORS_ONLN);
    if (systemCores == -1) {
        perror("sysconf");
        exit(EXIT_FAILURE);
    }
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(systemCores - 1, &mask);
    if (sched_setaffinity(getpid(), sizeof(cpu_set_t), &mask) == -1) {
        perror("sched_setaffinity");
        exit(EXIT_FAILURE);
    }

    variant master;
    initVariant(&master, VTYPE_master, argv, systemCores - 2);
    fprintf(stderr, "[MVEE] Master PID: %d\n", master.pid);
    resumeVariantFromWait(&master);

    variant replica;
    initVariant(&replica, VTYPE_replica, argv, systemCores - 3);
    fprintf(stderr, "[MVEE] Replica PID: %d\n", replica.pid);
    resumeVariantFromWait(&replica);

    while (1) {

        // Wait for any child
        int status;
        pid_t pid = waitpid(-1, &status, __WALL);


        if (pid == master.pid) {

            if (checkVariantExit(&master, &status) != 0) break;
            if (processVariantEvent(&master, &replica) != 0) {
                debug_printf(stderr, "Invalid State\n");
                exit(EXIT_FAILURE);
            }
```

```c


        } else if (pid == replica.pid) {

            if (checkVariantExit(&replica, &status) != 0) break;
            if (processVariantEvent(&replica, &master) != 0) {
                debug_printf(stderr, "Invalid State\n");
                exit(EXIT_FAILURE);
            }

        } else {
            debug_printf(stderr, "Unknown child %d", pid);
            exit(EXIT_FAILURE);
        }

    }

    fprintf(stderr, "[MVEE] Variants exited. Terminating.\n");
    exit(EXIT_SUCCESS);
}
```

mprotect.c

```c
#include "mprotect.h"
#include "../mappings.h"
#include "../security_manager.h"

int execSyscall_mprotect(variant *master, variant *replica) {
    /*
     * int mprotect(void *addr, size_t len, int prot);
     */

    int eq = 1;

    /* void *addr */
    /* Equivalence not required */
    void *targetAddr = (void *) master->syscallInfo.entry.args[0];

    /* size_t len */
    eq &= (master->syscallInfo.entry.args[1] == replica->syscallInfo.entry.args[1]);

    /* int prot */
    int newProt = (int) master->syscallInfo.entry.args[2];
    eq &= (newProt == replica->syscallInfo.entry.args[2]);

    if (eq) {

        variantMMap *map;
        if ((map = findMapping(targetAddr)) != NULL) {

            if (checkMMapPerms(newProt, map->flags) != 0) {
                return -1;
            }
        }

        resumeVariantFromWait(master);
        resumeVariantFromWait(replica);

        return 0;

    } else {

        return -1;
    }
```

write.c

```
#include "write.h"
#include "../debug.h"

#include <stdio.h>
#include <sys/syscall.h>
#include <sys/user.h>

int execSyscall_write(variant *master, variant *replica) {
    /*
     * ssize_t write(int fd, const void *buf, size_t count)
     */

    int eq = 1;

    /* int fd */
    int masterFd = (int) master->syscallInfo.entry.args[0];
    eq &= (masterFd == replica->syscallInfo.entry.args[0]);

    /* size_t count */
    size_t masterCount = (size_t) master->syscallInfo.entry.args[2];
    eq &= (masterCount == (size_t) replica->syscallInfo.entry.args[2]);

    /* void *buf */
    if (eq) {
        eq &= variantMemCmp(master, (void *) master->syscallInfo.entry.args[1], replica,
                            (void *) replica->syscallInfo.entry.args[1], masterCount) == 0;
    }

    if (!eq) {
        return -1;
    }

    /* Complete write in master */
    resumeVariantFromWait(master);
    if (waitVariant(master) != 0) return -1;
    requestSyscallInfo(master);
    ssize_t bytesWritten = master->syscallInfo.exit.rval;

    /* lseek in replica to update file position */
    struct user_regs_struct regs;
    ptrace(PTRACE_GETREGS, replica->pid, NULL, &regs);
    regs.rdi = masterFd; //unsigned int fd
    regs.rsi = bytesWritten; //off_t offset
    regs.rdx = SEEK_CUR; //unsigned int whence
    ptrace(PTRACE_SETREGS, replica->pid, NULL, &regs);
    setSyscall(master, SYS_lseek);

    resumeVariantFromWait(replica);
    if (waitVariant(replica) != 0) return -1;
    /* Set return value in replica*/
    setSyscallReturnValue(replica, bytesWritten);

    /* Resume from syscall exit */
    resumeVariantFromExit(master);
    resumeVariantFromExit(replica);

    return 0;
```

read.c

```c
#include "read.h"
#include "../debug.h"

#include <string.h>
#include <stdio.h>
#include <sys/user.h>
#include <sys/syscall.h>

int execSyscall_read(variant *master, variant *replica) {
    int eq = 1;

    /* int fd */
    int masterFd = (int) master->syscallInfo.entry.args[0];
    eq &= masterFd == replica->syscallInfo.entry.args[0];

    /* void *buf */
    void *masterBuf = (void *) master->syscallInfo.entry.args[1];
    void *replicaBuf = (void *) replica->syscallInfo.entry.args[1];
    /* Equality not required */

    /* size_t count */
    eq &= master->syscallInfo.entry.args[2] == replica->syscallInfo.entry.args[2];

    if (!eq) {
        return -1;
    }


    /* Complete read in master */
    resumeVariantFromWait(master);
    if (waitVariant(master) != 0) return -1;
    requestSyscallInfo(master);
    ssize_t bytesRead = master->syscallInfo.exit.rval;


    /* Copy read data to replica */
    if (bytesRead != -1) {

#ifdef USE_DIRECT_TRANSFER
        int ret = transferVariantMemDirect(master, masterBuf,
                                           replica, replicaBuf, bytesRead);
#else
        int ret = transferVariantMem(master, masterBuf, replica, replicaBuf, bytesRead);
#endif
        if (ret != 0) return -1;
    }

    /* lseek in replica to update file position */
    struct user_regs_struct regs;
    ptrace(PTRACE_GETREGS, replica->pid, NULL, &regs);
    regs.rdi = masterFd; //unsigned int fd
    regs.rsi = bytesRead; //off_t offset
    regs.rdx = SEEK_CUR; //unsigned int whence
    ptrace(PTRACE_SETREGS, replica->pid, NULL, &regs);
    setSyscall(replica, SYS_lseek);

    resumeVariantFromWait(replica);
    if (waitVariant(replica) != 0) return -1;

#ifndef USE_DIRECT_TRANSFER
    /* Set return value in replica*/
    setSyscallReturnValue(replica, bytesRead);
#endif

    /* Resume from syscall exit */
    resumeVariantFromExit(master);
    resumeVariantFromExit(replica);

    return 0;
}
```

poll.c

```c
#include <sys/poll.h>
#include "poll.h"

int execSyscall_poll(variant *master, variant *replica) {
    /*
     * int poll(struct pollfd *fds, nfds_t nfds, int timeout);
     */

    int eq = 1;

    /* nfds_t nfds */
    eq &= master->syscallInfo.entry.args[1] == replica->syscallInfo.entry.args[1];

    /* int timeout */
    eq &= master->syscallInfo.entry.args[2] == replica->syscallInfo.entry.args[2];

    /* struct pollfd *fds */
    void* masterFds = (void*) master->syscallInfo.entry.args[0];
    void* replicaFds = (void*) replica->syscallInfo.entry.args[0];
    size_t fdsSize = master->syscallInfo.entry.args[1] * sizeof(struct pollfd);
    if (eq) {
        eq &= variantMemCmp(master, masterFds,
                            replica, replicaFds,
                            fdsSize) == 0;
    }

    if (!eq) {
        return -1;
    }

    /* Run in master */
    resumeVariantFromWait(master);
    if (waitVariant(master) == -1) return -1;
    requestSyscallInfo(master);

    /* NOP in replica */
    setNopSyscall(replica);
    resumeVariantFromWait(replica);
    if (waitVariant(replica) == -1) return -1;

    /* Copy results back to replica */
    int ret = transferVariantMem(master, masterFds, replica, replicaFds,
                                 fdsSize);
    if (ret != 0) return -1;

    /* Set return value */
    setSyscallReturnValue(replica, master->syscallInfo.exit.rval);

    resumeVariantFromExit(master);
    resumeVariantFromExit(replica);

    return 0;
```

socket.c

```c
#include "socket.h"

int execSyscall_socket(variant *master, variant *replica) {
    /*
     * int socket(int domain, int type, int protocol);
     */

    int eq = 1;

    /* int domain */
    eq &= master->syscallInfo.entry.args[0] == replica->syscallInfo.entry.args[0];

    /* int type */
    eq &= master->syscallInfo.entry.args[1] == replica->syscallInfo.entry.args[1];

    /* int protocol */
    eq &= master->syscallInfo.entry.args[2] == replica->syscallInfo.entry.args[2];

    if (eq) {
        resumeVariantFromWait(master);
        resumeVariantFromWait(replica);

        return 0;
    } else {
        return -1;
    }
```

mmap.c

```c
#include "mmap.h"
#include "../mappings.h"
#include "../security_manager.h"

#include <sys/mman.h>
#include <stdio.h>

int execSyscall_mmap(variant *master, variant *replica) {
    /* void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset */

    int eq = 1;

    /* void *addr */
    /* Equality not required TODO ?*/

    /* size_t length */
    eq &= master->syscallInfo.entry.args[1] == replica->syscallInfo.entry.args[1];

    /* int prot */
    int masterProt = (int) master->syscallInfo.entry.args[2];
    eq &= masterProt == replica->syscallInfo.entry.args[2];

    /* int flags */
    int masterFlags = (int) master->syscallInfo.entry.args[3];
    eq &= masterFlags == replica->syscallInfo.entry.args[3];

    /* int fd */
    eq &= master->syscallInfo.entry.args[4] == replica->syscallInfo.entry.args[4];

    /* off_t offset */
    eq &= master->syscallInfo.entry.args[5] == replica->syscallInfo.entry.args[5];

    if (!eq) return -1;

    if (checkMMapPerms(masterProt, masterFlags) != 0) {
        return -1;
    }

    resumeVariantFromWait(master);
    if (waitVariant(master) != 0) return -1;
    requestSyscallInfo(master);
    void *mappedAdrmaster = (void *) master->syscallInfo.exit.rval;
    resumeVariantFromExit(master);

    addMapping(mappedAdrmaster,
               master->syscallInfo.entry.args[1],
               masterProt,
               masterFlags);

    resumeVariantFromWait(replica);

    return 0;
```

munmap.c

```c
#include <stdio.h>
#include "munmap.h"
#include "../mappings.h"
#include "../debug.h"

int execSyscall_munmap(variant *master, variant *replica) {
    /*
     * int munmap(void *addr, size_t length);
     */

    int eq = 1;

    /* void *addr */
    /* Equality not required */
    void *targetAddr = (void *) master->syscallInfo.entry.args[0];

    /* size_t length */
    eq &= master->syscallInfo.entry.args[1] == replica->syscallInfo.entry.args[1];

    if (!eq) {
        return -1;
    }

    variantMMap *map;
    if ((map = findMapping(targetAddr)) != NULL) {
        if (map->addr != targetAddr) {
            debug_printf(stderr, "munmap: TODO partial unmap handling not implemented.\n");
            debug_printf(stderr, "\tTarget: %p, in containing: %p at offset %ld\n",
                         targetAddr, map->addr, targetAddr - map->addr);
            printMappings();
            return -1;
        }
        removeMapping(map->addr);

    } else {
        debug_printf(stderr, "munmap Unexpected Error: no mapping found for address %p\n", targetAddr);
        printMappings();
        return -1;
    }

    resumeVariantFromWait(master);
    resumeVariantFromWait(replica);

    return 0;


```

D.2 Benchmark Scripts

Test file generator

```python
from random import randint

file_sizes = ["1_000", "10_000", "100_000", "1_000_000",
              "10_000_000", "100_000_000", "1_000_000_000"]

for size in file_sizes:
    filename = f"random_{size}.txt"
    with open(filename, "w") as f:
        for _ in range(int(size)):
            f.write(chr(ord('A') + randint(0, 25)))
        f.write('\n')
    print(filename, "done")
```

Cat/GZip Script

```bash
#!/bin/bash

# Cat
for file in random_*.txt; do
    echo "Testing $file..."

    echo "Baseline cat:"
    sudo perf stat -e task-clock -r 10 cat "$(pwd)/$file" 1>/dev/null
    echo "MVEE cat:"
    sudo perf stat -e task-clock -r 10 ../../cmake-build-debug/MVEE cat "$(pwd)/$file" 1>/dev/null
done

# Gzip
for file in random_*.txt; do
    echo "Testing $file..."

    echo "Baseline gzip:"
    sudo perf stat -e task-clock -r 10 gzip -k -f "$(pwd)/$file"
    echo "MVEE gzip:"
    sudo perf stat -e task-clock -r 10 ../../cmake-build-debug/MVEE gzip -k -f "$(pwd)/$file"
done

# Cleanup
rm ./*.gz
```