TOROS UNIVERSITY

ENGINEERING FACULTY

DEPARTMEN OF SOFTWARE ENGINEERING

SWE403

GRADUATION THESIS

By

21506008 Ulaş Tan Ersoyak

Advisor

Asst. Prof. Mehmet Ali Aktaş

Abstract

This thesis presents Process Processor (pp) , a tool for analyzing and modifying running programs on Linux systems. pp operates through a command-line interface, providing an interaface for process manipulation and analysis. The tool enables direct interaction with running processes through features such as memory inspection, runtime code modification, thread monitoring, and code disassembly.

At its core, pp provides fundamental operations like reading from and writing to process memory, viewing memory maps, and modifying memory permissions. These basic operations form the foundation for more complex features. The tool's functionality extends to process memory analysis, allowing users to examine how programs utilize their memory space, organize their functions, and understand their machine code through disassembly.

Advanced features of pp include dynamic code manipulation capabilities. The tool implements custom code injection into running processes, though this feature currently exhibits stability issues that require further development. pp supports shared library injection, enabling the introduction of new functionality into running programs. Additionally, the tool provides pattern searching and replacement functionality within process memory, allowing for runtime data modification. The disassembly feature helps users verify code modifications and understand program structure at the instruction level.

The implementation leverages various Linux system interfaces, including the /proc filesystem for process information gathering and the ptrace system call for process control. While pp successfully demonstrates basic process manipulation capabilities, it remains an experimental tool designed for learning and exploration rather than production use. The tool's development provides practical insights into Linux process management, memory organization, machine code analysis, and runtime program modification techniques.

Table of Contents

Chapter 1: Introduction

1.1 Background

In modern computing systems, understanding program behavior at runtime is essential for software development, debugging, and system analysis. Runtime program analysis provides developers and system administrators with critical insights into how programs utilize system resources, interact with the operating system, and manage their internal state. Linux systems offer sophisticated mechanisms for examining and interacting with running programs through kernel interfaces and specialized filesystems.

The analysis of running programs involves several fundamental aspects of operating system design. Process memory inspection reveals how programs organize their data and code in virtual memory space. System call interfaces provide controlled access to kernel functionality, enabling safe interaction between processes. The proc filesystem serves as a window into kernel data structures, exposing detailed information about process state and resource utilization.

These capabilities form the foundation for various software development and system administration tools. Debuggers leverage process control interfaces to examine program state, memory analysis tools track resource usage patterns, and monitoring utilities observe system behavior. Understanding these mechanisms requires practical implementation experience with Linux system programming interfaces.

1.2 Problem Statement

The complexity of modern software systems necessitates tools for runtime program analysis and modification. While existing tools provide debugging and monitoring capabilities, implementing such tools offers valuable insights into operating system internals and program behavior. This project addresses the need for practical experience with Linux process manipulation mechanisms through the development of an experimental analysis tool.

The implementation explores several critical areas of system programming. Process memory management mechanisms demonstrate how the Linux kernel isolates and protects program memory spaces. Function and variable organization within programs reveals compiler-level program structure decisions. System call interfaces illustrate kernel boundary crossing and privilege management. Assembly-level program analysis exposes low-level execution details and control flow patterns.

Understanding these concepts requires hands-on implementation experience with Linux system interfaces. This project provides this experience through the development of pp, focusing on learning and exploration rather than creating a production-ready tool. The implementation demonstrates practical application of system programming concepts while highlighting technical challenges in runtime program manipulation.

1.3 Objectives

The project explores Linux program internals through an experimental tool called pp, focusing on learning through implementation rather than creating production-ready software. The primary goal is to understand and manipulate program behavior at runtime through direct memory access and process control mechanisms.

1.3.1 Memory Access and Manipulation

Memory access and manipulation form the foundation of the project's objectives. This involves developing methods for reading and writing process memory while understanding the underlying protection mechanisms. Through these operations, the project explores how the Linux kernel manages memory isolation between processes and enforces memory access permissions at the hardware level. The implementation provides practical experience with memory region permissions, process isolation mechanisms, and the kernel's memory protection features.

1.3.2 Program Analysis Features

Program analysis capabilities extend beyond basic memory operations to examine program structure and function management. This includes techniques for locating functions in running programs and understanding how programs store and organize their executable code. The project places particular emphasis on function name demangling, especially for C++ programs where function names contain complex type information.

The analysis features encompass several specialized areas of program manipulation. Dynamic code injection serves as a practical example of runtime program modification, though this feature highlights the challenges of proper stack alignment and memory permission handling. The work provides insights into how programs execute injected code and manage runtime modifications.

Library injection capabilities explore the Linux dynamic linking system. This includes understanding library loading mechanisms, dependency management, and how programs interact with loaded libraries at runtime. The implementation demonstrates practical aspects of shared library manipulation and runtime library loading.

Memory pattern operations and process information gathering form another key component of the analysis features. These involve developing methods for memory searching and data replacement while understanding various memory region types. For process monitoring, the implementation focuses on thread management, memory usage analysis, and examination of process hierarchies and thread states.

The command interface development prioritizes user interaction design, creating an intuitive set of commands with clear parameter handling and informative error reporting. This aspect of the project makes complex system operations accessible through straightforward commands, ensuring that advanced features remain usable despite their technical complexity.

Throughout these objectives, pp serves as a practical platform for experimenting with Linux program internals, acknowledging that certain features, particularly code injection, remain in an experimental state. The focus remains on learning and exploration rather than creating a production-ready debugging tool.

Chapter 2: Technical Background

2.1 Process Memory Architecture

Process memory architecture defines how Linux organizes and protects program memory spaces. The implementation includes process isolation through virtual memory, memory region management, and permission control mechanisms. These components work together through kernel interfaces like the proc filesystem and memory-related system calls, enabling process analysis and modification capabilities.

2.1.1 Proc Folder

The /proc directory in Linux is a virtual filesystem that provides detailed information about the system and running processes. It is mounted at /proc and contains a variety of subdirectories and files that give insights into the kernel's state, system parameters, and process-specific information. Each running process has its own directory in /proc, named by its Process ID (PID), which contains files and subdirectories representing various aspects of the process.

One of the most important files for process memory analysis is /proc/[pid]/mem, which represents the virtual memory of a process as the program sees it. This file allows external tools and debuggers to read and manipulate the process's memory directly. For instance, reading from /proc/[pid]/mem provides an exact snapshot of a process’s memory space, offering raw access to the process's memory layout, including all its regions, stack, heap, and loaded libraries.

Another crucial file for understanding a process's memory layout is /proc/[pid]/maps. This file lists the memory regions the process has mapped into its address space, displaying details such as the start and end addresses of each region, its permissions, and the associated file or device, if any. A typical line in /proc/[pid]/maps may look like this:

7f8b24c45000-7f8b24e45000 r-xp 00000000 08:01 2097153 /lib/x86_64-linux-gnu/libc-2.31.so

This line provides the following information:

- **Virtual address range**: 7f8b24c45000 to 7f8b24e45000
- **Permissions**: r-xp (read, execute, private)
- **File offset**: 00000000 (where in the file this part starts)
- **Device**: 08:01 (major:minor device numbers)
- **Inode**: 2097153 (unique file number)
- **Path**: /lib/x86_64-linux-gnu/libc-2.31.so (the file mapped into memory)

By examining the /proc/[pid]/maps file, tools like debuggers can analyze how a program's memory is laid out, identifying key regions such as the stack, heap, and loaded shared libraries. This information is essential for tasks such as memory inspection, debugging, and manipulation, allowing for deeper insights into the internal workings of a process.

2.1.2 Virtual Memory Management

Linux gives each program its own virtual memory space up to 2^48 bytes (256 terabytes) on modern x86_64 systems. This space is split into two main parts:

- 0000000000000000 - 00007fffffffffff: Program (user) space
- fff8000000000000 - ffffffffffffffff: Kernel space

When a program tries to use memory, the Memory Management Unit (MMU) converts the program's virtual address into a real physical memory address.

This happens through four levels of page tables:

1. PGD (Page Global Directory):
    - Handles the highest bits of the address (47-39)
    - Points to the next level (PUD)

2. PUD (Page Upper Directory):
    - Uses address bits 38-30
    - Points to the PMD level
3. PMD (Page Middle Directory):
    - Uses address bits 29-21
    - Points to the final table (PTE)
4. PTE (Page Table Entry):
    - Uses address bits 20-12
    - Points to the actual memory page
    - The last 12 bits (11-0) show the exact byte in the page

Memory is split into pages of 4096 bytes (0x1000) each. pp can read this memory through the /proc/[pid]/mem file, which shows memory exactly as the program sees it.

2.1.3 Memory Mapping and Permissions

Memory in Linux programs is organized into regions, each with its own permissions. These permissions control what can be done with each memory region:

Permission Bits

- r (read) - 0x4: Memory can be read
- w (write) - 0x2: Memory can be changed
- x (exec) - 0x1: Code can be run from this memory

Additional Memory Flags

- p (private): Changes affect only this program
- s (shared): Changes can be seen by other programs
- t (CoW): Copy-on-Write, makes a private copy when changed

Common Memory Region Types

- Code regions are marked r-x (read and execute)
- Data regions are marked rw- (read and write)
- Stack is marked rw- (read and write)
- Some regions are marked --- (no access allowed)

Permission Enforcement

Linux enforces these permissions at the hardware level. If a program tries to:

- Write to read-only memory: Gets SIGSEGV
- Execute non-executable memory: Gets SIGSEGV
- Read from no-access memory: Gets SIGSEGV
-

2.1.4 Memory Allocation

In Linux systems, processes can request additional memory while they are running through a shared pool of system memory that Linux manages. The primary mechanism for this allocation is the mmap system call, which creates new memory regions in a process's address space.

Memory allocation provides several key capabilities for running processes. Programs can expand their available space for data storage, create new areas for dynamic code execution, establish shared memory regions for inter-process communication, and map files directly into their address space for efficient access. These capabilities make memory allocation a fundamental component of dynamic program behavior.

When requesting memory allocation, processes maintain significant control over the allocated regions. A process specifies the desired amount of memory and sets specific attributes for the region, including access permissions and sharing characteristics. The process can also request memory at specific addresses, though the kernel ultimately determines the final allocation location based on availability and system constraints.

The kernel handles the technical aspects of memory allocation through a series of operations. Upon receiving an allocation request, it searches the process's virtual address space for a suitable free region that meets the size and alignment requirements. Once found, the kernel updates the process's memory maps to include the new region and establishes the necessary page table entries to manage the memory. Finally, it returns the starting address of the allocated region to the process.

This memory allocation mechanism forms the foundation for dynamic memory management in Linux systems, enabling processes to adapt their memory usage based on runtime requirements. The flexibility of this system allows programs to efficiently manage their resources while maintaining memory protection and isolation between processes.

2.2 ELF File Format

The ELF (Executable and Linkable Format) is the standard file format for Linux executables, shared libraries, and object files. ELF files contain both data and metadata that tell the system how to load and run the program. This format provides a flexible and efficient way to organize program components.

2.2.1 File Structure

An ELF file's organization consists of three main components that work together to define the program's structure. The ELF Header contains fundamental information about the file type and organization, serving as a map for the rest of the file. Program Headers provide essential instructions for loading the program into memory, while Section Headers organize different types of program data, enabling efficient access to specific program components.

2.2.2 Program Sections

ELF files organize program content into specialized sections, each serving a specific purpose in the program's operation. The .text section houses the program's executable code, while the .data section contains initialized program variables. For memory efficiency, uninitialized variables are tracked in the .bss section. Program symbols and their names are managed through the .symtab and .strtab sections respectively. For dynamic linking support, .dynstr and .dynsym sections maintain information about shared library functions.

2.2.3 Symbol Tables

Symbol tables in ELF files serve as directories for locating functions and variables within the program. The Static Symbol Table (.symtab) maintains a comprehensive list of all program symbols, while the Dynamic Symbol Table (.dynsym) specifically tracks symbols used in shared library interactions. Each symbol entry contains detailed information including its size, type, and location, enabling efficient symbol resolution during program execution.

2.2.4 Segments

During program loading, the ELF loader groups related sections into segments based on their runtime requirements. The Text Segment combines code and read-only data, ensuring program instructions remain unmodified during execution. Writable program data resides in the Data Segment, allowing for runtime modification. The Dynamic Segment maintains information necessary for dynamic linking operations. Each segment maintains its own specific memory permissions and alignment requirements, ensuring proper memory protection and efficient memory access.

These segments provide a practical organization for program memory layout, balancing the needs for memory protection, efficient execution, and runtime flexibility. The segmentation system allows the program loader to efficiently map the file contents into memory while maintaining appropriate access controls and memory organization.

2.3 Dynamic Code Modification

Dynamic code modification enables runtime changes to program behavior through direct memory manipulation. The implementation involves function redirection, code injection, and memory permission management. These techniques allow program modification without changing the original executable, though they require careful handling of memory permissions and code execution states.

2.3.1 Function Hooking Techniques

Function hooking is a technique that changes how a program's functions work by redirecting them to different code while the program is running. This allows changing program behavior without modifying the original executable file. Through function hooking, developers can monitor function calls, modify program behavior, and add new functionality to existing programs while they run. The technique also enables interception and modification of function parameters, providing powerful capabilities for program analysis and modification.

The fundamental concepts behind function hooking involve code redirection, function interception, memory modification, and control flow alteration. These elements work together to enable runtime program modification. As an advanced technique, function hooking requires careful attention to memory management, program execution flow, system permissions, and code compatibility to maintain program stability.

2.3.2 Shared Library Injection

Shared library injection provides an alternative approach to runtime program modification. Unlike function hooking, which modifies existing program behavior by redirecting functions, shared library injection introduces new code through loadable library files. This approach modifies program functionality without altering existing code structures.

The key distinction lies in their operational methods. Function hooking directly modifies existing program functions, requiring exact function locations and making permanent changes to function code. In contrast, shared library injection works through the program's native library loading system, adding new functions while leaving original program code intact. This approach allows modifications to be reversed by simply unloading the library.

The injection process follows the program's standard library loading mechanisms. When a library is injected, it becomes available to the program through normal library loading channels. This integration makes all the library's functions accessible while maintaining the integrity of existing program functions. The use of standard loading mechanisms contributes to greater stability compared to function hooking, as it preserves the program's memory layout and avoids direct code modification.

2.3.3 Memory Permission Handling

Memory permissions serve as a fundamental protection mechanism in program execution, controlling access to different memory regions. In the context of dynamic code modification, understanding and managing these permissions becomes crucial. Code execution requires executable memory permissions, while data modifications need write permissions. Some memory regions are intentionally protected, and permission changes can enable or prevent specific operations.

The operating system enforces these permissions as a security measure, preventing accidental code overwrites and protecting against code injection attacks. This protection mechanism ensures safe memory sharing between processes and protects sensitive data from unauthorized modification.

When implementing dynamic code modifications, special attention must be paid to memory permissions during operations like adding new code, modifying existing code, loading libraries, and managing shared memory. These operations often require careful manipulation of memory permissions while maintaining system security and stability.

2.4 Code Disassembly

Disassembly plays a critical role in program analysis by converting machine code back into assembly language. This translation process reveals the low-level operation of programs, converting raw executable bytes into human-readable assembly instructions. For instance, machine code sequences like 48 89 C7 and FF E0 translate to meaningful assembly instructions such as 'mov rdi, rax' and 'jmp rax' respectively.

The disassembly process supports multiple aspects of program analysis. It enables developers to understand program behavior, analyze function implementation, locate specific instruction patterns, and verify code modifications. This capability proves particularly valuable in debugging and program analysis tasks.

The x86-64 architecture's use of variable-length instructions, ranging from 1 to 15 bytes, makes accurate disassembly especially important. This variability impacts several aspects of program analysis, including the identification of function boundaries, pattern location in code, control flow analysis, and verification of code integrity. Proper disassembly becomes essential for understanding and working with program code at the instruction level.

Chapter 3: Design and Implementation

3.1 Architecture Overview

pp's architecture is organized into distinct modules that work together to handle process manipulation and analysis. The design separates core functionality from supporting features, creating a modular system that facilitates maintenance and expansion.

3.1.1 Core Components

The core components form the fundamental operational base of pp. The Process Management module (process) serves as the central coordinator, representing target processes and managing their state and information. It maintains control over process functions and handles the organization of memory regions, providing a unified interface for process manipulation.

3.1.1.1 Memory Handling

Memory Handling (memory_region) focuses on the intricate details of process memory management. This module defines memory regions and their attributes, implementing controls for memory permissions and managing I/O operations. It provides a comprehensive representation of memory areas within processes, ensuring proper memory access and modification.

3.1.1.2 Debugging Interface

The Debugging Interface (debugger) implements higher-level process manipulation features. This component coordinates process control operations, manages memory allocation, and implements advanced features such as function hooking and library injection. It also provides access to process registers, enabling detailed state examination and modification.

3.1.1.3 Thread Management

Thread Management (thread) handles the complexities of multi-threaded program analysis. This module manages thread states, provides access to thread-specific information, and enables register manipulation at the thread level. It works closely with other core components to maintain consistent process state during analysis.

3.1.2 Supporting Components

Supporting components enhance the core functionality with user interface and utility features. The Command Interface (cli) manages user interaction through command parsing and error handling, providing a consistent interface for accessing pp's capabilities.

Code Analysis (disassembler) provides essential program examination features through instruction decoding and analysis. This module converts machine code into readable assembly representations, enabling detailed examination of program behavior at the instruction level.

The Utility Functions module (util) supplies various support operations that other components rely on. It handles file operations, performs name demangling for C++ symbols, manages ELF file processing, and provides various helper functions used throughout the system. These utilities ensure consistent handling of common operations across all components.

This modular architecture enables pp to maintain separation of concerns while providing comprehensive process analysis capabilities. The interaction between core and supporting components creates a flexible system capable of handling complex process manipulation tasks while remaining maintainable and extensible.

3.2 Process Access and Memory Operations

Modern operating systems provide multiple mechanisms for processes to inspect and understand the memory layout of other processes. pp implements a set of features that leverage these mechanisms, particularly focusing on Linux's /proc filesystem and process inspection interfaces. These analysis capabilities range from examining memory maps and calculating usage statistics to inspecting detailed process and thread states. Through these features, users can gain deep insights into a process's memory structure, resource utilization, and runtime behavior. The following sections explore each analysis feature in detail, examining both their implementation and practical applications in debugging and process analysis.

3.2.1 Process Identification

Linux systems store process information in the /proc filesystem, organizing processes through a directory structure where each running process has its own directory named after its Process ID (PID). pp implements a systematic approach to identify and validate target processes within this filesystem structure.

pp scans through the directory entries, focusing specifically on directories with numeric names that represent running processes. Each of these numeric directories could indicate a target process that needs to be analyzed.

Once potential process directories are identified, pp performs detailed validation of each candidate. For each PID directory discovered, the system reads the corresponding /proc/[pid]/comm file to obtain the process name. This name is then compared against the target program name specified by the user. Additionally, the system verifies that the process is accessible for analysis, ensuring that pp has the necessary permissions to interact with it.

The final phase of process identification involves comprehensive result processing. pp aggregates information about all processes that match the specified criteria, supporting scenarios where multiple instances of the target program might be running simultaneously. For each matching process, the system collects and organizes relevant process details that will be needed for subsequent analysis operations. If no matching processes are found, pp generates an appropriate error message to inform the user.

This systematic process identification mechanism serves as the foundation for all subsequent process manipulation operations in pp. By ensuring accurate and thorough target process selection, the system establishes a reliable base for memory analysis and modification operations. The careful validation and error handling in this phase help prevent issues that could arise from attempting to analyze incorrect or inaccessible processes.

3.2.2 Basic Memory Operations

Process memory manipulation requires several fundamental operations that form the foundation of more complex features. These operations include reading from and writing to process memory, searching and replacing patterns in memory, and modifying memory region permissions. Each operation utilizes specific Linux system calls and mechanisms to ensure safe and efficient memory access between processes.

3.2.2.1 Memory Reading and Writing

Linux provides two specialized system calls, process_vm_readv and process_vm_writev, for direct inter-process memory access. These system calls are fundamental to how pp interacts with target process memory. Unlike traditional memory access mechanisms that require data to pass through kernel space buffers, these calls create a direct path between process address spaces.

The process_vm_readv operation performs direct memory reads from a target process. It takes a source address from the target process's memory space and transfers the data directly to a specified address in the calling process's memory space. This direct transfer mechanism not only improves performance by eliminating unnecessary copying but also ensures memory safety by maintaining process boundaries and permission checks.

The process_vm_writev operation works similarly but in reverse, writing data directly from the calling process's memory space into the target process's memory. The direct nature of these transfers ensures that memory modifications are as atomic as possible, reducing the risk of partial writes or inconsistent memory states. Both operations respect memory permissions and process boundaries, throwing errors if access violations occur.

When these operations fail, they can indicate various conditions such as invalid memory addresses, insufficient permissions, or process state issues. This error reporting helps maintain memory safety and system stability during memory manipulation operations.

3.2.2.2 Pattern Searching and Replacement

Pattern operations in pp provide mechanisms for locating and modifying specific sequences of bytes in process memory. The system handles both hexadecimal byte patterns and ASCII text patterns.

3.2.2.3 Pattern Search Implementation

The search operation traverses the memory regions of a target process to locate specific byte patterns. When examining a process, the system first obtains a complete map of memory regions. For each readable region, the operation buffers the region's contents locally for pattern matching.

Pattern matching works differently based on the input format. For hexadecimal patterns, the system interprets pairs of characters as byte values, creating a sequence for matching. ASCII patterns are converted directly to their byte representations. The matching algorithm moves through each memory buffer sequentially, identifying all occurrences of the target pattern.

When a match is found, the system calculates its true address within the process's memory space by combining the region's base address with the pattern's offset in the buffer. This translation ensures that reported addresses accurately reflect locations in the target process's memory space rather than the local buffer.

3.2.2.4 Pattern Replacement

Pattern replacement extends the search functionality by modifying found patterns in memory. The operation requires two patterns: one to search for and another to replace matches with. The system supports pattern input in both hexadecimal and ASCII formats, with hexadecimal allowing precise byte-level control.

The replacement algorithm first ensures pattern length compatibility. If the replacement pattern is shorter than the search pattern, it is extended with space characters to match the original length. This padding prevents unintended memory alterations that could occur from size mismatches.

The system processes memory regions that have both read and write permissions. For each candidate region, the algorithm reads the memory content, identifies matches, and performs replacements at each matched location. The operation can be limited to a specific number of replacements, allowing controlled modification of memory content.

3.2.2.5 Memory Safety Considerations

Memory operations incorporate several safety mechanisms to maintain system stability. The system verifies memory permissions before each operation, ensuring that searches only occur in readable regions and replacements only in regions with both read and write access. When memory access fails for any region, the system continues with subsequent regions, ensuring partial failures don't prevent the overall operation from completing.

Pattern operations carefully maintain memory boundaries and region integrity. The system tracks the scope of each memory region to prevent pattern matching or replacement from crossing region boundaries. This boundary awareness ensures that operations remain confined to their intended memory areas and prevents accidental modification of adjacent memory regions.

3.2.2.6 Memory Permission Modification

Memory permission modification in Linux systems operates through the mprotect system call. Since processes can only modify their own memory permissions, pp injects the mprotect system call directly into the target process. This is achieved by writing an instruction sequence into an executable region of the target process.

The instruction sequence consists of a syscall (0F 05), followed by an int3 breakpoint (CC), and padding with three nop instructions (90 90 90). Before execution, the process registers are set up with specific values: rax receives

the mprotect syscall number (0x0A), rdi gets the target memory address, rsi holds the region size, and rdx contains the desired permission flags.

When executed, this sequence triggers the mprotect system call in the target process's context, modifies the permissions, then hits the breakpoint instruction. The breakpoint returns control to pp, allowing it to restore the original instructions and register state. This mechanism ensures the target process performs the permission modification on its own memory, adhering to Linux security mechanisms.

3.2.2.7 Region Allocation

Linux systems use the mmap (memory map) system call as their fundamental mechanism for memory allocation. While smaller allocations typically use the heap through malloc, larger memory allocations and memory-mapped files rely directly on mmap. This system call creates new mappings in a process's virtual address space, either backed by files or as anonymous memory regions. Anonymous mappings, relevant for memory allocation, create private memory areas that are not associated with any file.

When mmap creates a new memory region, it can either place it at a specific address requested by the process or allow the kernel to choose a suitable location in the process's address space. The call provides fine-grained control over the mapping's properties, including read, write, and execute permissions, as well as whether the mapping should be private to the process or shared with other processes.

pp implements memory allocation by injecting the mmap system call into target processes. Similar to other injection operations, this requires executing system calls within the target process's context, as processes can only allocate memory for themselves under Linux security mechanisms.

The allocation process begins by locating an executable memory region within the target process. This region serves as the staging area for the injected allocation code. The implementation preserves the original contents of this region by reading them before modification. The instruction pointer (rip) is set to the executable region to execute the injected code sequence.

pp injects an instruction sequence that leverages the mmap system call to perform the actual allocation. This sequence consists of a syscall instruction (0F 05), followed by an int3 breakpoint (CC), and three nop instructions (90 90

90) for padding. The registers are configured according to the syscall convention: rax receives the mmap syscall number (0x09), rdi is set to zero to let the kernel choose the allocation address, rsi specifies the region size, rdx contains the permission flags (PROT_READ | PROT_WRITE | PROT_EXEC), r10 sets the mapping flags (MAP_PRIVATE | MAP_ANONYMOUS), and r8 and r9 are set to -1 as they're unused for anonymous mappings.

After the allocation completes, pp restores the target process to its original state by replacing the injected instructions and restoring the original register values. The operation returns a memory_region object that encapsulates the newly allocated memory space.

3.2.3 Runtime Process Analysis

pp provides multiple commands for inspecting and analyzing process information, allowing users to examine various aspects of running processes. These commands range from basic process identification to detailed analysis of memory regions and thread states. This section explores the different process information retrieval capabilities, their implementation, and practical applications in runtime analysis and debugging.

3.2.3.1 Memory Mapping Analysis

Memory mapping analysis in pp leverages the Linux proc filesystem to examine the memory layout of target processes. The system parses the /proc/[pid]/maps file, which provides a comprehensive view of all memory regions allocated to a process, including their addresses, sizes, permissions, and mapped files.

The implementation processes each memory region entry from the maps file, converting the information into structured memory_region objects. These objects encapsulate essential characteristics of each memory region, including its starting address, size, access permissions, and associated file mappings if present. For improved readability, the system implements intelligent size formatting, automatically converting byte sizes to appropriate units (B, K, M, G) based on the region's size.

When displaying memory maps, pp presents the information in a structured format that includes:

- Address range showing the start and end addresses of each region
- Size of the region in human-readable format
- Access permissions (read, write, execute, shared)
- Region name or "[anonymous]" for unnamed mappings

This memory mapping analysis serves as a foundation for other memory-related operations in pp, providing essential information about process memory layout and characteristics.

3.2.3.2 Memory Statistics

Memory statistics analysis in pp provides a comprehensive overview of a process's memory utilization by analyzing its memory regions. The implementation leverages the memory region data obtained from /proc/[pid]/maps to calculate various memory usage metrics that help understand how a process utilizes system memory.

The analysis aggregates several key statistics from the memory regions:

- Total memory usage across all regions
- Amount of executable memory (regions with execute permissions)
- Amount of writable memory (regions with write permissions)
- Count of anonymous memory regions (regions without associated file mappings)

The implementation iterates through each memory region of the target process, examining their sizes and permissions to build these statistics. Each region's permissions are checked to determine if it contributes to executable or writable memory totals. Anonymous regions are identified by the absence of a region name, providing insight into dynamic memory allocation patterns.

3.2.3.3 Process and Thread Information

Process and thread inspection in pp leverages various Linux kernel interfaces to provide detailed runtime information about processes and their threads. Through the proc filesystem and ptrace functionality, pp implements inspection capabilities that allow users to examine both process-wide and thread-specific states.

For process information, pp gathers data from multiple proc filesystem sources. The system determines the process base address by examining the first entry in the process memory map, typically representing the main executable region. Memory usage statistics are calculated using /proc/[pid]/statm, where pp tracks resident pages to provide accurate memory utilization information. The process executable path is obtained through the /proc/[pid]/exe symbolic link, which provides the actual path to the running executable.

Thread inspection operates at two levels: enumeration and detailed analysis. Thread enumeration is implemented by scanning the /proc/[pid]/task directory, which contains subdirectories for each thread in the process. Each thread's name is obtained from its respective comm file in the proc filesystem (/proc/[pid]/task/[tid]/comm). For detailed thread inspection, pp utilizes the ptrace interface to access thread register states, providing visibility into the execution context of individual threads. This includes access to the full x86_64 register set through the PTRACE_GETREGS operation.

The combined process and thread information provides users with a view of process execution state, from high-level process characteristics to detailed thread-level execution contexts.

3.3 Code Analysis and Injection

Runtime code analysis and modification features in pp provide capabilities for examining and altering program behavior during execution. The implementation includes function hooking through source code compilation and injection, disassembly for examining machine code, and shared library injection for extending program functionality. These features leverage Linux's memory management and process control mechanisms to enable program manipulation while maintaining system stability.

3.3.1 Disassembly

The disassembly feature in pp provides machine code analysis capabilities through the Capstone disassembly framework. This implementation wraps Capstone's x86-64 disassembler, providing a clean interface for examining process memory regions. The disassembler converts raw machine code into

human-readable instruction representations, displaying each instruction's address, mnemonic, operands, and size.

The disassembly process begins by reading a specified memory region from the target process. Each instruction is decoded into components including its mnemonic (like "mov" or "call") and operand string (like "rax, rbx"). The output format follows a consistent pattern of address: mnemonic operands (size), making the disassembly easy to read and analyze. By leveraging Capstone's capabilities, pp supports the full x86-64 instruction set.

This feature serves as a verification tool for runtime code modification. After function hooks are installed, disassembly can verify the correct placement of jump instructions and ensure the target function's modification was successful. Similarly, when analyzing control flow, the disassembly output reveals branch instructions (like jmp, call, ret) that can be modified using pp's memory writing functionality to alter program behavior. This combination of disassembly and memory modification enables runtime code manipulation while providing immediate feedback on the effects of changes.

3.3.2 Shared Library Injection

Linux systems include dynamic linking capabilities through the dlopen function, which is provided by the C standard library (libc). This function enables runtime loading of shared libraries, allowing programs to extend their functionality after startup. During libc compilation, dlopen is automatically included as part of the standard dynamic linking interface. The function accepts a path to a shared library and loading flags, handling the complex process of library loading, symbol resolution, and dependency management.

The implementation injects shared libraries by leveraging the target process's existing libc installation. pp locates libc in the process memory space by examining memory regions and searching for "libc.so" in their names. After finding libc, the implementation parses its ELF structure to locate the dlopen function's address through the dynamic symbol table.

Unlike previous injection methods that assembled syscall instructions (0F 05) followed by an int3 breakpoint (CC) and padding with nop instructions (90 90 90), shared library injection requires more setup. The implementation injects an eight byte instruction sequence (90 90 90 CC D3 FF 90 90) into an executable memory region. This sequence performs a call to the rbx register which contains the dlopen address, surrounded by nop instructions for

padding and an int3 breakpoint to return control to pp. The instruction pointer (rip) is set to the executable region offset by 2 bytes to skip the padding and execute the call instruction. To prepare the function call, registers are configured according to the x86_64 calling convention: rdi contains the path to the shared library (first argument), rsi contains RTLD_NOW flag (second argument), and rbx holds the dlopen function address. Additionally, rsp and rbp are set to a newly allocated stack region to maintain proper stack alignment.

The process registers are configured to call dlopen with the library path and RTLD_NOW flag, ensuring immediate symbol resolution. After injection completes, pp restores the original process state, including register values and the modified code region.

3.3.2.1 Injected Library Behavior

The behavior of injected libraries depends entirely on their implementation. When a library is successfully loaded through dlopen, it behaves as if it had been loaded during program startup through normal dynamic linking. The library's constructors (functions marked with __attribute__((constructor))) execute automatically during loading, allowing the library to perform any necessary initialization.

The injected library can modify the target process in various ways depending on its design. Common uses include adding new functions that the original process can call, intercepting existing function calls through symbol interposition, modifying program data, or establishing communication channels with external programs. The exact functionality depends on the library's purpose and implementation.

This flexibility allows users to extend program functionality at runtime by designing and injecting appropriate libraries. However, the injected library must be compatible with the target process's architecture and have access to any required dependencies, as incompatibilities can cause process instability.

3.3.3 Function Hooking

Function hooking in pp enables runtime modification of program behavior by redirecting function calls to custom code. The implementation requires users to provide source code containing a hook_main function that implements the desired behavior. This source code is compiled with specific flags (-O1 for faster compilation time, -fPIC for position-independent code, and -fno-exceptions to simplify the generated code) to create a shared object that can be injected into the target process.

3.3.3.1 Hook Implementation

The hooking process begins by compiling the source file containing the hook code. After compilation, pp parses the resulting ELF file to locate the hook_main function. This involves traversing the ELF's section headers to find the symbol tables (.symtab and .strtab) and the text section (.text) where the actual code resides. Once the hook_main symbol is located, pp calculates its offset within the text section and extracts just the function's compiled code.

Memory allocation for the hook follows the same pattern as other injection operations. pp allocates a new memory region in the target process and writes the extracted function code into this region. The allocated region must be executable to allow the injected code to run.

To redirect the original function to the hook, pp modifies the target function's beginning with a specific instruction sequence:

- A REX prefix (48) indicating 64-bit mode
- A mov instruction (B8) that loads the hook's address into rax
- A jmp instruction (FF E0) that jumps to the address in rax
- A ret instruction (C3) for proper function return

3.3.3.2 Technical Challenges

Function hooking presents several significant technical challenges that make it one of the more complex features to implement reliably:

1. Code Relocations: The compiled hook function may contain references to external symbols or data that need to be properly relocated when the code is moved to a new address. The current implementation doesn't handle these relocations, limiting the complexity of hooks that can be safely implemented.

2. Stack Alignment: x86-64 calling conventions require the stack to be properly aligned, especially before function calls. When redirecting functions, maintaining proper stack alignment becomes challenging as the hook interrupts the normal function prologue.
3. Function Arguments: Accessing the original function's arguments in the hook requires careful stack manipulation, as the arguments may be passed in registers or on the stack depending on their types and the calling convention.

These challenges make function hooking particularly prone to subtle bugs that can be difficult to diagnose. Memory corruption, crashes, or unexpected behavior can occur if any aspect of the function transition isn't handled correctly. Moreover, debugging these issues is complicated because the problems often manifest in seemingly unrelated parts of the program due to corrupted program state.

The complexity of proper function hooking explains why many debugging and instrumentation tools either implement limited hooking capabilities or rely on other techniques like breakpoints for function interception. A complete solution would require extensive handling of relocations, careful stack and register management, and thorough testing across various function types and calling patterns.

3.3.4 Function Analysis and Symbol Resolution

Function analysis in pp provides capabilities for locating, examining, and understanding functions within running processes through ELF symbol parsing and runtime memory analysis. The system implements multiple approaches to function discovery and analysis, from exact name matching to pattern-based searches with symbol demangling support.

3.3.4.1 Function Symbol Resolution

The function resolution process operates by parsing the ELF symbols from the target process's executable file. The system examines both dynamic (SHT_DYNSYM) and static (SHT_SYMTAB) symbol tables to build a comprehensive list of program functions. For each symbol table section, pp extracts function symbols (STT_FUNC type) and their associated names from the string table.

To determine actual runtime addresses of functions, pp calculates the load address adjustment by examining the program headers. It identifies the first PT_LOAD segment to establish the base load address, then adjusts each function's address by combining the process's base address with the symbol's value offset from the load address. This adjustment ensures accurate function location even in processes with address space layout randomization (ASLR).

3.3.4.2 Function Discovery and Name Demangling

pp implements two distinct function discovery commands. The find-func command performs exact name matching, searching for a specific function name within the process's symbol tables and returning its precise runtime address when found. For more flexible searching, the find-fn command enables pattern-based function discovery, locating all functions whose names contain a specified pattern.

Name mangling is a technique used by C++ compilers to encode additional information about functions into their symbol names. This encoding is necessary because C++ supports function overloading, namespaces, and classes, which create situations where multiple functions can share the same base name. The compiler mangles these names by encoding the parameter types, namespaces, and class information into the symbol name.

For example, a simple C++ function:

void MyClass::setValue(int value)

Might be mangled into a symbol name like:

_ZN7MyClass8setValueEi

The mangled name encodes the class name (MyClass), function name (setValue), and parameter type (i for int). This encoding ensures each function has a unique symbol name in the compiled binary, allowing the linker to correctly resolve function calls.

The pattern matching system supports demangled symbol names through the --demangle option. When enabled, pp uses the C++ ABI demangling facility to convert these mangled symbol names back to their original C++ function signatures before pattern matching. This capability makes it easier to locate C++ functions in the binary, as users can search using the original function names rather than needing to know the mangled symbol format.

3.3.4.3 Function Analysis

The function analysis capabilities focus on examining the memory context and initial content of identified functions. When analyzing a function, pp determines its location within process memory and examines the containing memory region's characteristics. The analysis provides several key pieces of information:

The system reports the function's absolute address in process memory and identifies the memory region containing the function code. It examines the region's permission attributes to understand how the function's code is protected in memory. For functions in shared libraries or modules, pp identifies the specific module containing the function code.

To aid in function verification and analysis, pp examines the initial bytes of the function code. This examination provides a hexadecimal display of the function's beginning, allowing verification of function boundaries and initial instructions. The analysis helps users understand both the location and content of functions within the process memory space.

3.4 Command Line Interface Implementation

The CLI uses a hash map to store and dispatch commands, with each command containing a name, description, argument list, and handler function. Handler functions return either a success value or an error, allowing error propagation through the command chain.

3.4.1 Command Processing Architecture

The parser validates input through three steps:

1. Command existence check in the hash map
2. Argument count validation against command specification
3. Handler-specific argument validation and execution

When validation fails, the system prints usage information and exits. For successful validation, the parser extracts arguments into a span and passes them to the handler function.

3.4.2 Error Handling

Error handling occurs at three levels:

1. Parameter validation checks required arguments and formats
2. Type conversion verifies numeric inputs and address formats
3. Operation execution catches system call and memory access failures

Errors propagate through std::expected return values, preserving the original error message while adding context at each level. This enables tracking of error sources through the operation chain.

3.4.3 Parameter Types

The system processes three main parameter types:

1. PIDs - Validated against process table
2. Memory addresses - Parsed as hexadecimal, checked against memory maps
3. Sizes - Parsed as decimal, validated for alignment requirements

3.4.4 Command Categories

Commands divide into functional groups:

Process Access:

- Process identification (pidof, name)
- Information gathering (info, maps)
- Memory statistics (memstat)

Memory Operations:

- Direct access (read, write)
- Pattern matching (search, replace)
- Permission control (chmod)
- Memory management (allocate)

Analysis Features:

- Function analysis (analyze-func)
- Disassembly (disasm)
- Symbol management (functions)

Runtime Modification:

- Function hooking (hook)
- Library injection (inject)
- File loading (load)

Thread Management:

- Thread listing (threads)
- Thread inspection (thread-info)

3.4.5 Command Registration

The registration system adds commands to the parser through a single registration function. Each registration includes the command name, description, argument specification, and handler implementation. This centralized registration enables consistent command behavior while allowing specialized argument handling per command.

Chapter 4: Conclusion

4.1 Implementation Summary

pp demonstrates basic process manipulation and analysis capabilities on Linux systems. The implementation explores process memory management, runtime code modification, and thread analysis through direct interaction with Linux system interfaces. While achieving core functionality, certain features like function hooking remain experimental and require additional development.

4.2 Core Features Memory Operations

The implementation provides fundamental memory access through process_vm_readv and process_vm_writev system calls. Memory manipulation features include pattern searching, data replacement, and permission modification. These operations leverage the proc filesystem and ptrace interface for process interaction.

4.2.1 Analysis Capabilities

Process analysis features examine memory maps, calculate usage statistics, and inspect thread states. The implementation uses ELF parsing for function symbol resolution and the Capstone framework for code disassembly. These capabilities enable runtime program inspection and modification.

4.2.2 Code Modification

Runtime code modification operates through function hooking and shared library injection. The function hooking implementation compiles source code into position-independent code for injection. Library injection leverages the target process's existing dynamic linker through dlopen. Both features demonstrate basic operation while highlighting technical challenges in stable code modification.

4.3 Technical Limitations

- Function hooking stability issues from incomplete relocation handling
- Stack alignment challenges during code injection
- Limited symbol resolution for complex C++ functions
- Basic pattern matching implementation without optimization

4.4 System Integration

The implementation uses multiple Linux interfaces:

- Proc filesystem for process information
- Ptrace for process control
- System calls for memory operations
- ELF parsing for binary analysis

These mechanisms provide the foundation for process manipulation while maintaining system security boundaries.

4.5 Future Development

Potential improvements include:

- Enhanced function hooking reliability
- Optimized pattern searching
- Extended thread control features
- Improved error handling

These developments would expand pp's capabilities while maintaining its focus on Linux process manipulation and analysis.

Resources

System Call Documentation

- System call numbers (NR) and their corresponding symbolic names: https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

Dynamic Library Loading

- Linux manual page for dlopen(3): https://www.man7.org/linux/man-pages/man3/dlopen.3.html
- Dynamic linking and loading tutorial : https://www.youtube.com/watch?v=Y57ruDOwH1g&list=PLpM-Dvs8t0Vak1rrE2NJn8XYEJ5M7-BqT&index=2

Memory Management and Memory Mapping

- Linux manual page for mmap(2): https://www.man7.org/linux/man-pages/man2/mmap.2.html
- Memory mapping implementation details: https://stackoverflow.com/questions/5877797/how-does-mmap-work

Memory Protection

- Linux manual page for mprotect(2): https://www.man7.org/linux/man-pages/man2/mprotect.2.html
- Memory protection implementation details: https://stackoverflow.com/questions/3825018/how-does-mprotect-work

Code Analysis

- Capstone disassembler C language documentation: https://www.capstone-engine.org/lang_c.html

Linux Process Management

Proc Filesystem

- Linux kernel documentation for /proc filesystem: https://www.kernel.org/doc/html/latest/filesystems/proc.html
- Virtual memory mapping details: https://stackoverflow.com/questions/20480608/how-is-kernel-virtual-memory-mapped-to-physical-memory

ELF File Format

- ELF executable memory loading: https://unix.stackexchange.com/questions/70506/which-parts-of-an-elf-executable-get-loaded-into-memory-and-where

Assembly Tools

- Online x86 assembler and opcode reference: https://defuse.ca/online-x86-assembler.htm