Endokernl with Software Fault Isolation

Zhenghui Guo
Rice University

Jerry Zhang
Rice University

Abstract

Endokernel provides a new way to easily add efficient and extensible security architecture by utilizing Intel technology Erim with memory protection keys(MPK) to implement memory isolation at the intra-process level, separating the trusted and untrusted domain sets. MPK, one of Intel's memory isolation mechanisms for the x86 platform, has some limitations concerning architecture dependence that prevent it from being widely adopted. To overcome the hardware limitations the Endokernel has, we will apply another software mechanism, SFI, for memory isolation to achieve architecture independence and compare performance costs. In this project, we'll demonstrate the benefit of using SFI other than MPK in our work to build Endokernal in a portable way to suit a modern diverse architectural environment.

Keywords— Security, In-Process Isolation,SFI, Sandboxing

1 Introduction

Process abstraction contains a high-level weight of interaction at the kernel and user levels. The modern process is also a multiprogramming environment, where the process runtime operates on-behalf-of code and data belonging to distinct and mutually distrusting clients, leaving no ability to thwart corruption or leakage of sensitive data.[1] The current OS design approach, process-level memory isolation, jams all content into one fault domain, compromising security and reliability. However, all of these languages enforce rigid interfaces that, while quite expressive, are not extensible and, by nature, introduce threats to unrelated parts of the kernel. Beyond that, they also fail to monitor and enforce resource control at the subprocess level.[4] To lose the level of weight and better isolate those processes by putting them in different protection domains.

Recently, researchers proposed the idea of an "Endokernel", The Endokernel introduces a new virtual machine abstraction for representing subprocess authority, enforced by an efficient self-isolating monitor that maps the abstraction to system-level objects (processes, threads, files, and signals)[4]A kernel within a process and only include what is necessary for memory isolation and enable extensibility through app-layer level OS servers[1]New hardware-based technology Inteltech MPK and Erim bright Endokernel' researcher's attention. MPK enforces memory protection within the same address space, allowing it to separate and provide privileged memory access of the same process to its own address space. ERIM and Hodor systems have built-in sandboxes that prevent adversaries from bypassing the isolation scheme by executing unsafe instructions that modify the PKRU register[8]. Although, compared to traditional inter-process memory isolation, which requires a context switch and changing page table when switching between two processes, they provide fast authorization check for all memory access with the PKRU register and the protection bits stored in each page table entry.

Unfortunately, Erim and Hodor suffer from security and usability problems. 1) these sandbox implementations have known security and usability problems[8]. To neutralize any unsafe instructions in the protected program, ERIM, for example, utilizes static binary instrumentation (SBI). SBI cannot reliably distinguish code from data, and ERIM could leave some unsafe instructions untouched[3]. 2) ERIM's sandbox also marks pages that contain unsafe instructions as non-executable, which could lead to usability issues. Hodor's sandbox uses hardware breakpoints to ensure the program cannot execute unsafe instructions. This approach does not rely on SBI like ERIM's, but both systems can still be bypassed using the kernel as a confused deputy [2]. 3)Erim is based on MPK, a mechanism only available for x86 CPUs and lacks support for recently fast-growing ARM processors.4)The

MPK could still suffer from malicious attacks, such as hackers injecting code to execute the update PKRU instruction. The above security features of Endokernel have been compromised since it was built on Erim.

To address Erim's problem, we have turned our attention to a more stable, portable secure, and reliable memory isolation technique that software instances have widely adopted, SFI(Software Fault Isolation)[9], for the Endokernel. The SFI splits an address space into different fault domains and prevents memory access from one fault domain to another by checking the target address both at compile time for static memory access and runtime for dynamic memory access. It's widely recognized that the SFI could be slower than MPK since it needs to check all dynamic memory access by comparing the target segment identifier with the current one at runtime. The cross-domain RPC communication that SFI proposed is also much slower than the WRPKRU instruction used by MPK because that includes additional instructions for saving and restoring registers and preparing call stubs and return stubs. Nevertheless, the question for our project is: *how much faster would MPK be over SFI and would there be some corner cases where the SFI would be faster than MPK?* Additionally, using the advantage of the SFI, a software-based technique that would be architecture-independent, we further install the Endokernel incorporated with SFI on ARM CPU processors and evaluate its performance there. There are three goals in our project: **1**) **We will replace Erim(MPK) with SFI in Endokernel. 2**) **Compare the performance Endokernel with SFI in arm and x86. 3**)**We will use CPI+MPK to achieve a system call. (optional)**

2 Background

2.1 Endokernel

The operating system and kernel industry has been thinking about memory isolation as the complexity of operating systems and software grows. The most common and popular method for memory isolation is the paging and page table which provides process-level memory isolation. However, the need for more isolation beyond just achieving process-level isolation is growing. If users want to isolate a piece of memory, they have to create a new process that causes overhead when doing context switches and changing page tables. Not until recently, this brought attention to both the software and hardware industry and intel proposed MPK as new hardware primitive to support intra-process memory isolation and thread-local permission control on groups of pages[7]. The Endokernel further uses the MPK to support its memory isolation and protection mechanism, but the MPK has the downside of only supporting x86 CPUs and limiting the Endokernel's portability. We switch our attention to a more universal architecture-independent approach, SFI.

2.2 Software Fault Isolation

SFI uses inline reference monitors within a single memory space. The core technique of SFI is to allocate a logical enough "fault domain", and at the same time, check each memory access at compile time and runtime to ensure they do not go beyond their fault domain. The SFI also provides low-latency cross-fault-domain RPC calls to allow communications between fault domains and isolating syscalls via delegating all syscalls to a trusted domain with an arbitration code to determine the authority to make that call. Recently, Apple silicon has been built based on ARM architecture; Hardware like the Apple M1 chip has shown they are much more energy-efficient than x86, as apple silicon almost replaced the role of x86 in Mac, it already occupies a giant chunk of the computer market in recent years. Paying more attention to the Arm architecture seems like a mandatory task for future Endokernal development.

2.3 Code Pointer Integrity

Endokernal provides W xor X to limit reading and writing to improve system calls. This method is very good, but it increases the complexity of the policy. We try to use a very popular method to widely use CPI[5] to solve the problem of system calls. Traditionally methods such as CPI(code pointer integrity) is been widely used with more stability.

3 SFI implementation

We examined two ways to implement SFI and discuss their strength and weakness. Our results indicates that it's much easier to implement SFI with LLVM rather than directly editing the assembly file. Finally, we present some benchmark comparisons between the software with SFI-enabled programs and SFI-disabled programs.

3.1 Editing assembly file

There are three major steps to implement the SFI address sandboxing

1. Statically analyze the assembly instructions and find all the memory access instructions.
2. For each memory access instruction, classify them into safe instruction and unsafe instruction. Where safe instructions are instructions that access the memory address that can be statically analyzed and verified to be in the correct segment and vice-versa for unsafe instructions.
3. For each unsafe instruction, insert assembly code to perform address segment matching or address segment sandboxing.

The implementation was built on a legacy 32-bit SFI implementation. Since we only need to read through the assembly code and find all instructions with opcodes like mov, st, ld, je, etc., we can perform step 1) and step 2) very easily. However, we found that assembly code differs depending on the operating systems/compilers and the CPU architectures. For example, the x86 uses mov for loading/storing the memory while the OSX with ARM Apple M1 CPU uses st/ld to store and load memory. This would require a lot of effort for our analyzer to be compatible with different platforms. Furthermore, doing step 3), inserting assembly instructions is much more complicated since it would require rewriting the whole assembly code and correctly updating the memory address of jmp instruction since they would be affected by the inserted instructions. With obstacles we discussed above, the whole system would be hard to implement and it would also be hard to make it fits on all different kinds of platforms since the instructions for them are slightly different.

3.2 Using LLVM toolchain

We further explore the other method for implementing the SFI, by using the LLVM toolchain[6]. The LLVM toolchain separates the compiler to the frontend which generates LLVM IR(Intermediate Representation) and the backend which takes the LLVM IR and compile it into an executable file. Since the LLVM backend supports all major platforms we are using today, we only need to edit the LLVM frontend to insert sandboxing instructions to LLVM IR and the LLVM backend will compile that into an executable file, saving us lots of effort of rewriting the assembly code directly. With the LLVM toolchain, we will write function pass and LLVM intrinsics to analyze the LLVM IR and insert check each memory access to see if it's safe and for unsafe memory access, insert assembly instruction to check it at runtime.

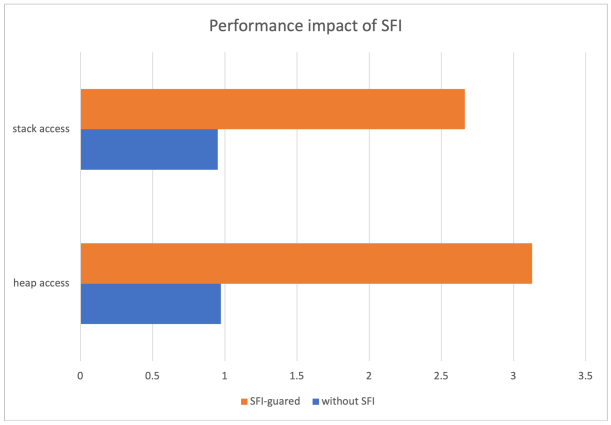

Figure 1: Runtime with and without SFI

3.3 Performance evaluation

We have created a demo program for a simple performance analysis of the SFI program. The demo program dynamically allocates an array in heap and keeps trying to set the value of an element in the array to some value(Thus trying to store data in the memory.) This is an unsafe store because the compiler could not find the memory destination at compile time. So it will create overhead for checking the segment at runtime. We also created another demo that access memory in the stack. The performance comparison for these two demos with and without SFI safeguarded is shown in Figure 3.3. We found that using SFI creates lots of overhead in checking the memory segment. But this is for the memory access-sensitive programs, we expect to see much lower overhead for programs less frequently access the memory or only access the static memory objects.

References

[1] Anonymous Author. Endokernel: An operating system organization for protection within processes. *Not yet*, 2022.

[2] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. {PKU} pitfalls: Attacks on

{PKU-based} memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426, 2020.

[3] Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis. Proceedings of the 2011th european conference on machine learning and knowledge discovery in databases-volume part ii, 2011.

[4] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. The endokernel: Fast, secure, and programmable subprocess virtualization. *arXiv preprint arXiv:2108.03705*, 2021.

[5] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[6] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[7] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[8] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 266–282, 2022.

[9] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.