Raven: A Novel Kernel Debugging Tool on RISC-V

Hongyi Lu
luhy2017@mail.sustech.edu.cn
Research Institute of Trustworthy Autonomous Systems
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, Guangdong, China

Fengwei Zhang*
zhangfw@sustech.edu.cn
Department of Computer Science and Engineering
Research Institute of Trustworthy Autonomous Systems
Southern University of Science and Technology
Shenzhen, Guangdong, China

Abstract

Debugging is an essential part of kernel development. However, debugging features are not available on RISC-V without the use of external hardware. In this paper, we leverage a security feature called Physical Memory Protection (PMP) as a debugging primitive to address this issue. Based on this debugging primitive, we design Raven, a novel kernel debugging tool with the standard functionalities (breakpoints, watchpoints, stepping, introspection). A prototype of Raven is implemented on a SiFive Unmatched development board. Our experiments show that Raven imposes a moderate but acceptable overhead to the kernel. Moreover, a real-world debugging scenario is set up to test its effectiveness.

Keywords

Debugger, Kernel Debugging, RISC-V

ACM Reference Format:
Hongyi Lu and Fengwei Zhang. 2022. Raven: A Novel Kernel Debugging Tool on RISC-V. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530583

1 Introduction

One essential part of kernel development is debugging, which helps developers to track abnormal behavior and build a more robust kernel. Unlike in the user space, where we have support from the kernel, debugging in the kernel space has to rely on special tools.

Existing kernel-debugging tools generally fall into two categories: hardware and software debugging. A hardware approach typically requires an additional hardware device like a JTAG probe [6], while a software debugger leverages on-board hardware features or a hypervisor to perform debugging. However, both options suffer from limitations on the RISC-V platform.

The hardware debugging approaches have the following limitations. (i) Hardware vendors may not expose the debugging port due to security or cost considerations. This is very common on mass-production models. For example, the Nezha D1 [8] from Allwinner Tech [1] hides its debugging pins inside the SD card slot, which has to be connected through a special adapter. (ii) Current hardware debuggers for RISC-V typically require complicated configurations. Take the HiFive Unmatched [4] from SiFive [15] for instance; its heterogeneous architecture requires complicated configuring for its debugger to work. (iii) The debugging specifications of RISC-V remain a working draft [12], resulting in divergent implementations by different vendors [1, 9, 15]. For instance, to debug on the Nezha D1 [8], one has to use its customized debugger, CKLink [2], which is tied to this specific series of processors.

Meanwhile, existing software kernel debugging approaches can be further categorized into two types, hypervisor-based debugging and built-in debugging. A hypervisor-based debugger runs the kernel inside a virtual environment like QEMU [17]. This approach isolates the kernel from the actual hardware, making it hard to debug hardware related components in kernel. A built-in debugger like kGDB [7] or WinDBG [16] normally relies on kernel modules to provide debugging functionality. This leads to the following limitations. (i) They are OS-specific, since these kernel modules are tied to a specific OS and impractical to port. (ii) They use software breakpoints, which replace certain instructions inside the kernel. This breaks the integrity of the kernel and leads to a failure of the integrity check. Since hardware breakpoints are not available on RISC-V without an external device, the software breakpoint is the only option for these debuggers. There are bare-metal kernel debugging tools on other platforms like x86 and ARM [23, 27] that overcome the above limitations by leveraging hardware features such as the Performance Monitoring Unit (PMU) [10] and Embedded Trace Macrocell (ETM) [3]. However, these features have not yet been fully implemented on the RISC-V platform.

In this paper, we propose a debugging design called Raven that addresses the above issues by utilizing an off-the-shelf hardware feature called Physical Memory Protection (PMP) on RISC-V. PMP can divide physical memory into different permission regions with a granularity of up to 4 bytes. Raven is able to use this feature as a debugging primitive. For example, when a breakpoint is needed, Raven can mask the instruction at the breakpoint as non-executable. Then when the processor tries to execute this instruction, PMP generates an exception to trap into Raven to perform further introspection. Note that since PMP is a hardware feature, it does not alter any content in the kernel. Thus, Raven does not break the integrity of the kernel itself.

To test the feasibility of our design, we implement a prototype of Raven with the official firmware OpenSBI [14] and test it on a QEMU virtual machine and a SiFive Unmatched board. We also implement a set of GDB-like debugging commands such as breakpoints, watchpoints, and memory introspection in the Raven prototype. This set of basic commands not only provides a user-friendly

*Fengwei Zhang is the corresponding author

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530583

debugging interface for Raven but also enables it to be easily incorporated with other debugging tools like IDA [5] and Radare2 [11]. Moreover, we successfully implement Raven with both fine- and coarse-granularity PMP to achieve high portability across different RISC-V boards.

Our contribution is summarized as follows.

- We propose a novel approach called Raven that leverages a hardware feature of RISC-V to perform kernel debugging. To the best of our knowledge, Raven is the first work on RISC-V that achieves non-invasive kernel debugging without using external hardware.
- We implemented a prototype of Raven within the official firmware OpenSBI [14] of RISC-V, which can be directly deployed on any off-the-shelf RISC-V devices to start kernel debugging.
- We tested the prototype of Raven to show that it has the debugging functionalities largely equivalent to that of an external debugger.

The rest of this paper is organized as follows. Section 2 provides relevant background for Raven and debugging. Section 3 focuses on the overall design of Raven. Section 4 discusses the implementation of Raven in detail. Section 5 evaluates the effectiveness and performance of Raven. Section 6 discusses the compatibility and limitations of Raven. Section 7 concludes this paper and discusses future work.

2 Background and Related Work

In this section, we introduce the relevant background and existing work about debugging tools on the RISC-V platform.

2.1 RISC-V Architecture

2.1.1 RISC-V Privilege Levels The current RISC-V standard specifies three different privilege levels, User Mode (U-Mode), Supervisor Mode (S-Mode) and Machine Mode (M-Mode) [13]. Each has different permissions for operations such as configuring registers, manipulating memory, and handling traps. As illustrated in Figure 1, user applications run in U-Mode with the least privilege, whilst the OS kernel resides in the S-Mode, which is privileged to perform operations such as virtual memory mapping. Raven runs in the firmware inside the M-Mode with the highest privileges so that it performs PMP configuration and introspection into the kernel.

Application	Application	U-Mode
OS Kernel		S-Mode
Firmware & Bootloader		M-Mode

Figure 1: RISC-V Privilege Levels

2.1.2 RISC-V Physical Memory Protection RISC-V provides PMP to control access to certain memory regions [13]. More specifically, each PMP entry consists of two registers, an address register (*pmpaddr*) and a configuration register (*pmpcfg*). A combination of access permissions (R/W/X) and an address region can be assigned to an entry through these registers, so the entry constrains access to its region accordingly. The number and granularity of these PMP entries are implementation dependent. For example, the Nezha D1 [8] only provides 8 PMP entries with a coarse granularity of 4KB, while the QEMU emulator equips 16 entries and a fine granularity of 4 bytes. Apart from that, RISC-V defines a set of registers to provide relevant information about exceptions. Table 1 illustrates the content of these registers when a PMP exception occurs.

Table 1: RISC-V Exception Registers

Register	Information
`mepc`	Program counter before the exception
`mcause`	Cause of the exception
`mtval`	Protected address

2.1.3 RISC-V Instruction Format The base RISC-V Instruction Set Architecture (ISA) only consists of fixed-length 32-bit instructions, which are naturally aligned on 4-byte boundaries [13]. This coincides with the granularity of PMP provided by RISC-V. However, in order to reduce the size of the program on the RISC-V platform, the standard defines a "C" standard extension to add support for 16-bit instructions on RISC-V. These two types of instructions can be distinguished by their lowest two bits. All the 32-bit instructions of base ISA end with `11`, while the compressed 16-bit instructions end with `00`, `01` or `10`.

2.1.4 RISC-V Debugging Support RISC-V defines a special privilege level called Debug Mode [12]. It provides various debugging functionalities such as hardware breakpoints, program buffers, and single stepping. However, this mode is only available with an external hardware debugger connected. Neither firmware nor the operating system has access to these features. Thus, we still categorize this debugging support as external debugging.

2.2 Related Work

2.2.1 PMP-assisted Systems Current research leveraging PMP focuses on its security features. For example, Penglai [20] and Keystone [22] use PMP to provide a Trusted Execution Environment (TEE) for sensitive applications. LIRA-V [24] utilizes PMP to establish trusted communication channels and perform mutual attestation between two devices. Moreover, PMP can also be used to force runtime pointer checks to detect vulnerabilities such as heap overflow and use-after-free [18]. Raven differs from the previous work because it uses PMP for debugging.

2.2.2 Debugging Systems There are a variety of well-known debugging tools running within the operating system. For instance, IDA Pro [5] and Radare2 [11] are both powerful tools for debugging user applications. Since they rely on the operating system to provide their runtime environment, they cannot be used to debug kernels without combining a hypervisor like QEMU [17].

Besides QEMU, there are also other hypervisor-based approaches for debugging. For example, V2E leverages hardware virtualization and software emulation to perform malware analysis [26] while Ether only uses hardware virtualization to achieve high transparency for malware analysis [19]. Although it is possible to use these systems to perform kernel debugging, they still focus on malware analysis. Further, they both rely on hardware virtualization, which is not fully implemented on RISC-V.

Many operating systems also offer a built-in kernel debugging mechanism, such as kGDB [7] and WinDBG [16]. Like Raven, they

use a debug client connect to the target machine through serial or network to perform debugging. However, both kGDB and WinDBG require kernel modules residing in the operating system. This ties them to a specific OS and makes them hard to port. Further, the trusted codebase of these OS-specific debuggers is the entire kernel, making them unable to debug kernel-level rootkits, since these rootkits are capable of tampering with the behavior of these debuggers.

There are also debugging systems that leverage hardware features. MalT [27] uses the System Management Mode to implement debugging functionalities on x86 platforms, while Ninja [23] leverages the PMU and ETM features on Arm. Neither of them relies on the OS and both successfully achieve non-invasive debugging without external hardware. However, these hardware features they rely on are either not available or not fully implemented on RISC-V.

Additionally, Jang and Kang [21] leverage Arm debugging features as security primitives to enforce kernel integrity, which is opposite to what Raven is trying to do.

3 Design of Raven

3.1 Overview

We first introduce an overview of Raven. As Figure 2 illustrates, Raven receives debugging commands from the debug client via serial or network, sets PMP registers or performs introspection accordingly, and sends results back to the client. Raven is a piece of software inside M-Mode, and we categorize it as a software debugger. Hence, we focus on addressing the limitation of current software kernel debuggers on RISC-V.

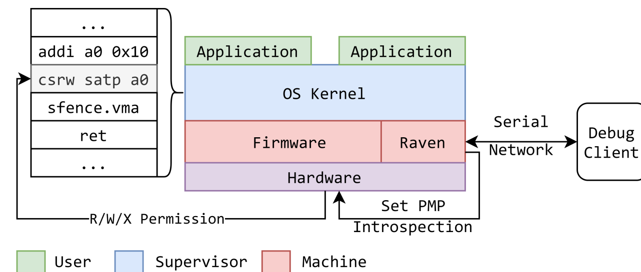

Figure 2: Overview of Raven

3.2 Design Goals and Challenges

As our goal is to overcome the limitations mentioned in Section 1 and Section 2, we need to fulfill the following properties besides the basic functionalities of a debugger.

P1: Non-invasive Debugging. As mentioned in Section 2, the hardware breakpoints on RISC-V are only available when an external debugger is connected. This causes the software debuggers on RISC-V to use software breakpoints, which replace instructions inside the kernel and breaks its integrity. Raven solves this issue by utilizing PMP as a debugging primitive to implement various debugging functionalities.

P2: No Hypervisor. Hypervisor-based kernel debugging typically runs the kernel in a virtualized environment such as QEMU. Inside a virtualized environment, the kernel may act differently, increasing the difficulty of debugging. We solved this issue by designing Raven as a bare-metal debugger inside the firmware, which does not rely on any hypervisor.

P3: Operating System Agnostic. Existing built-in kernel debuggers like kGDB and WinDBG are generally OS-specific. However, Raven is built along side the firmware instead of the kernel. Thus, it is OS-agnostic and can be used to debug different kernels.

To achieve these three properties, the following challenges have to be resolved. (i) How can PMP be utilized as a debugging feature (§3.3)? (ii) How can it be extended into various debugging functionalities (§3.4)? (iii) How to resolve the issues brought by the granularity of PMP (§3.5)? (iv) As its name suggests, PMP only recognizes the physical memory address; how to synchronize between physical and virtual addresses (§3.6)?

3.3 Utilization of PMP

In this part we introduce how we solve the first challenge, using PMP as hardware primitives to facilitate debugging. We utilize PMP as two different primitives, breakpoints and watchpoints. As Figure 3 shows, to set a breakpoint, Raven uses PMP to mask the corresponding instruction as non-executable and halts the kernel when this instruction is executed. In a similar approach, both read-only and write-only watchpoints can be set with their respective permissions.

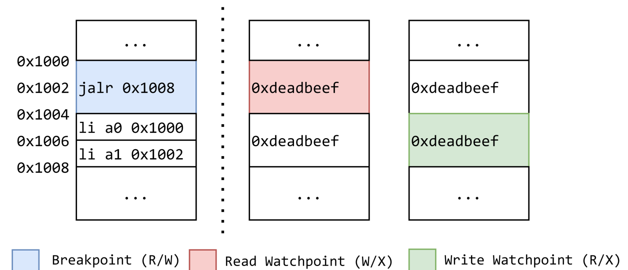

Figure 3: PMP as Debugging Primitives

However, since PMP is implementation specific, some boards may equip with PMP of only 4 kilobyte granularity, which is impractical to set up breakpoints with coarse-granularity PMP. To address this issue while keeping Raven non-invasive, we propose a more compatible utilization of PMP that only requires 4 kilobyte granularity. As Figure 4 shows, instead of setting breakpoints directly with PMP, we use PMP to protect the region where the breakpoints reside and emulate the result. This achieves breakpoint functionality while maintaining the non-invasive property of Raven.

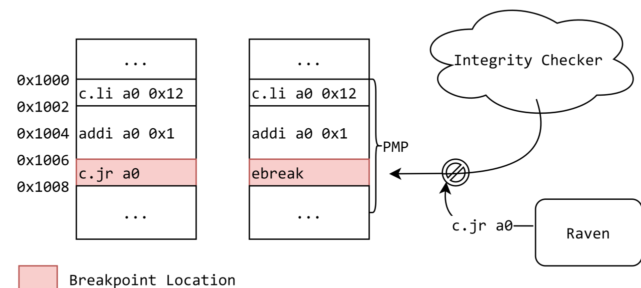

Figure 4: Breakpoints with Coarse-granularity PMP

3.4 Extension of Functionality

With these two primitives available, now we introduce how to achieve stepping in Raven, which is another key functionality for a debugger. One natural intuition is to update the breakpoint as the kernel runs. Raven also follows this method with control transferring instructions processed separately. As Figure 5 shows, if the current instruction is not a control transfer instruction, Raven sets the breakpoint at the next instruction; otherwise, Raven dissects the control transfer instruction to predict its destination and sets the breakpoint correspondingly.

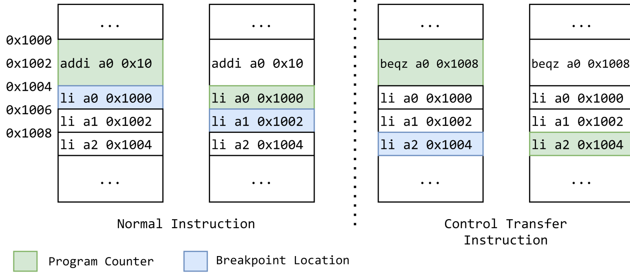

Figure 5: Control Transfer Process

3.5 Solution of Granularity Issues

Since the granularity of PMP is at most 4 byte, the standard specifies that its address must also be 4-byte aligned. However, the standard defines a set of compressed 2-byte instructions. This difference of alignment brings the "hidden instruction" issue for Raven. As shown in Figure 6, during a stepping process, the PMP is always 4-byte aligned as the kernel runs. This causes a misalignment between PMP and the current pc. If the misalignment happens at a control transfer instruction like `c.jr a0` in this figure. Raven will lose the track of current pc and cease stepping.

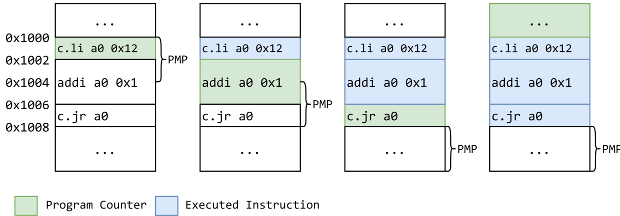

Figure 6: Hidden Instruction Issue

Raven overcomes this issue by using a look-ahead technique. As Figure 7 demonstrates, if Raven detects the program counter is not aligned up to 4-byte boundary, then this indicates a possible "hidden instruction". Thus, when stepping, besides the current instruction, Raven also look one instruction ahead. If either of them is a control transfer instruction, Raven will dissect it and set PMP at its destination.

3.6 Synchronization of Physical and Virtual Addresses

Since PMP only recognizes physical addresses, it is necessary to synchronize the memory mapping of breakpoints between kernel and Raven. To address this issue, we leverage a hardware feature of

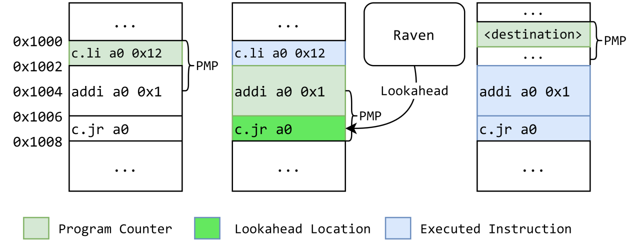

Figure 7: Look-ahead Technique

RISC-V called Trap Virtual Memory (TVM) [13]. When the kernel tries to modify the page table root or flush the Translation Lookaside Buffer (TLB), an exception is generated if TVM is enabled, and Raven can update its PMP according to the new page table.

4 Implementation

We implemented a prototype of Raven on a QEMU virtual machine and a SiFive Unmatched Development board [4] to evaluate our design. This prototype is based on an off-the-shelf firmware of RISC-V called OpenSBI [14]. By default, Raven performs the following steps to start debugging. (i) Raven sets up TVM to intercept page table modification. (ii) An initial breakpoint is set at `0x80202000`, the starting address of Linux. (iii) Raven starts boot up process and halts the kernel at the initial breakpoint. (iv) Raven waits for debugging commands. The list of supported debugging commands are described in Table 2. With these commands, Raven is able to perform most debugging functionalities.

Table 2: Debugging Commands Supported by Raven

Command Format	Description
b <address>	Set a breakpoint at <address>
w <address>	Set a watch point at <address>
pr (pw) <address>	Read(Write) memory content at <address>
rr (rw) <reg>	Read(Write) register content of <reg>
map <address>	View the memory mapping of <address>
csrr (csrw) <csr>	Read(Write) control status register of <csr>
s	Single-step execution
c	Continue execution after a breakpoint
<GPIO Switch>	Send an external interrupt to halt the kernel

5 Evaluation

In this section, we evaluate the effectiveness and performance of Raven as follows. §5.1 evaluates the codebase size of Raven. §5.2 provides a case study where Raven is used to locate and fix a real-world bug. §5.3 uses LMBench to evaluate its overall overhead. §5.4 further investigates its synchronization overhead.

5.1 Codebase Size

Table 3 shows the Line of Code (LoC) of each component of Raven. In total, we add 568 lines of C code into the OpenSBI [14] firmware. The codebase of Raven is small compared to the firmware, which consists of about 16, 000 LoCs. This shows that Raven can be easily modified and ported to fit other use cases.

Table 3: LoC of Raven Components

Components	Lines of Code
Initialization	65
Breakpoints	150
Stepping	242
Serial	111
Total	568

5.2 Case Study: A Buggy Device Tree

5.2.1 Real-world Debugging Scenario Device trees are used by Linux to pass hardware information from an early bootloader to the kernel. If an incorrect device tree is used, the kernel will not be able to recognize its hardware, resulting in a kernel crash. To test the effectiveness of Raven, we set up a real-world debugging scenario where we reproduced a similar bug and located it via Raven by the following steps. (i) Modifying the address of the interrupt controller to craft a buggy device tree. (ii) Booting Linux up with this device tree to cause a kernel crash. (iii) Using Raven to locate and fix this bug.

5.2.2 Debugging with Raven As this bug occurs in the early booting stage, the kernel does not give any output or respond to any input. This makes this bug very hard to locate without an external debugger. However, with Raven, we can do the following to locate this bug. To start with, we use the GPIO switch to send an external interrupt to halt the kernel. By observing the *sepc* register, we notice that the kernel is stuck at its exception handler. Then we set up a breakpoint at this exception handler (*handle_exception*) to see what causes the kernel to crash. After the breakpoint is triggered, we find that this unhandled exception is caused by a load fault (*scause=5*) during the initialization of the interrupt controller (`irq-sifive-plic.c`). Thus, we successfully locate this bug via Raven. By introspecting this initialization process with Raven, the incorrect address (`0xa002080`) can be easily identified and fixed. This debugging process is demonstrated in Figure 8.

```
At 0x80202000 0x80202000
[Raven] Input command: b 0xffffffe0002011e8      Exception Handler
[Raven] Input command: c
At 0xffffffe0002011e8 0x804011e8
[Raven] Input command: csrr $sepc
$sepc: 0xffffffe000017d96                        Current Instruction
[Raven] Input command: csrr $scause                                      Relevant Information
$scause: 5                                       Exception Cause         of Exception
[Raven] Input command: csrr $stval
$stval: 0xffffffe000002080                       Exception Address
[Raven] Input command: map 0xffffffe000002080
[Raven] Map of virtual address 0xffffffe000002080 is 0xa002080   Buggy Address
                                                                 (should be 0xc002080)
[Raven] Input command: pr 0xffffffe000017d96
[Raven] *(0xffffffe000017d96)=0x420c   Current Instruction: ld a0 0(a2)
                                       (driver/irqchip/irq-sifive-plic.c)
[Raven] Input command: rr a0
```

Figure 8: Debugging with Raven

5.3 LMbench

To study the overhead imposed by Raven, we ran LMbench [25] with Raven and compared it to vanilla Linux. To simulate a real workload, Raven was configured to synchronize memory mapping change with one dummy breakpoint that does not halt the kernel.

5.3.1 Context Switch Performance Figure 9 shows the context switch latency of the kernel with Raven under different process numbers and sizes; the maximum overhead of Raven is about 5μs. We believe the main slowdown is caused by the synchronization between the physical and virtual addresses which occurs when the kernel performs a context switch and updates the page table.

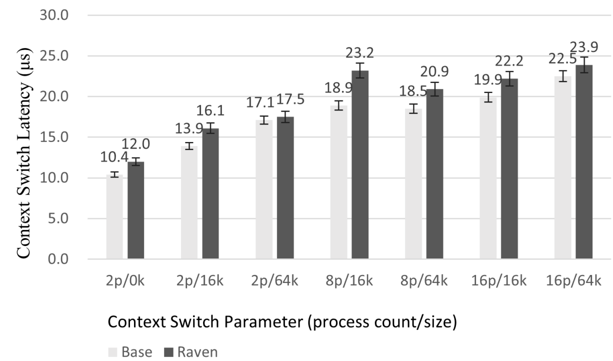

Figure 9: Context Switch Performance

5.3.2 File Operation Performance Figure 10 demonstrates the performance overhead of Raven during file operations. During creation and deletion of files, Raven only imposes a negligible overhead. However, Raven slows down file reading by about 43%. We believe the reason is that the kernel updates its page table frequently when reading a file via `mmap`. This kind of overhead can be reduced by temporarily disable the synchronization mechanism of Raven if the breakpoints or watchpoints are not set in the memory-mapped area.

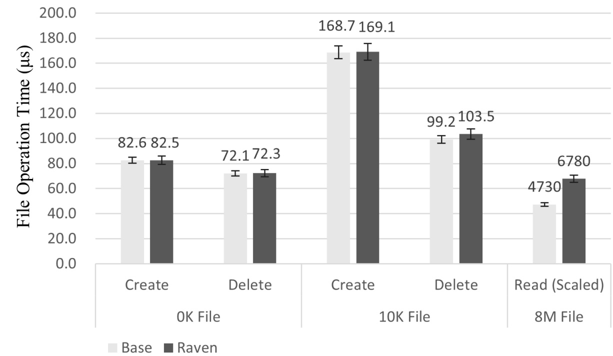

Figure 10: File Operation Performance

5.4 Synchronization Overhead

To further investigate the slowdown caused by Raven, we chose a typical sample of the Linux kernel—its booting process. It is the initialization process of the kernel, which involves numerous page table updates, making it a good sample for evaluating the performance impact of the synchronization between virtual and physical addresses.

Table 4: Booting Process Performance

Metrics (Avg.)	Without Raven	With Raven
Boot Time	4.713s	4.719s
Page Table Update	\	~11000
Time per Updates	\	~0.6μs

From Table 4, we can see that the kernel performs about 11,000 page table updates during booting, and each update takes about $0.6\mu s$ to complete. In contrast to the performance test of LMbench, the overhead of the booting process caused by Raven is only about 0.1%. This is probably because the booting process itself takes a long time to complete, thus leading to a lower overhead.

6 Discussion

6.1 Compatibility

In our approach, the number of PMP entries is essential to the functionality of Raven. In both modes of Raven, the total number of breakpoints and watchpoints is bounded by the number of PMP entries. As shown in Table 5, most modern RISC-V boards support at least 8 PMP entries with 4 KB granularity, which meets the requirement for the more compatible implementation of Raven.

Table 5: PMP Support on RISC-V Boards

RISC-V Board	# PMP	PMP Granularity
QEMU Virtboard [17]	16	4 byte
HiFive Unleashed [15]	8	4 byte
HiFive Unmatched	8	4 kilobyte
HiFive Rev B	8	4 byte
Allwinner Nezha D1 [8]	8	4 kilobyte

6.2 Limitations

Since PMP is not designed for debugging, it brings the following limitations for Raven. (i) Raven cannot achieve instruction-level single stepping since there exist 2-byte instructions in the "C" extension of RISC-V standard and, PMP only has up to 4-byte granularity. Though this can be addressed by disabling this extension or using software breakpoints with PMP as mentioned in Section 3, this still weakens the overall functionality of Raven. (ii) As mentioned in Section 3, with only coarse-granularity PMP available, Raven still needs to use software breakpoints to replace instructions in the kernel. Although these breakpoints are protected by PMP, this still has a noticeable effect on the system, weakening the non-invasive property of Raven.

7 Conclusion & Future Work

We utilize the physical memory protection facilities to implement Raven, a software kernel debugger that fulfills three key properties. (i) Raven is non-invasive: it does not break the overall integrity of the kernel. (ii) Raven directly resides in the firmware and does not require any hypervisor. (iii) Raven is OS-agnostic and can be used with different kernels.

As for future work, we plan to integrate Raven with a standard GDB debugging protocol, which will allow Raven to communicate with GDB clients and improve its debugging efficiency. Apart from that, after this paper is accepted, we plan to open source our work to the community to facilitate kernel debugging.

Acknowledgments

The authors would like to thank Dr. Adrian Rowland and Dr. Zhengyu Ning for proofreading and giving out their valuable suggestions to this paper. This work is supported by the National Natural Science Foundation of China under Grant No.: 62002151, and Science, Technology and Innovation Commission of Shenzhen Municipality under Grant No.: SGDX20201103095408029.

References

[1] 2021. Allwinner Official Website. https://www.allwinnertech.com/
[2] 2021. CKLink Debugger Manual.
[3] 2021. Embedded Trace Macrocell Architecture Specification. https://developer.arm.com/documentation/ihi0014/q/Introduction/About-Embedded-Trace-Macrocells
[4] 2021. HiFive Unmatched - SiFive. https://www.sifive.com/boards/hifive-unmatched
[5] 2021. IDA Pro – Hex Rays Official Website. https://hex-rays.com/ida-pro/
[6] 2021. JTAG Official Website. https://www.jtag.com/
[7] 2021. KGDB Wiki. https://kgdb.wiki.kernel.org/index.php/Main_Page
[8] 2021. Nezha D1 Development Board. https://d1.docs.aw-ol.com/en/d1_dev/
[9] 2021. Nucleisys Official Website. https://www.nucleisys.com/
[10] 2021. Performance Monitor Unit Technical Reference Manual. https://developer.arm.com/documentation/ddi0433/c/performance-monitoring-unit
[11] 2021. Radare Official Website. https://rada.re/n/
[12] 2021. RISC-V Debug Specification. https://github.com/riscv/riscv-debug-spec original-date: 2017-01-20T20:58:54Z.
[13] 2021. RISC-V Instruction Set Manual. https://github.com/riscv/riscv-isa-manual original-date: 2017-02-02T03:24:54Z.
[14] 2021. RISC-V Open Source Supervisor Binary Interface (OpenSBI). https://github.com/riscv-software-src/opensbi original-date: 2018-11-06T00:50:48Z.
[15] 2021. SiFive Official Website. https://www.sifive.com/
[16] 2021. WinDbg Official Website. http://www.windbg.org/
[17] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATC '05)*. USENIX Association, USA, 41.
[18] Asmit De and Swaroop Ghosh. 2021. HeapSafe: Securing Unprotected Heaps in RISC-V. *CoRR* abs/2105.08712 (2021). arXiv:2105.08712 https://arxiv.org/abs/2105.08712
[19] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, Peng Ning, Paul F. Syverson, and Somesh Jha (Eds.). ACM, 51–62. https://doi.org/10.1145/1455770.1455779
[20] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 275–294. https://www.usenix.org/conference/osdi21/presentation/feng
[21] Jinsoo Jang and Brent ByungHoon Kang. 2019. Revisiting the ARM Debug Facility for OS Kernel Security. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 110. https://doi.org/10.1145/3316781.3317897
[22] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 38:1–38:16. https://doi.org/10.1145/3342195.3387532
[23] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 33–49. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning
[24] Carlton Shepherd, Konstantinos Markantonakis, and Georges-Axel Jaloyan. 2021. LIRA-V: Lightweight Remote Attestation for Constrained RISC-V Devices. In *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*. IEEE, 221–227. https://doi.org/10.1109/SPW53761.2021.00036
[25] Carl Staelin. 2005. *lmbench*: an extensible micro-benchmark suite. *Softw. Pract. Exp.* 35, 11 (2005), 1079–1105. https://doi.org/10.1002/spe.665
[26] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2E: Combining Hardware Virtualization and Softwareemulation for Transparent and Extensible Malware Analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (London, England, UK) *(VEE '12)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/2151024.2151053
[27] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *2015 IEEE Symposium on Security and Privacy*. 55–69. https://doi.org/10.1109/SP.2015.11