BootRIST: Detecting and Isolating Mercurial Cores at the Booting Stage

Yihao Luo[1,2,*], Yunjie Deng[1,2,*], Jingquan Ge[4], Zhenyu Ning[3,2], and Fengwei Zhang[2,1(✉)]

[1] Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
{12032472,12032869}@mail.sustech.edu.cn
[2] Department of Computer Science and Engineering, Southern University of Science and Technology, China
zhangfw@sustech.edu.cn
[3] Hunan University, China
zning@hnu.edu.cn
[4] Nanyang Technological University, Singapore
jingquan.ge@ntu.edu.sg

Abstract. With the shrinking of transistor size and their own increasing complexity, CPUs have become more fault-prone. The mercurial cores, one type of processor failure, can randomly return silent miscomputation results and have been discovered in commercial CPUs. To address the mercurial core problem, we propose BootRIST, a software-based solution to detect and isolate mercurial cores in CPUs at the booting stage. BootRIST provides a framework to execute the testing instructions and store the result for each CPU core. Based on the execution result, BootRIST leverages the CPU hotplug mechanism to isolate mercurial cores. With BootRIST, the system and software can correctly run on CPUs with mercurial cores. We have implemented a prototype of BootRIST and deployed it on an Arm Linux server to perform an extensive evaluation. The empirical studies demonstrate that BootRIST can effectively detect and isolate mercurial cores at the booting stage.

1 Introduction

During the very-large-scale integration era, the increasing performance and the decreasing cost of CPUs have made them widely used in servers and endpoint devices. As the basic and essential component of computer systems, CPUs must handle all computation requests and logic operations correctly. Since CPU manufacturers already apply strict and sophisticated testing to CPUs, we naturally think that the CPU is reliable; the system design and software development also implicitly assume that the CPUs are fail-stop [26].

At the same time, the scale and complexity of CPUs constantly increases. Dozens of cores are integrated into a single CPU, and several extensions, such as

* Co-first authors. ✉ The corresponding author.

Single Instruction/Multiple Data (SIMD) and hardware encryption instructions, are supported in modern CPUs. However, the shrinking hardware and complex function increase the probability of physical flaws in processors [3, 4, 6, 21, 28]. Even in a normal environment without extreme radiation, the CPUs have become more fault-prone than before. Google [11] and Facebook [7] have discovered that some server CPUs can repeatedly produce random wrong results without any error logs. Further investigation reveals that the errors typically occur on specific cores on multi-core CPUs rather than the entire CPUs. Such cores are named *mercurial cores* [11].

Although rough statistical analysis shows that the rate of mercurial cores is relatively low (a few mercurial cores per several thousand machines) [30], the large number of servers and cloud machines makes it impossible to ignore the existence of mercurial cores. Moreover, current systems lack an efficient detection and correction mechanism to handle mercurial core errors. Since the errors are produced by CPU computations, the existing fault-tolerance mechanism in the system (e.g., ECC in DRAM and RAID [5] in disk) cannot automatically correct the corruption. Even worse, the undetected corruption can be propagated to memory or files and further induce huge risks. For instance, broken configuration data can cause applications to crash, or a corrupt secret key can cause encrypted critical data to be permanently inaccessible. Consequently, there is an urgent need to address the threats of mercurial cores for server systems. A solution must not only detect mercurial cores effectively but also to keep these defective machines in use to reduce the cost to the cloud service provider.

In this paper, we propose BootRIST, a software-based solution to perform a **R**andom **I**nstruction **S**equence **T**est at the **Boot**ing stage to detect mercurial cores and isolate them to preserve the utility of the system. The design principle of BootRIST is to propose a generic and automatic mercurial core detection and isolation tool that can be widely applied to the existing Linux system. Therefore, the design of BootRIST is based on the common features of Linux and can be integrated into the Linux kernel without any hardware-level modification. Furthermore, BootRIST provides an effective framework to accelerate instruction testing. We slightly modify the Linux exception handler to reduce the switching overhead when an instruction testing error occurs. The modification to the kernel only affects the system at the booting stage, while it introduces virtually no overhead to system runtime.

The execution flow of BootRIST contains three stages: instruction testing stage, detection stage, and isolation stage. The instruction testing stage executes a set of fault-prone instructions and records the execution result for all CPU cores. During the detection stage, BootRIST detects mercurial cores based on the execution result. Without requiring a correct default result to identify mercurial cores, BootRIST applies a voting algorithm to elect a base core. Finally, for the cores with different results from the base core, BootRIST isolates them from the operating system by using the CPU hotplug mechanism.

We implement a prototype BootRIST on openEuler 20.03 [12] with Linux kernel 4.19.90. We deploy the prototype on TaiShan Server [13], an Arm archi-

tecture server with two Kunpeng 920 processors (a total of 96 cores). We first measure the efficiency of instruction testing with different instruction workloads. To further demonstrate the effectiveness of BootRIST, we inject the corruption results to verify the detection function and evaluate the isolation time of mercurial cores. We measure the system performance with a system-level benchmark UnixBench and an application-level benchmark SPEC2017 to demonstrate the overhead caused by BootRIST. The experimental results indicate that BootRIST achieves efficient instruction testing and can detect and isolate mercurial cores in the system. Furthermore, BootRIST only incurs negligible overhead at the system runtime.

We consider the contributions of our work as follows:

- We propose BootRIST, a software-level solution to detect and isolate mercurial cores at the booting stage. This solution does not need to modify the hardware, which can be widely deployed on the existing system.
- We design an efficient instruction testing framework to accelerate the execution of instructions. The framework improves the testing performance and reduces the testing time.
- We implement the BootRIST on an Arm architecture server and evaluate the efficiency and effectiveness of the BootRIST prototype. We also evaluate the BootRIST-enabled system on UnixBench and SPEC2017, and demonstrate that the BootRIST introduces virtually no overhead to the system runtime.

2 Preliminaries

2.1 What is Mercurial Core?

The concept of mercurial cores was initially introduced by Google [11] to describe the intermittent and undetectable miscomputations that occur on certain CPUs. Similar errors on CPUs have also been reported by Facebook [7] and Alibaba Cloud [30]. These errors manifest themselves in processor cores following seemingly innocuous changes in low-level software libraries, such as the utilization of faster but rarely-used instructions to enhance computational performance.

Further investigation shows that mercurial cores behave normally in most cases, while they return corrupt execution errors in some occasional and unpredictable conditions. Currently, the detection of mercurial cores mainly relies on manual reports, while the automatic machine check is less powerful. Without the immediate detection and correction of error results, mercurial cores may expose the computation data to large corruption risks. Researchers speculate that one fundamental reason for the occurrence of mercurial cores is the shrinking of CMOS scaling [24]. As the size of silicon decreases, it approaches physical limitations, thereby increasing the risk of latent failures. Another contributing factor is the growing complexity of processor architecture. The extensive integration of cores within a single CPU and the complex extension of instructions also elevate the risk of mercurial cores.

Solving the mercurial core problems necessitates collaborative efforts from both CPU manufacturers and server vendors. The CPU vendors are responsible for improving manufacturing testing or using the hardware-assisted way to reduce the rate of mercurial cores. As for the server vendors, they should apply the necessary fault-tolerance mechanism or isolation scheme to avoid the potential mercurial core corrupting user data. Existing solutions mainly add redundancy from both hardware and software layers to avoid its impact on the system [9,19]. In this paper, BootRIST provides a software-based isolation scheme for server vendors to preserve the normal function of machines with mercurial cores.

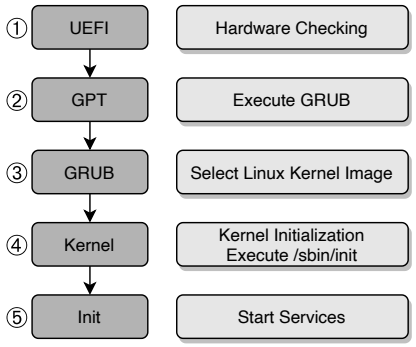

Fig. 1: The booting process of Linux.

2.2 Linux Booting Process

The entire booting process of the Linux system typically contains six steps after the power-up. First, the Unified Extensible Firmware Interface (UEFI) (①) performs the system hardware integrity check and initializes the hardware components. After the check, the BIOS executes and hands over the control to the boot loader in GUID Partition Table (GPT) (②). The MBR or GPT contains information about GRand Unified Bootloader (GRUB), which is loaded and executed in the next steps. Then the GRUB (③) allows the user to select one installed Linux kernel image as the boot image or directly use the default one according to the GRUB configuration file. In addition to the kernel image, the GRUB also loads and executes the initial RAM disk (`initrd`) images, which contain the necessary executables and system files to support the Linux system. Next, the Linux kernel image is executed to initialize the necessary kernel settings (④), such as CPU, MMU, and interrupt. Finally, according to the root file system provided by `initrd`, the Linux kernel executes the first program – `/sbin/init` (⑤) to start the services. Based on the booting process, BootRIST is executed during the kernel initialization stage (④) to detect and isolate mercurial core.

2.3 CPU State and Hotplug in Linux Kernel

For reliability, availability, and serviceability (RAS), the Linux kernel provides the CPU hotplug mechanism to allow the system dynamically plug or unplug CPU cores. The Linux kernel maintains the following four bitmaps to manage all CPU core states.

- *possible*: The CPUs are defined in the Device Tree Source (DTS) and can be plugged into the system.
- *present*: The CPUs that are plugged.
- *online*: The CPUs that are available for scheduling.
- *active*: The CPUs that are available for receiving task migration.

The four aforementioned states are initially set to `true` by default to enable all CPU cores. However, when unplugging a CPU core from the system, the Linux kernel follows a specific process. Firstly, it migrates all running tasks and IRQs away from the core being unplugged. Subsequently, the Linux kernel sets the states of *present*, *online*, and *active* to `false` for the unplugged CPU core, preventing any new tasks from being scheduled or assigned to that core. In this paper, BootRIST remove mercurial cores to preserve the system functionality by leveraging the Linux CPU hotplug mechanism.

2.4 `SIGILL` in Linux System

`SIGILL`, a special signal in the Linux kernel, is typically triggered when processors execute an undefined instruction in user-mode. By default, when a program encounters the `SIGILL` signal, it is interrupted and crashes. However, it is possible for the program to avoid crashing by registering a `SIGILL` handler, which allows for handling the `SIGILL` signal in a customized manner. In this paper, we make use of the undefined instruction and the `SIGILL` handler to interrupt the test program and perform necessary operations during testing.

3 Fault Model and Assumptions

We consider a conservative fault model for transient hardware faults on CPU core. The mercurial core behaves as normal in most cases, while it can introduce silent corruption when executing specific types of instructions [11], such as floating-point instructions, vector instructions, and SIMD instructions. This silent corruption can lead to incorrect results being returned to registers or memory, thereby compromising the reliability of system.

To mitigate the impact of corruption on system runtime, BootRIST aims to detect and isolate the failure by testing fault-prone instructions at the booting stage. We assume that the server provider utilizes a fault-prone instruction set, allowing BootRIST to repeatedly execute a series of random instructions from this set during the booting stage to detect mercurial cores. Similar to existing research on processor failure detection [16, 17, 25], we do not consider faults in the memory subsystem and assume that it is protected by ECC.

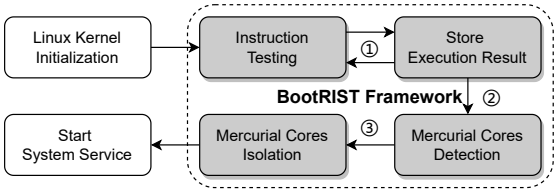

Fig. 2: BootRIST overview.

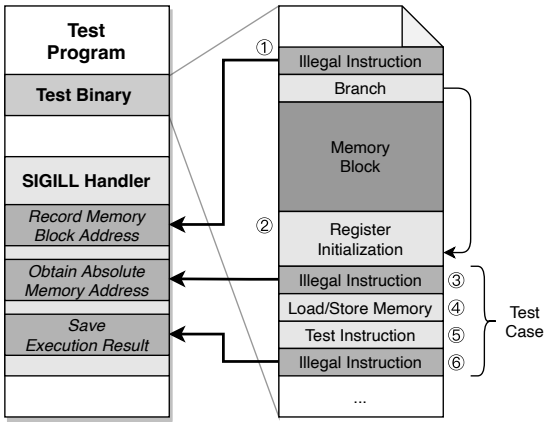

Fig. 3: The details of `risu` workflow. The test program loads and executes the test binary, whose layout is demonstrated on the right side.

4 Design

4.1 System Overview

As illustrated in Figure 2, BootRIST is integrated into Linux kernel and executed during the initialization of Linux kernel. Without affecting the high-level applications, BootRIST finishes the detection and isolation of mercurial cores before starting system services. The execution flow of BootRIST is split into three steps: ① Instruction Testing, ② Mercurial Core Detection, and ③ Mercurial Core Isolation. BootRIST first performs Instruction Testing (①) to execute a series of fault-prone test instructions on all CPU cores, storing the execution result of each CPU core. For the Mercurial Core Detection (②), BootRIST compares the execution result core-by-core and utilizes a voting algorithm to find mercurial cores based on the comparison. When step ② detects the existence of mercurial cores, BootRIST further isolates them from normal cores to avoid unpredictable fault (i.e., ③ Mercurial Core Isolation).

4.2 Instruction Testing

The design of the instruction testing framework is based on `risu` [18], a tool used to find inconsistencies between simulators and real machines. `risu` can generate

a test binary that contains a sequence of instructions that randomly selected from the instruction set. Moreover, `risu` provides a framework for executing the test binary and comparing the resulting execution outputs. Based on the design principle of `risu`, BOOTRIST leverages the `risu` to detect the mercurial core in processor. In this part, we first introduce the workflow of `risu`. Then we describe how to optimize the workflow to improve test efficiency.

4.2.1 `risu` Workflow To achieve the test, `risu` provides a test program to collaborate with the generated test binary. Note that the test binary is not a single executable file, while it requires to be loaded by a test program and executed as a function. As illustrated in Figure 3, we elaborate on important components in the test binary and test program. The test program contains an executable memory to execute the test binary and a `SIGILL` handler to support the dedicated operations and tackle unintentional errors. As for the test binary, it contains the following parts:

- **Memory Block**: A continuous memory region that is used to load and store memory data.
- **Register Initialization**: Initialize all general purpose registers and vector registers with pre-defined values.
- **Test Instruction**: The randomly generated arithmetic instructions according to the given instruction set.
- **Illegal Instruction**: The instructions that are intentionally inserted by `risu` to interrupt the execution and enter the `SIGILL` handler. Based on the type of illegal instruction, the handler can perform the following different operations: records the address of the memory block, obtains the load or store address in the memory block, or saves the value of registers and the memory block. After completing one of the above operations, the handler skips the illegal instructions and continues executing the binary.

In the beginning, the binary contains the first illegal instruction (①), which notifies the `SIGILL` handler to record the base address of the memory block. Following that, the binary executes a branch instruction and proceeds to the register initialization phase where all general purpose registers and vector registers are set. After the initialization, the binary proceeds to execute all test cases until reaching the end.

Each test case comprises memory load/store instructions, test instructions, and illegal instructions. Before accessing the memory, the binary first stores the offset of the memory block in `x0` register and executes a dedicated illegal instruction (③). Based on the given offset and the recorded address of the memory block, the `SIGILL` handler calculates the absolute address, which is then stored in the `x0` register as the target memory address for the subsequent load/store instructions (④). The test instructions (⑤) are executed with the input given by the load/store instructions. After executing the test instructions, the test case reaches the final illegal instructions and notifies the `SIGILL` handler to save execution results in registers and the memory block (⑥).

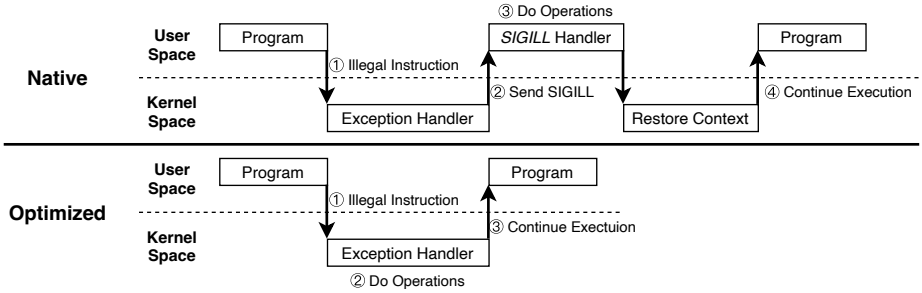

Fig. 4: The workflow comparison of the native `SIGILL` handler and the optimized illegal instructions handler.

4.2.2 Compress Execution Result During the Instruction Testing, BootRIST requires executing a series of test instructions and storing the execution result for each test case. However, directly storing the execution result of all test cases can lead to significant memory consumption. To mitigate this issue, a scheme is devised to reduce memory usage. One possible solution is to execute the program in parallel on each CPU core and compare the results of each test case simultaneously. However, this approach requires strict parallel execution and synchronization among all CPU cores, which can introduce substantial slowdowns to the test progress and reduce the test efficiency.

To tackle these challenges, BootRIST utilizes a hash compression algorithm to reduce memory consumption in storing execution results. The approach involves allocating a dedicated memory region of identical size to the memory block and CPU registers for each CPU core within the Linux kernel. At the beginning of the test program, BootRIST initializes the hash state of the allocated memory. When the `SIGILL` handler receives the request to save the execution result, BootRIST employs a message digest algorithm to update the hash state based on the current state, as well as the values stored in the registers and memory block. After executing all test cases, the hash states in the reserved memory regions are used to identify any inconsistencies in execution. As a result, BootRIST effectively reduces the memory required to store the execution results while ensuring the discrimination of mercurial core.

4.2.3 Optimize the `SIGILL` Handler In `risu`, the `SIGILL` handler is used to perform the specific operations base on the type of illegal instruction. However, the test program's frequent interactions with the `SIGILL` handler can introduce additional complexity and result in a slowdown during testing. To address this issue, we aim to design an efficient illegal instruction handler that improves test performance by reducing the overhead associated with entering the `SIGILL` handler.

To optimize the test performance, we begin by examining the workflow that utilizes the `SIGILL` handler to handle the illegal instructions. As illustrated on the top side of Figure 4, once the program executes an illegal instruction (①), the

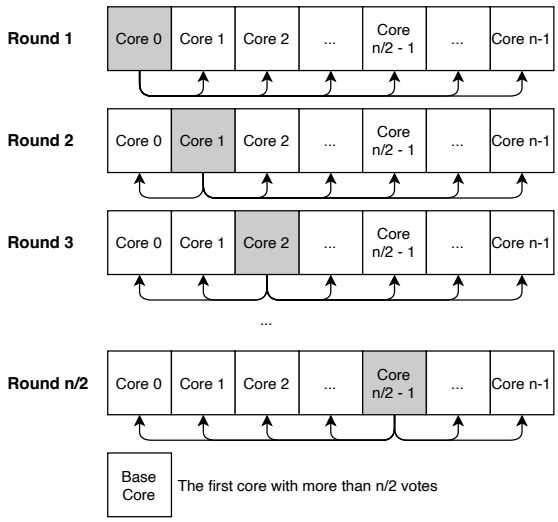

Fig. 5: Voting algorithm for mercurial core detection.

CPU generates an exception and traps the system into Linux exception handler. The exception handler stores the program context, and sends the `SIGILL` signal with the saved context registers to the program `SIGILL` handler (②). With the given context registers values, the `SIGILL` handler can identify the type of illegal instruction and perform corresponding operations (③). Once the `SIGILL` signal is handled appropriately, the handler returns to the kernel space. Subsequently, the kernel restores the program's state and resumes its execution (④). However, the complex Linux signal mechanism and frequent switch between the user space and kernel space can significantly slow down the test procedure. To address this, we improve the test performance by replacing the `SIGILL` handler with a efficient illegal instruction handler.

As shown in the bottom side of Figure 4, we integrate the necessary `risu` operations into the Linux exception handler. For the illegal instruction that comes from BootRIST test, the exception handler directly performs the operations base on the type of illegal instruction. After finishing the operation, the exception handler skips the illegal instruction and resumes the program execution without sending the `SIGILL` signal to the program. Note that the modification of exception only affects the `SIGILL` raised by BootRIST, while the `SIGILL` from other programs works as normal. Consequently, BootRIST reduces the redundant context switch and eliminates the heavy signal mechanism, which improves the test performance and can find the mercurial core efficiently.

4.3 Mercurial Core Detection

To detect mercurial cores, a straightforward approach is to compare the execution results with pre-calculated correct results. However, this method requires executing tests on a fault-free machine to obtain accurate reference results, which

adds extra effort. Considering that the occurrence rate of mercurial cores is typically low, it implies that the majority of cores will produce correct results. Exploiting this observation, we can employ a voting algorithm to identify the mercurial core without the need for additional pre-calculation efforts. By comparing the execution results core-by-core, the voting algorithm determines the cores that deviate from the expected outcome and identifies them as mercurial cores. This approach allows for efficient mercurial core detection without relying on pre-calculated results.

Figure 5 illustrates the process of the voting algorithm. We label the cores from 1 to n and apply the voting algorithm $\frac{n}{2}$ rounds, where n is the number of processor cores. During round i, we compare the result of core i with all other cores. The vote of core i is counted by the number of the same execution results compared with core i. For the first core with more than $\frac{n}{2}$ votes, BootRIST marks it as the base core and regards its execution result as reliable. Then the cores with different results from the base core are marked as mercurial cores, which will be isolated in the later stage. Note that the algorithm proceeds at most $\frac{n}{2}$ rounds since the base core cannot exist in the last $\frac{n}{2}$ cores. If no core can obtain more than $\frac{n}{2}$ votes, we regard the result is contaminated by unexpected factors, such as the unintentional modification to the memory from other program, because the rate of mercurial cores is typically low. In this case, BootRIST cleans all recorded results and executes the Instruction Testing and Mercurial Core Detection again to find mercurial cores.

4.4 Mercurial Core Isolation

The Mercurial Core Isolation is executed only if the detection result indicates the existence of mercurial cores. BootRIST leverages the CPU hotplug mechanism to isolate mercurial core. More specifically, BootRIST first migrates all the running tasks and IRQ in mercurial cores to other normal cores. After the migration, we change the *present*, *online*, and *active* state to `false`, disabling mercurial cores. As a result, any tasks can not be assigned to the unplugged mercurial core by scheduling or task binding.

5 Implementation

Section 4 focuses on the design principle of the BootRIST. In this section, we further demonstrate the necessary implementation details of our design.

5.1 Instruction Generation

To detect Mercurial cores, we need to generate error-prone instruction test cases. In AArch64 execution mode, ARMv8 runs the A64 instruction set. A64 instruction set consists of 32-bit fixed-length encoded instructions and uses 64-bit addressing mode. Each instruction encoding includes bits for Opcode, Condition Code, Immediate, Register, Shift Operation, etc. For a specific instruction, some

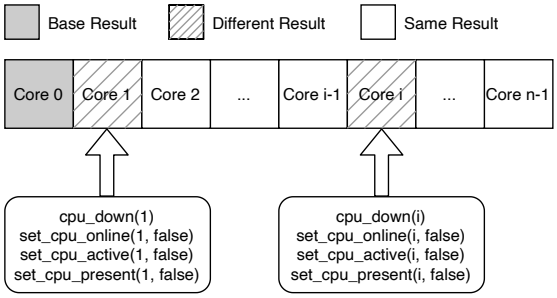

Fig. 6: Mercurial core isolation.

bits, like the Opcode, are fixed, while others, such as register and immediate values, are variable. The Risu tool provides us with instruction templates for the A64 instruction set, detailing the encoding format and value constraints for the variable fields of each instruction. By parsing these templates, we can select the types and numbers of instructions we want to test and generate multiple different legal instructions based on the constraints for testing.

5.2 Parallel Execution

To improve the test performance, we implement the Instruction Testing with parallel execution on each CPU core. During the kernel initialization, BootRIST first leverages the kernel function *call_usermodehelper()* to execute the testing program. Then we duplicate the program process n times through leveraging the *fork()* function, where n is the number of CPU cores. For each forked process, we utilize the *sched_setaffinity()* function to bind it to different processor cores. Consequently, the test procedure can be parallel executed on all CPU cores.

5.3 Modification to Exception Handler

As mentioned in Section 4.2.3, we integrate the illegal instruction handler into Linux exception handlers to improve the test performance. Therefore, we modify the kernel exception handler *do_undefinstr()* to handle the intentionally inserted illegal instruction and process the dedicated operations, such as recording execution results and obtaining the memory block address. Note that we only migrate the illegal instruction handler part into kernel exception handlers, while the `SIGILL` handler is still preserved to tackle the rare unintentional execution error.

5.4 Mecurial Core Isolation

BootRIST achieves core isolation by leveraging the CPU hotplug mechanism in Linux kernel. Figure 6 presents an example of mercurial core detection result and illustrates how to isolate mercurial cores in detail. From the figure, we can

see that core 0 is the base core, while core 1 and core i are mercurial cores as their execution results are different from the base result. BootRIST first utilizes the function *cpu_down()* to migrate the running task and IRQ from mercurial cores to normal cores. Then we switch the *present*, *online*, and *active* state to false to further remove the core from the scheduling list and unplug the core from the system. The switch of core states is achieved by employing Linux kernel APIs *set_cpu_online()*, *set_cpu_active()* and *set_cpu_present()*. Consequently, BootRIST can isolate mercurial cores from the normal cores and preserve the system's functionality.

6 Evaluation

In this section, we detail the performance evaluation of BootRIST. First, our evaluation environment is introduced in Section 6.1. In Section 6.2, we demonstrate the category of testing instructions and the testing performance of our framework. Furthermore, we measure the time consumption of mercurial core isolation in Section 6.3. Finally, we evaluate the system performance overhead caused by the deployment of the BootRIST in Section 6.4.

6.1 Evaluation Environment

We select Huawei TaiShan 200 (Model 2280) Server [13] as our experimental platform. The platform has two HiSilicon Kunpeng 920-4826 CPUs [10] with a total of 96 ARMv8-A cores. Moreover, it equips 382 GB of DDR4 memory and 2.1TB of hard drive storage. To record the execution result, we reserve 8,192 bytes and 1,024 bytes memory to store the hash state of the memory block and register values for each CPU core (i.e., 0.75 MB and 96 KB in total), respectively. We prototype BootRIST based on Linux kernel 4.19.90 running in openEuler 20.03 [12].

6.2 Performance of Instruction Testing

We leverage `risu` [18] to randomly generate a various number of test instructions from the Arm instruction set [1, 2] to demonstrate the test performance. As previous work suggests that mercurial cores are more likely to be observed when executing complex instructions [11, 27] (e.g., vector operations, floating-point instructions), we use such instructions as a fault-prone instruction set to generate test binary. More specifically, the test binary contains the floating-point operations instructions, Single Instruction/Multiple Data (SIMD), and Scalable Vector Extension (SVE) instructions.

To demonstrate the test performance, we evaluate the performance of BootRIST with a different number of test instructions and compare the execution time between BootRIST and different experiment settings in Table 1. The column **Instructions** demonstrates the number of test instructions generated by the `risu` [18]. The column **BootRIST** and `risu` demonstrate the performance

Table 1: Comparison of instruction test time (seconds) and its percentage in total booting time.

Instructions	BootRIST		risu [18]	
	Test Time	Percentage	Test Time	Percentage
10	0.01	0.01%	0.01	0.01%
100	0.02	0.01%	0.03	0.02%
1000	0.18	0.11%	0.23	0.14%
10000	1.86	1.13%	2.31	1.40%
100000	18.78	10.34%	23.10	12.42%
500000	93.75	36.53%	115.36	41.46%

of instruction testing on BootRIST and native `risu`, respectively. Note that **BootRIST** leverages the exception handler to handle the illegal instruction, while `risu` uses the `SIGILL` handler. As shown in the table, the comparison between BootRIST and `SIGILL` demonstrates that our efficient illegal instruction handler achieves approximately 20% acceleration to the instruction testing. The experimental results show that BootRIST provides an efficient instruction test framework. Furthermore, we measure the time consumption of the booting process and evaluate the percentage of test time in total booting time. The native Linux system takes 162.87 (s) to boot. And the **Percentage** columns in the Table 1 show that BootRIST does not severely increase the booting time. Since the reboot in server systems is infrequent, the instruction testing at the booting stage would not significantly affect high-level users.

6.3 Mercurial Core Detection and Isolation

To demonstrate the effectiveness of BootRIST, we further evaluate the detection and isolation of mercurial cores. We manually emulate mercurial cores by injecting faults into the execution result. For the broken execution results, BootRIST regards the corresponding cores as mercurial cores and isolates them from other cores.

We further evaluate the time consumption caused by isolating a various number of mercurial cores. Recall from Section 4.3 that our voting algorithm rejects the discrimination result if the number of mercurial cores is equal to or greater than half of the total number of cores. Since our experimental platform equips with 96 cores, we evaluate the isolation time for no more than 47 mercurial cores and present the result in Figure 7. The figure shows that the isolation time almost increases linearly with the increasing number of cores. Moreover, the isolation time is less than 2 seconds even if we isolate 47 cores, demonstrating that our mercurial core isolation scheme is efficient.

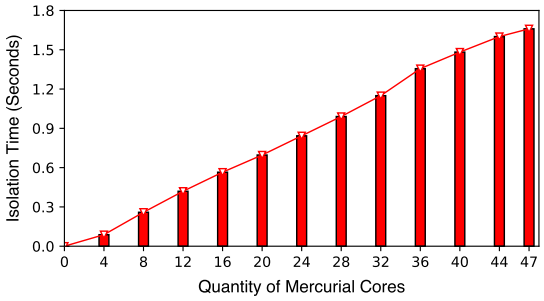

Fig. 7: Isolation time of mercurial cores.

6.4 System Performance Analysis

We further evaluate the impact of BootRIST with system-level and application-level benchmarks to demonstrate that the deployment of BootRIST incurs virtually no overhead to the system runtime.

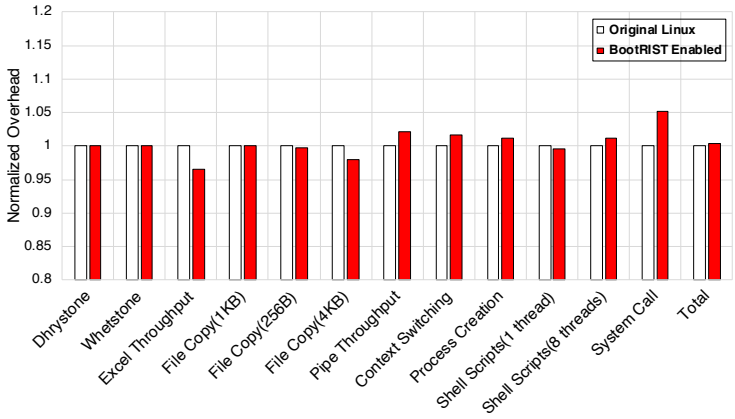

Fig. 8: The performance overhead of BootRIST on UnixBench.

UnixBench [15]: UnixBench is a benchmark tool to evaluate the performance of a Unix-like system. To demonstrate the performance overhead of BootRIST on the system, we perform UnixBench to evaluate the affection to critical system operations. Figure 8 illustrates the comparative evaluation of BootRIST relative to the native Linux system on UnixBench. The white column and red column represent the Linux system with native settings and the Linux system with BootRIST, respectively. As shown in Figure 8, the deployment of BootRIST does not incur significant overhead on all test items. The largest overhead occurs in the *System Call* item because all system calls execute the `svc` instruction and enter the exception handler in Arm Linux. The modification

of the exception handler incurs such overhead, which can be futher reduced by optimizing the BootRIST code added in the exception handler. Nevertheless, the total overhead indicates that BootRIST only incurs negligible overhead. The experiment results demonstrate that BootRIST incurs almost no impact on the system runtime.

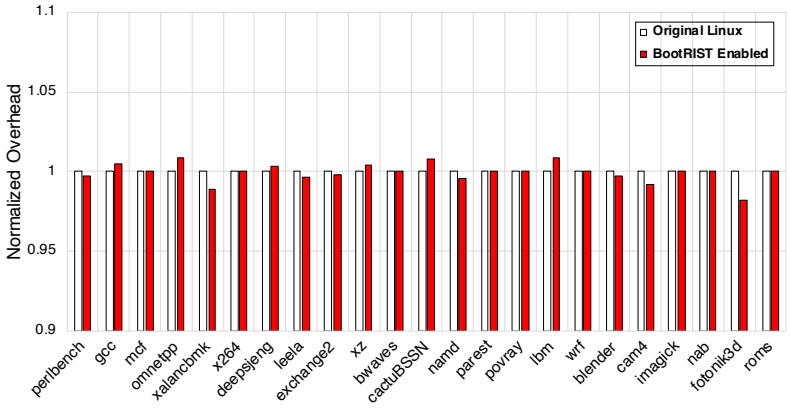

Fig. 9: The performance overhead of BootRIST on SPEC2017.

SPEC_CPU 2017 [29]: SPEC_CPU 2017 benchmark package provides testing suites to measure the performance of compute-intensive real user applications. Therefore, we further employ SPEC_CPU 2017 to evaluate the impact of BootRIST on the user application performance. We evaluate all of SPEC_CPU 2017 INT and FP applications and present the result in Figure 9. As illustrated in Figure 9, the performance of BootRIST and original Linux system are almost same, and only present some minor fluctuations on different user applications. Moreover, the average performance overhead shows that BootRIST introduces virtually no overhead on the user application.

7 Discussion and Future Work

The design principle of BootRIST is to provide an automatic mercurial core detection framework. We achieve the automatic testing and detection at the booting stage with pre-generated instruction binary, which contains a series of fault-prone instructions. However, we consider that the fault-prone instruction coverage may not be enough to discover mercurial cores. The two aspects mainly cause insufficient coverage. The first aspect is the limitation of testing time. Since BootRIST is running at the booting stage, the testing time cannot occupy the booting process too long to affect the normal usage of the system. The second aspect is the incomplete summary of the fault-prone instruction set. We assume that the server provider already knows the fault-prone instruction set. With such

a set, the mercurial cores can be detected even if the reboot is infrequently and the booting time is short. However, the investigation of the fault-prone instruction set relies on manual bug reports [11], which is less efficient and is hard to discover all fault-prone instructions. Therefore, we consider future work can improve the fault-prone instruction coverage by prolonging the testing time or designing an automatic way to complete the fault-prone instruction set. For instance, BootRIST can integrate with an online instruction generator to achieve an online test and leverage fuzzing technologies to guide the discovery of fault-prone instructions.

8 Related Work

8.1 Processor Failure

The previous studies have proposed some schemes to detect unexpected processor failure. Farron [30]is the first work to observe and characterize SDC on large-scale server clusters. Nostradamus [20] employs extra hardware to overcome the computation errors in out-of-order processor cores. It predicts the expected result of instruction, and corrects the output if a failure occurs. Besides the hardware solution, some works [16, 17, 22, 25] design a software-level redundancy scheme to overcome the soft-error. They modify the compiler to leverage the unused instruction-level resources to add redundancy. SiliFuzz [27] leverages fuzzing technology to detect potential mercurial cores. They use the known bad machines to guide the test generator and create fault-prone test cases. OpenDCDiag [23] is an open-source framework to find defects and bugs in CPUs. OpenDCDiag requires user manually writing test code and providing pre-calculated golden result. The CPUs failure is detected by comparing the execution result with the golden result. Intel In-Field-Scan can run circuit level test on a CPU core to detect processor failures. However, it only supports on a part of upcoming Xeon Sapphire Rapids processor, which means it can not be applied to other Intel CPUs or other manufacturers' CPUs (such as AMD and Arm).

8.2 Discover Inconsistent Execution

Several works are proposed to discover inconsistent execution to improve reliability. iScanU [8] aims to find undocumented instructions on RISC processors. It executes a single instruction and determines whether it is an undocumented instruction based on whether the SIGLL signal is triggered. EXAMINER [14] focuses on finding the inconsistent execution between simulators and real machines. It leverages the Arm architecture specification language (ASL) to generate representative instruction streams to find inconsistent instruction efficiently. `risu` [18] is an open-source random user-space instruction generator intended to test the implementation of simulators of Arm architecture. In BootRIST, we leverage the `risu` to randomly generate test instructions on a fault-prone instruction set and find the inconsistent execution between CPU cores to detect mercurial cores.

9 Conclusions

With the shrinking of transistor size and their own increasing complexity, CPUs have become more fault-prone. Recently, the mercurial cores, which are caused by permanent hardware faults and may return the corrupt result, have been discovered in modern CPUs. The existence of mercurial cores brings a huge data risk to the system and can severely affect the system's functionality. Therefore, we propose BootRIST, a software-based solution to detect and isolate mercurial cores at the booting stage, and can be integrated into the existing system. We design an efficient instruction testing framework to improve the testing pressure and reduce the testing time. Furthermore, without requiring the default correct execution result, we design a voting algorithm to identify the possible mercurial core. Moreover, BootRIST fully leverages the Linux CPU hotplug mechanism provided to isolate mercurial cores from normal cores. Based on the design, we implement a prototype BootRIST on a server, as well as perform the experimental studies. The experiment results demonstrate that BootRIST achieves efficient instruction testing and effective mercurial core detection and isolation on the system. Furthermore, our evaluation demonstrates that the deployment of BootRIST only incurs virtually no overhead to the system.

Acknowledgments

We would like to thank the anonymous reviewers and COMPASS members for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No.62372218 and No.62102175.

References

1. Arm: Arm A64 Instruction Set Architecture. `https://developer.arm.com/documentation/ddi0596/latest`
2. Arm: Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. `https://developer.arm.com/documentation/ddi0487/latest/`
3. Bohr, M.T., Young, I.A.: CMOS scaling trends and beyond. IEEE Micro **37**(6), 20–29 (2017)
4. Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro **25**(6), 10–16 (2005)
5. Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H., Patterson, D.A.: RAID: High-performance, reliable secondary storage. ACM Computing Surveys (CSUR) **26**(2), 145–185 (1994)
6. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. IEEE micro **23**(4), 14–19 (2003)
7. Dixit, H.D., Pendharkar, S., Beadon, M., Mason, C., Chakravarthy, T., Muthiah, B., Sankar, S.: Silent data corruptions at scale. arXiv preprint arXiv:2102.11245 (2021)
8. Dofferhoff, R., Göebel, M., Rietveld, K., Van Der Kouwe, E.: iScanU: A portable scanner for undocumented instructions on risc processors. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 306–317. IEEE (2020)

9. Dong, S., Kryczka, A., Jin, Y., Stumm, M.: Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In: 19th USENIX Conference on File and Storage Technologies (FAST 21). pp. 33–49 (2021)
10. HiSilicon: Kunpeng 920-4826 - hisilicon. https://en.wikichip.org/wiki/hisilicon/kunpeng/920-4826
11. Hochschild, P.H., Turner, P., Mogul, J.C., Govindaraju, R., Ranganathan, P., Culler, D.E., Vahdat, A.: Cores that don't count. In: Proceedings of the Workshop on Hot Topics in Operating Systems. pp. 9–16 (2021)
12. HUAWEI: openEuler. https://www.openeuler.org/en/
13. HUAWEI: Taishan Server. https://e.huawei.com/en/products/servers/taishan-server
14. Jiang, M., Xu, T., Zhou, Y., Hu, Y., Zhong, M., Wu, L., Luo, X., Ren, K.: EXAMINER: automatically locating inconsistent instructions between real devices and CPU emulators for ARM. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 846–858 (2022)
15. Kelly Lucas: byte-UnixBench. https://github.com/kdlucas/byte-unixbench
16. Kuvaiskii, D., Faqeh, R., Bhatotia, P., Felber, P., Fetzer, C.: HAFT: Hardware-assisted fault tolerance. In: Proceedings of the Eleventh European Conference on Computer Systems. pp. 1–17 (2016)
17. Kuvaiskii, D., Oleksenko, O., Bhatotia, P., Felber, P., Fetzer, C.: ELZAR: Triple modular redundancy using intel avx (practical experience report). In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 646–653. IEEE (2016)
18. Linaro: risu. https://git.linaro.org/people/pmaydell/risu.git/
19. Mamone, D., Bosio, A., Savino, A., Hamdioui, S., Rebaudengo, M.: On the analysis of real-time operating system reliability in embedded systems. In: 2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). pp. 1–6. IEEE (2020)
20. Nathan, R., Sorin, D.J.: Nostradamus: Low-cost hardware-only error detection for processor cores. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2014)
21. Nightingale, E.B., Douceur, J.R., Orgovan, V.: Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs. In: Proceedings of the sixth conference on Computer systems. pp. 343–356 (2011)
22. Oh, N., Shirvani, P.P., McCluskey, E.J.: Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability **51**(1), 63–75 (2002)
23. OpenDCDiag: OpenDCDiag. https://github.com/opendcdiag/opendcdiag
24. Papadimitriou, G., Gizopoulos, D.: Silent data corruptions: Microarchitectural perspectives. IEEE Transactions on Computers (2023)
25. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software implemented fault tolerance. In: International symposium on Code generation and optimization. pp. 243–254. IEEE (2005)
26. Schlichting, R.D., Schneider, F.B.: Fail-stop processors: An approach to designing fault-tolerant computing systems. ACM Transactions on Computer Systems (TOCS) **1**(3), 222–238 (1983)
27. Serebryany, K., Lifantsev, M., Shtoyk, K., Kwan, D., Hochschild, P.: SiliFuzz: Fuzzing CPUs by proxy. arXiv preprint arXiv:2110.11519 (2021)

28. Shivakumar, P., Kistler, M., Keckler, S.W., Burger, D., Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proceedings International Conference on Dependable Systems and Networks. pp. 389–398. IEEE (2002)
29. SPEC: Spec cpu 2017. https://www.spec.org/cpu2017/
30. Wang, S., Zhang, G., Wei, J., Wang, Y., Wu, J., Luo, Q.: Understanding silent data corruptions in a large production cpu population. In: Proceedings of the 29th Symposium on Operating Systems Principles. pp. 216–230 (2023)