PESC: A Per System-Call Stack Canary Design for Linux Kernel

Jiadong Sun, Xia Zhou, Wenbo Shen, Yajin Zhou, Kui Ren
Zhejiang Univeristy, China
{simonsun,zhouxia_icsr,shenwenbo,yajin_zhou,kuiren}@zju.edu.cn

ABSTRACT

Stack canary is the most widely deployed defense technique against stack buffer overflow attacks. However, since its proposition, the design of stack canary has very few improvements during the past 20 years, making it vulnerable to new and sophisticated attacks. For example, the ARM64 Linux kernel is still adopting the same design with StackGuard [27], using one global canary for the whole kernel. The x86_64 Linux kernel leverages a better design by using a per-task canary for different threads. Unfortunately, both of them are vulnerable to kernel memory leaks. Using the memory leak bugs or hardware side-channel attacks, e.g., Meltdown or Spectre, attackers can easily peek the kernel stack canary value, thus bypassing the protection.

To address this issue, we proposed a fine-grained design of the kernel stack canary named PESC, standing for **Pe**r-**S**ystem-Call **C**anary, which changes the kernel canary value on the system call basis. With PESC, attackers cannot accumulate any knowledge of prior canary across multiple system calls. In other words, PESC is resilient to memory leaks. Our key observation is that before serving a system call, the kernel stack is empty and there are no residual canary values on the stack. As a result, we can directly change the canary value on system call entry without the burden of tracking and updating old canary values on the kernel stack.

Moreover, to balance the performance as well as the security, we proposed two PESC designs: one relies on the performance monitor counter register, termed as PESC-PMC, while the other one uses the kernel random number generator, denoted as PESC-RNG. We implemented both PESC-PMC and PESC-RNG on the real-world hardware, using HiKey960 board for ARM64 and Intel i7-7700 for x86_64. The synthetic benchmark and SPEC CPU2006 experimental results show that the performance overhead introduced by PESC-PMC and PESC-RNG on the whole system is less than 1%.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

kernel, buffer overflow, stack canary, system call

Wenbo Shen is the corresponding author.

CODASPY '20, March 16–18, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7107-0/20/03…$15.00
https://doi.org/10.1145/3374664.3375734

ACM Reference Format:
Jiadong Sun, Xia Zhou, Wenbo Shen, Yajin Zhou, Kui Ren. 2020. PESC: A Per System-Call Stack Canary Design for Linux Kernel. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy (CODASPY '20), March 16–18, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3374664.3375734

1 INTRODUCTION

Memory corruption bugs are one of the oldest computer security problems [41]. These bugs give the attacker opportunities to inject new code, change the control flow, or tamper with the data. Among all memory corruption bugs, buffer overflow is the most common one. The first computer worm Morris worm was using a buffer overflow bug to escalate the privilege on computer systems in 1988 [16]. Due to the stack memory layout, the buffer overflow on the stack enables the attacker to tamper with the security-critical variable, i.e., the return address, stored on the stack. This allows the attacker to subvert the control flow [37]. To defend against the stack buffer overflow, Cowan et al. proposed StackGuard [27] in 1998, which utilizes the compiler to insert a *canary* word (a.k.a. stack cookie) between the allocated buffer and the return address. By doing so, any overflow from the buffer to the return address will be detected by checking the canary value when a function returns. Since then, the canary based protection has been widely adopted and deployed on most of the computer systems. It is considered as one of the most commonly used techniques against stack buffer overflow attacks.

However, even after more than 20 years, the evolution of canary based protection is lagging far behind. Until the Linux kernel v4.19, it still uses a global universal canary (termed as ***global canary***) for the ARM64 architecture, which is the same with the original design proposed in 1998. In contrast, the attacking techniques have become much more sophisticated than the ones in 20 years ago, with new types of memory leaks [23] and hardware bugs [33]. Better than ARM64 kernel, x86_64 Linux kernel adopts a fine-grained canary design by assigning different canaries for each thread when forking (termed as ***per-task canary***).

Sadly, both global canary and the per-task canary designs are vulnerable to memory leaks. For example, in the global canary design, the canary is assigned to a random value at kernel boot-up and never changes. All processes and threads from user space or kernel space share the same canary. As a result, one stack leak allows the attacker to learn the canary values for all stacks of all threads on Linux. Even for per-task canary, once the value is assigned to a thread, it never changes, allowing the attacker to exploit kernel stack leak vulnerabilities or side-channel attacks to probe the per-task canary and infer the canary values.

Unfortunately, such information leak vulnerabilities are common nowadays. For example, the incorrect check of `/proc/pid/stack` allows the attacker to leak kernel stack content, including the canary

value on the stack content [5, 46]. The uninitialized memory in KVM leaks the stack memory in x86 Linux kernel [12]. Security researchers have demonstrated more practical attacks against the kernel stack canary via memory leaks in 2018 [3] and INFILTRATE Security Conference 2019 [6]. Besides these software vulnerabilities, the recent infamous Meltdown [33] or Spectre [32] and their numerous variants, allow attackers to infer the kernel canary values by probing the kernel canary address, without triggering any kernel software bugs.

With the leaked canary value, the kernel stack overflow attack will roll back to the old days. The attacker can easily craft a special payload containing the overflow payload and the correct kernel canary values, therefore bypasses all canary checks and launches the buffer-overflow attack successfully. On one side, the stack overflow vulnerabilities are still very common. Since 2014, Google project zero team has reported more than 40 stack overflow vulnerabilities [13], including mainstream operating system kernels, such as Linux kernel, iOS kernel, and Windows kernel. As a result, the existence of the stack overflow, and information leak vulnerabilities make it easy for attackers to bypass the widely deployed stack canary based defense. Due to this reason, there is a pressing need to improve its design and implementation.

To this end, in this paper, we proposed a memory leak resilient kernel stack canary design, named as PESC. PESC represents **Pe**r-**S**yscall **C**anary, in which a **new** random stack canary will be generated for each system call. Compared with the global canary and per-task canary, PESC has two main advantages. First, by changing the canary value on every system call entry, PESC invalids the leaked old stack canary, so that attackers cannot accumulate any knowledge of prior canary across system calls. Second, as the stack canary will be generated on the fly when the attacker is trying to trigger the stack overflow via a system call, he or she has no way to obtain the new stack canary value beforehand, even though the kernel contains memory leak vulnerabilities or Meltdown/Spectre related hardware problems. In other words, under PESC, attackers cannot reuse leaked canaries, nor can they trigger vulnerabilities to leak canaries beforehand.

Even though the basic idea of generating a new stack canary for each system call is conceptually simple, PESC still needs to resolve two technical challenges. First, it is hard to choose the critical code location to change the kernel stack canary during a system call. In kernel, when calling a function with point-to address writes (using arrays or pointers), the compiler will automatically generate code to push stack canary at function prologue and check it at function epilogue [9]. We cannot check the canary here as the old canary is already on the kernel stack. Changing canary value will lead to stack canary check failure. In fact, the canary value can only be changed when no canary value has been pushed to stack yet. To resolve this problem, we propose to change the canary value at the `kernel_entry` point, where the kernel stack is empty.

Second, we need to balance the performance as well as the randomness of the newly generated canary. PESC requires to generate a new canary with enough randomness at each system call. considering that system calls happen frequently, the system call dispatcher is carefully designed to be short and efficient, usually less than 100 instructions. However, generating a random number in the kernel usually involves multiple functions, with hundreds of lines of code. Therefore, invoking the heavy random number generation functions at the system call dispatcher will introduce certain performance overhead. To improve the performance, we proposed two PESC designs: PESC-PMC and PESC-RNG. PESC-PMC relies on the **P**erformance **M**onitor **C**ounter to generate the new canary, which makes it a lightweight design, adding only a couple of instructions to the kernel entry code. PESC-RNG relies on kernel **R**andom **N**umber **G**enerator to produce the new canary. It adds certain performance overhead to every system call but has a fully randomized canary. Our evaluation shows that for both PESC-PMC and PESC-RNG implementations, the performance overhead to the whole system is less than 1%.

The contributions of this paper are in three-fold:

- We proposed a more fine-grained kernel stack canary design named PESC (**PE**r-**S**ystem call **C**anary), which changes the kernel stack canary on the system call basis, making it impossible for the attacker to accumulate canary knowledge between system calls.
- To balance security and performance, we proposed the performance counter register based design termed as PESC-PMC and the kernel random number generator based design denoted as PESC-RNG. We further implemented both designs for ARM64 and x86_64 Linux kernel on real-world hardware.
- We evaluated the implemented designs for both ARM64 and x86_64 Linux kernel. The Android synthetic benchmark experiments show that the average performance overhead of PESC-PMC and PESC-RNG are 0.27% and 0.40%, respectively. For x86_64 implementation, the performance overhead of SPEC CPU2006 experiments are 0.09% and 0.15%.

2 BACKGROUND

In this section, we will introduce the necessary background knowledge, including buffer overflow, kernel stack canary, Linux kernel system call handling and the performance monitor counter.

2.1 Buffer overflow and stack canary

Buffer overflow is a common type of memory corruption bugs. The root cause is missing boundary check. A stack buffer overflow usually happens when the buffer is allocated on stack and allows to write without any size checking. As a result, the buffer write can go beyond the boundary of the allocated buffer, causing the stack buffer overflow. For example, the attacker can exploit the stack buffer overflow to overwrite the return address with the address of shellcode, or gadget addresses which are used to perform return-oriented programming attacks [18].

To defend against stack buffer overflow, a special value named stack canary (a.k.a, stack cookies) has been used as "a canary in a coal mine", to warn any stack buffer overflow. More specifically, the stack canary is a value that is inserted between the buffer and the return address, so that the overflow to the return address has to go through the canary. As a result, the canary value will be changed, and the overflow can be detected by comparing the canary values when the function returns.

The stack canary design has been widely deployed and is the most widely used stack buffer overflow defense technique. However,

```
1  canary_test:
2    ......
3    adrp  x19, ffffff800907d000
4    add   x19, x19, #0x6c8
5    ldr   x6, [x19]
6    str   x6, [x29,#104]
7    ...   function body   ...
8    ldr   x1, [x29,#104]
9    ldr   x0, [x19]
10   eor   x0, x1, x0
11   cbnz  x0, ffffff800860d7dc
12   ......
13   bl    ffffff80080b1d90 <__stack_chk_fail>
```

Figure 1: Compiler inserted canary logic of ARM64.

```
1  canary_test:
2    ......
3    mov    %gs:0x28,%rax
4    mov    %rax,0x40(%rsp)
5    ...   function body   ...
6    mov    0x40(%rsp),%rax
7    xor    %gs:0x28,%rax
8    jne    ffffffff815df1cf
9    add    $0x48,%rsp
10   retq
11   callq  ffffffff810611b0 <__stack_chk_fail>
12   ......
```

Figure 2: Compiler inserted canary logic of x86_64.

it is vulnerable to memory leaks. Once the stack canary gets leaked, the attacker can bypass the stack canary checking easily.

2.2 Current status of kernel canary

The Linux kernel v4.19 (released in October 2018) adopts two canary designs to protect kernel stacks: global canary for ARM64 and per-task canary for x86_64.

2.2.1 Global Canary. ARM64 (a.k.a., AArch64) Linux kernel v4.19 uses one single global canary variable `__stack_chk_guard` for kernel stacks of all processes.

Canary initialize: ARM64 Linux kernel initializes the global canary during kernel boot-up. The first kernel function `start_kernel` will call the function `boot_init_stack_canary` to assign a pseudo-random value to `__stack_chk_guard`. It is worth noting that after the initialization, the value of the global canary `__stack_chk_guard` will never change until kernel reboots. In other words, the same canary value will be used for all processes, all the time. As a result, if the kernel canary is leaked from one process, the attacker is able to know the canary used for all processes and launch further attacks.

Canary use: The ARM64 compiler will insert the canary logic automatically during compiling. For example, when the kernel config `CONFIG_STACKPROTECTOR_STRONG` is enabled, for a function that uses register local variables, local variable's address or array as the right-hand side of an assignment, the compiler will add the canary-push logic at function prologue and canary-check logic at the function epilogue automatically [20].

Figure 1 shows the inserted canary-push and canary-check instructions for ARM64 architecture. For canary-push logic, Line 3-4 will load `__stack_chk_guard`'s address, which is `0xffffff800907d6c8`, to register x19, and then Line 5 will load the canary value to x6 and Line 6 will push canary to the stack.

When the function returns, Line 8 will load the saved canary value from the stack, while Line 9 will load the original canary value from the global variable `__stack_chk_guard`. Line 10-11 will compare the stack canary value with the original canary value. If they do not match, which means the stack is corrupted, the execution will jump to Line 13, which calls the function `__stack_chk_fail` to crash the kernel.

2.2.2 Per-task Canary. Different with ARM64's global canary design, the x86_64 Linux kernel is using per-task canary design, in which each process has its own stack canary.

Canary initialize: In per-task canary, each process will maintain a `stack_canary` variable in its process control block `task_struct`. When creating a new process, the `dup_task_struct` function will assign a pseudo-random number to `stack_canary` of the newly created process.

Canary use: Similar to the ARM64 compiler, the x86_64 compiler will also insert the canary logic into the generated binaries automatically. However, rather than relying on the global canary value, the inserted x86_64 canary logic uses the canary that belongs to current process. In x86_64, other than `task_struct->stack_canary`, a second canary copy is saved to the `stack_canary` field of a per-cpu data structure, called `irq_stack_union`. During process switching, the per-task canary `task_struct->stack_canary` of the switch-in process will be copied to the CPU's `irq_stack_union->stack_canary`, which will be used for the subsequent reads, rather than the global canary value `__stack_chk_guard` used in ARM64.

Figure 2 shows the inserted canary logic in a Linux kernel function of x86_64 architecture. The per-task canary `irq_stack_union` is saved in the thread-local-storage structure, pointed by the register gs, with an offset of 0x28. Therefore, Line 3-4 will load per-task canary value from `irq_stack_union->stack_canary` to register `rax`, and put its value to stack for stack protection. At function return, Line 6 will read the stack canary value from the stack, while Line 7 will load the original canary value from thread-local-storage structure `irq_stack_union->stack_canary` again. Line 8 will compare these two canary values and jump to `__stack_chk_fail` on mismatch, same with ARM64 design.

2.3 Kernel system call handling

Generally, the user space process on Linux uses the system call wrappers in standard C library or calls the system call directly, to request kernel fulfill certain privileged operations. As shown in Figure 3, to handle a system call, the execution will switch from user mode to kernel mode. The system call entry in kernel needs to save the user space context and load the kernel registers. The system call entry is defined as the `kernel_entry` on ARM64 and `entry_SYSCALL_64` on x86_64. After the system call entry, a system call dispatcher will invoke the correct system call according to the user-pass system call number to fulfill the system call request.

2.4 Performance monitor counter

Performance monitor counter provides a method to measure CPU cycles, it is available on both ARM64 and x86_64 CPUs. ARM64 PMUv3 provides a set of performance monitoring registers [17],

one of which is the cycle counter PMCCNTR_EL0. Its value increases on every processor clock cycle or every 64 processor clock cycle according to different configurations of the control register PMCR_EL0[8]. Moreover, ARM64 adopts instruction msr to read the value of PMCCNTR_EL0. For x86_64, Intel provides the performance-monitoring counter (PMC) to measure CPU performance. PMC registers can be set to count different events, such as unhalted core cycles[11]. And instructions rdpmc would read the the value of PMC register into register EAX and EDX, which contain low 32 bits and high 32 bits respectively.

Note that on both ARM64 and x86_64, userspace access to the performance monitor counter registers is usually disabled for security purpose. And this doesn't hurt the userspace timing capability as the userspace can still access the timer (such as using rdtsc to read time-stamp counter) or invoke kernel routines via system calls (such as gettimeofday) to get the timing information.

3 ASSUMPTIONS AND THREAT MODELS

We assume that the attacker has full control of the user space, but it cannot change the kernel image. In other words, the attacker cannot tamper with the existing kernel code or inject new kernel code. This assumption is reasonable as the secure boot and trust boot techniques are pretty mature nowadays, the boot-loader is able to check the integrity of the kernel image before loading it [22].

We further assume that the attacker has the arbitrary kernel memory read capability. In other words, the attacker can read any kernel memory by exploiting kernel memory leak bugs or by launching side-channel attacks such as Meltdown and Spectre.

For the kernel memory write capability, the attacker can overwrite the kernel stack by overflowing kernel stack variables, such as overflowing pointers or arrays. Note that the overflow must be sequential, which means the attacker cannot skip kernel canary and just overwrite the return address or the frame pointer saved on the kernel stack. Moreover, we assume that the overflow is triggered by calling system calls from user space. This is a valid assumption as system calls are designed to be only kernel entry points for the user space. Finally, we trust the random number generator on Linux kernel, and the attacker cannot predict the next random number that will be generated.

4 PESC DESIGN AND IMPLEMENTATION

In this section, we will first present the design of PESC, including how to generate the new canary value and where to update the canary. Then, we will talk about the implementation details of PESC on both ARM64 and x86_64.

4.1 Overview

For both the global canary as well as the per-task canary design, the attacker can leverage the leaked stack canary to craft the overflow payload so that stack canary will be overflowed by the correct canary value. In other words, with the leaked canary value, the overflow attack can be conducted without being detected as the stack canary value after overflow is correct. Although current canary design with the string terminator is able to defend against strcpy overflow attacks, however, there still exist other overflow attacks that can bypass the string terminator canary, such as overflowing

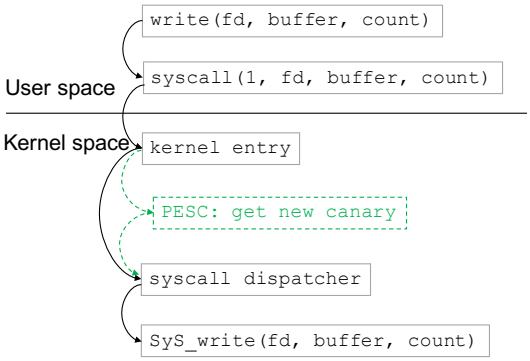


Figure 3: PESC design. PESC represents Per System-call Canary.

via memcpy. Therefore, a leaked stack canary is still a huge threat to the canary based stack overflow protection.

To address this problem, we proposed PESC, representing Per System-call Canary. The key observation behind PESC design is that the kernel stack is empty before-serving/after-finishing one system call so that PESC is able to generate a brand new kernel stack canary without the burden of tracking and updating all previous saved old stack canary values on the kernel stack. Let's use Figure 3 to illustrate the design of PESC: when the user space calls the write system call, the execution flow will switch to kernel space. The very first piece of code is called kernel entry, which performs the user-to-kernel context switching. Then according to the user space passed system call number, the system call dispatcher will find the correct system call (i.e., SyS_write) from the system call table sys_call_table and jump to it to fulfill the user space system call request. With PESC, a per system-call canary generation logic is inserted into the system call handling path, as shown by the green dashed box in Figure 3, so that a new stack canary will be generated for each system call. As a result, PESC can achieve the per system-call canary capability.

From the attacker's perspective, to bypass the stack canary protection, and launch the stack buffer overflow attack, he/she must get the leaked kernel stack canary value at first, by exploiting meltdown vulnerabilities or other information leak bugs. After that, the attacker can use the leaked canary value to craft a special overflow payload and inject the payload to kernel space using another system call so that the leaked canary value overwrites the canary value on kernel stack. As a result, the overflow attack is conducted successfully even under the protection of global and per-task canary protection. On the contrary, the life cycle of a PESC stack canary is limited to that system call only, which is significantly shortened compared to the global canary and the per-task canary design. As a result, the previously leaked stack canary is invalided by the new canary generated when the attacker is injecting the crafted payload the kernel via a new system call. In other words, PESC can protect the kernel stack even with canary leakages.

PESC is achieved by generating a new stack canary at system call handling path. Even though the idea sounds straightforward, PESC needs to resolve two technical challenges to achieve high security

and low performance overhead. First, for the new canary value generation, PESC needs to balance the randomness of newly generated canary and the introduced performance overhead. Calling the time consuming random number generator for every system call can be a performance killer. To trade-off, PESC proposes two canary value generation approaches, relying on the performance monitor counter and kernel pseudo-random number generator, respectively. We will present the details in §4.2.

Second, the kernel stack canary has to be changed before any old canary value got pushed to the stack; otherwise, on a function return, a stack canary mismatch will happen between the new canary value and the old canary on the stack, which will crash the kernel. To address this problem, PESC chooses the canary update location in the kernel entry carefully, before any canary pushing operations. The details will be covered in §4.3.

Note that the kernel thread stack is protected by the per-task canary design as each kernel thread gets its own canary when being forked. As system call is the only way that a user process can invoke kernel functions, its stack is more likely to be corrupted. Therefore, the system call stack should be protected more cautiously.

4.2 New canary value generation

For generating a new canary value, the intuitive approach would be directly calling kernel pseudo-random number generator. However, kernel pseudo-random number generation functions usually invoke dozens of functions, containing hundreds of lines of code. Calling these functions on every system call will introduce non-trivial performance overhead. Therefore, to balance the canary randomness and the performance, PESC proposes two approaches.

The first approach is to use the performance monitor counter register (PMC) as the random source for the new canary, and we termed this approach as PESC-PMC. With PMC system register as the random source, PESC only needs several instructions to fetch the value from the PMC system register, and implement the canary update logic with a handful of instructions, which has minimal performance overhead. For brevity, we defer the details to §4.4 and §4.5. However, the canary value fetched from PMC register is not fully randomized, which is the main security concern. Fortunately, our evaluation shows that even the attacker can try to leak the canary value right before the attacking system call, the success rate is still very low, as shown in Figure 5.1.

The second approach relies on the kernel pseudo-random number generation function `get_random_long` to generate the new kernel stack canary, termed as PESC-RNG. While the performance of PESC-RNG is not as good as one of PESC-PMC, but it is still acceptable. The performance evaluation result in §5 shows that the performance overhead of PESC-RNG on the whole system is less than 1%. Again, the details of PESC-RNG are covered in §4.4 and §4.5.

4.3 New canary update location

For updating the canary value, a straightforward solution would be generating a new canary and updating all canary value on the stack to avoid a canary check mismatch. For example, if the canary is changed after its value got pushed to the stack, a canary mismatch will be detected on the function return, which will panic the kernel. Therefore, all old canary values on the stack need to be updated when generating a new stack canary. As a result, this straightforward approach requires to record all canary addresses and update all canary values on kernel stack on every system call, which introduces non-negligible performance overhead.

To improve the performance, PESC leverages the key insight that the kernel stack is empty before-serving/after-finishing one system call. Instead of updating all old canary values on the stack, PESC proposes to generate the new stack canary at the very beginning of kernel entry, before any canary pushing instructions, so that no residual canary value gets pushed to the stack, hence no need to update old canary values on the stack. The exact location of the new canary update depends on the implementation and architecture, and details will be given in §4.4 for ARM64 and §4.5 for x86_64.

4.4 PESC implementation on ARM64

PESC ARM64 is implemented on Android HiKey960 Linux kernel [10], on the version of 4.19.36. We have implemented both PESC-PMC and PESC-RNG.

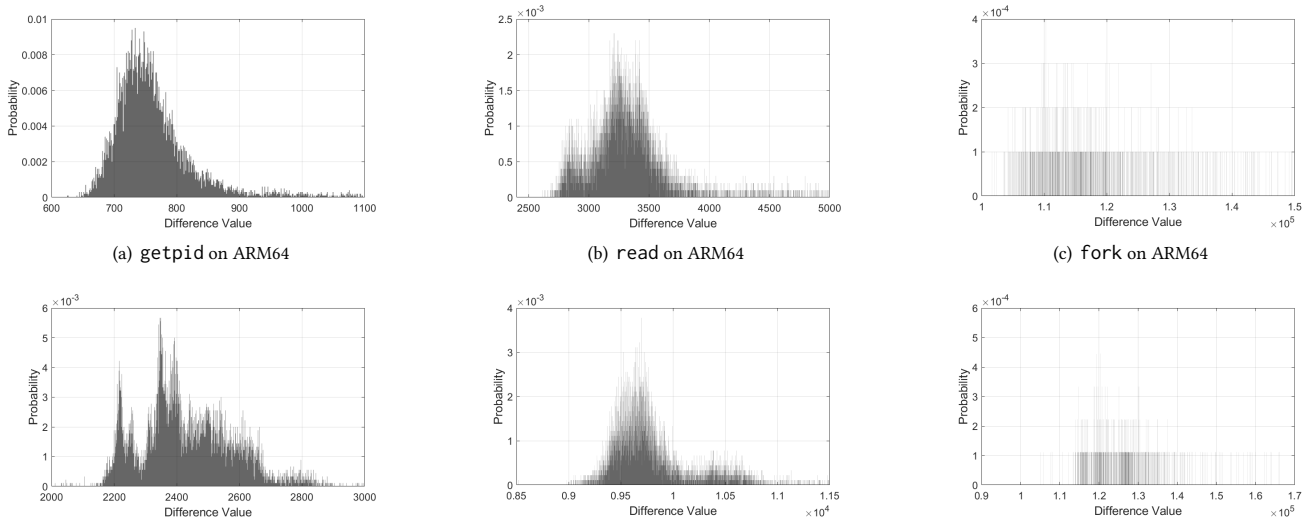
4.4.1 PESC-PMC on ARM64. On ARM64, the performance monitors cycle count register is `PMCCNTR_EL0`, which holds the cycle counts. As mentioned before, though ARM64 now is using a global canary design, it already includes a per-task stack canary member `stack_canary` in its process control block `task_struct`. Therefore, for generating the new canary value, PESC-PMC fetches the PMC register using instruction `mrs x19, PMCCNTR_EL0`, and saves the value to the per-task stack canary storage `task_struct->stack_canary`.

Before reading the PMC register value, PESC must configure PMC properly. The performance monitor is usually disabled on production devices. Therefore, PESC-PMC needs to enable performance monitor on system boot-up. When the kernel boots up, PESC-PMC will first clear `PMUSERENR_EL0.EN` bit, so that the user space access to performance monitor registers is disabled. Second, PESC-PMC will set the `PMCNTENSET_EL0` register to enable the cycle counter. Finally, PESC-PMC clears the `PMCR_EL0.D` bit to make `PMCCNTR_EL0` count every cycle and sets `PMCR_EL0.E` to enable `PMCCNTR_EL0`. By repeating the above steps on every core, PESC-PMC enables all performance monitor counters. Therefore, no matter which cores one process is running on, it can fetch performance monitor counters and use it as the new canary value.

For the new canary update location, PESC-PMC modified the `kernel_entry` assembly macro to insert the above code, so that new canary value will be generated for every user-to-kernel transition (i.e., system calls). Note that besides user-to-kernel transition triggered by system calls, `kernel_entry` macro also handles the kernel-to-kernel entries, such as interrupts. As the attacker cannot inject payload via interrupts, PESC-PMC does not change the canary value on kernel-to-kernel transitions.

Note that the performance monitor enabling code only runs once during the system boot-up, the canary update logic at every system call only involves a couple of assembly instructions. Therefore, the performance impact of PESC-PMC is minimal.

4.4.2 PESC-RNG on ARM64. For PESC-RNG, we inserted a kernel function `get_random_canary` call to the function `el0_svc_handler`, before the real system call dispatcher `el0_svc_common`. The function `get_random_canary` relies on `get_random_long`, will generate a



(a) `getpid` on ARM64 (b) `read` on ARM64 (c) `fork` on ARM64

(d) `getpid` on x86_64 (e) `read` on x86_64 (f) `fork` on x86_64

Figure 4: Figure (a), (b), and (c) show probability distribution using `getpid`, `read` and `fork` system calls, respectively. Figure (d), (e), and (f) show the corresponding probability distributions of x86_64. The bin size of the histogram is set to 1.

pseudo-random canary and assign it to `task_struct->stack_canary` of current task. Same with PESC-PMC, the insertion location is carefully chosen so that no kernel canary exists on the kernel stack, therefore no need to update the stack. Different from PESC-PMC, PESC-RNG depends on the kernel pseudo-random number generator, and does not require to enable the performance monitor counter.

Note that currently ARM64 architecture has no hardware support for random number generation. As a result, there is no fast and secure way of generating a random number. This is the reason why PESC needs trade-off. Fortunately, the 5th generation of ARM64 architecture extension ARMv8.5 provides the hardware-backed random number generator. More specifically, ARMv8.5 introduces two registers, `RNDR` and `RNDRRS`, and makes sure that reads to these registers return a 64-bit random number [7]. With the hardware support, PESC is able to generate the 64-bit random canary by using a single instruction, achieving both security and performance.

Note that the latest ARM64 Linux kernel used by Android Hikey board is v4.19, which only supports the global canary, we backport the per-task canary from Linux kernel v5.0 to v4.19 and use GCC9.1 compiler to compile the kernel to enable per-task canary on our ARM64 Hikey board.

4.5 PESC implementation on x86_64

Same with the ARM64 architecture, we also implement both PESC-PMC and PESC-RNG on X64_64. The implementation is based on Ubuntu 18.04 with a kernel version of 5.0.0.

4.5.1 PESC-PMC on x86_64. On x86_64, the performance monitoring counter register is `PMC`. For generating new canary values, PESC-PMC uses the instruction `rdpmc` to read the performance monitoring counter, the lower 32 bits and higher 32 bits are saved into different registers. Then PESC-PMC concatenates these values and uses it as the new canary value. As mentioned in §2.2.2, the per-task canary design on x86_64 has two copies of the canary, one is in task struct while the other one is in thread-local-storage for current running task. Therefore, when updating the new canary value, PESC-PMC updates both places. The PESC-PMC implementation only contains about one dozen of instructions in total, the performance impact is guaranteed to be small.

For the new canary update location, PESC-PMC inserts the new canary generation code to the beginning of the `do_syscall_64` function to make sure no old canary values exist on kernel stack. For security reasons, PESC-PMC disables user space direct access to the performance monitor registers by clearing the `CR4.PCE` (Performance-monitoring Counter Enable) bit on x86_64 Ubuntu. As a result, only ring 0 can execute `rdpmc` instruction. The user space has no way to access the value of `PMC` register. This will not hurt the userspace timing capability as the user program can still use `rdtsc` to read the time-stamp counter.

4.5.2 PESC-RNG on x86_64. On x86_64, most of PESC-RNG design is the same as PESC-PMC. The only difference is that PESC-RNG replaces the `rdpmc` instruction with `get_random_canary` function call, which calls `get_random_long` to generate a 64-bit pseudo-random value. Similarly, the newly generated canary is assigned to the canary copies in both task struct and thread-local-storage.

Note that current x86_64 CPUs provides hardware support for the random number generation, the corresponding instruction `RDRAND` can generate 16-bit, 32-bit or 64-bit random numbers. However, the overhead of `RDRAND` is high. On Intel Core i7-7700K processor (Kaby Lake-S micro-architecture), with a frequency of 4.5 GHz, one `RDRAND` instruction needs 110 ns or 463 CPU cycles [14]. Even worse, one execution of `RDRAND` is not guaranteed to generate a random number for sure, and Intel document recommends that 10 retries in a tight loop are likely to get a new random number [4]. As time needed for generating a random number using `RDRAND` cannot

be determined, therefore we choose kernel pseudo-random number generation function to generate the new canary for PESC-RNG.

5 EVALUATION

In this section, we first examine the security of PESC-PMC design, and then evaluate the performance impact of both PESC-PMC and PESC-RNG. The experiments for ARM64 were conducted on HiKey960 board [10], with 4 Cortex A73 and 4 Cortex A53 cores, 3GB DRAM and 32GB flash storage, running Android 10.0-rc3 with Linux kernel 4.19.36. For x86_64, we conducted experiments on Ubuntu 18.04 LTS with Intel i7-7700 CPU and 16GB RAM, the Linux kernel version is v5.0.0.

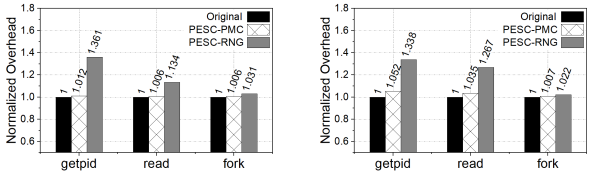
5.1 Security evaluation

PESC-RNG uses kernel pseudo-random number generator to generate the canary value, the randomness of the newly generated canary is guaranteed. Different from PESC-RNG, for the performance reason, PESC-PMC chooses to use the value of performance monitor counter as the new canary value directly. The value of performance monitor counter cannot provide the same randomness entropy compared with the PESC-RNG. Therefore, in this section, we examine the randomness entropy of performance monitor counter value thoroughly.

As mentioned in before, for both ARM64 and x86_64, the user space direct access to the performance monitor counter is disabled. As a result, the attacker cannot directly read the PMC register value. As the new canary value will be generated on-fly when the attacker is trying to trigger the buffer overflow attack, there is no way the attacker can infer the canary beforehand. The best an attacker can do is to leak the old canary value and use it to guess the new value generated by the next system call. For example, the attacker can either use information leak vulnerabilities or side-channel techniques, to leak the canary value saved in `task_struct->stack_canary` or on the stack. Based on the old canary value, the attacker can infer the range of the PMC value and guess the canary that will be generated on the next system call.

Therefore, to test PESC-PMC security, we need to evaluate the `PMC` value variations between two consecutive system calls, which is the best the attacker can achieve. To cover different system call lengths, we choose three system calls, `getpid`, `read` and `fork`, in which `getpid` is the shortest system call in Linux kernel [1], `read` system call reads 100 bytes of data, representing mid-length system calls while `fork` is the heavy system call. For each system call, we call it twice consecutively and record the canary values generated by PESC-PMC. We repeat this process every 100 milliseconds for 10000 times. Finally, we removed the outlier values and calculated the difference values between 9000 consecutive canary pairs. The results are shown in Figure 4.

From Figure 4, it is easy to see that even for the shortest system call `getpid`, which is unlikely to have the memory leak bugs, the highest chance that the attacker can guess the PMC right is less than 1% on both ARM64 and x86_64, as shown by Figure 4(a) and Figure 4(d). For a mid-length system call `read`, the highest correct guess chance falls to less than 0.5% on both ARM64 and x86_64, while for heavy system call like `fork`, the correct guess chance falls to less than 1‰ on both ARM64 and x86_64, which is close to zero.



(a) ARM64 (b) x86_64

Figure 5: Performance overhead on individual system call. Lower is better. The y-axis is the normalized overhead. The system call time of the original kernel is normalized to 1.

In other words, the PMC guess successful rate by the attacker on ARM64 is always less than 1% both on ARM64 and on x86_64.

Note that the above experiments measure the PMC differences between two consecutive system calls. In a real attacking scenario, the attacker needs to craft the payload using the leaked canary, which takes more time, making the PMC value on the next system call even harder to predict.

5.2 Performance evaluation

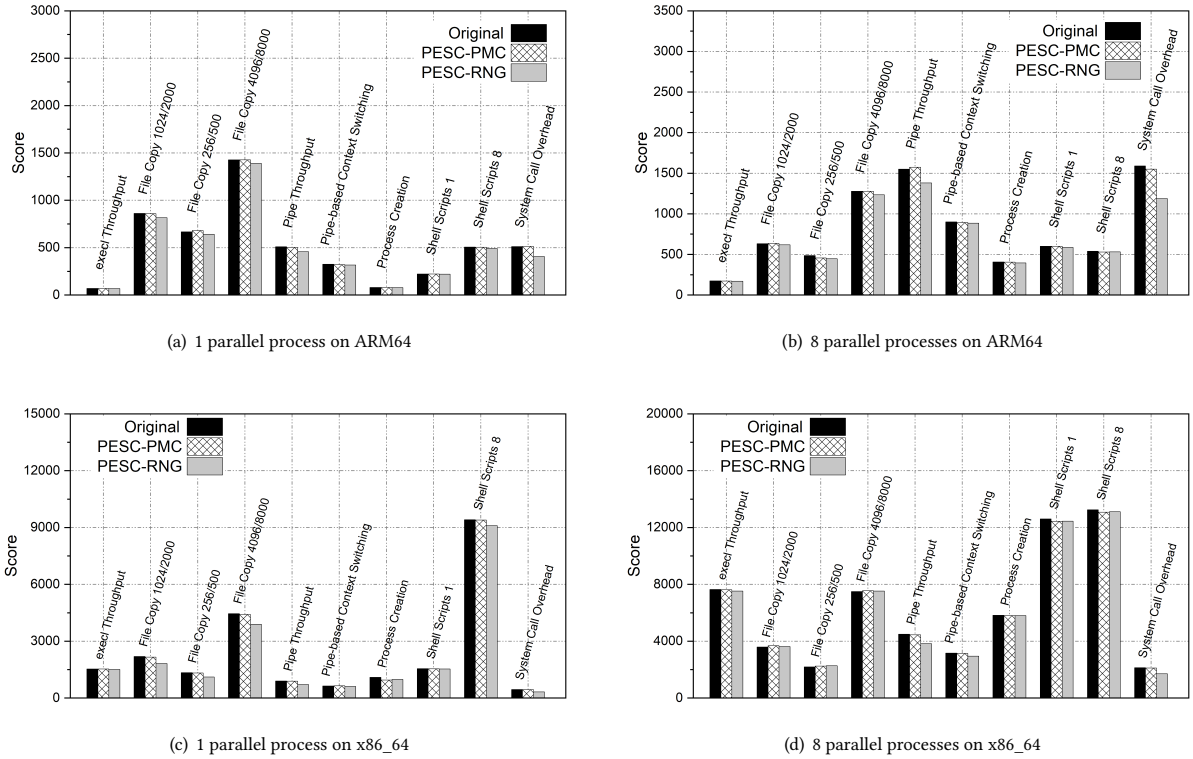
Both PESC-PMC and PESC-RNG require to add code in system call handling code to generate new kernel canaries for each system calls. The newly added code will impact performance. Therefore in this section, we want to evaluate the performance overhead introduced by PESC.

The performance evaluation consists of two tests: micro test and macro test. In the micro test, as we know the added code is mainly in system call handling. Therefore, we need to understand how much slow down the PESC will introduce to individual system calls. In the macro test, we want to know the performance impact of PESC on the whole system. We evaluated the performance overhead of PESC-PMC and PESC-RNG on both ARM64 and x86_64.

5.2.1 Micro Test. In the micro test, we want to understand the performance impact of PESC on individual system calls; therefore, we conducted the performance evaluation using both selected individual system calls and the UnixBench [2].

For the performance of individual system call, we choose the same three system calls `getpid`, `read`, and `fork` used in Security Evaluation 5.1, representing short system calls, mid-length systems, and heavy system calls, respectively. For each system calls, we test the performance for three kernel settings: original kernel without PESC, the kernel with PESC-PMC, and the kernel with PESC-RNG. We calculated the average time cost of 1000 consecutive `getpid` calls, 1000 consecutive `read` calls, and 1000 consecutive `fork` calls (the parent process did not wait child processes to finish) for each kernel settings. Every `read` system call reads 100 bytes data.

The results are shown in Figure 5. The y-axis represents the normalized performance overhead while the system call time of the original kernel setting is normalized to 1. It is easy to see that for both ARM64 and x86_64, the overhead of PESC-PMC is low. Average performance overhead is close to zero (about 0.8%) on ARM64, and is about 3% on x86_64, since PESC-PMC only adds several instructions to the system call entry. In PESC-RNG, system calls spend more time than on both ARM64 and x86_64 due to



(a) 1 parallel process on ARM64

(b) 8 parallel processes on ARM64

(c) 1 parallel process on x86_64

(d) 8 parallel processes on x86_64

Figure 6: UnixBench results. 1 parallel process and 8 parallel processes settings are used. Original means the original Linux kernel without PESC. For the result, higher is better.

Table 1: The scores of Android synthetic benchmarks on ARM64.

Name	Original	PESC-PMC	Overhead	PESC-RNG	Overhead
Linpack	2307	2306	0.04%	2307	0
GeekBench					
Single Core	1874	1871	0.16%	1871	0.16%
Multi Core	4424	4379	1.02%	4380	0.99%
Vellamo					
Browser	5400	5410	-0.19%	5383	0.37%
Metal	3240	3229	0.34%	3228	0.49%
average	-	-	0.27%	-	0.40%

many functions are called for random number generation. As `getpid` is very short, the added random number generation functions in kernel entry will make a big impact (about 36% on ARM64 and 34% on X86_64). But for mid-length system calls and heavy system calls, the performance overhead drops to 13% and 3% on ARM64, 27% and 2% on X86_64, respectively.

Besides self-picked individual system calls, UnixBench also gives the performance evaluation of individual system calls as well as combined operations. For ARM64, We compiled UnixBench using Clang/LLVM in termux-0.84 and ran it on HiKey960 board. we set the iteration parameter of UnixBench to 1 and disconnected the display device of HiKey960 board for all UnixBench experiments to minimize the board heating, as HiKey960 board is very sensitive to temperature.

For UnixBench experiments, we choose both single process and multi-processes settings. Combining with the two architectures ARM64 and x86_64, we have 4 experiment scenarios. For each scenario, we ran UnixBench eight times, removed the highest one, and the lowest one and calculated the average on the remaining six results to minimize the deviation. The results are shown in Figure 6. From the figure, we can see that the performance of PESC-PMC is very close to the performance of the original kernel (higher is better). On average, the performance overhead of PESC-PMC is less than 1% on both ARM64 and x86_64, for both 1 process

Table 2: The SPEC CPU2006 benchmark performance overhead on x86_64

program	Original (s)	PESC-PMC (s)	Overhead (%)	PESC-RNG (s)	Overhead (%)
401.bzip2	297	297	0	298	0.34%
403.gcc	165	166	0.61%	166	0.61%
429.mcf	174	174	0	175	0.57%
445.gobmk	309	309	0	309	0
456.hmmer	245	244	-0.41%	244	-0.41%
458.sjeng	337	337	0	337	0
462.libquantum	222	222	0	224	0.90%
iheight464.h264ref	324	324	0	323	-0.31%
471.omnetpp	227	227	0	227	0
473.astar	270	270	0	270	0
483.xalancbmk	127	128	0.79%	127	0
average	-	-	0.09%	-	0.15%

and 8 processes test settings. For PESC-RNG, the random number generation code is added to each system call, which incurs large performance overhead. As a result, the performance of PESC-RNG, in general, is 7% and 8% slower than the original kernel, on ARM64 and x86_64, respectively.

5.2.2 Macro Test. In macro test, we evaluate the performance impact of PESC-PMC and PESC-RNG on the whole system by using the synthetic benchmarks.

For ARM64, we choose three popular benchmarks from the Android play store: Linpack, GeekBench, and Vellamo. As mentioned before, HiKey960 board is very easy to have the heating problem, and the board itself is very sensitive to temperature rise. To make sure the experiments are conducted fairly and accurately, we turn off the board to cool down for about 10 minutes after every benchmark test. Also, we keep the environmental temperature stable, to minimize the environmental impacts. Similar to the UnixBench tests, we ran each benchmark five times and also removed the highest one and the lowest one to minimize the deviation. Table 1 lists the benchmark scores of original kernel without PESC, PESC-PMC kernel, and PESC-RNG kernel on HiKey960 board, higher is better. Column 2, 3 and 5 show the benchmark scores while column 4 and 6 show the corresponding degradation of PESC-PMC and PESC-RNG. The average score degradation is 0.27% and 0.40% for PESC-PMC and PESC-RNG, respectively. In other words, the performance overhead imposed by PESC-PMC and PESC-RNG on the whole system is small.

For x86_64, We use SPEC CPU2006 benchmark [19] to evaluate the performance of the whole system. We ran SPEC CPU2006 benchmark 2 times for each kernel. The result is shown in Table 2. Column 2,3 and 5 show the run time of the tests on the system with the original kernel, kernel with PESC-PMC and kernel with PESC-RNG. Column 3 and 5 calculate the slowdown of PESC-PMC and PESC-RNG, respectively. From the table, it is easy to see that the average performance overhead is 0.09% and 0.15% for PESC-PMC and PESC-RNG, respectively. In other words, neither PESC-PMC nor PESC-RNG introduce a big performance overhead to the whole system. A possible explanation is that system calls are not called so frequently when compared to other functions. As we add code in system call handling, which influences the system call performance only, the low proportion of system calls in the benchmark dilutes the impact of the added code in PESC.

5.3 Limitations

Current PESC design provides the per-system canary for system call stacks. However, for kernel thread stacks and interrupt stacks, PESC only provides the same protection as per-task canary does.

Moreover, PESC can defeat the canary leak and payload injection via two separated system calls. Unfortunately, PESC cannot defend against attacks (mentioned as the the bridging gadget in [44]) that leak the kernel stack canary and inject crafted payload in one single system call.

6 RELATED WORKS

PESC is in general related to the stack canary design and improvement studies. In this section, we will compare PESC with these related works.

6.1 Static stack canary

Stack canary was first proposed in StackGuard [27] in 1998. The basic idea of original stack canary design, including **StackGuard** [26, 27, 42], **Propolice** [30] and **GS (Buffer Security Check)** [15], is to put a canary word between local variables and the return address and check the canary value on function return to detect any return address corruptions. The content of canary word can be divided into three types. The first type is **terminator canary** [21, 28, 39, 42]. It contains string terminators; therefore any buffer overflow caused by the string copy will be defeated since the terminators will terminate the overflow string automatically. **Random canary** [21, 28, 39, 42] is another type of canary. As the canary is a random number which is hard to guess, it is able to prevent all sequential overflows. **Random XOR canaries** [21, 39, 42] is a random number which encrypts control data on stack using exclusive-or.

However, for all these designs, the canary value remains the same after initialization, which makes them vulnerable to memory leaks. For example, attacks [25, 29, 39, 40, 45] are able to defeat the protection of stack canary. Even worse, security researchers have released an open-source framework named CookiesCrumbler, which is able to analyze different stack canaries designs and launch corresponding bypassing attacks [24]. Unfortunately, even after 20

years, ARM64 Linux kernel still uses the very basic global static kernel canary design, making it vulnerable to all memory leaks and modern side-channel attacks, which is the direct motivation of our PESC design.

6.2 Dynamic stack canary

A series of canary enhancement techniques are proposed to secure the stack canary.

RAF-SSP (Re-new After Fork Stack Smashing Protector) [35] enhanced canary design by differentiating the child process's canary from its parent. Therefore, RAF-SSP is able to defeat byte-by-byte attacks that exploit the inherited address space of the child process right after a `fork` system call. It updates child process reference canary with a random number right after the child process is created. With this technique, the attacker is unable to infer any stack canary information of the parent process even by byte-by-byte attacking the canary of the child process. And RAF-SSP does not allow the child process to return since it assumes child process on a server end with an `exit()` function, which limits its adoption. Besides RAF-SSP, **SSPFA** [36] is proposed as an enhancement of SSP for Android devices, which has a similar design with RAF-SSP. However, for both RAF-SSP and SSPFA, they can not protect the stack if the attacker has the arbitrary memory read capability by exploiting memory leak or side-channel vulnerabilities.

DynaGuard [38] uses a canary linked list stored in thread-local-storage (TLS) to implement a dynamic canary design. It updates the canary in both the TLS and all inherited stack frames after the `fork` system call. Thus, all canaries on the stack of the newly fork child process are different from that of its parent process, which can not be exploited to guess the parent canary value. However, the pin-based [34] DynaGuard introduces a run-time overhead of 170.66%, which is too high for practical deployments.

DCR (Dynamic Canary Randomization) [31] uses a canary linked list whose head node is stored in the location of the original GCC SSP in TLS. When a user specified function is invoked, the system re-randomizes canary value, stores it on stack and inserts it into the canary linked list. Then it checks all the canary values on the stack since buffer overflow might happen in previous functions. When the function returns, DCR is able to find the head of the list using embedded offset in canary value and then checks it. The average run-time overhead of DCR is more than 24%, and it also needs extra space to store canaries.

DiffGuard (Different function frames with different canaries) [47] will update all stack canaries of newly forked child process for `fork` system call. It also implements different canaries for each function call. It uses a random canary buffer (RCB) in the TLS to store canaries. When a new function is called, a new canary will be fetched from RCB, and the index of RCB will be incremented. When the function returns, the process is able to check whether the canary is corrupted using this index. DiffGuard realizes a per-frame canary design and its average run-time overhead is 3.2%. DiffGuard has a higher overhead than PESC. Moreover, it requires extra storage to hold the RCB.

P-SSP (Polymorphic Stack Smashing Protector) [43] keeps the structure of TLS unchanged and uses a pair of shadow canary words to defeat byte-by-byte attacks. One word is a random number generated when the child process is forked, and the other one is calculated by XORing this random number and the original canary. Therefore, the original canary can be recovered by using XOR operation on these two words. Instead of storing original canary on stack, these two words are stored on stack. Therefore, P-SSP is able to defeat byte-by-byte attacks and the exhaustive search. By storing these two shadow canary word on stack rather than the original canary, attackers cannot get accumulated information about the original canary.

Sadly, all of these designs store their canaries either in the memory or in TLS, which is also part of the main memory. As a result, a single memory leak vulnerability or hardware related side-channel attack can bypass all of them. To the contrary, PESC is designed to protect kernel stack against memory leaks. Even the attacker has the arbitrary memory read capability, he/she still cannot bypass the protection of PESC.

7 CONCLUSION

In this paper, we proposed PESC, a new Per System-call Canary design for Linux kernel. The basic idea of PESC is to generate a new kernel stack canary at every system call entry. Compared with existing global canary design and per-task canary design, PESC has two benefits. First, by generating a new canary for every system call, PESC invalidates the leaked kernel canaries. As a result, the attacker cannot accumulate the knowledge of canaries across system calls. Second, by generating the new canary on-fly for every system call, PESC ensures that the attacker has no way to leak the stack canary beforehand, even though the kernel has memory leak vulnerabilities.

To achieve PESC, we propose to generate the new canary value at the system call entries, before any canary being pushed to the stack, so that the performance overhead is reduced as there is no need to update the canary value on the stack. To balance performance and security, we propose two implementations of PESC: PESC-PMC and PESC-RNG. The canary value is fetched from the performance monitor counter register on PESC-PMC and is generated by the kernel random number generator on PESC-RNG.

We implemented PESC-PMC and PESC-RNG on both ARM64 and x86_64 and conducted both security and performance evaluations on ARM64 HiKey960 board and the x86_64 Intel i7-7700 CPU. Our security evaluation shows that for PESC-PMC, even with the arbitrary kernel read vulnerabilities, the chance of the attacker predicting new stack canary is still small. Performance-wise, for ARM64 implementations, the Android synthetic benchmark experiments show that the average performance overhead of PESC-PMC and PESC-RNG is 0.27% and 0.40%, respectively. For x86_64 implementation, the performance overhead of SPEC CPU2006 experiments is 0.09% and 0.15%.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. This work is partially supported by the National Natural Science Foundation of China under Grants No. 61772236 and Zhejiang Key R&D Plan under Grant No. 2019C03133.

REFERENCES

[1] 2011. Approximate Overhead of System Calls. https://www.ibm.com/developerworks/community/blogs/kevgrig/entry/approximate_overhead_of_system_calls9?lang=en. (2011).

[2] 2017. byte-unixbenchs. https://github.com/kdlucas/byte-unixbench. (2017).

[3] 2018. Exploit Mitigation Techniques - Stack Canaries. https://0x00sec.org/t/exploit-mitigation-techniques-stack-canaries/5085. (2018).

[4] 2018. Intel Digital Random Number Generator (DRNG) Software Implementation Guide. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide. (2018).

[5] 2018. Issue 1657: Linux: semi-arbitrary task stack read on ARM64 (and x86) via /proc/pid/stack. https://bugs.chromium.org/p/project-zero/issues/detail?id=1657. (2018).

[6] 2019. 2PAC 2Furious: Envisioning an iOS Compromise in 2019. https://downloads.immunityinc.com/infiltrate2019-slidepacks/marco-grassi-liang-chen-2pac-2furious/infiltrate19_final.pdf. (2019).

[7] 2019. Arm Architecture Reference Manual, Issue E.a. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile. (2019).

[8] 2019. Cycle Count Register. https://developer.arm.com/docs/ddi0433/a/performance-monitoring-unit/performance-monitoring-register-descriptions/performance-monitor-control-register. (2019).

[9] 2019. Function prologue. https://en.wikipedia.org/wiki/Function_prologue. (2019).

[10] 2019. HiKey960 board. https://source.android.com/setup/build/devices. (2019).

[11] 2019. IntelÂő 64 and IA-32 Architectures Software DeveloperâĂŹs Manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. (2019).

[12] 2019. Issue 1759: KVM: uninitialized memory leak in kvm_inject_page_fault. https://bugs.chromium.org/p/project-zero/issues/detail?id=1759. (2019).

[13] 2019. Issues - project zero. https://bugs.chromium.org/p/project-zero/issues/list?can=1&q=stack+overflow&colspec=ID+Status+Restrict+Reported+Vendor+Product+Finder+Summary&num=100. (2019).

[14] 2019. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf. (2019).

[15] 2019. Microsoft /GS(Buffer Security Check). https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?redirectedfrom=MSDN&view=vs-2019. (2019).

[16] 2019. Morris Worm. https://en.wikipedia.org/wiki/Morris_worm. (2019).

[17] 2019. Performance monitoring register descriptions. https://developer.arm.com/docs/ddi0433/a/performance-monitoring-unit/performance-monitoring-register-descriptions. (2019).

[18] 2019. Return-oriented programming - Wikipedia. https://en.wikipedia.org/wiki/Return-oriented_programming. (2019).

[19] 2019. SPEC CPU 2006. http://www.spec.org/cpu2006/. (2019).

[20] 2019. STACKPROTECTOR_STRONG. https://elixir.bootlin.com/linux/v5.0/source/arch/Kconfig#L473. (2019).

[21] Steven Alexander. 2005. Defeating compiler-level buffer overflow protection. *The USENIX Magazine; login* (2005).

[22] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 90–102.

[23] Brian Belleville, Wenbo Shen, Stijn Volckaert, Ahmed M Azab, and Michael Franz. 2019. KALD: Detecting Direct Pointer Disclosure Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2019).

[24] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. 2018. Smashing the Stack Protector for Fun and Profit. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 293–306.

[25] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[26] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. 1999. Protecting systems from stack smashing attacks with StackGuard. In *Linux Expo*.

[27] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX Security Symposium*, Vol. 98. San Antonio, TX, 63–78.

[28] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, Vol. 2. IEEE, 119–129.

[29] Yu Ding, Zhuo Peng, Yuanyuan Zhou, and Chao Zhang. 2014. Android low entropy demystified. In *2014 IEEE International Conference on Communications (ICC)*. IEEE, 659–664.

[30] Hiroaki Etoh. 2019. GCC extension for protecting applications from stack-smashing attacks (ProPolice)(2003). *URL http://www. trl. ibm. com/projects/security/ssp* (2019).

[31] William H Hawkins, Jason D Hiser, and Jack W Davidson. 2016. Dynamic canary randomization for improved software security. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. ACM, 9.

[32] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[35] Hector Marco-Gisbert and Ismael Ripoll. 2013. Preventing brute force attacks against stack canary protection on networking servers. In *2013 IEEE 12th International Symposium on Network Computing and Applications*. IEEE, 243–250.

[36] Héctor Marco-Gisbert and Ismael Ripoll-Ripoll. 2019. SSPFA: effective stack smashing protection for Android OS. *International Journal of Information Security* (2019), 1–14.

[37] Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.

[38] Theofilos Petsios, Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2015. Dynaguard: Armoring canary-based protections against brute-force attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 351–360.

[39] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*. ACM, 1–8.

[40] Laszlo Szekeres, Mathias Payer, Lenx Tao Wei, and R Sekar. 2014. Eternal war in memory. *IEEE Security & Privacy* 12, 3 (2014), 45–53.

[41] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[42] Perry Wagle, Crispin Cowan, et al. 2003. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*. Citeseer, 243–255.

[43] Zhilong Wang, Xuhua Ding, Chengbin Pang, Jian Guo, Jun Zhu, and Bing Mao. 2018. To detect stack buffer overflow with polymorphic canaries. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 243–254.

[44] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. {KEPLER}: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1187–1204.

[45] A Zabrocki. 2010. Scraps of notes on remote stack overflow exploitation. *Phrack* 63, 15 (2010).

[46] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. PeX: A Permission Check Analysis Framework for Linux Kernel. In *28th USENIX Security Symposium 2019*. 1205–1220.

[47] Jun Zhu, Weiping Zhou, Zhilong Wang, Dongliang Mu, and Bing Mao. 2017. Diffguard: Obscuring sensitive information in canary based protections. In *International Conference on Security and Privacy in Communication Systems*. Springer, 738–751.