Is the Canary Dead? On the Effectiveness of Stack Canaries on Microcontroller Systems

Xi Tan
CactiLab, University at Buffalo
USA
xitan@buffalo.edu

Sagar Mohan
CactiLab, University at Buffalo
USA
sagarmoh@buffalo.edu

Md Armanuzzaman
CactiLab, University at Buffalo
USA
mdarmanu@buffalo.edu

Zheyuan Ma
CactiLab, University at Buffalo
USA
zheyuanm@buffalo.edu

Gaoxiang Liu
CactiLab, University at Buffalo
USA
gliu25@buffalo.edu

Alex Eastman
CactiLab, University at Buffalo
USA
alexeast@buffalo.edu

Hongxin Hu
University at Buffalo
USA
hongxinh@buffalo.edu

Ziming Zhao
CactiLab, University at Buffalo
USA
zimingzh@buffalo.edu

ABSTRACT

Microcontroller units (MCUs) are compact computers tailored for embedded and Internet-of-Things (IoT) applications. MCU-based devices primarily run software systems coded in low-level languages such as C, making them susceptible to memory corruption attacks like stack-based buffer overflows. Stack canaries are a low-overhead buffer overflow detection mechanism that offers a certain level of protection and is frequently used in microprocessor systems in both the kernel and application layers. However, their effectiveness and overhead on microcontroller systems have not been extensively studied. As a result, the community naively assumes that the stack canary mechanism on microcontrollers provides the same level of security as it does on microprocessor systems.

In this paper, we present a study that centers on the implementation and utilization of stack canaries in microcontroller systems. More specifically, we delve into the support for stack canaries across libraries, compilers, and system layers. Our findings suggest that the implementations of stack canaries on microcontroller systems are generally less secure than their counterparts on microprocessors. Additionally, we conducted measurements to assess the overhead of stack canaries within Zephyr, a popular real-time operating system for microcontrollers. We aim for this paper to illustrate the limitations of stack canaries on microcontrollers and advocate for the exploration of alternative solutions.

CCS CONCEPTS

• **Security and privacy → Systems security**.

SAC '24, April 8–12, 2024, Avila, Spain
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0243-3/24/04…$15.00
https://doi.org/10.1145/3605098.3635925

KEYWORDS

Microcontroller systems; stack canaries

ACM Reference Format:
Xi Tan, Sagar Mohan, Md Armanuzzaman, Zheyuan Ma, Gaoxiang Liu, Alex Eastman, Hongxin Hu, and Ziming Zhao. 2024. Is the Canary Dead? On the Effectiveness of Stack Canaries on Microcontroller Systems. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605098.3635925

1 INTRODUCTION

Microcontroller units (MCUs), such as Arm Cortex-M [3], find extensive use in resource-constrained devices, including applications in smart home gadgets, drones, and wearables. In the fourth quarter of 2020 alone, Arm reported that its partners had collectively shipped 4.4 billion Cortex-M MCUs [26]. These MCUs typically operate using either bare-metal software or real-time operating systems (RTOSs), especially for specific tasks that require a deterministic response under constraints of memory, power consumption, and cost. Microcontroller systems predominantly use low-level programming languages, such as C. However, the use of such languages brings along inherent challenges, particularly concerning memory corruption issues. These issues give rise to vulnerabilities, notably buffer overflows, which have the potential to compromise system security by allowing attackers to fully control the system.

While many techniques for enhancing memory safety exist, they often come with high performance overheads [44]. One notable exception is stack canaries [32], a low-overhead solution for mitigating stack-based buffer overflow attacks. Stack canaries can detect unauthorized overwrites to critical stack data, such as return addresses and frame pointers, offering a balance between security and performance. First introduced in 1998 through StackGuard [32], stack canaries are widely supported by compilers and have been incorporated into microprocessor systems, such as in x86/x64 and Arm Cortex-A based systems. Over the years, various research has been introduced to bolster the security of stack canaries, including defenses against brute-force attacks [41], hardware-assisted dynamic canary generation [40], and fine-grained stack canaries at the per-system call level in the Linux kernel [43]. However, these security mechanisms are primarily designed for microprocessor systems, and the effectiveness of stack canaries on microcontroller systems has not been thoroughly studied.

In this paper, we undertake an analysis of stack canaries in microcontroller system implementations and their practical usage. Using a C standard library [13], GCC [27], and Linux distributions [14] on x86/x64 as the baseline for comparison, our analysis encompasses various microcontroller platforms, including libraries, compilers, and non-Linux RTOSs. Our study delves into several aspects, including the extent of support for stack canaries in microcontroller systems, the intricacies of their implementation, and an assessment of their performance and efficiency.

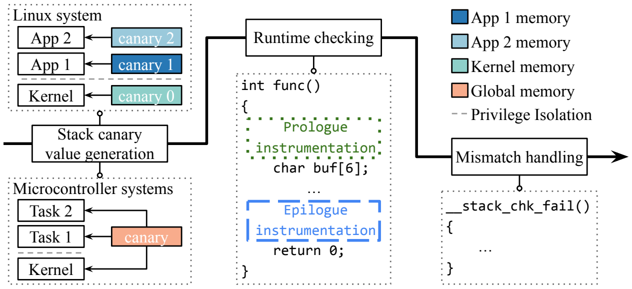
Our first finding is that only a few microcontroller systems have integrated with stack canaries. This can be attributed to three main reasons. First, stack canaries are not enabled by default for compilers targeting MCU programs. Instead, developers must take explicit steps to activate this security feature. Secondly, the inclusion of stack canaries primarily hinges on compiler instrumentation. However, it is important to highlight that the responsibility for initializing the canary value lies with the system itself. Unfortunately, this crucial aspect is inadequately supported by many bare-metal systems and RTOSs. Finally, the use of stack canaries in microcontroller programs results in significantly higher overhead compared to their microprocessor program counterparts. This is primarily due to the compact nature of microcontroller programs.

Our second finding underscores that, despite the support for stack canaries in some systems, several limitations persist. A prevalent issue revolves around the *prolonged reuse* of a single canary value, a practice commonly configured either during compilation or at system boot time. This persistence of canary values is attributed to two key challenges in MCU environments: (1) *timing for canary updates*. MCU systems often lack a clear and opportune time point for enforcing the introduction of new canary values, contributing to the extended reuse of existing values; (2) *limited random number generation*. MCUs face inherent limitations in generating genuinely random numbers required for canary initialization. This sharing of canary values presents a significant security risk, especially considering that most microcontroller systems do not implement mechanisms like privilege isolation and memory access control [49].

The remainder of the paper is organized as follows. In §2, we present an overview of the stack canaries mechanism. We discuss compiler instrumentation and canary mismatch handling in §3 and system support for canary generation in §4. In §5, we present the security analysis of stack canaries on microcontroller systems, followed by a discussion in §6.

2 OVERVIEW OF STACK CANARY

Stack canaries are values that are inserted into a function's stack frame, typically placed before essential data, such as the frame pointer and return address. As illustrated in Figure 1, we divide the stack canary mechanism into three phases in chronological order.



```
int func()
{
   char buf[6];
      ...
   return 0;
}
```

```
__stack_chk_fail()
{
    ...
}
```

Figure 1: Three Phases of the Stack Canary Mechanism

In the first stage of *stack canary value generation*, canary values are generated and made available to functions that require them at the variable `__stack_chk_guard`. For applications compiled against the GNU C Library (glibc), this variable is defined within the C library. However, for programs like the kernel, not compiled against glibc, it is the program's responsibility to define this variable. The randomness and security of canary values are critical to their effectiveness, as only random and unknown values can thwart attackers' guessing attempts. We will discuss how different systems generate stack canary values in §4.

In the second stage of *runtime checking*, a canary-protected function fetches the value from the variable `__stack_chk_guard` and places it onto its stack frame in its prologue. Prior to the function's return, during its epilogue, the program checks if this canary value remains intact. If altered, it signifies a buffer overflow attempt. The rationale behind this is that sequential buffer overflows operate by overwriting memory from lower to higher memory addresses. Consequently, to gain control by tampering with the return pointer or frame pointer, it is imperative to also overwrite the canary value. Since the stack canary is designed to be a transparent security feature for software developers, the operations of fetching the canary and comparing it are actually performed by instructions that are instrumented by compilers. The instrumented instructions used by different compilers are similar in this step, as we will discuss in §3.

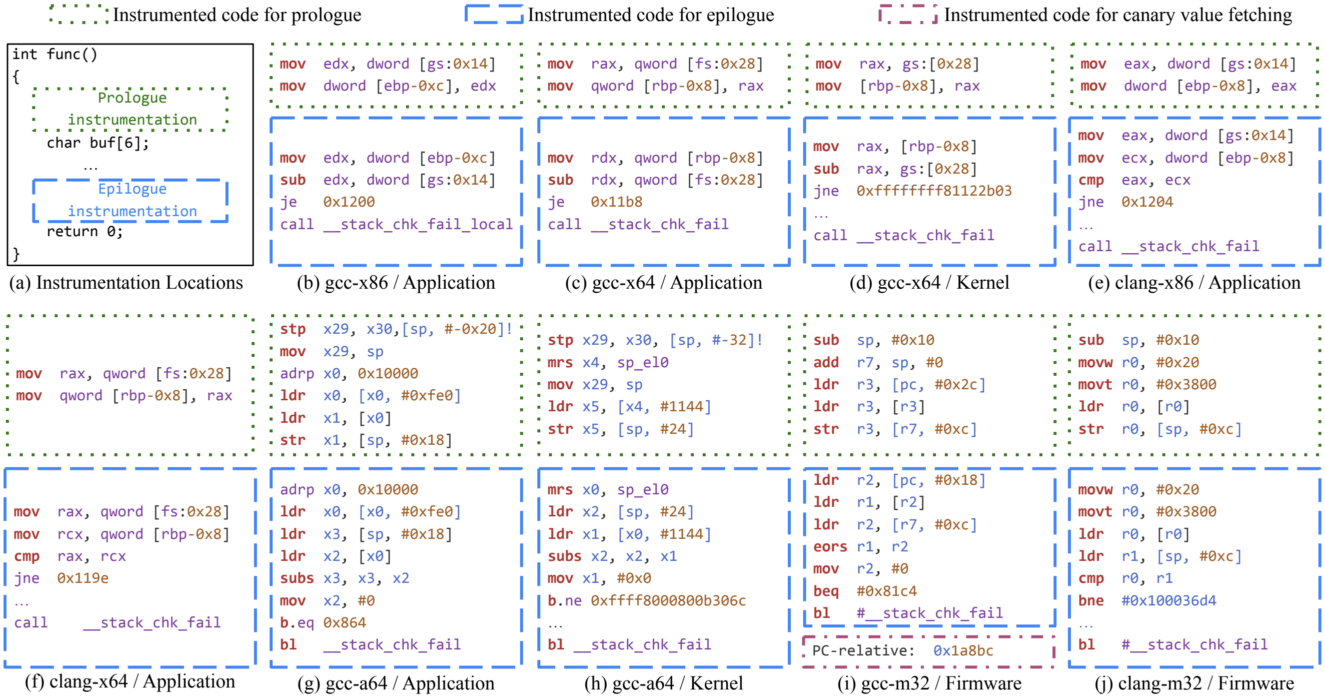
If an overwriting of the canary value is detected, the third phase of *mismatch handling* occurs. This is also accomplished through instrumented code that calls stack check failure functions. Depending on the nature of the protected functions, e.g., application function or kernel function, the mismatch handling will vary, as we will discuss in §3.

3 COMPILER INSTRUMENTATION AND CANARY MISMATCH HANDLING

In this section, we discuss stack canaries support from compilers and libraries.

3.1 Compiler Instrumentation and Options

To safeguard a function's stack frame with canaries, the compiler instruments instructions into both the prologue and epilogue of the function. In the prologue, these instructions serve two primary purposes: (P1) fetching the canary value from a specified source and (P2) placing this canary value onto the stack. Meanwhile, in the epilogue, the instrumentation performs the following actions:


Instrumented code for prologue
Instrumented code for epilogue
Instrumented code for canary value fetching

```
int func()
{
    Prologue
    instrumentation
   char buf[6];
       ...
    Epilogue
    instrumentation
   return 0;
}
```

(a) Instrumentation Locations

```
mov   edx, dword [gs:0x14]
mov   dword [ebp-0xc], edx

mov   edx, dword [ebp-0xc]
sub   edx, dword [gs:0x14]
je    0x1200
call  __stack_chk_fail_local
```

(b) gcc-x86 / Application

```
mov   rax, qword [fs:0x28]
mov   qword [rbp-0x8], rax

mov   rdx, qword [rbp-0x8]
sub   rdx, qword [fs:0x28]
je    0x11b8
call  __stack_chk_fail
```

(c) gcc-x64 / Application

```
mov   rax, gs:[0x28]
mov   [rbp-0x8], rax

mov   rax, [rbp-0x8]
sub   rax, gs:[0x28]
jne   0xffffffff81122b03
…
call  __stack_chk_fail
```

(d) gcc-x64 / Kernel

```
mov   eax, dword [gs:0x14]
mov   dword [ebp-0x8], eax

mov   eax, dword [gs:0x14]
mov   ecx, dword [ebp-0x8]
cmp   eax, ecx
jne   0x1204
…
call  __stack_chk_fail
```

(e) clang-x86 / Application

```
mov   rax, qword [fs:0x28]
mov   qword [rbp-0x8], rax

mov   rax, qword [fs:0x28]
mov   rcx, qword [rbp-0x8]
cmp   rax, rcx
jne   0x119e
…
call     __stack_chk_fail
```

(f) clang-x64 / Application

```
stp   x29, x30,[sp, #-0x20]!
mov   x29, sp
adrp  x0, 0x10000
ldr   x0, [x0, #0xfe0]
ldr   x1, [x0]
str   x1, [sp, #0x18]

adrp  x0, 0x10000
ldr   x0, [x0, #0xfe0]
ldr   x3, [sp, #0x18]
ldr   x2, [x0]
subs  x3, x3, x2
mov   x2, #0
b.eq  0x864
bl    __stack_chk_fail
```

(g) gcc-a64 / Application

```
stp x29, x30, [sp, #-32]!
mrs x4, sp_el0
mov x29, sp
ldr x5, [x4, #1144]
str x5, [sp, #24]

mrs x0, sp_el0
ldr x2, [sp, #24]
ldr x1, [x0, #1144]
subs x2, x2, x1
mov x1, #0x0
b.ne 0xffff8000800b306c
…
bl __stack_chk_fail
```

(h) gcc-a64 / Kernel

```
sub   sp, #0x10
add   r7, sp, #0
ldr   r3, [pc, #0x2c]
ldr   r3, [r3]
str   r3, [r7, #0xc]

ldr   r2, [pc, #0x18]
ldr   r1, [r2]
ldr   r2, [r7, #0xc]
eors  r1, r2
mov   r2, #0
beq   #0x81c4
bl    #__stack_chk_fail

PC-relative:  0x1a8bc
```

(i) gcc-m32 / Firmware

```
sub   sp, #0x10
movw  r0, #0x20
movt  r0, #0x3800
ldr   r0, [r0]
str   r0, [sp, #0xc]

movw  r0, #0x20
movt  r0, #0x3800
ldr   r0, [r0]
ldr   r1, [sp, #0xc]
cmp   r0, r1
bne   #0x100036d4
…
bl    #__stack_chk_fail
```

(j) clang-m32 / Firmware

Figure 2: Examples to showcase instrumented stack canary code across different architectures and compilers.

(E1) retrieves the canary value from the designated source again, (E2) compares it with the value residing on the stack and triggers an error handling function if the values do not match.

Figure 2 shows examples of instrumented instructions by different compilers for different targets and architectures. We evaluated the following compilers, including GNU GCC for x86 (gcc-x86), x64 (gcc-x64), Cortex-A (gcc-a64), and Cortex-M (gcc-m32), as well as Clang for Cortex-A (clang-a64) and Cortex-M (clang-m32). Our evaluation encompasses multiple targets and platforms, namely, userland applications on Linux, the Linux kernel, and firmware for Cortex-M devices.

As Figure 2 (b)(c)(e)(f) show, when compiling a userland application for Linux on x86/x64, the instrumented instructions fetch the canary value from `[gs:0x14]` or `[fs:0x28]` as `gs` register or the `fs` register has the address of the userland thread local storage (TLS). Before the function returns, the instrumented instructions fetch the canary value again from TLS and compare it with the one stored on the stack. If it is a match, the function returns normally. Otherwise, a call to `__stack_chk_fail_local()` or `__stack_chk_fail()` is triggered (the distinction will be discussed in §3.2). When compiling the Linux kernel on x64, as illustrated in Figure 2 (d), the instrumented instructions retrieve the canary value from `[gs:0x28]`, as `gs` in the Linux kernel mode has the address of the `fixed_percpu_data` structure, which we will discuss in Section 4.1.

When compiling userland applications and the kernel for the Arm Cortex-A 64-bit architecture, Figure 2 (g)(h) indicate differences when fetching the canary value. For userland applications, the instrumented instructions first calculate the address of a 4KB memory region close to the current PC and write this address to the `x0` register. Next, these instructions retrieve the canary value from an offset (`0xfe0` in this example) of this address and store it on the stack. In the kernel, the instrumented instructions first fetch the value of the `sp_el0` register, which points to the currently executing userland process’s `task_struct` structure. The canary value is then obtained from an offset (`#1144` in this example) of this address, which points to the `stack_canary` field of the `task_struct` structure.

Figure 2 (d)(h) show the architecture-specific stack canary designs of the Linux kernel. Specifically, for x86/x64, the Linux kernel uses a single global canary variable, which is applicable to kernel stacks across all processes. An offset of `#28` in this example from the kernel segment base stores the canary value. In contrast, the Cortex-A Linux kernel employs a per-task canary approach, where each kernel task maintains its own unique stack canary value in its `task_struct` structure.

When compiling firmware for Cortex-M, Figure 2 (i) demonstrates that the instrumented instructions first employ PC-relative addressing `[pc, #0x18]` to fetch the address of the canary value into `r2`. All of the functions use the same canary address. On the other hand, Figure 2 (j) reveals that clang-m32 instruments instructions to obtain the canary value directly from a memory location, specifically `#0x38000020` in this instance. All functions fetch the canary value from the same memory location.

Moreover, both GCC and Clang provide three compilation options that determine which functions to instrument. The first, *-fstack-protector*, activates buffer-overflow checks for functions containing vulnerable objects, such as local character arrays. The second, *-fstack-protector-strong*, offers protection for functions with

```c
void
__attribute__ ((noreturn))
__stack_chk_fail (void)
{
  __fortify_fail ("stack smashing detected");
}

strong_alias (__stack_chk_fail, __stack_chk_fail_local)
```

(a) Code snippet from `stack_chk_fail.c` file

```c
extern void __stack_chk_fail (void) __attribute__ ((noreturn));
/* On some architectures, this helps needless PIC pointer setup
that would be needed just for the __stack_chk_fail call.  */
void __attribute__ ((noreturn)) attribute_hidden
__stack_chk_fail_local (void)
{
  __stack_chk_fail ();
}
```

(b) Code snippet from `stack_chk_fail_local.c` file

Listing 1: Code snippets for canary mismatch handling on glibc-2.38.

```c
1  __visible noinstr void __stack_chk_fail(void)
2  {
3    instrumentation_begin();
4    panic("stack-protector: Kernel stack is corrupted in: %pB",
5    __builtin_return_address(0));
6    instrumentation_end();
7  }
```

(a) Linux kernel

```c
uintptr_t __stack_chk_guard = (uintptr_t) STACK_CHK_GUARD;

__attribute__((noreturn)) void __stack_chk_fail(void)
{
  core_panic(PANIC_SSP, "ssp: stack smashing detected");
}
```

(b) RIOT-OS

```c
void _StackCheckHandler(void)
{
  z_except_reason(K_ERR_STACK_CHK_FAIL);
  CODE_UNREACHABLE; /* LCOV_EXCL_LINE */
}

FUNC_ALIAS(_StackCheckHandler, __stack_chk_fail, void);
```

(c) Zephyr

Listing 2: Code snippets for canary mismatch handling on Linux kernel and microcontroller systems.

various array types (like integer arrays), functions referencing local variables, memory allocation functions such as `alloca()`, and functions possessing buffers larger than 8 bytes. The third, *-fstack-protector-all*, extends the protection to all functions, regardless of their contents. The *-fstack-protector-strong* option has been the default for compiling userland applications in major Linux distributions, such as Ubuntu and their derivatives.

3.2 Canary Mismatch Handling

When a canary mismatch is detected, a call to either function `__stack_chk_fail()` or function `__stack_chk_fail_local()` is made. The implementations of these functions vary depending on where they are implemented. As illustrated in Listing 1 (a)(b), in the GNU C Library (glibc), this function calls `__fortify_fail()`, which in turn calls glibc `abort`. The `abort` function raises a `SIGABRT` signal on Linux. Listing 2 (a) shows the implementation of function `__stack_chk_fail()` in the Linux kernel, which triggers a kernel panic and prints out the symbolic name of the function from which `__stack_chk_fail()` was called (line 4). The `instrumentation_*` functions on line 3 and line 6 are part of the Linux kernel's built-in infrastructure for function tracing. Similarly, the implementations of `__stack_chk_fail()` on microcontroller systems typically log the error and halt the kernel. Listing 2 (b) shows that in RIOT-OS if a canary value mismatch occurs, it immediately sends the kernel into a panic [25]. Listing 2 (c) demonstrates that in Zephyr, if the canary value check fails, the kernel triggers a fatal stack overflow error, thereby halting the system [46].

4 SYSTEM SUPPORT FOR STACK CANARY GENERATION

In this section, we first present glibc and the Linux kernel as state-of-the-art implementations of stack canary value generation. Then, we discuss how stack canary values are generated in microcontroller systems. We also assess a system's support for stack canaries from several aspects. The findings are summarized in Table 1. Additionally, we select Zephyr, an RTOS for MCUs, as a case study to evaluate the performance of stack canaries within microcontroller systems. In particular, we assess the security of implemented stack canary mechanisms from the following aspects:

Canary Randomness: A canary value should be random and not predictable.

Canary Lifespan: Security is enhanced when the lifespan of a canary value is short and not reused.

Canary Size: A larger canary size increases the range of possible values; hence, the canary will be harder to be brute-forced.

Initialization Time: When a stack canary is initialized impacts its effectiveness in securing the system, exposure windows, and resilience to replay and brute-force attacks.

Enabled by Default: Enabling stack canaries by default minimizes configuration errors and lowers the chance of introducing vulnerabilities for developers who are not fully acquainted with a system's security suite.

4.1 The GNU C Library (glibc) and Linux Kernel Implementations

4.1.1 Linux Userland Application. When a new userland application is launched on a Linux system, e.g., via the `execve` system call, the kernel generates random data using the `get_random_bytes()` function. Then, the random data is passed to the userland application through the auxiliary vector structure. During the application's initialization in the userland, glibc fetches the random data passed by the kernel and invokes the `_dl_setup_stack_chk_guard()` function to produce the canary value. Note that child processes created by the `fork` system call will have the same canary value as their parent.

4.1.2 Linux Kernel. To support stack canary in the kernel, the compiler options `STACKPROTECTOR` and `STACKPROTECTOR_STRONG` must be enabled. On Cortex-A and RISC-V architectures, the additional `STACKPROTECTOR_PER_TASK` option is offered to support per-task canary values. Listing 3 shows that during the system boot phase, the Linux kernel invokes the `boot_init_stack_canary()` function, which eventually calls `get_random_long()` to generate

```
1   extern unsigned long __stack_chk_guard;
2
3   static __always_inline void boot_init_stack_canary(void)
4   {
5     unsigned long canary = get_random_canary();
6
7     current->stack_canary = canary;
8     #ifndef CONFIG_STACKPROTECTOR_PER_TASK
9     __stack_chk_guard = current->stack_canary;
10    #endif
11  }
```

(a) Cortex-A

```
1   static __always_inline void boot_init_stack_canary(void)
2   {
3     unsigned long canary = get_random_canary();
4
5     current->stack_canary = canary;
6     #ifdef CONFIG_X86_64
7     this_cpu_write(fixed_percpu_data.stack_canary, canary);
8     #else
9     this_cpu_write(__stack_chk_guard, canary);
10    #endif
11  }
```

(b) x86/64

Listing 3: Code snippets for stack canary value generation on Linux kernel.

Table 1: Comparison of stack canary value generation and mismatch handling across systems

Systems that implement canary value generation and/or mismatch handling							
	Version/ Commit	Size (byte)	Enabled by Default?	Initialization Time	Randomness	Storage	Mismatch Handling
Linux App. (glibc [13])	`v2.38`	4	Yes	application execution	PRNG	Thread Local Storage	kill the thread
Linux Kernel [14]	`v6.5.5`	4	Yes	boot/task creation	PRNG	task_struct	log and panic
LiteOS [15]	`2f8fdf9`	4	No	compile	hardcoded/user-defined RNG	global	log and panic
RIOT-OS [22]	`724e6e0`	4	No	compile	CSPRNG	global	log and panic
Zephyr [45]	`v3.5.0`	4	No	boot	TRNG/PRNG	global	log and panic
Nuttx [2]	`a506f9f`	-	No	-	-	-	panic
Systems that do not implement canary value generation or mismatch handling							
TizenRT (`3.0_GBM`) [24], Mynewt (`6972a1b`) [1], TinyOS (`c4fcab7`) [29], Azure RTOS (`b1b21dd`) [5], RT-Thread (`b1b21dd`) [23], OpenWrt (`12f5372`) [20], Contiki-NG (`b6e22a2`) [10], Mongoose OS (`39b05dd`) [16], FreeRTOS (`4e2a034`) [11], TI-RTOS (`2.21.xx`) [28]							

-: not applicable. The systems discussed in this table utilize 32-bit architectures.

a random canary value. The random value is then stored in the `current task_struct` structure. If the kernel is compiled with `STACKPROTECTOR_PER_TASK`, all tasks will have their own random canary values in their `task_struct` structures. Please note that this per-task canary value is the one the kernel employs when executing on behalf of the task, distinct from the userland canary value discussed in the previous subsection. Additionally, as shown in line 7 of Listing 3 (b), on the x86/64 architecture the canary value is stored in the `fixed_percpu_data`. As detailed in §3.1, on x86/64 compiler-instrumented code access the canary value from this address (`gs:0x28`).

4.1.3 Randomness of the Linux Application and Kernel Stack Canary Values. The randomness of the canary values for both userland applications and the kernel is derived from the kernel's entropy pool, which produces pseudo-random numbers. The pool is periodically seeded with entropy from various sources, including keyboard presses, mouse movements, and disk activity. On 32-bit architectures, the canary value will be 4 bytes, while on 64-bit architectures it will be 8 bytes.

4.2 Microcontroller System Implementations

As shown in Table 1, we studied 14 open-source microcontroller systems [7, 8]. Among these, three discuss how they generate canary values, and four implement mismatch handling functions, which eventually send the system into a panic state. In detail, LiteOS [15] assigns the canary value as `0x000a0dff` for 32-bit architectures and `0x000a0dff000a0dff` for 64-bit architectures. It also provides a weak function, `ArchStackGuardInit()`, which developers can replace to initialize the canary value with their implementation. RIOT-OS [22] uses a cryptographically secure pseudo-random number generator (CSPRNG) to ensure a unique 4-byte canary value for each build. Zephyr [45] generates canary values using a non-cryptographic random number generator (RNG). This RNG derives entropy from a physical source (if compatible with the device) or a system timer clock for pseudo-entropy.

For the ten microcontroller systems that do not implement the canary value generation or mismatch handling, compiling the system results in errors: *undefined symbol* `__stack_chk_guard` *and undefined symbol* `__stach_chk_fail`.

4.3 Overhead on Microcontroller Systems: A Case Study of Zephyr

To evaluate the impact of stack canaries on performance and code size within microcontroller systems, we conducted tests on Zephyr using the Nucleo F412ZG board [18]. This board is powered by an ARM Cortex-M4 core running at 100 MHz with 1MB of flash memory and 512KB of SRAM.

We evaluated three projects on Zephyr (v3.5.0) [30]: (i) *Blinky* [9] implements a single kernel thread that toggles an LED via GPIO API at a predetermined time interval. We modified the infinite toggling to occur 100 times in order to measure the time consumption, (ii) *Producer/consumer* [21] is a userspace example that implements two userspace threads and one kernel thread. It simulates a driver and uses two interconnected userspace tasks: task A fetches and buffers data from a driver, while task B processes this data in a secure environment before task A writes the processed output back to the

Table 2: Performance measurement (%) of Zephyr on three projects with the optimization level -O3

	Benchmark Name	Baseline	Stack Canary Enabled
Zephy on Cortex-M	Blinky	961,101,963	0.000075%
	Producer/consumer	145,793,296	0.012418%
	Multi-threading	265,447,389	0.003991%
Reported overhead on x86 [33]	SPECint	-	2.8%
	SPECCfp	-	2.5%
	SPEC CPU	-	2.7%

-: not applicable. Our evaluation outcomes are influenced not only by the architecture but also by the experimental benchmarks.

Table 3: Code size overhead of Zephyr on three projects with the optimization level -O3.

	Baseline (bytes)*	Stack Canary Enabled		
		overhead*	# instrumented func./all func.*	# instrumented func. called by `main`/... `main`
Blinky	26,206	18.99%	122/160 (76.25%)	15/34 (44.11%)
Producer /consumer	86,184	14.95%	314/371 (84.64%)	34/57 (59.65%)
Multi-threading	36,648	15.94%	141/182 (77.47%)	37/61 (60.66%)

*: results for the whole project. Since the scheduler operates commonly across all projects, they were not clarified in this table.

driver, (iii) *Multi-threading* [17] refers to a scheduler example that utilizes condition variables in a multithreaded application. This example comprises 20 worker threads and one main thread, all of which are kernel threads. The main thread and worker thread switch execution when the worker thread signals the main thread, which is waiting on the condition variable.

For compilation, we employed the Zephyr-specific GCC cross-compiler designed specifically for the ARM architecture. We set the optimization level to `-O3` to match the optimization used by [33] for a reliable comparison with the x86 architecture. To assess performance overhead, we monitored CPU cycle usage across the entire `main()` function. Furthermore, we statically analyzed the binary files using Ghidra [12] to understand the instrumentation details for each function. Zephyr does not enable stack canaries by default. Therefore, the default-configured Zephyr was used as our baseline.

Table 2 shows that the performance overhead of Zephyr running on Cortex-M is significantly lower than that of benchmarks run on x86. *Blinky*, *Producer/Consumer*, and *Multi-threading* recorded overheads of 0.000075%, 0.012418%, and 0.003991%, respectively. Table 3 presents a substantial increase in code size for these projects: 18.99% for *Blinky*, 14.95% for *Producer/Consumer*, and 15.94% for *Multi-threading*, respectively. This increase is attributed to over 75% of the functions in these projects being instrumented for canary runtime checks. Our focus was further narrowed to functions called by `main()`, aligning with our performance study. Table 3 also indicates that *Producer/Consumer* has 59.65% of its functions instrumented, which is slightly less than the 60.66% in *Multi-threading*. Nevertheless, *Producer/Consumer* incurs a higher performance overhead compared to *Multi-threading*. This is largely due to the more frequent execution of functions with canary checks during runtime.

5 SECURITY ANALYSIS OF STACK CANARY ON MICROCONTROLLER SYSTEMS

In this section, we discuss the weaknesses of canary value generation on microcontroller systems. We also discuss the fundamental reasons for these weaknesses and emphasize the common challenges faced by stack canary value generation on MCUs.

5.1 Weaknesses

5.1.1 Weakness 1: A Single Canary Value in the Address Space. Microcontroller systems use a global canary value for all kernel and task functions. Therefore, attackers only need to deduce one value in order to compromise stacks across the system. For MCUs, the kernel and tasks often reside within the same physical memory address space. Ensuring proper isolation under such conditions requires additional measures, often involving the use of MPUs. However, Zhou et al. [48, 49] have observed that MPUs are not widely adopted in commercial products and often do not function as intended due to high overhead and conflicts with the existing system design.

5.1.2 Weakness 2: No or Weak Randomness. LiteOS relies on developers to implement an RNG function for randomizing canary values. In the absence of RNG, it assigns default canary values, which, if known to attackers, are vulnerable to circumvention. Furthermore, the canary value stays fixed until the system reboots or recompiles, even in systems equipped with RNG capabilities. The static nature of the canary makes it susceptible to attackers who can deploy brute-force tactics, where different values are tried until they find the right one.

5.1.3 Weakness 3: Lack of Good Entropy for Randomness. Many MCUs lack good entropy sources, which are essential for random number generation because of their design priorities. Specifically, MCUs are built for simplicity, energy efficiency, and affordability. Adding reliable entropy sources can increase their complexity, energy use, and cost. In addition, many microcontroller systems prioritize predictable, real-time responses, making the introduction of randomness potentially counterproductive. However, without a good RNG, the system cannot ensure the randomness of a stack canary value. For devices without TRNG, Zephyr offers a PRNG that uses the system timer for entropy. However, since attackers might manipulate the system's boot time, controlling the system timer and potentially retrieving the canary value becomes feasible.

6 DISCUSSIONS

6.1 Reduce Attack Surface

One way to reduce the attack surface of stack canaries is to avoid storing the reference canary in insecure memory, where it could be read or overwritten by an attack. Introduced for ARM microcontrollers, the Pointer Authentication (PA) mechanism [4, 36] acts as a countermeasure against memory corruptions. It generates and verifies the Pointer Authentication Code (PAC) for pointers using the QARMA block cipher. PCan [40] leverages PA to dynamically generate canaries while preventing the exposure of referenced canaries in memory. In addition to the PAC, the Physical Unclonable

Function (PUF) [42] offers another layer of hardware support, providing a unique root of key that remains concealed from developers. PUFCanary [34] employs the PUF to randomize canary generation, thereby eliminating the need to store sensitive canary words in memory or CPU registers.

6.2 Enlarge Entropy Source Pool

One way to enhance the randomness is to bolster the entropy pool, which will improve the randomness of the canary value. For MCUs that lack TRNG, we recommend merging various entropy sources. The system timer utilized by Zephyr is one potential source of randomness. Additionally, other resources, such as unused SRAM, sensor outputs like temperature monitors, and audio and video streams, can also contribute to the entropy source. For instance, μArmor [31] introduces μRNG, a CSPRNG design that utilizes SRAM startup values, a range of oscillators, and analog-to-digital converters as sources of entropy.

6.3 Use a Memory-safe Language

Memory-safe languages like Rust provide memory safety assurances without compromising performance. Many microcontroller systems, including Tock OS [39] and Bern RTOS [6], harness the advantages of Rust to address memory safety concerns in their development.

6.4 Other Efficient Security Mechanisms

In addition to stack canaries, several other security mechanisms have garnered extensive research attention. GCC and Clang support the `FORTIFY_SOURCE` [19] macro with the glibc to enhance security. Unlike broader mechanisms like stack canaries, `FORTIFY_SOURCE` specifically improves certain C standard library functions, like `strcpy` and `sprintf`, by replacing them with safer versions when buffer sizes are predictable at compile-time, thereby minimally impacting code size. Additionally, Silhouette [47] and Kage [35] utilize unprivileged store and load instructions to protect the shadow stack, enhancing control-flow security in microcontroller systems. Meanwhile, CRT-C [38] and EC [37] advocate for efficient compartmentalization techniques. Such approaches offer a finer security granularity compared to stack canaries and achieve this with modest performance and code size implications.

7 CONCLUSION

In this paper, we delve into the three phases of the stack canary mechanism. We conduct an in-depth examination of stack canaries' implementation in compilers, libraries, and systems. We also use Zephyr as a case study to explore the overhead on microcontroller systems. We observed that the generation of canary values in microcontroller systems not only lacks emphasis but also robustness, unveiling five distinct weaknesses. Unfortunately, some weaknesses are inherent and cannot be easily rectified. Ultimately, it appears as though the stack canary might not be the best fit for microcontroller systems. This underscores the need to explore alternative security mechanisms.

ACKNOWLEDGMENT

This material is based upon work supported in part by a National Science Foundation (NSF) grant (2237238) and a National Centers of Academic Excellence in Cybersecurity grant (H98230-22-1-0307). Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

REFERENCES

[1] 2023. Apache Mynewt. https://github.com/apache/mynewt-core. (2023).
[2] 2023. Apache NuttX. https://nuttx.apache.org/. (2023).
[3] 2023. ARM Cortex-M. https://www.arm.com/products/silicon-ip-cpu?families=cortex-m&showall=true. (2023).
[4] 2023. Armv8.1-M Pointer Authentication and Branch Target Identification Extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension. (2023).
[5] 2023. Azure RTOS ThreadX. https://github.com/azure-rtos/threadx. (2023).
[6] 2023. Bern RTOS. https://bern-rtos.org/. (2023).
[7] 2023. Best IoT Operating Systems of 2023. https://slashdot.org/software/iot-operating-systems/. (2023).
[8] 2023. Best Real-Time Operating Systems (RTOSs) of 2023. https://slashdot.org/software/real-time-operating-systems-rtos/. (2023).
[9] 2023. Blinky sample. https://docs.zephyrproject.org/latest/samples/basic/blinky/README.html. (2023).
[10] 2023. Contiki-NG: The OS for Next Generation IoT Devices. https://www.contiki-ng.org/. (2023).
[11] 2023. FreeRTOS: Real-time operating system for microcontrollers. https://www.freertos.org/. (2023).
[12] 2023. Ghidra website. https://ghidra-sre.org/. (2023).
[13] 2023. GNU C library. https://elixir.bootlin.com/glibc/glibc-2.38/source. (2023).
[14] 2023. Linux version 6.5.5. https://elixir.free-electrons.com/linux/v6.5.5/source. (2023).
[15] 2023. LitOS Github. https://github.com/LiteOS/LiteOS. (2023).
[16] 2023. Mongoose OS. https://mongoose-os.com/. (2023).
[17] 2023. Multi-threding sample. https://docs.zephyrproject.org/latest/samples/kernel/condition_variables/simple/README.html. (2023).
[18] 2023. NUCLEO-F412ZG board. https://www.st.com/en/evaluation-tools/nucleo-f412zg.html. (2023).
[19] 2023. Object size checking to prevent (some) buffer overflows. https://gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg02055.html. (2023).
[20] 2023. OpenWrt. https://openwrt.org/start. (2023).
[21] 2023. Producer/consumer sample. https://docs.zephyrproject.org/latest/samples/userspace/prod_consumer/README.html. (2023).
[22] 2023. RIoT. https://github.com/RIOT-OS/RIOT. (2023).
[23] 2023. RT-Thread. https://www.rt-thread.io/. (2023).
[24] 2023. Samsung TizenRT. https://github.com/Samsung/TizenRT. (2023).
[25] 2023. Samsung TizenRT: stack_protector.c. https://github.com/Samsung/TizenRT/blob/1c9e6fdbb53006a50702eca23abbf4b5cca5c1c7/os/board/rtl8730e/src/component/soc/amebad2/atf/lib/stack_protector/stack_protector.c. (2023).
[26] 2023. The Arm ecosystem ships a record 6.7 billion Arm-based chips in a single quarter. https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter. (2023).
[27] 2023. The GNU Compiler Collection. https://gcc.gnu.org/. (2023).
[28] 2023. TI-RTOS (RTOS Kernel) Overview. https://software-dl.ti.com/simplelink/esd/simplelink_cc13x0_sdk/4.10.02.04/exports/docs/proprietary-rf/proprietary-rf-users-guide/proprietary-rf-guide/tirtos-index.html. (2023).
[29] 2023. TinyOS. https://github.com/tinyos/tinyos-main. (2023).
[30] 2023. Zephy. https://github.com/zephyrproject-rtos/zephyr/releases/tag/v3.5.0. (2023).
[31] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. 2019. Challenges in designing exploit mitigations for deeply embedded systems. In *European Symposium on Security and Privacy (EuroS&P)*. IEEE.
[32] 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*.
[33] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security*.
[34] Asmit De, Aditya Basu, Swaroop Ghosh, and Trent Jaeger. 2020. Hardware assisted buffer protection mechanisms for embedded RISC-V. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).

[35] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J Walls, and John Criswell. 2022. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *USENIX Security Symposium*.

[36] Qualcomm Technologies Inc. 2023. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf. (2023).

[37] Arslan Khan, Dongyan Xu, and Dave Jing Tian. 2023. Ec: Embedded systems compartmentalization via intra-kernel isolation. In *Symposium on Security and Privacy (S&P)*. IEEE.

[38] Arslan Khan, Dongyan Xu, and Dave Jing Tian. 2023. Low-cost privilege separation with compile time compartmentalization for embedded systems. In *Symposium on Security and Privacy (S&P)*. IEEE.

[39] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *Symposium on Operating Systems Principles*.

[40] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. 2019. Protecting the Stack with PACed Canaries. In *Workshop on System Software for Trusted Execution*. ACM.

[41] Hector Marco-Gisbert and Ismael Ripoll. 2013. Preventing brute force attacks against stack canary protection on networking servers. In *International Symposium on Network Computing and Applications*. IEEE.

[42] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. 2002. Physical one-way functions. *Science* (2002).

[43] Jiadong Sun, Xia Zhou, Wenbo Shen, Yajin Zhou, and Kui Ren. 2020. PESC: A Per System-Call Stack Canary Design for Linux Kernel. In *Data and Application Security and Privacy*. ACM.

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

[45] Zephyr. 2023. Zephyr RTOS. https://zephyrproject.org/. (2023).

[46] Zephyr Project: compiler_stack_protect.c. 2023. Stack Overflows Detection. https://github.com/zephyrproject-rtos/zephyr/blob/078967671c9038367edeb60818c0e69015320e32/kernel/compiler_stack_protect.c. (2023).

[47] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. 2020. Silhouette: Efficient protected shadow stacks for embedded systems. In *USENIX Security Symposium*.

[48] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2019. Good motive but bad design: Why ARM MPU has become an outcast in embedded systems. *arXiv preprint arXiv:1908.03638* (2019).

[49] Wei Zhou, Zhouqi Jiang, and Le Guan. 2023. Understanding MPU Usage in Microcontroller-based Systems in the Wild. (2023).