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  
mdaranu@buffalo.edu

Zheyuan Ma  
CactiLab, University at Buffalo  
USA  
zheyuann@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.

KEYWORDS
Microcontroller systems; stack canaries

ACM Reference Format:

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 RTOs. 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 RTOs. 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 re-use 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 re-use 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.

![Figure 1: Three Phases of the Stack Canary Mechanism](image)

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

SAC ’24, April 8–12, 2024, Avila, Spain

<table>
<thead>
<tr>
<th>Instrumentation Locations</th>
<th>Instrumented code for prologue</th>
<th>Instrumented code for epilogue</th>
<th>Instrumented code for canary value fetching</th>
</tr>
</thead>
<tbody>
<tr>
<td>int func()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instrumentation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char buf[6];</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instrumentation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov edx, dword [gs:0x14]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov dword [ebp-0x4], edx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>je 0x1200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call __stack_chk_fail</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a) Instrumentation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locations</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Prologue ...
call __stack_chk_fail

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 variable 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_fail1() 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_per_cpu_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 (0xf300 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_e10 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 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
was called (line 4). The __stack_chk_fail() 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].

### 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 random data for the stack canary.
As shown in Table 1, we studied 14 open-source microcontroller systems, and four implement mismatch handling functions, which is stored in the value discussed in the previous subsection. Additionally, as shown in line 7 of Listing 3 (b), on the x86/64 architecture the canary value will be 4 bytes, while on 64-bit architectures it will be 8 bytes.

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 __stack_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

---

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

```c
extern unsigned long __stack_chk_guard;
static __always_inline void boot_init_stack_canary(void) {
    unsigned long canary = get_random_canary();
    current->stack_canary = canary;
    __stack_chk_guard = current->stack_canary;
} #ifdef CONFIG_STACKPROTECTOR_PER_TASK
    __stack_chk_guard = current->stack_canary;
#else
    this_cpu_write(__stack_chk_guard, canary);
#endif

static __always_inline void boot_init_stack_canary(void)
{
    unsigned long canary = get_random_canary();
    current->stack_canary = canary;
    if(defined CONFIG_X86_64)
    this_cpu_write(fixed_percpu_data.stack_canary, canary);
    else
    this_cpu_write(__stack_chk_guard, canary);
} #endif
```

---

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

<table>
<thead>
<tr>
<th>Systems that implement canary value generation and/or mismatch handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version/Commit</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Linux App. (glibc [13]) v2.38</td>
</tr>
<tr>
<td>Linux Kernel [14] v6.5.5</td>
</tr>
<tr>
<td>LiteOS [15] 2f8fdff</td>
</tr>
<tr>
<td>RIOT-OS [22] 724e6e0</td>
</tr>
<tr>
<td>Zephyr [45] v3.5.0</td>
</tr>
<tr>
<td>Nuttx [2] a506f9f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Systems that do not implement canary value generation or mismatch handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>TizenRT (3.0_GBM) [24], Mynnewt (6972a1b) [1], TinyOS (c4f7ca7b) [29], Azure RTOS (b1b21dd) [5], RT-Thread (b1b21dd) [23], OpenWrt (12f5372) [20], Contiki-NG (bb6e22a2) [10], Mongoose OS (39b85dd) [16], FreeRTOS (4e2a834) [11], TI-RTOS (2.21.xx) [28]</td>
</tr>
</tbody>
</table>

---

*: not applicable. The systems discussed in this table utilize 32-bit architectures.
Table 2: Performance measurement (%) of Zephyr on three projects with the optimization level -O3

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Baseline</th>
<th>Stack Canary Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blinky</td>
<td>961,101,963</td>
<td>0.000075%</td>
</tr>
<tr>
<td>Producer/consumer</td>
<td>145,793,296</td>
<td>0.012418%</td>
</tr>
<tr>
<td>Multi-threading</td>
<td>265,447,389</td>
<td>0.003991%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Baseline (bytes)*</th>
<th>Stack Canary Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blinky</td>
<td>26,206</td>
<td>18.99%</td>
</tr>
<tr>
<td>Producer/consumer</td>
<td>86,184</td>
<td>14.95%</td>
</tr>
<tr>
<td>Multi-threading</td>
<td>36,648</td>
<td>15.94%</td>
</tr>
</tbody>
</table>

Table 2 shows that the performance overhead of Zephy 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, Zephy 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 Unclo
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


Is the Canary Dead? On the Effectiveness of Stack Canaries on Microcontroller Systems SAC ’24, April 8–12, 2024, Avila, Spain
www.qualcomm.com/content/dam/qcomm-maritech/dm-assets/documents/p
[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 separa-
tion with compile-time compartmentalization for embedded systems. In Sym-
posium on Security and Privacy (S&P). IEEE.
Prabal Dutta, and Philip Leviu. 2017. Multiprogramming a 64k 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.
against stack canary protection on networking servers. In International Sym-
posium on Network Computing and Applications. IEEE.
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.
https://github.com/zephyrproject-rtos/zephyr/blob/078967671c9038367edeb60
818c6e6901532e32/kernel/compiler_stack_protect.c. (2023).
design: Why ARM MPU has become an outcast in embedded systems. arXiv