future internet

Article

Through the Window: Exploitation and Countermeasures of the ESP32 Register Window Overflow †

Kai Lehniger [1,*] and Peter Langendörfer [1,2]

1 IHP—Leibniz-Institut für Innovative Mikroelektronik, 15236 Frankfurt (Oder), Germany
2 BTU Cottbus-Senftenberg, 03046 Cottbus, Germany; peter.langendoerfer@b-tu.de or langendoerfer@ihp-microelectronics.com
* Correspondence: lehniger@ihp-microelectronics.com; Tel.: +49-335-5625-277

† This article is a revised and expanded version of the paper entitled "Through the Window: On the exploitability of Xtensa's Register Window Overflow", which was presented at "International Telecommunication Networks and Applications Conference 2022 (ITNAC 2022)".

Abstract: With the increasing popularity of IoT (Internet-of-Things) devices, their security becomes an increasingly important issue. Buffer overflow vulnerabilities have been known for decades, but are still relevant, especially for embedded devices where certain security measures cannot be implemented due to hardware restrictions or simply due to their impact on performance. Therefore, many buffer overflow detection mechanisms check for overflows only before critical data are used. All data that an attacker could use for his own purposes can be considered critical. It is, therefore, essential that all critical data are checked between writing a buffer and its usage. This paper presents a vulnerability of the ESP32 microcontroller, used in millions of IoT devices, that is based on a pointer that is not protected by classic buffer overflow detection mechanisms such as Stack Canaries or Shadow Stacks. This paper discusses the implications of vulnerability and presents mitigation techniques, including a patch, that fixes the vulnerability. The overhead of the patch is evaluated using simulation as well as an ESP32-WROVER-E development board. We showed that, in the simulation with 32 general-purpose registers, the overhead for the CoreMark benchmark ranges between 0.1% and 0.4%. On the ESP32, which uses an Xtensa LX6 core with 64 general-purpose registers, the overhead went down to below 0.01%. A worst-case scenario, modeled by a synthetic benchmark, showed overheads up to 9.68%.

Keywords: Xtensa; memory corruption; buffer overflow; register windows; windowed ABI; return-oriented programming

check for updates

Citation: Lehniger, K.; Langendörfer, P. Through the Window: Exploitation and Countermeasures of the ESP32 Register Window Overflow. *Future Internet* **2023**, *15*, 217. https://doi.org/10.3390/fi15060217

Academic Editor: Claude Chaudet

Received: 27 April 2023
Revised: 14 June 2023
Accepted: 15 June 2023
Published: 19 June 2023

1. Introduction

Xtensa is a processor architecture for application-specific designs. It offers a large set of configuration options, starting from various-size options of caches and registers up to the inclusion of instruction-set extensions and the possibility to include its own instructions in the design to accelerate computation. It is used for fast audio and video processing, but also in microcontrollers, such as the ESP32.

One of the configuration parameters is the Application Binary Interface (ABI). Xtensa offers two different ABIs, call0 and windowed [1]. The call0 ABI has standard calling conventions using a function's prologue and epilogue to save and restore registers, whereas the windowed ABI offers a larger number of physical registers and a sliding-window mechanism similar to the SPARC architecture to increase speed and code density [1].

A register window is a mapping of a virtual register set to a larger physical set. When a function is called, instead of saving registers to the stack, the mapping of the register window can be changed to make new registers available. When the function returns, the mapping is moved back.

As the number of physical registers is limited, it happens that the processor runs out of unused registers when a new function is called. In this case, registers must be saved onto the stack. For Xtensa, a window-overflow exception is thrown. Later, when the control flow returns to a function call that had its registers saved onto the stack, a window underflow exception is thrown by the return instruction to load the registers' content back [2].

Past works have demonstrated that the window underflow exception handler can be exploited to change the control flow of a program, allowing Return-Oriented Programming (ROP) attacks [3,4]. ROP attacks change the control flow of a program by overwriting a vulnerable return address on the stack [5]. This work now focuses on the mechanisms of the window-overflow exception handler and tries to give a detailed analysis regarding the potential exploit.

Although not as potent as the window underflow exception handler, due to the limitations we discovered, this paper demonstrates how a stack buffer overflow vulnerability in combination with a window overflow can be used to modify values outside of the stack segment. In addition, we describe how this attack can be used to bypass Stack Canaries and, under rare conditions, be extended to a ROP attack. Stack Canaries offer protection against stack buffer overflows that insert a guard word after local variables in a stack frame [6]. This guard word is checked for modification before the function returns. A register window overflow can be triggered after a stack buffer overflow that accesses data that was manipulated by the aforementioned buffer overflow. As this can be done before the Stack Canary check is executed, the manipulation can be hidden, and the overflow attack goes undetected. The restrictions of this method are discussed, and a possible countermeasure is presented.

This paper focuses only on the Xtensa LX architecture used in the ESP32 microcontroller. There is already a successor, the Xtensa NX architecture, which uses a simplified register window mechanism, which does not contain the vulnerability anymore. Nevertheless, there are still millions of devices using the Xtensa LX architecture. Therefore, we believe our analysis and the presented patch are still of great use to the community.

This work is an extension to a paper first presented at the International Telecommunication Networks and Applications Conference 2022 (ITNAC 2022) [7]. The contributions of the original paper were the introduction of a new vulnerability that utilizes the window-overflow exception handlers of the Xtensa LX architecture. Countermeasures were introduced to mitigate the vulnerability. One particular countermeasure that involves a small patch of the exception handlers was evaluated. With this extension, we hope to provide a better introduction to the field as well as a more detailed discussion of the vulnerability. We also extended our evaluation by moving to a real board instead of relying on simulations, leading to unexpected results as well as an adaptation to our benchmark methodology using a synthetic benchmark as an addition to the CoreMark benchmark. We also extended our discussion of the feasibility of the vulnerability with regard to memory safety measures of the ESP32 microcontroller.

The rest of this paper is structured as follows. Section 2 describes the current state of the art, and Section 3 describes the window mechanism of the Xtensa architecture. Section 4 describes the exploit in detail and Section 5 discusses countermeasures. In Section 6, our proposed countermeasure is evaluated. Section 7 concludes this paper.

2. State of the Art

Buffer overflows are a common vulnerability that has been known for decades. The usage of unsafe languages such as C, especially for embedded programming, short time-to-market, and the lack of security update support for a lot of cheap IoT devices are just some reasons why buffer overflows are relevant up to this date.

First, stack buffer overflows were used to inject binary code into the stack segment and overwrite the return address to change the control flow to the injected code [8]. These so-called shellcode injection attacks can be prevented by marking the stack segment as non-executable, a technique implemented by most modern processors.

Return-into-libc attacks were developed to manipulate function arguments and jump to a system function call to create malicious behavior. This was later generalized into ROP [5], an attack that uses existing code snippets called gadgets. All these gadgets consist of a small number of instructions and a return. By overwriting a return address on a stack with the address of such a gadget, it is possible to change the control flow and execute the instructions of the gadget. The return of the gadget can then pick-up the next gadget address from the stack, which creates a gadget chain. This allows an attacker to create arbitrary behavior with a sufficiently large set of gadgets.

To prevent the manipulation of the return address, various mechanisms have been implemented. The most popular ones are Stack Canaries and Address Space Layout Randomization (ASLR) [9]. ASLR randomizes the location of segments when a process is loaded, which requires an OS to implement it. Therefore, it is often missing in deeply embedded systems where the application is linked together with OS components into a single binary. Although not directly protecting return addresses, it makes the manipulation of something more complicated as an attacker first needs to find out the addresses before he can redirect the control flow.

Recently, it has become common to protect the control flow directly with Control Flow Integrity (CFI) [10]. CFI does that by including runtime checks into the program that either checks against a common ruleset in coarse-grained CFI [11–13], or against a pre-calculated Control Flow Graph (CFG) in fine-grained CFI, as described in the original implementation [10]. The problem with the CFG approach is that backward edges (function returns) can lead to many positions in the code, making CFI weak. Shadow Stacks are a great alternative to protect these backward edges instead of using a CFG [14]. A Shadow Stack is a separate stack that contains a copy of all return addresses. Before a return address is used, it is compared against its copy.

A more recent protection mechanism of return addresses is chaining them together [15,16]. Unused bits of a 64-bit address are used to store additional information for authentication. Without this knowledge, an attacker cannot insert a manipulated return address without breaking this chain. The most recent authentication information is hidden in registers, so it is not possible to manipulate the beginning of the chain. All return addresses are checked before usage. Although, in [15] custom hardware, extensions for a RISC-V core are used, the authors in [16] use the Pointer Authentication (PAC) instruction-set extension for ARM, which has recently been shown to be vulnerable against micro-architectural side-channel attacks on Apple's M1 processor [17].

Stack Canaries [18] are implemented by the compiler by inserting a canary word in between a buffer and the return address and checking it for manipulation before executing the return. There are a variety of different implementations for canaries, each with their own pros and cons. This starts with the canary word itself. It can either consist of random bytes, pre-defined terminator characters, or a combination of both [19]. The idea behind using terminator characters is to make it impossible to write past the canary word with functions that operate on strings that are commonly exploited for stack buffer overflow attacks, i.e., *strcpy*.

Stack Canary protection has been broken in the past with different methods. On 32-bit systems, it is possible to overcome canary protection with simple brute force, especially when a combination of random and terminator bytes is used and only 1 or 2 bytes are random. Another possibility is to leak the value of the canary word. In an attack called stack reading [20] or a byte-by-byte [21] attack, applications that are forking child processes are attacked by only overwriting one byte of the canary word. When the process crashes, the guess for the byte was wrong; when it continues, the bytes were guessed correctly and the attack continues with the next byte until the whole canary word is leaked. This works when, during a fork operation, the canary word is not re-randomized, which was later done by [22].

Other protection mechanisms already instrumentalize exception handlers for architectures with register windows. In [23], the return address is XORed with a secret value to obfuscate it in a SPARC processor. The detection is made by the fact that SPARC's instructions must be four-byte aligned, with pointers to different locations raising an exception. Since the chance for an attacker to guess a return instruction that still results in the last two bits being zero after being XORed with the secret value is relatively high, they also discussed different applications of exception handler instrumentation. They present the concept of an early Shadow Stack implementation using this mechanism. For the Xtensa architecture, exception handler instrumentation was used to implement a variation of Stack Canaries, called Window Canaries, where the instrumentation is moved from function prologues and epilogues into the exception handlers [24].

3. The Xtensa Register Window Mechanism

The Xtensa architecture has a configuration to use a register window mechanism. With this configuration, the regular number of 16 general-purpose registers `a0`–`a15` stays the same, but is virtualized and backed up by a larger physical register set. The physical register set contains either 32 or 64 registers `ar0`–`ar31`/`ar63`, depending on the configuration.

An example of such a mapping is illustrated in Figure 1. In the example, the mapping starts at register `ar24` and ends at register `ar39`. The current mapping is changed when a new function is called, or a function returns. The direction is shown in the figure. The number of registers around which the register window is rotating depends on the call instruction that is used for a function call. In total, there are eight different call instructions for the Xtensa LX architecture, four for direct, and four for indirect calls. Each group contains call instructions to move the register window by either 0, 4, 8, or 12 registers. The instructions `call0` and `callx0`, which do not move the register window, are only used within the call0 ABI. As the registers shift by at most 12 registers, there is always an overlapping between the old and new register windows. This is used to pass arguments and return values between the caller and the callee.

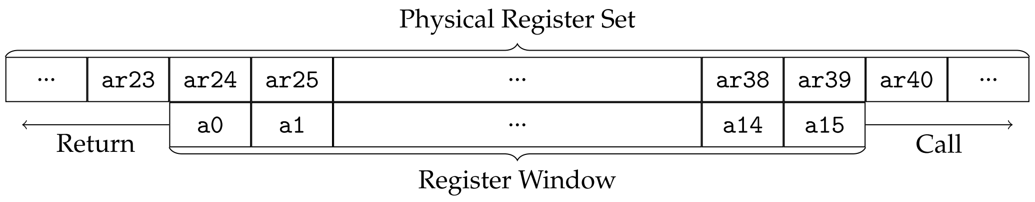

Figure 1. Example of a register window mapping of logical registers `a0`–`a15` to physical registers `ar24`–`ar39`.

In general, this mechanism increases code density and efficiency as it eliminates the necessity of saving and restoring register values in the prologue and epilogue of a function. With a new function, the register window can simply shift to a new set of registers. It is only limited by the number of physical registers. When a program reaches a certain call depth, there are no registers left that are not already in use. In such a case, the processor detects memory access to such a register and generates a window-overflow exception. Please note that the overflow is not generated when a function is called but only when a register that already belongs to a different window is accessed. The overflow exception is responsible for saving register values of the non-overlapping part of the register window to the stack before returning to the regular program flow. For example, in Figure 2, if *Function A* with a corresponding register window W_A is calling *Function B* using a `call8` instruction and at a later points its register values are saved by a window overflow, a *_WindowOverflow8* handler is invoked, saving only registers `a0`–`a7` of *Function A*. The other registers are overlapping with the window of *Function B* and are therefore associated with its window W_B.

The processor keeps track of which windows are currently in the physical register set and which have been written to the stack. When a program returns from a function using a windowed return instruction `retw`, it is checked whether or not register values must be loaded back into the physical register set or not. In such a case, a window underflow exception is invoked that restores all register contents.

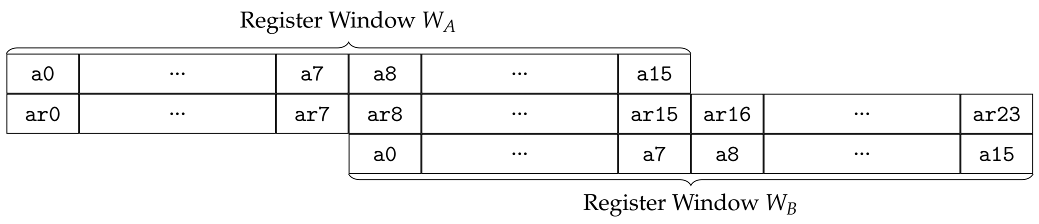

Figure 2. Example of two overlapping register windows W_A and W_B.

4. Exploiting the Window-Overflow Exception Handler

This register window mechanism is not a security problem itself. The vulnerability comes from the specific implementation of the window exception handlers and the way that register values are saved. For each function call, there are pre-allocated memory areas in the stack. There are two types of areas allocated for each stack frame, namely the Base Save Area (BSA) and the Extra Save Area (ESA). The BSA has a fixed size of 16 bytes, containing values for registers `a0`–`a3`. These registers represent the minimal number of registers that move out of the current window. Any potential additional registers are stored in the ESA. The size of an ESA can therefore vary, from 0 bytes up to 32 bytes.

The position of these memory areas is critical too. Figure 3 shows the stack for a scenario where the registers of a function call i are about to be saved onto the stack by an overflow exception. It shows a total of three stack frames—Frame $i - 1$, Frame i, and Frame $i + 1$—as well as save areas that are relevant for exception handling. Other parts of the stack frames, including ESAs other than ESA i are only represented by "..." for simplicity.

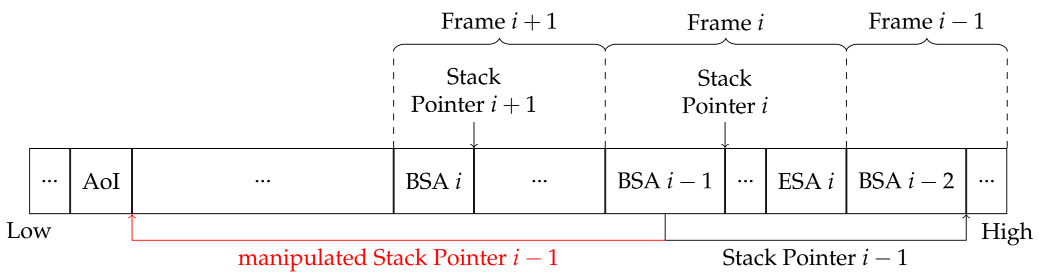

Figure 3. Stack layout and relevant pointers used during a window-overflow exception to save registers into BSA i and ESA i, including Stack Pointer $i - 1$, which is loaded from the stack and can be manipulated by an attacker to point to an Area of Interest (AoI).

The placement of ESA i is straight forward. It is placed at the highest address of Frame i. Its size depends on the call instructions used inside the function body. This way, the compiler can determine how much space to allocate for the ESA. The BSA i, on the other hand, is in Frame $i + 1$. This is used to be able to backtrack the stack, as the BSA contains register `a1`, which acts as the stack pointer.

When an overflow occurs, the window is rotated to a position where the stack pointers for Frame i and $i + 1$ are in the register window. For example, during an overflow of eight, Stack Pointer $i + 1$ is in register `a9`, and Stack Pointer i is in register `a1`. The BSA is located at the lowest address part of a stack frame, right below the stack pointer. This allows it to have a fixed offset to the register values in the BSA. Finding the position of ESA i is a little more complicated. Since stack frame sizes can differ, the offset of ESA i in relation to Stack Pointer i is unknown for the exception handler. To reach it, it uses Stack Pointer $i - 1$, which is stored in BSA $i - 1$. Since the size of a BSA is constant, it is possible to address the values in ESA i using Stack Pointer $i - 1$ with constant offsets.

Since BSA $i - 1$ was already written to the stack by a previous window-overflow exception, this is the part where an attacker can use a stack buffer overflow to alter the value of Stack Pointer $i-1$, illustrated in Figure 3 by the red arrow. This moves the position of the ESA to any position the attacker chooses, here named the Area of Interest (AoI). This area could contain sensible data that the attacker wants to overwrite, or maybe is used for memory-mapped registers, which would allow the attacker to change the configuration of the IoT device. Another possibility is to target an output buffer to leak the saved register values.

For the attack to work, there are few more requirements than just a simple stack buffer overflow vulnerability. This is illustrated in a simple example in Listing 1. The shown function *vulnerable*first copies the *input* into a local *buffer*. For an attack that targets the return address, this would already be enough. However, widely used defense mechanisms such as Stack Canaries would detect the buffer overflow before the return is executed. Having another function *compute* or access to a high register can potentially trigger a window-overflow exception. Therefore, the use of a manipulated pointer before the canary word is checked.

Listing 1. Typical stack buffer overflow vulnerability.

```c
void vulnerable(char *input){
    char buffer[BUF_SIZE];
    strcpy(buffer, input);  // potential buffer overflow
    compute(buffer);        // can trigger a window overflow
    return;                 // stack canary check before return
}
```

4.1. Attack Model

We assume the attacker can perform a single consecutive write operation starting from any local variable in the stack segment. This is a relatively weak assumption compared to other papers that focus on defense mechanisms against more advanced memory corruption attacks, which mostly assume the attacker is capable of arbitrary read and write operations [25]. The reason for this is that the whole purpose of the mechanism discussed in this paper is to leverage a simple stack buffer overflow into the ability to write to arbitrary memory addresses.

We can extend this model by allowing arbitrary read and write operations in the stack segment. This would give the attacker the ability to circumvent canary-based protection mechanisms more easily. The more important part is that chaining the exploit or combining it with other attacks would become easier. The reason for this, as well as how realistic we think this scenario is, is discussed in Section 4.4.

4.2. Example Attack: Overcome the Stack Canary Protection

As an example, in this subsection, we describe an attack that targets the Stack Canary protection to overcome it and enable further attacks. Stack Canaries are a widely used, compiler-based detection mechanism for stack buffer overflows. A data word is placed onto the stack during the prologue of a function and checked for modification in the epilogue. The idea behind this mechanism is that a stack buffer overflow must overwrite this canary value to be able to modify sensible data such as the return address.

This implementation uses a reference canary value. It is placed in a different memory location to the stack itself and it is assumed to be unreachable by an attacker that is only exploiting a stack buffer overflow vulnerability. Furthermore, modification of the reference canary value can be mitigated, for example, by using a Memory Management Unit (MMU) or Memory-Protection Unit (MPU), if present. The ESP32 offers both. Their impact on the subsequent vulnerability is explored in the following section.

There are two general types of canaries, namely random canaries and terminator canaries. The random canary word is, as its name suggests, a random data word. For the random canary word to work, it is important to re-randomize it for every start of a process, or even with a *fork* of a child process to not be vulnerable against certain attacks such as stack reading [20]. Since embedded devices typically do not operate with strict processes that are created and loaded, re-randomization of the canary may take more effort and cannot fully be covered by the compiler alone.

Terminator canaries are an alternative, as they do not require randomness or secrecy to be secure. They rely on the idea that many buffer overflows are caused by unsafe functions that operate on strings, i.e., *strcpy*. These functions typically use a terminator character that indicates the end of the string instead of a length input parameter. An attacker can use this to create a buffer overflow simply by supplying a very long string, if said string is not checked before processing. Terminator canaries use this by embedding these terminator characters into the canary word. This way, an attacker that tries to overflow a buffer using one of the unsafe string functions can no longer overflow a buffer past the canary word without modifying it.

Another factor that will become important for the attack is that canaries are typically not applied to all stack frames to not impact performance too much. Instead, compilers rely on heuristics to determine which functions to instrument. Both GCC and Clang support the stack protector feature that allows the implementation of Stack Canaries with different levels of granularity [26,27].

- `-fno-stack-protector`: disables canaries.
- `-fstack-protector`: adds canaries to all functions that have local arrays with a size of at least eight bytes, including all usages of the *alloca* function.
- `-fstack-protector-strong`: adds canaries to all functions containing local arrays, or when the address of a local variable is taken.
- `-fstack-protector-all`: adds canaries to all functions.

The specific attack to disable the canary protection will not work if all functions are protected by canaries, for a reason that will be discussed later in detail. The attack itself is divided into two parts, illustrated in Figure 4. More details are to be found in Sections 4.2.1 and 4.2.2.

- First part: where a register value is leaked to gain information about the attacked application, and
- Second part: where the reference canary value is manipulated to disable the Stack Canary protection.

Both parts are carried out using the same vulnerability. The main idea is that the leaked register information is used to overwrite the reference canary value. The attacker can use the leaked information to place the new canary value in the attack payload at appropriate places. With the ability to overwrite the reference canary value, there is no need to leak the original reference canary value. It also defeats terminator canaries as the terminator characters are replaced by other values.

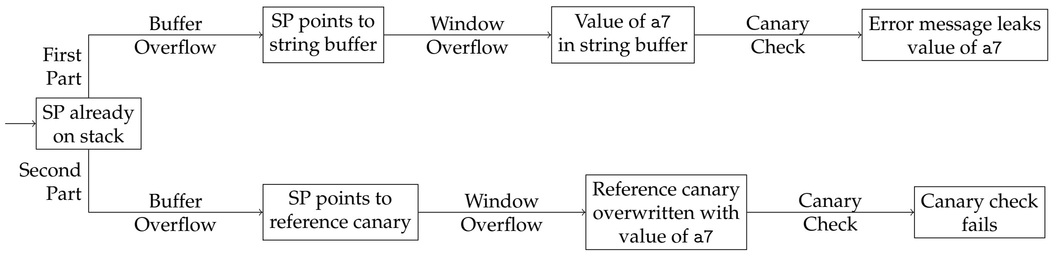
First Part
Buffer Overflow
SP points to string buffer
Window Overflow
Value of a7 in string buffer
Canary Check
Error message leaks value of a7
SP already on stack
Second Part
Buffer Overflow
SP points to reference canary
Window Overflow
Reference canary overwritten with value of a7
Canary Check
Canary check fails

Figure 4. Schematic of the window overflow attack to overcome Stack Canary protection.

4.2.1. Leak Application Information

The first step is mainly taken to gather information about the system. If the attacker has other means to gain all or some of this information, i.e., reverse engineering or debugging, this step can be (partly) skipped. Information such as the position of the reference canary word may be easier to acquire with different methods. Therefore, the explanation will focus on how to leak register values that are needed for the second step.

Figure 5 shows how ESA *i* must be placed for the attacker to leak register a7. The reason to only leak one register is to minimize the chance of overwriting any of the characters with a terminator character as well as leaving the '\n' character at the end of the string intact to make sure the output buffer is flushed before the program terminates. Knowing the value of a single register is also sufficient to carry out the next step.

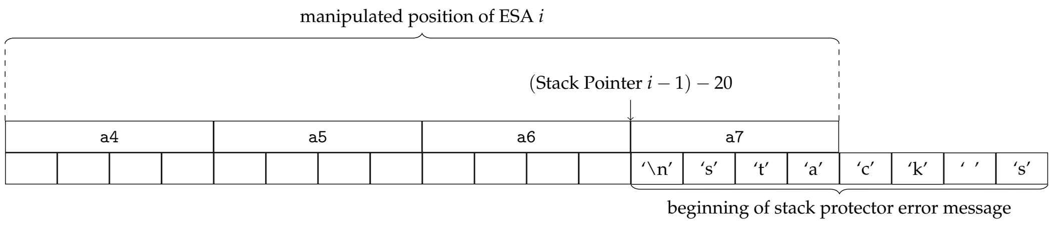
manipulated position of ESA *i*
(Stack Pointer *i* − 1) − 20
a4
a5
a6
a7
'\n'
's'
't'
'a'
'c'
'k'
' '
's'
beginning of stack protector error message

Figure 5. Using the manipulated stack pointer to overlap ESA *i* with the error message of the stack protector implementation.

A part of the execution sequence, based on the code snipped in Listing 1, is depicted in Figure 6. After *buffer* was overflown using the *strcpy* function, *compute* is called, which triggers a window overflow. This is where the error message string is overwritten. Only when *vulnerable* is about to return is the stack buffer overflow detected by checking the overwritten canary value. The *__stack_ckh_fail* function is called, which prints the manipulated error message and leaks the value of a7 accordingly.

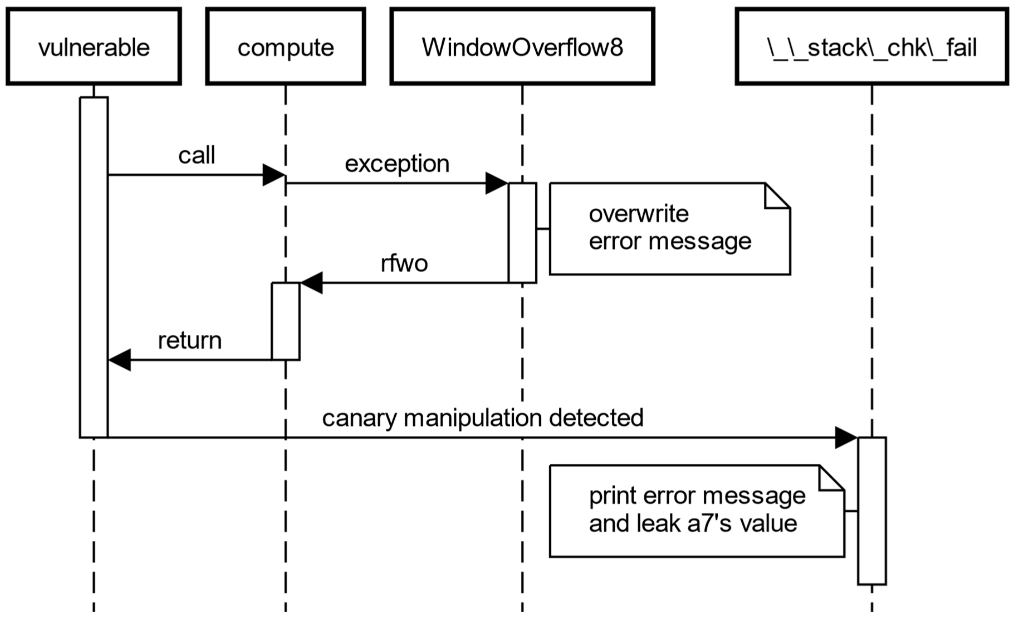

Figure 6. Execution sequence of a vulnerable program to leak a register value [7].

4.2.2. Overwrite the Reference Canary Value

This pattern of attack is very similar. Instead of writing the value of register a7 into a string buffer, we now target the reference canary value. As a result, the reference canary value is overwritten with the value of a7, which was leaked in the previous step. To bypass the canary check, the leaked value just needs to be placed at the correct position in the payload, and the check will pass as both canary values are now containing the leaked value of a7.

4.2.3. Limitations

To make this attack work, some requirements need to be fulfilled. The Stack Canary implementation itself must print a static string that can be overwritten. This condition is met by several implementations [28]. The value of register `a7` should not contain any terminator characters, and the positions of the error message string and the reference canary value must be known or be obtained by brute-forcing it. To ensure the correct prediction of the value of register `a7`, step one of the attacks should be performed multiple times to make sure that either

- the value stays constant throughout multiple runs, or
- directly depends on the payload the attacker sends,
- or neither of the above.

Depending on the result, the attack may become easier or impossible. In a best-case scenario, the register values depend directly on the input, which makes more elaborate attacks easier, i.e., writing specific values to memory-mapped registers, influencing the device configuration. If the register value does not depend on the input, it may stay constant, in which it may contain a terminator character that stops the attack, following a recognizable pattern such as being used for counting incoming messages, making the attack more complicated but not impossible, or is completely random, which also breaks the attack.

It is also possible that *compute* does not trigger an overflow directly (because *vulnerable* called another function such as *strcpy* before) and further function calls inside *compute* are necessary. In this case, it is critical that *compute* is not protected by canaries itself as it would load the old canary value to the stack, but would use the manipulated reference value for the check. As *compute*'s stack frame is located at a lower address than *vulnerable*'s stack frame, it is not possible to manipulate its canary value. This condition is met when using the `-fstack-protector` flag, which is the most common, but not when using the `-fstack-protector-all` flag, which adds canary checks to all functions.

This attack has an advantage against terminator canaries. As the value of the canary is actively changed, removing the terminator characters, stack buffer overflows caused by functions such as *strcpy* can be exploited, which is not possible by simply leaking the reference canary value.

4.3. MPU and MMU Protection

It is common practice to isolate memory regions that contain critical data from the rest of the application to prevent the data from being tampered with. In a deeply embedded environment, this might be even worse, as it is possible that there is no separation between user- or kernel-space, or separation between different tasks. Although there are pure-software solutions, such as Software Fault Isolation (SFI) [29], that can be used for that purpose, solutions that rely on hardware support are often preferred due to lower overheads.

The ESP32 offers an MPU as well as an MMU, depending on the memory region that should be protected [30]. Both protection units control access based on eight different Process Identifiers (PIDs). PIDs 0 and 1 act as privileged PIDs, with access to all physical addresses, and are not affected by the MMU memory mappings. For the other PIDs, the MPU blocks all memory access and the MMU allows the creation of custom memory mappings for each PID. Additionally, the PID controller provides a way to switch to PID 0 on an interrupt. Switching to another PID can be done manually when currently having a privileged PID.

There are two problems with this feature set. The limit of only six different user-level PIDs is very restrictive, even for an embedded system. This makes it complicated to implement memory-protection features such as the FreeRTOS MPU Support [31], which assigns a different PID to each task. It can be used to realize a simple kernel–user–space separation, as done in Apache NuttX. The problem is that this implementation switches to the kernel mode for window overflow and underflow handlers, and defines hooks that switch back to the previous PID at the and of the handler [32]. As a result, all memory access, including the vulnerable one, is done with PID 0, breaking the memory protection for the window-overflow vulnerability.

4.4. Applicability and Conflict with Return-Oriented Programming

Just bypassing the stack buffer overflow detection is most likely not the goal of an attacker. Therefore, we investigated how the exploit can be further developed into manipulating the control flow, in particular a ROP attack.

Without going into too many details on how ROP works on Xtensa's windowed ABI, it is important to know that window underflow exceptions are used to overwrite the return address and the stack pointer of register values that have been placed onto the stack before. It is important to control the stack pointer to be able to continue the gadget chain, as all registers are loaded with offsets to the stack pointer. This gives us the following order of events:

1. an overflow exception that saves the return address and stack pointer to the stack,
2. a buffer overflow that overwrites these values, and finally
3. an underflow exception that loads the manipulated values back into the register set.

On the other hand, we have the following order of events to execute the second step to bypass the canary protection:

1. a buffer overflow that overwrites a stack pointer, so it points to the location of the canary value, and
2. an overflow exception that loads this stack pointer and overwrites the canary value.

This gives us two overflow exceptions that need to happen, i.e., one before and one after the buffer overflow. This also means that two corresponding underflow exceptions are occurring in reverse order. That means the first underflow exception restores the registers of the overflow exception that happens after the buffer overflow. In conclusion, the register values cannot be manipulated with the buffer overflow itself.

To make matters worse, if we want to use the second underflow exception to load a gadget address and stack pointer, it becomes apparent that the position for the stack pointer is already occupied in the payload by the pointer to the canary value. That is because we manipulated the stack pointer that was loaded by backtracking the stack one frame, which is the frame that was stored by the first overflow exception.

In theory, we need a third underflow exception that loads the manipulated values; however, with the second underflow exception loading the pointer to the location of the reference canary value as the stack pointer, the position of the stack is now at a completely different position and will most likely crash the program when we try to continue to use it as a stack. Due to this fact, ROP attacks are very unlikely to be executed successfully, although they are not impossible.

To be able to execute a ROP attack after bypassing the canary value, we need to add more requirements to the attacked program, which decreases the chances that a program can be attacked in this way. The problem is the stack pointer that points outside of the stack segment, so we must either overwrite this value again, which would require another buffer overflow, or we need to skip the underflow exception that loads this value into a register. This can be achieved by mechanisms that subvert the regular control flow, such as exception handling in C++ or *setjmp*/*longjmp* in C.

This is the case when we assume the attacker has capabilities explained in the extended attack model. Being able to write to the stack multiple times, i.e., by triggering multiple buffer overflows, would allow it to manipulate another stack pointer after it was written to the BSA by a window-overflow exception that already executed the exploit, to chain an arbitrary number of random-access writes together. It would also make ROP attacks easier to execute for the same reason. This is also a more realistic scenario as it would only require two memory corruptions, the first before a window overflow for the overflow exploit, and one afterwards to fix the manipulated stack pointer and inject the ROP payload.

Besides ROP, other techniques would benefit from disabled Stack Canaries, e.g., Jump-Oriented Programming (JOP). As JOP relies mainly on indirect jumps, it should not be affected by the overflow/underflow mechanism. However, to the best of our knowledge, JOP was not researched for the Xtensa architecture, and remains an open topic.

5. Countermeasures

The vulnerability is specific to the LX version of the Xtensa architecture. In the newer version, Xtensa NX, the number of different call instructions is reduced and only a window rotation of eight registers is supported. This also results in a simplification of the window overflow and underflow exception handlers, as there is no longer a separation into BSA and ESA. No longer having an ESA also eliminates the need for stack backtracking, and with that closes the vulnerability. However, current microcontrollers are still using the Xtensa LX architecture and are probably staying in the field inside IoT devices for many years before being replaced. Therefore, efficient countermeasures are still relevant.

With the Xtensa NX architecture, there is already a version of the Xtensa architecture that only uses a window shift with a single size of eight and only uses a single register save area without backtracking. Taking this as an inspiration and the fact that compilers exclusively use the window shift of eight for code generation, a fix could be to apply the fixed-window implementation of the Xtensa NX architecture to Xtensa LX cores. More difficult, probably, is the adoption of low-level firmware code. Switching to a fixed-window implementation could require lots of adoptions instead of just simply changing the used function call instructions. We therefore cannot fully estimate the necessary work that this change would require.

The problem of this vulnerability is not that it is hard to detect, but that it uses a new kind of pointer before existing countermeasures can take effect. Although there are general working countermeasures such as SFI, a more targeted solution with minimal overheads is more desirable. In this section, we present countermeasures to detect an exploitation of the vulnerability.

5.1. Plausibility Check

The basic idea is to use the inherent property of the pointer being used for backtracking the stack. As a result, we know that the stack frame this pointer points to must be at a higher address than the current stack pointer. Since the attacker is already able to manipulate memory at higher addresses using the buffer overflow itself, we believe this attack is most likely to be used to affect lower addresses. We will later also discuss possible scenarios where this assumption holds not to be true.

With this assumption, however, we can create a simple test inside the relevant window-overflow exception handlers to verify if a backtracking pointer is being tampered with. Listing 2 shows the `_WindowOverflow8` exception handler before the changes on the left side and after the changes on the right side. The implementation for the `_WindowOverflow12` handler looks equivalent and the `_WindowOverflow4` handler is not changed as there is no stack backtracking. There are three relevant pointers: `a1` and `a9` contain stack pointers of adjacent windows, and line 2 loads a third stack pointer into register `a0`. We use one of the already-present stack pointers to check if the recently loaded stack pointer in register `a0` is located at a higher address. We selected register `a9`, but register `a1` would work just as well.

In case we detect manipulation of the value in register `a0`, we force the program to crash by setting the value of `a0` to 32. This results in the instruction in line 10 accessing address null. This is enough for demonstration purposes, but a cleaner implementation would probably invoke an error handler instead of simply crashing the program. At this point, it suffices to say that invoking an error handler within the interrupt context, especially with a manipulated stack and register values, comes with its own complications, which are discussed in broader detail in [24].

Listing 2. Overflow exception handler before (left) and after (right) adding a check for the loaded pointer [7].

```
_WindowOverflow8:
    s32e    a0, a9, -16
    l32e    a0, a1, -12



    s32e    a1, a9, -12
    s32e    a2, a9,  -8
    s32e    a3, a9,  -4
    s32e    a4, a0, -32
    s32e    a5, a0, -28
    s32e    a6, a0, -24
    s32e    a7, a0, -20
    rfwo
```

```
_WindowOverflow8:
    s32e    a0, a9, -16
    l32e    a0, a1, -12
    bgeu    a0, a9, _L
    movi.n  a0, 32
_L:
    s32e    a1, a9, -12
    s32e    a2, a9,  -8
    s32e    a3, a9,  -4
    s32e    a4, a0, -32
    s32e    a5, a0, -28
    s32e    a6, a0, -24
    s32e    a7, a0, -20
    rfwo
```

5.2. Breaking with the Assumption

The core assumption for this protection mechanism is that the attacker uses the vulnerability to access addresses that are located at lower addresses than the stack, as higher addresses could be reached with the stack buffer overflow anyway. That said, in an embedded environment, an attacker might have reasons to reach for higher addresses instead of lower ones. In a multi-task scenario, one task that contains the stack buffer overflow vulnerability could be used purely to modify values on another task's stack, which is located at a higher address. The reason could be that stack buffer overflows can be sufficiently detected so the task is being shut down. Of course, if this leads to the whole device being restarted, this is not an option. It can be assumed that access to other stacks is possible, either due to the lack of memory-protection mechanisms or the fact that they can be circumvented, as described in Section 4.3.

To account for the cases, the proposed check could be extended to also check for an upper bound of the loaded stack pointer. This, however, raises some problems. The lower bound can be defined by another stack pointer, but there is no stack pointer available for a frame that is located at a higher address. Additionally, since all tasks share the same exception handlers, it is not possible to use a fixed address as an upper bound. One possibility is to use the largest stack size in the application as a bound. This would be an upper bound of how much the loaded stack pointer would be allowed to increment compared to the previous stack pointer. As this would only be a rough estimation, it would still be possible for an attacker to reach past a stack frame. To make sure no memory corruption is possible outside of the affected stack, a reasonable amount of memory must be used as a buffer between the stack and the next memory region. In the case of only one large stack frame in the entire program, this could lead to huge amounts of unusable memory.

6. Evaluation

We evaluated our countermeasure using the simulation of an Xtensa LX7 core with 32 physical registers as well as an ESP32-WROVER-E development board with two Xtensa LX6 cores with 64 physical registers. The simulation results were part of a previous publication [7].

6.1. Simulation Results

For simulations, we used a minimalistic Xtensa configuration with 32 physical registers. We profiled our implementation with a cycle-accurate instruction-set simulator by Cadence. As a benchmark, we selected CoreMark [33], a CPU benchmark for embedded systems. We did not use the reported performance numbers of the benchmark itself but instead, we used the results of the profiling tool, as we just wanted to have a complex-enough application that represents typical tasks of an embedded system and makes use of the greater power of the profiler.

The application was compiled with the xt-clang and the xcc compilers, and the test was run three times with different optimization levels, as optimization techniques such as inline functions can heavily affect the number of window exception handler calls. The programs were compiled with

- -O0 for no optimization,
- -Os for size optimization, which is commonly used for embedded devices to create the smallest binary possible, and
- -O3 to enable all optimizations to increase performance, even if it increases the code size (i.e., inline function).

The results for the two compilers were very different; for xt-clang, they are shown in Table 1 and for xcc, in Table 2. The top half shows the number of different window overflows that occurred, and the bottom half displays the base number of clock cycles without the protection mechanism, as well as the absolute and relative overhead introduced by it. Although the overheads for the exception handler itself stay almost constant at around 36.3%, the overall overheads for the application greatly vary from 0.024–0.036% for xt-clang, and up to 0.104–0.423% for xcc, depending on the level of optimization.

These large differences can be explained by looking at the differences in the number of window overflows for the different configurations. Although xcc shows great improvements with higher optimization levels, the number of window overflows (and, with that, also the absolute overhead) stays almost constant. In contrast, xt-clang heavily decreases the number of overflows with higher optimization levels (e.g., by inlining functions) resulting in smaller absolute overhead.

Table 1. Simulation results for 10 iterations of CoreMark compiled with xt-clang measured in clock cycles [7].

	-O0	**-Os**	**-O3**
Number of exception handler calls			
_WindowOverflow4	3	3	3
_WindowOverflow8	1104	453	288
_WindowOverflow12	0	0	0
Base (+ protection overhead) clock cycles			
Total	17,359,869 +4230 +0.024%	5,025,383 +1807 +0.036%	4,522,967 +1139 +0.025%
_WindowOverflow8	12,188 +4416 +36.232%	4987 +1812 +36.334%	3172 +1152 +36.318%

Table 2. Simulation results for 10 iterations of CoreMark compiled with xcc measured in clock cycles [7].

	-O0	**-Os**	**-O3**
Number of exception handler calls			
_WindowOverflow4	3	3	3
_WindowOverflow8	3929	3924	3920
_WindowOverflow12	0	0	0
Base (+ protection overhead) clock cycles			
Total	15,015,694 +15,672 +0.104%	5,313,717 +16,022 +0.302%	3,726,070 +15,752 +0.423%
_WindowOverflow8	43,224 +15,716 +36.359%	43,168 +15,696 +36.360%	43,124 +15,680 +36.360%

6.2. ESP32 Results

We repeated the benchmark on a recent microcontroller using the ESP32-WROVER-E development board based on the ESP-IDF Software Development Kit (SDK). The test cases are slightly different, as the SDK currently only supports the gcc compiler. Additionally, the optimization flags that are available in the configuration are sightly different. It supports -Og, which allows some optimizations if debugging is not affected, and -O2, which also optimizes for performance but without having a negative impact on code size such as with -O3. Table 3 shows the results for the different optimization flags. In these cases, we used the reported results of the benchmark, measured with a high-resolution timer with a microsecond resolution. We also increased the number of iterations from 10 to 100 to have more precise results. Measurement results between two identical experiments may vary within the range of one microsecond.

Naturally, the evaluation shown in Table 3 also misses the other information we received from the profiling tool of the simulator, such as the exact number of window-overflow exceptions and their execution time. We see, however, a drastically reduced overhead compared to the simulations. Even the largest overhead of 0.0065% is drastically lower than the best case for the simulation with 0.024%. when only comparing results from the same compiler family, the differences are even larger, with 0.104% being the best simulation result. The big difference between the simulated system and the microcontroller is the number of physical registers. This is 32 for the simulation, but 64 for the microcontroller. We assume that this drastically decreases the number of window exceptions during the benchmark. We therefore tried to use a debugger to count the number of window-overflow exceptions, but had crashes for most configurations, which we think has to do with using breakpoints in the interrupt context. Our only successful run was with the -Og flag, where we counted 105 window overflows for 10 iterations. As this flag generated our highest absolute overheads, we assume that the number of window overflows is even lower for the other flags.

Table 3. CoreMark results on an ESP32-WROVER-E development board with 100 iterations in µs.

	-O0	**-Og**	**-Os**	**-O2**
Base value	1,364,352	534,468	432,150	318,670
Absolute overhead	+28	+35	+24	+9
Relative overhead	+0.0021%	+0.0065%	+0.0056%	+0.0028%

6.3. Synthetic Benchmark

As the very low overheads might suggest, a benchmark may be a very beneficial use case for the evaluation of our countermeasure. Benchmarks are more computationally heavy, instead of complex or deep, when it comes to the control flow. We therefore designed a synthetic program that represents a worst-case scenario for the control flow. This, of course, depends on the number of functions being called in relation to the actual work being done inside the functions. However, just calling a lot of functions, i.e., in a loop, would not create many window overflows, as, with every returning function, the window moves back again. To get the most window exceptions, the window must be moved in the same direction as long as possible. All other function calls will only increase the runtime of the benchmark, but not result in additional overheads generated by our protection mechanism.

We therefore designed a benchmark that consists of a very simple control flow. Its structure is depicted in Figure 7. Functions only contain the minimum of instructions that are required to maintain the control flow. These are `entry` instructions which rotate the window at the beginning of the function, `call8` instructions to call the next function, and `ret.w` to rotate the window back and return to the previous function. The call depth can be configured. Additionally, we use `nop.n` operations to simulate an artificial workload. (Although we previously associated window rotations with function calls, the actual rotation is not performed by a `call` instruction, but by an `entry` instruction, which is also responsible for stack space allocation.) We choose these instructions as they have no side effects. The whole benchmark contains no conditional jumps, except for the loop that is used to execute it multiple times.

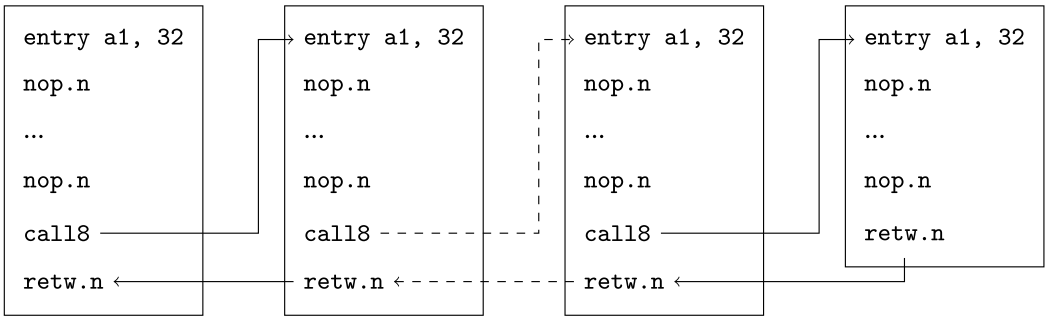

Figure 7. Structure of the synthetic benchmark.

We ran the benchmark for 1000 iterations each with different configurations. We changed the call depth as well as the number of `nop.n` instructions inside each function. The results are shown in Figure 8. Each point represents a run of the benchmark with the specified call depth on the x-axis. The number of `nop.n` instructions inside each function is marked in the legend. For each set of benchmarks, we run it with and without our protection mechanism. These sets share the same mark, but use different colors, specifically red for the unprotected and blue for the protected runs. We altered the transparency for every other set of graphs for better distinguishability. The call depth ranges from 1 to 30, and we selected 5, 15, 25, 35, and 45 `nop.n` instructions. More measurements for different numbers of `nop.n` instructions in between were made but not included as they would decrease distinguishability and fall within the expected results anyway. Figure 9 shows all our results in a single graph for the sake of completion, but we will discuss the details based on Figure 8. It shows the regularity of our results and how benchmark results would look for different parameters.

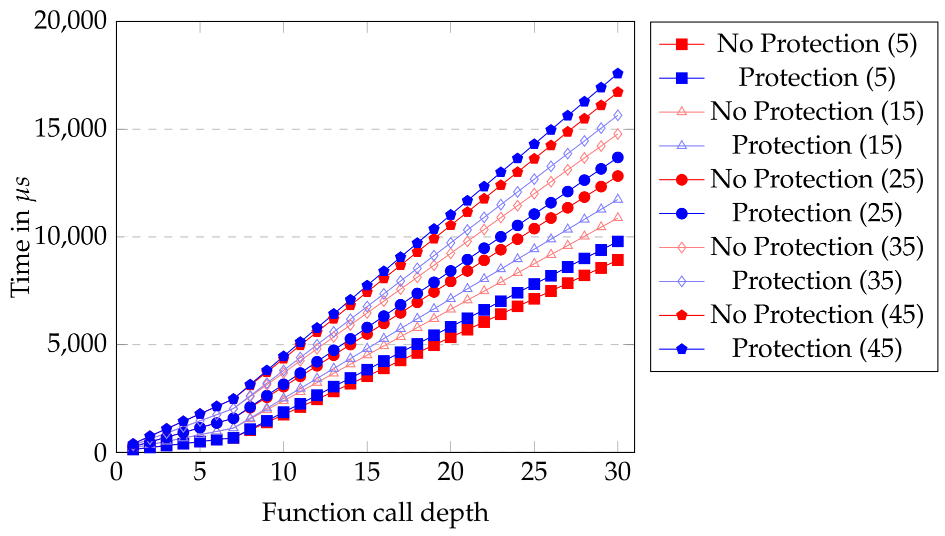

Figure 8. Selected synthetic benchmark runtimes in µs based on the call depth and the number of `nop.n` instructions in the function (noted in parenthesis) with and without the window-overflow protection.

The different measurement sets in Figure 8 show the effect of the window mechanism. There is a noticeable increase in every graph at a call depth of eight. This directly correlates with our expectations, where the effect of the window overflow and underflow exceptions should be noticeable, as there are 64 physical registers, and each call (`entry`) shifts the window by 8. This also shows the effects of the exception handlers as a whole very well. With greater call depths, we can observe a growing gap between the protected and unprotected runs. The actual work done in the program (represented by the `nop.n` operations) does not influence the absolute overhead generated by our protection mechanism, but on the relative overhead. This was to be expected. This can also be observed in Figure 10, which shows the measured difference between each protected and unprotected run. It also shows that once the window exceptions start to take effect, the overhead increases nearly linearly with around 36–38 µs per additional function call.

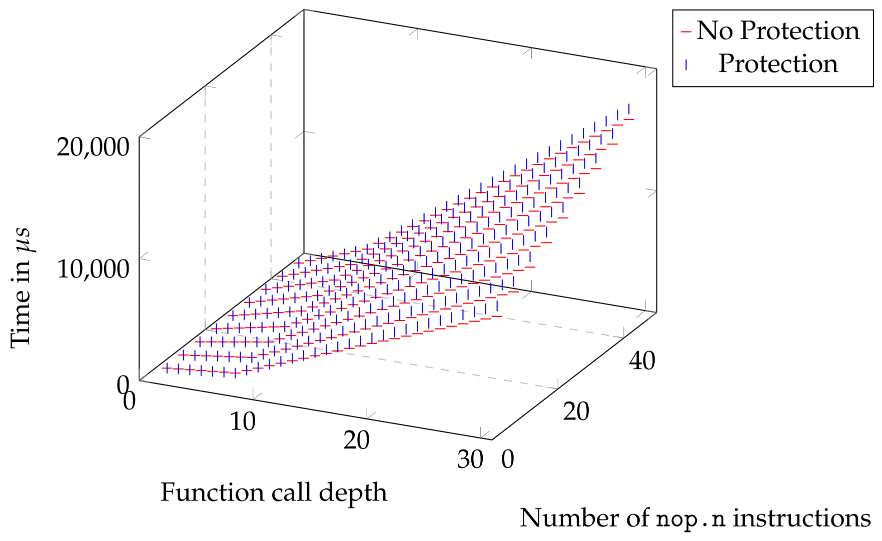

Figure 9. Synthetic benchmark runtimes in μs based on the call depth and the number of `nop.n` instructions in the function with and without the window-overflow protection

Consequently, the highest measured overhead occurs with the least number of instructions executed in between window overflows. With only 5 `nop.n` instructions, the absolute overhead with a call depth of 8 was 39 µs, and for a call depth of 30, it was 864 µs, resulting in a relative overhead of 3.77% or 9.68%, respectively. This is still a very acceptable outcome for an artificially constructed worst-case scenario. Of course, the overhead could be increased even further with a larger call depth, but we argue that a call depth of 30 is already very deep for an embedded system, occupying 960 bytes of stack space for register save areas alone.

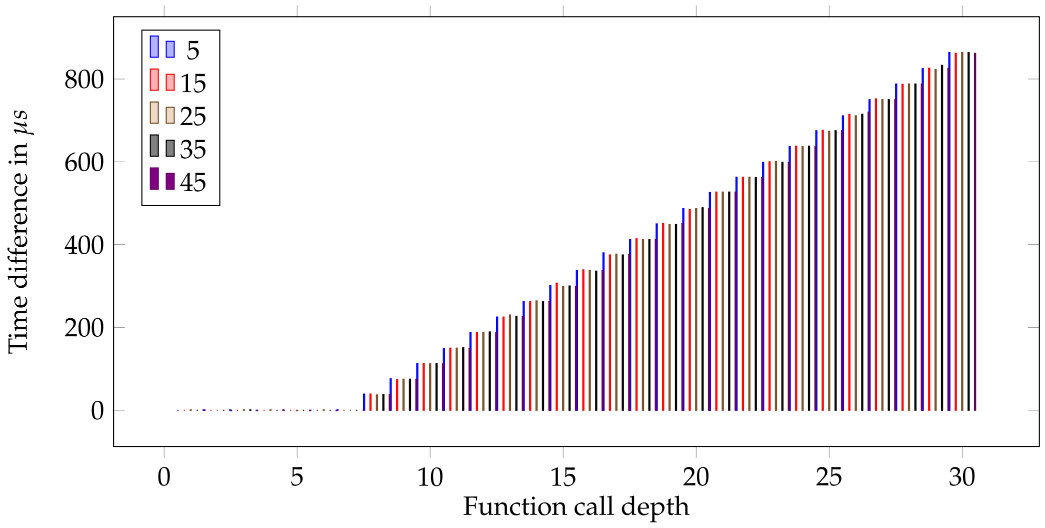

Figure 10. Runtime difference between protected and unprotected benchmark runs.

Just by looking at an average of 45 instructions per function, the overhead already decreases to 1.22% for a call depth of 8 and 5.15% for a call depth of 30. When we assume that the average number of instructions in each function is much higher (considering effects such as loops, inlining of smaller functions, etc.) and the control flow of the program normally is not as straightforward as in the worst-case scenario, we can assume that multiple hundreds and even thousands of instructions are executed between each window overflow.

If we assume a call depth of 8 (which corresponds exactly to one window overflow per benchmark iteration), we can extrapolate the existing data to see that we reach the 1% threshold at around 60 `nop.n` instructions, or 60 × 8 = 480 for the whole benchmark. To cross the 0.1% overhead, we already are at around 750 instructions per function. To get at least near the results of the CoreMark benchmark, we would need to increase the number of instructions per function to around 4850.

7. Conclusions

This paper extended the contents of [7]. This previous paper introduced a new vulnerability that allows the attacker to use an unprotected stack pointer to manipulate arbitrary memory addresses. The paper demonstrated how this vulnerability can be used to disable Stack Canaries and even discussed possibilities to combine it with other attack techniques such as ROP. It introduced a low-cost software patch for this vulnerability and used simulations in combination with the CoreMark benchmark to evaluate its overhead. The simulated Xtensa core only had 32 general-purpose registers, and the overhead ranged from 0.104% to 0.423%. It also showed the overhead for the exception handlers only, which increased by 36.60%. These results were also included in this paper.

This paper extended the content using an ESP32-WROVER-E development board for evaluation instead of simulations. This board uses two Xtensa LX6 cores, each with 64 general-purpose registers. We believe this difference led to significant differences in our measured results. The overhead of the patch measured for the microcontroller was between 0.0021% and 0.0065%. Additionally, a new benchmark was developed to showcase the worst-case scenario. It was designed to maximize the occurrence of window overflows, but can also scale and represent larger programs. In the worst case, we showed that we were able to reach 9.68% overhead, which is still far below the 36.60% measured in the simulation. Limits of the attack as well as defense mechanisms were discussed in detail.

In conclusion, the presented vulnerability can be a threat to systems that may have a buffer overflow vulnerability and insufficient protection. Our software patch should be easy to apply to existing systems with negligible consequences for the system performance. It is also easy to integrate into an existing code base to protect systems. In the future, the vulnerability will hopefully be resolved by replacing vulnerable devices either with devices that are patched, or with devices using the next generation of Xtensa cores that no longer contain the vulnerability.

Author Contributions: Conceptualization, K.L. and P.L.; Data curation, K.L.; Funding acquisition, P.L.; Methodology, K.L.; Project administration, P.L.; Software, K.L.; Supervision, P.L.; Visualization, K.L.; Writing—original draft, K.L.; Writing—review and editing, P.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Federal Ministry of Education and Research (BMBF) under research grant number 01IS18065E. The publication of this article was funded by the Open Access Fund of the Leibniz Association.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ABI	Application Binary Interface
AoI	Area of Interest
ASLR	Address Space Layout Randomization
BSA	Base Save Area
CFG	Control Flow Graph
CFI	Control Flow Integrity
ESA	Extra Save Area
IoT	Internet of Things
JOP	Jump-Oriented Programming
MMU	Memory Management Unit
MPU	Memory-Protection Unit
PID	Process Identifier
ROP	Return-Oriented Programming
SDK	Software Development Kit
SFI	Software Fault Isolation

References

1. Cadence Design Systems, Inc. *Xtensa Instruction Set Architecture (ISA) Reference Manual*; Cadence Design Systems, Inc.: San Jose, CA, USA, 2019.
2. Cadence Design Systems, Inc. *Xtensa Microprocessor Programmer's Guide*; Cadence Design Systems, Inc.: San Jose, CA, USA, 2018.
3. Lehniger, K.; Aftowicz, M.J.; Langendorfer, P.; Dyka, Z. Challenges of Return-Oriented-Programming on the Xtensa Hardware Architecture. In Proceedings of the Euromicro Conference on Digital System Design, Kranj, Slovenia, 26–28 August 2020; Trost, A., Žemva, A., Skavhaug, A., Eds.; IEEE: Piscataway, NJ, USA, 2020. https://doi.org/10.1109/dsd51259.2020.00034.
4. Amatov, B.; Lehniger, K.; Langendörfer, P. Return-Oriented Programming Gadget Catalog for the Xtensa Architecture. In Proceedings of the 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops), Pisa, Italy, 21–25 March 2022; pp. 655–660. https://doi.org/10.1109/PerComWorkshops53856.2022.9767489.
5. Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 2 November–31 October 2007; pp. 552–561.
6. Perry, W.; Crispin, C. Simple Stack Smash Protection for GCC. In Proceedings of the GCC Developers Summit, Ottawa, ON, Canada, 25–27 May 2003; pp. 243–255.

7. Lehniger, K.; Langendörfer, P. Through the Window: On the exploitability of Xtensa's Register Window Overflow. In Proceedings of the 2022 32nd International Telecommunication Networks and Applications Conference (ITNAC), Wellington, New Zealand, 30 November–2 December 2022; pp. 353–358.
8. Aleph One. Smashing the stack for fun and profit. *Phrack Mag.* **1996**, *7*, 14–16.
9. PaX Team. Pax Address Space Layout Randomization (aslr). Available online: https://pax.grsecurity.net/docs/aslr.txt (accessed on 3 May 2023).
10. Abadi, M.; Budiu, M.; Erlingsson, Ú.; Ligatti, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM Transactions on Information and System Security (TISSEC)*; ACM Digital Library: New York, NY, USA, 2009; Volume 13, pp. 1–40.
11. Li, J.; Wang, Z.; Bletsch, T.; Srinivasan, D.; Grace, M.; Jiang, X. Comprehensive and Efficient Protection of Kernel Control Data. *IEEE Trans. Inf. Forensics Secur.* **2011**, *6*, 1404–1417. https://doi.org/10.1109/tifs.2011.2159712.
12. Fratrić, I. *ROPGuard: Runtime Prevention of Return-Oriented Programming Attacks: Technical Report*; IEEE: Piscataway, NJ, USA, 2012.
13. Cheng, Y.; Zhou, Z.; Miao, Y.; Ding, X.; Deng, R.H. *ROPecker: A Generic and Practical Approach for Defending against ROP Attack*; Internet Society: Reston, VA, USA, 2014.
14. Clercq, R.d.; Verbauwhede, I. A survey of Hardware-based Control Flow Integrity (CFI). *arXiv* **2017**, arXiv:1706.07257, 2017.
15. Li, J.; Chen, L.; Xu, Q.; Tian, L.; Shi, G.; Chen, K.; Meng, D. *Zipper Stack: Shadow Stacks Without Shadow*; Springer: Cham, Switzerland, 2020; pp. 338–358. https://doi.org/10.1007/978-3-030-58951-6{_}17.
16. Liljestrand, H.; Nyman, T.; Gunn, L.J.; Ekberg, J.; Asokan, N. *PACStack: An Authenticated Call Stack;* USENIX Association: Berkeley, CA, USA, 2021.
17. Ravichandran, J.; Na, W.T.; Lang, J.; Yan, M. PACMAN. In Proceedings of the 49th Annual International Symposium on Computer Architecture, New York, NY, USA, 18–22 June 2022; Salapura, V., Ed.; ACM Digital Library: New York, NY, USA, 2022; pp. 685–698. https://doi.org/10.1145/3470496.3527429.
18. Cowan, C.; Pu, C.; Maier, D.; Walpole, J.; Bakke, P.; Beattie, S.; Grier, A.; Wagle, P.; Zhang, Q.; Hinton, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the USENIX Security Symposium, San Antonio, TX, USA, 26–29 January 1998; Volume 98, pp. 63–78.
19. Cowan, C.; Wagle, F.; Calton Pu.; Beattie, S.; Walpole, J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In Proceedings of the Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00, Hilton Head, SC, USA, 25–27 January 2000. https://doi.org/10.1109/discex.2000.821514.
20. Bittau, A.; Belay, A.; Mashtizadeh, A.; Mazieres, D.; Boneh, D. Hacking Blind. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; pp. 227–242. https://doi.org/10.1109/SP.2014.22.
21. Wang, Z.; Ding, X.; Pang, C.; Guo, J.; Zhu, J.; Mao, B. To Detect Stack Buffer Overflow with Polymorphic Canaries. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg, 25–28 June 2018. https://doi.org/10.1109/dsn.2018.00035.
22. Marco-Gisbert, H.; Ripoll, I. Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers. In Proceedings of the 2013 IEEE 12th International Symposium on Network Computing and Applications, Cambridge, MA, USA, 22–24 August 2013; pp. 243–250. https://doi.org/10.1109/NCA.2013.12.
23. Frantzen, M.; Shuey, M. *StackGhost: Hardware Facilitated Stack Protection*; USENIX Association: Berkeley, CA, USA, 2001.
24. Lehniger, K.; Langendorfer, P. Window Canaries: Re-thinking Stack Canaries for Architectures with Register Windows. *IEEE Trans. Dependable Secur. Comput.* **2022**, 1–11. https://doi.org/10.1109/TDSC.2022.3230748.
25. Burow, N.; Zhang, X.; Payer, M. SoK: Shining Light on Shadow Stacks. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 985–999. https://doi.org/10.1109/SP.2019.00076.
26. Instrumentation Options (Using the GNU Compiler Collection (GCC)), 2021-07-28T07:17:14.000Z.
27. Clang Command Line Argument Reference—Clang 15.0.0git Documentation, 2022–03-08T06:03:13.000Z.
28. Bierbaumer, B.; Kirsch, J.; Kittel, T.; Francillon, A.; Zarras, A. Smashing the Stack Protector for Fun and Profit. In *ICT Systems Security and Privacy Protection*; IFIP Advances in Information and Communication Technology; Janczewski, L., Kutyłowski, M., Eds.; Springer: Cham, Switzerland, 2018; Volume 529, pp. 293–306. https://doi.org/10.1007/978-3-319-99828-2.
29. Wahbe, R.; Lucco, S.; Anderson, T.E.; Graham, S.L. Efficient software-based fault isolation. In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principes, Asheville, NC, USA, 5–8 December 1993; Operating Systems Review; Black, A.P., Liskov, B., Eds.; Association for Computing Machinery: New York, NY, USA, 1993; pp. 203–216.
30. Espressif Ssytems. *ESP32 Technical Reference Manual*; Version 4.8; Espressif Ssytems: Shanghai, China, 2022.
31. FreeRTOS-MPU—Memory Protection Unit Support in FreeRTOS. Available online: https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html (accessed on 12 April 2023).
32. GitHub Apache/Nuttx esp32_Window_Hooks.s. Available online: https://github.com/apache/nuttx/blob/master/arch/xtensa/src/esp32/esp32_window_hooks.S (accessed on 12 April 2023).
33. EEMBC. CoreMark. Available online: https://github.com/eembc/coremark (accessed on 8 March 2022).