In-Vivo Stack Overflow Detection and Stack Size Estimation for Low-End Multithreaded Operating Systems using Virtual Prototypes

Sören Tempel[1] Vladimir Herdt[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
tempel@uni-bremen.de, vherdt@uni-bremen.de, drechsler@uni-bremen.de

***Abstract*—Constrained IoT devices with limited computing resources are on the rise. They utilize low-end multithreaded operating systems (e.g. RIOT) where each thread is assigned a fixed stack size during the development process. In this regard, it is important to choose an appropriate stack size which does not cause stack overflows and at the same time does not waste scarce memory resources by overestimating the required thread stack size.**

In this paper we propose an in-vivo technique for stack overflow detection and stack size estimation that leverages *Virtual Prototypes* (VPs) and is specifically tailored for low-end multithreaded IoT operating systems. We focus on SystemC-based VPs which operate on the TLM abstraction level. VPs are an industrial proven modeling standard to enable early software development and testing. We propose a non-intrusive extension for existing VPs which allows detecting stack overflows and provides a stack size estimation, which is beneficial to a VP-based development process. Our analysis works in-vivo, hence no modification of the executed software binary is required between testing and deployment. Our evaluation using the RIOT operating system revealed two previously unknown stack overflows in RIOT and identified potential stack size overestimation.

***Index Terms*—Memory Usage, Constrained Devices, Virtual Prototypes, RISC-V, RIOT, Stack Overflows, IoT**

I. Introduction

With the rise of the *Internet of Things* (IoT), heavily constrained devices with limited computing resources are becoming increasingly popular. As per RFC 7228 [1], these limitations exist to reduce the production cost and/or the physical size of these devices [1, Section 2.1]. A common example of a limited resource on constrained devices is available memory. While conventional devices (e.g. laptops, desktops, or servers) have several gigabytes of memory at their disposal, constrained devices only have access to a few hundred kilobytes of memory [1, Section 3]. For this reason, software written for these devices does not use dynamic memory allocations to avoid memory fragmentation. Instead, memory is statically allocated. In a multithreaded environment this creates an interesting problem regarding the allocation of stack space for different threads executed by the software. As the stack space is also statically allocated at compile-time, the programmer must choose an appropriate maximum stack size before deploying the software. If the code executed in a given thread does not use the allocated stack space in its entirety, memory is wasted thus potentially causing increased production costs. If the stack size—chosen by the programmer—is too small for the thread using it, a stack overflow might occur.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001.

On conventional devices, stack overflows will typically result in a crash due to the violation of employed memory protections. These memory protections are implemented using *Memory Management Units* (MMU) or *Memory Protection Units* (MPU). Both are specialized hardware components allowing for the fine-grained protection of selected memory regions. Unfortunately, a survey by Wetzels found that only $47.1\,\%$ of surveyed constrained devices supported an MMU while $11.8\,\%$ supported an MPU [2]. Meaning, almost one half of surveyed constrained devices provided no hardware support for memory protections, likely to further reduce production costs. As such, stack overflows and similar memory corruptions are often not detected on constrained devices [3]. In the worst case, an undetected stack overflow might allow an attacker to subvert program control flow and achieve remote code execution.

We believe it to be essential to detect stack overflows in a development environment before the software is being deployed in production. Furthermore, doing so would allow for an estimate of the stack size required for different threads executed by the software. In this regard, it is desirable to estimate stack size requirements as early as possible in the development process to assess production costs and determine how code changes—affecting stack usage—would impact these costs.

Software for constrained devices interacts closely with hardware peripherals and often relies on custom peripherals. Unfortunately, custom hardware is not commonly available during early stages of software development. For this reason, *Virtual Prototypes* (VPs) are employed in this domain during early development phases. VPs provide an abstract model of the utilized hardware platform and allow early execution of software targeting this platform. They are predominantly created in SystemC TLM [4]. While prior work has already

presented techniques for detecting stack overflows and estimating stack size requirements, these approaches do not integrate well with VPs and therefore do not address challenges specific to the development of constrained IoT devices.

To address these challenges, we built upon prior work by Park *et al.* which proposes a dynamic stack overflow detection and stack size estimation technique. This technique requires custom compilers and software instrumentation [5]. We illustrate that sanity checks originally performed in the software—using code inserted by a custom compiler—can also be performed in the VP. Thereby allowing the analysis to be performed in-vivo, without any instrumentation or modification of the executed software. Meaning, the analyzed binary can afterwards be flashed on the constrained devices as-is, thereby guaranteeing that observed stack usage behavior will not change due to employed instrumentations. We evaluate our approach by applying it to the RIOT operating system where we identified two previously unknown stack overflows and potential stack size overestimation.

II. Related Work

Prior work has already proposed a variety of techniques to prevent stack overflows and/or reduce stack usage on embedded systems. One solution for reducing stack usage is to resize thread stacks dynamically as needed [6], [7], [8]. In this regard, Biswas *et al.* propose adding additional sanity checks to the compiled software to detect stack overflows and resize the stack segment if an overflow is detected [6]. Similarly, Kim *et al.* measure stack usage periodically at run-time and perform stack reallocations if needed [7]. Behren *et al.* present an operating system which adopts a dynamic stack allocation technique [8]. Unfortunately, these techniques impact run-time behavior and can also cause memory fragmentation both of which we believe to be undesirable, especially on constrained devices which provide realtime guarantees.

For this reason, a different branch of research focuses on determining worst case stack usage prior to software deployment. A popular approach for doing so is static analysis [9], [10], [11]. Regehr *et al.* and Brylow *et al.* present such an approach which specifically focuses on interrupt-driven embedded software [9], [11]. However, static approaches cannot handle recursive functions and indirect function calls. This problem is resolved in prior work by requiring programmers to provide annotations for loop bounds and indirect calls. For example, prior work by Kästner *et al.* requires programmers to manually add annotations in the formal AIS language [10]. Unfortunately, manual effort makes it more laborious to employ such techniques.

Lastly, different dynamic testing approaches for finding stack overflows have been proposed [5], [12], [13]. Regehr uses random testing to determine worst case stack usage and compares results with the aforementioned static approach by the same author [13]. Prior work by Zhang *et al.* relies on memory protection mechanisms provided by the processor [12]. Related work by Park *et al.* uses a modified C compiler to add sanity checks to the preamble of each compiled function to detect stack overflows and estimate worst case stack size [5]. This hinders adaption of such approaches. In order to ease employment in the constrained devices domain, we believe it to be desirable to detect stack overflows during normal testing already performed today with VPs in early stages of software development. Contrary to prior dynamic testing approaches, our approach allows for an in-vivo analysis requiring no modification or instrumentation of the executed software.

1: **procedure** CHECK_STACK($func$)
2: $thread \leftarrow current_thread()$
3: $required \leftarrow current_stackuse + func.stackuse$
4:
5: **if** $required > thread.stacksize$ **then**
6: $handle_stack_overflow()$
7: **else if** $required > max_stackuse[thread]$ **then**
8: $max_stackuse[thread] \leftarrow required$
9: **end if**
10: **end procedure**

Fig. 1. Stack overflow detection and stack size estimation algorithm [5].

III. Background

This section will serve as a brief primer on VPs. Furthermore, a dynamic algorithm for detecting stack overflows will be presented. This algorithm is based on prior work by Park *et al.* [5].

A. Virtual Prototypes

VPs provide an abstract model of a hardware platform, thereby allowing early execution of software targeting this platform. VPs are often written in SystemC [4], a C++ class library which enhances the C++ programming language with facilities for modeling hardware systems. VPs utilize SystemC *Transaction-Level modeling* (TLM) to describe hardware interactions based on a bus architecture where transactions are exchanged over this bus [4, p. 413]. Compared to more accurate modeling levels (e.g. RTL), TLM operates on a higher abstraction level which improves simulation performance and makes it easier to add new peripherals. This eases modeling of constrained devices with custom peripherals. By allowing early execution of embedded software, VPs can also be used to verify this software during early development stages.

B. Stack Overflow Detection

In order to identify stack overflows in a multithreaded *Operating System* (OS), the following information needs to be available to the overflow detection technique:

1) The **currently active thread** executed by the OS at a particular point in time.
2) The allocated **thread stack size**, i.e. the total size of the stack memory region for the currently active thread.
3) The amount of **used thread stack** memory at a particular point in time during thread execution.

Interestingly, the same information is required to estimate stack size requirements of executed threads. Figure 1 presents an algorithm for stack overflow detection and stack size estimation which utilizes the aforementioned information. The algorithm exploits the fact that information is stored on a

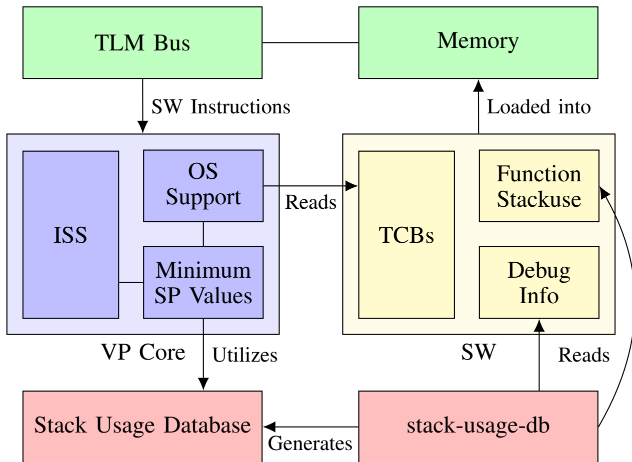

Fig. 2. Architectural overview of our proposed approach.

per-function basis on the stack. The algorithm runs *before* each function execution which is central to preemptively detect stack overflows, thereby preventing the system from malfunctioning.

Firstly, the algorithm determines the currently active thread (Line 2) on which the given function is about to be executed. Afterwards, in Line 3, the amount of stack space required to safely execute the function is computed. This computation is based on the amount of stack space currently in use and the stack space required by the given function (e.g. memory used for local function variables). The resulting value is then compared to the total thread stack size in Line 5. If the required thread stack size exceeds the total allocated thread stack size, execution of the current function would lead to a stack overflow and a stack overflow handler is invoked in Line 6. Otherwise, the maximum stack usage for the current thread is updated, if it exceeds a previously measured value (Line 7 - Line 8).

Prior work by Park *et al.* presents an implementation of this algorithm which relies on a modified C compiler and instrumentations of the executed software [5]. In section IV, we propose a VP-based implementation of this algorithm which allows performing the analysis in-vivo without any software modifications.

IV. Approach

In the following subsections, we will describe how we implemented the algorithm from subsection III-B in a VP context.

A. Overview

Figure 2 shows an overview of our approach. In order to integrate the stack overflow detection and stack size estimation algorithm with a SystemC-based VP, only the *Instruction Set Simulator* (ISS) has to be modified. The ISS (left to the center of Figure 2) is responsible for fetching, decoding, and executing instructions. On each instruction execution we check if the currently executed instruction—as specified by the *Program Counter* (PC) register—corresponds to the entry address of a new function. If so, we execute the algorithm from subsection III-B before executing the instruction specified by the PC.

For execution of this algorithm, the following information needs to be obtained and managed by the VP alongside software execution: The currently active thread (1), the allocated thread stack size (2), and the amount of thread stack memory in use (3). Figure 2 provides an overview of the components required to extract this information. Extracting the currently active thread (1) and the allocated thread stack size (2) is an OS-specific process. Our architecture provides an abstract OS support component (left to the center of Figure 2) which is responsible for extracting this information. Commonly, metadata information for threads is stored in so-called *Thread Control Block* (TCB) by the OS (right to the center of Figure 2). The OS support component accesses these TCBs to extract the active thread and the total thread stack size for OS threads. This component needs to be manually implemented for the specific OS utilized by the executed software. We present an exemplary OS support component implementation for RIOT in section V.

Regarding the extraction of currently used stack memory (3), recall that the stack is just a memory region where the executed software stores information about active functions (e.g. local variables). From the VP perspective, an access to information on the stack is just a load/store instruction relative to the current position in the stack memory region. Instruction set architectures provide a general purpose register, called the *Stack Pointer* (SP), to store the current position in the stack memory region. As such, the amount of currently used stack memory can be determined by consulting this register. Assuming the stack grows downward, the maximum stack usage of a given thread can be determined by storing the minimum SP value measured on a per-thread basis (left to the center of Figure 2).

On each executed instruction, the VP consults a generated database to check if the instruction addresses matches the start address of a function. This database is referred to as Stack Usage Database at the bottom left of Figure 2. It provides a mapping *Function Address* $\mapsto$ *Function Stack Usage* and is thus used by the VP to predict the total required stack space for each executed function as done in Line 3 of Figure 1. The stack usage database is generated through a custom tool called `stack-usage-db` using information extracted from the compiled software (bottom right of Figure 2). This process is further described in the following section.

B. Stack Usage Database

Modern versions of the GCC compiler toolchain support a command-line flag called `-fstack-usage`. This flag causes the compiler to emit stack usage information for individual functions. In C, the compilation process involves compiling separate translation units into separate object files, these object files are then passed to a linker which generates an executable file from the object files [14, Section 5.1.1]. With the `-fstack-usage` command-line flag activated, GCC outputs a separate file with stack usage information for each compiled translation unit in addition to the object file.

```
1  nano-vfprintf.c:392:1:__sfputc_r 0   static
2  nano-vfprintf.c:403:1:__sfputs_r 32  static
3  nano-vfprintf.c:348:1:__sprint_r 16  static
4  stdio.h:503:5:_vfprintf_r        176 static
5  stdio.h:206:5:vfprintf           0   static
```

Fig. 3. Example `-fstack-usage` file generated by GCC.

An example stack usage file is shown in Figure 3. The file consists of multiple lines, each representing information about a function defined in the associated translation unit. Each line consists of three fields providing different information. The first field states the source file where the function is defined, the line/column number, and the function name. The second field states the stack usage in bytes. The third is a qualifier which further specifies how the function uses the stack. A function may have unbounded stack usage if it uses *Variable Length Arrays* (VLAs) where the size of an object on the function stack depends on a variable [14, Section 6.7.5.2]. Functions with an unbounded stack are not supported by our approach but are automatically identified by our tooling—using the aforementioned qualifier—and cause an error message to be emitted. The experiments we performed with RIOT indicate that VLAs are not widely used in the low-end IoT domain.

The problem with the format shown in Figure 3, is that it does not contain any information about text segment addresses of these functions because this information is only available after linking all object files into a binary. However, as per subsection IV-A functions must be identified by their address. For this reason, we wrote a tool—referred to as `stack-usage-db` in Figure 2—which merges multiple `-fstack-usage` files into a single stack usage database which is indexed by function text segment addresses. In order to identify the text segment addresses of utilized functions, the tool operates on a linked ELF binary. It iterates over all function symbols defined in the binary and determines the `-fstack-usage` file for a given symbol from DWARF [15] debug information contained in the binary. That is, the DWARF source line information, which describes where a symbol is defined, is compared against the first field of all `-fstack-usage` files generated by the compiler. If the `-fstack-usage` file for a given symbol was found, the stack usage in bytes for the function represented by this symbol is added to the database. Thereby iteratively creating a mapping *Function Address* $\mapsto$ *Function Stack Usage*. Lastly, the database generated by the `stack-usage-db` tool is passed to the VP on simulation start. This enables the VP to determine whether a new function is being executed (by checking if the database contains a function starting at PC) and allows determining the stack usage requirements of this function. The source code of the `stack-usage-db` tool is available on GitHub[1].

[1]https://github.com/agra-uni-bremen/stack-usage-db

C. OS Integration

Apart from stack usage information about individual functions, the VP also needs to determine the currently executed thread and its associated stack size. As discussed in subsection IV-A, extracting this information requires OS-specific code because multithreaded systems represent information about threads in different ways. However, in order to implement a scheduler the OS will store metadata information for threads in TCBs. TCBs are stored in memory, therefore it is possible for the VP to extract information about a specific thread by accessing the memory location where this information is stored.

Similarly, operating systems often include symbols to allow a debugger to determine the currently active thread, the offset of information in the TCBs, et cetera. Our approach relies on the OS to provide such symbols. As part of the OS support component, we extract the address of these symbols in the executed ELF file using `libdwfl` from `elfutils`[2] directly in the VP. For example, this allows us to determine the memory address of the variable which stores the currently active thread. In the VP context, a SystemC TLM read transaction is then emitted for this address, thereby causing SystemC to retrieve the active thread ID from guest memory. Many operating systems (e.g. RIOT) store the allocated total stack size in the TCB as well, thereby allowing the VP to access this information via the OS support component[3]. The next section further describe extraction of information from TCBs using RIOT as an example OS.

V. Evaluation

We have implemented our proposed technique on top of `riscv-vp`, an open source SystemC-based VP for the RISC-V architecture. The original `riscv-vp` source code is freely available on GitHub[4] [16]. In total we had to modify roughly 600 LOC in `riscv-vp` which shows that a non-intrusive integration of our technique into existing VPs is possible. We evaluated our implementation by applying it to RIOT, an open source OS for the low-end IoT which is further described in a publication by Baccelli *et al.* [17]. We choose RIOT as it supports the RISC-V architecture and the SiFive HiFive1[5] platform, which is also supported by `riscv-vp`. Furthermore, a survey conducted by Hahm *et al.* among operating systems for the low-end IoT domain identified RIOT as the "most prominent open source OS" with multithreading support [18, p. 732].

Multithreading is a core concept of RIOT. Most importantly, RIOT's default network stack (GNRC) implements each network protocol as a separate thread. Different protocol implementations communicate with each other using message passing, a form of interprocess communication. For example, the IPv6 and UDP implementation run in separate threads and the IPv6 thread passes network packets to the UDP thread for further processing [19]. As such, RIOT-based embedded

[2]https://sourceware.org/elfutils/
[3]If this information is not available in the TCB, it can be supplied separately.
[4]https://github.com/agra-uni-bremen/riscv-vp
[5]https://www.sifive.com/boards/hifive1

applications consist of multiple threads. Each thread has its own statically allocated thread stack, thus the pre-allocated stack space has a significant impact on RIOT's memory footprint. In 2018, a minimal RIOT configuration required $3.2\,\mathrm{kB}$ of ROM and $2.8\,\mathrm{kB}$ of RAM, of which $2.2\,\mathrm{kB}$ where thread stack space [17, p. 4436]. Presently, RIOT thread sizes are approximated through pre-defined macros such as `THREAD_STACKSIZE_DEFAULT`.

In the following, we will describe how we integrated our technique with RIOT (subsection V-A) and discuss stack overflows we encountered in RIOT during this integration (subsection V-B). Furthermore, we will report results on the measured thread stack size in preexisting RIOT test applications and compare our results with the approximated pre-allocated stack space (subsection V-C). Lastly, we will evaluate the performance impact our proposed technique has on VP execution speed (subsection V-D). The artifacts for this evaluation are available on Zenodo [20].

A. Integration

As described in section IV, our approach does not require instrumentations or modifications of the tested software. Instead, we extract information about active threads by reading guest memory from the VP. RIOT already provides dedicated debug symbols which allow a debugger to determine information about active threads. For example, RIOT provides a symbol which allows retrieving an identifier for the currently active thread. This information is already used by debuggers, such as OpenOCD[6], to allow for selective debugging of individual RIOT threads. Based on these symbols, we extract the TCBs for RIOT threads and offsets for information stored inside the TCBs.

In RIOT, all thread stack spaces are disjunct. We infer the currently active thread from the SP value by iterating over all thread stack spaces and checking to which stack space the current SP value belongs. Alternatively, it would also be possible to determine the current thread by ID. However, RIOT uses a dedicated stack for handling interrupts (referred to as ISR stack in the following). Code executed on the ISR stack does not belong to any thread and has no thread ID, identifying the current thread by SP allowed us to detect two stack overflows on the ISR stack which will be further described in the next subsection.

B. Stack Overflows

As part of our experiments, we encountered two stack overflows on RIOT's ISR stack. Both occurred during debugging of our RIOT integration. For debugging purposes, RIOT includes several builtin `printf` invocations which are abstracted through a pre-processor `DEBUG` macro and normally disabled. These debug statements can be enabled on a per-file basis. When doing so, the stack space of the associated thread is normally increased by `THREAD_EXTRA_STACKSIZE_PRINTF`. This is necessary as the `printf` family of functions has a comparatively large stack usage. Unfortunately, this approach does not work for the ISR stack since this stack is not allocated in a C file as a static `char` array but instead pre-allocated in the linker script. As such, enabling debug statements in functions executed on the ISR stack causes stack overflows and can lead to weird behavior during debugging. We encountered this issue while debugging the RIOT thread creation code from `core/thread.c`. We have also reported this issue to RIOT developers, at the time of writing it has not yet been fixed[7].

TABLE I
STACK SIZE ESTIMATED FOR RIOT'S GNRC TEST CASES.

Test Case	Thread	MS	CS	Used
gnrc_ipv6_nib	ISR	128 B	512 B	25 %
	idle	76 B	256 B	29.69 %
	ipv6	544 B	1024 B	53.12 %
	main	716 B	1280 B	55.94 %
	mockup_eth	516 B	1024 B	50.39 %
gnrc_ndp	ISR	128 B	512 B	25 %
	idle	72 B	256 B	28.12 %
	main	456 B	1280 B	35.62 %
	test-netif	440 B	1024 B	42.97 %
gnrc_rpl_p2p	ISR	96 B	512 B	18.75 %
	idle	80 B	256 B	31.25 %
	ipv6	228 B	1024 B	22.27 %
	main	304 B	1280 B	23.75 %
gnrc_sock_udp	ISR	176 B	512 B	34.38 %
	idle	80 B	256 B	31.25 %
	ipv6	388 B	1024 B	37.89 %
	main	576 B	1280 B	45 %
	udp	304 B	1024 B	29.69 %

RIOT also includes a build configuration, called `DEVELHELP`, which enables more helpful error messages but does not allow for verbose debugging of individual files. As an example, the trap handler for the RISC-V architecture prints the value of several RISC-V *Control and Status Registers* (CSRs), using `printf`, if `DEVELHELP` is enabled and an unknown trap is encountered. This is useful for easily determining where an unexpected trap occurred. However, since the trap handler is also executed on the ISR stack and the ISR stack is too small to execute `printf` functions, this code path also results in a stack overflow. We encountered this stack overflow as we initially raised a custom trap in the VP when encountering stack overflows. Since this trap is unknown to RIOT, it would cause RIOT to attempt to print the aforementioned debug information which would then result in a stack overflow on the ISR stack and a nested raise of the corresponding trap. We also reported this issue to RIOT developers, one way of fixing it would be switching to a more stack space efficient method for printing CSR values in the trap handler[8]. The fact that we managed to identify two edge cases where a stack overflow occurs in RIOT illustrates the effectiveness of our proposed technique.

C. Stack Size Estimation

In order to evaluate the stack size estimation aspect of our proposed technique, we measured the maximum stack usage for several preexisting test cases for RIOT's network stack GNRC. The results are shown in Table I. Each application

[6]http://openocd.org/

[7]https://github.com/RIOT-OS/RIOT/issues/16395

[8]https://github.com/RIOT-OS/RIOT/issues/16448

TABLE II
BENCHMARKS RESULTS FOR TESTS/BENCH_RUNTIME_COREAPIS.

Benchmark	Modified VP	Baseline	Slowdown
nop loop	0.71 s	0.33 s	53.52 %
mutex_init	0.0 s	0.0 s	0 %
mutex lock/unlock	14.8 s	6.7 s	54.73 %
thread_flags_set	7.22 s	3.3 s	54.29 %
thread_flags_clear	3.98 s	1.61 s	59.55 %
thread flags set/wait any	19.96 s	8.5 s	57.41 %
thread flags set/wait all	17.37 s	7.33 s	57.8 %
thread flags set/wait one	21.31 s	8.91 s	58.19 %
msg_try_receive	10.75 s	4.26 s	60.37 %
msg_avail	3.03 s	0.99 s	67.33 %
Average	-	-	52.32 %

starts multiple threads and was executed until a pre-defined cancellation point was reached (e.g. `TestRunner_end`). For each thread in each test case, Table I shows the measured maximum stack usage (*MS*) and the configured stack size (*CS*) in bytes. Lastly, the percentage of configured stack space that was actually used by the test case is shown in the last column of Table I.

The majority of executed test cases use less than 50 % of the configured stack space. This confirms our initial hypothesis that RIOT threads are largely overprovisioned in terms of stack size. It also noticeable that stack sizes are often re-used and not tailored to a specific application. Most notably, all executed test cases use the default main stack size of 1024 B. Please note though that the executed test cases are not specifically designed to yield the worst case stack usage. As such, measurements from Table I only indicate the maximum stack usage measured but not necessarily the worst possible stack usage. Nonetheless, they serve as a good indicator and may help developers in iteratively optimizing the stack sizes of their application. We focus on in-vivo dynamic analysis in this publication, we will further discuss techniques to address the aforementioned shortcoming in section VI.

D. Performance Impact

As discussed in section IV, our approach relies on sanity checks performed during the execution of RISC-V instructions and thus has an impact on execution performance. We measured this impact by executing RIOT benchmarks specifically designed to gather performance statistics.

The utilized benchmark performs several consecutive invocations of different functions from the RIOT API. In Table II we compare our implementation against the original `riscv-vp` version as found on GitHub[9]. All tests have been performed on an Intel i7-8565U system running Alpine Linux. The first column in Table II (Benchmarks) shows the executed benchmark function, the second the time it took to execute it with our implementation (Modified VP), and the third the execution time with the original `riscv-vp` (Baseline). The fourth column (Slowdown) displays the relative slowdown caused by our implemented stack overflow detection and stack size estimation technique. On average, execution is slowed down by 52.32 % through our employed technique.

[9]All benchmarks have been performed with the `riscv-vp` command-line options `--use-dmi` and `--tlm-global-quantum=10000` enabled.

We believe this to be an acceptable overhead during development. Currently, our implementation performs sanity checks for each executed instruction to implement the algorithm from subsection III-B. The performance impact may be reduced by only performing these checks after jump instructions, which are commonly used by compilers to execute a new function.

VI. Discussion and Future Work

The evaluation demonstrates that our approach is capable of uncovering stack overflows in real-word software for constrained devices. While we focus on the in-vivo analysis technique itself in this paper, we believe that our approach can be improved further by not relying on pre-defined test cases or manual testing to discover stack overflows in the executed software. Since we rely on performed tests, our employed stack size estimation technique does not offer any guarantees that the measured stack usage is actually the worst case stack usage of a given thread. An interesting direction for future work would be investigating whether our approach can be improved by employing software testing techniques which enumerate (all) reachable programs paths automatically. A promising technique in this regard is symbolic execution. Symbolic execution engines—like KLEE [21], FIE [22], or S²E [23]—mark input variables as symbolic and enumerate reachable program paths based on these symbolic inputs. Symbolic execution has also been recently integrated with SystemC-based VPs [24]. In regards to constrained devices, this is especially interesting when considering our proposed stack size estimation technique. Due to memory limitations, the state space on constrained devices is often smaller than on conventional ones, potentially even allowing for a complete analysis using symbolic execution in this domain [22, p. 23]. If a complete analysis is possible, this could allow offering guarantees regarding the estimated thread stack size.

VII. Conclusion

We presented a stack overflow detection and stack size estimation technique for multithreaded operating systems which we implemented as an in-vivo analysis using VPs (section IV). This allows finding stack overflows and gives programmers an estimate regarding thread stack usage during early software development, without any modifications or instrumentations of the executed software. By avoiding software modifications, we can ensure that the observed behavior does not change when deploying the software. Our implementation is specifically tailored to constrained devices where stack overflows would normally go undetected due to the lack of memory protections. In this regard, our technique enabled us to find two previously unknown stack overflows in the low-end IoT operating system RIOT which we reported to RIOT developers (subsection V-B). Additionally, preliminary results obtained using our stack size estimation technique indicate that existing RIOT application potentially overestimate thread stack sizes, thereby wasting memory (subsection V-C). We intend to further improve our proposed technique in future work by combining it with symbolic execution (section VI). Our current implementation is freely available on GitHub[10].

[10]https://github.com/agra-uni-bremen/fdl21-stackuse-vp

References

[1] C. Bormann, M. Ersue, and A. Keränen, *Terminology for Constrained-Node Networks*, RFC 7228, May 2014. DOI: 10.17487/RFC7228. [Online]. Available: https://rfc-editor.org/rfc/rfc7228.txt.

[2] J. Wetzels, "Internet of Pwnable Things: Challenges in Embedded Binary Security," *USENIX ;login:*, vol. 42, no. 02, pp. 73–77, 2017. [Online]. Available: https://www.usenix.org/publications/login/summer2017/wetzels.

[3] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *The Network and Distributed System Security Symposium 2018*, ser. NDSS, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf.

[4] System C Standardization Working Group, "IEEE Standard for Standard SystemC Language Reference Manual," IEEE, Tech. Rep., 2012, pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619. [Online]. Available: https://standards.ieee.org/standard/1666-2011.html.

[5] S. H. Park, D. K. Lee, and S. J. Kang, "Compiler-Assisted Maximum Stack Usage Measurement Technique for Efficient Multi-threading in Memory-Limited Embedded Systems," in *Computers, Networks, Systems, and Industrial Engineering 2011*, R. Lee, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 113–129, ISBN: 978-3-642-21375-5.

[6] S. Biswas, T. Carley, M. Simpson, B. Middha, and R. Barua, "Memory Overflow Protection for Embedded Systems Using Run-Time Checks, Reuse, and Compression," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 4, pp. 719–752, Nov. 2006, ISSN: 1539-9087. DOI: 10.1145/1196636.1196637.

[7] S. C. Kim, H. Kim, J. Song, and P. Mah, "A Dynamic Stack Allocating Method in Multi-Threaded Operating Systems for Wireless Sensor Network Platforms," in *2007 IEEE International Symposium on Consumer Electronics*, 2007, pp. 1–6. DOI: 10.1109/ISCE.2007.4382142.

[8] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable Threads for Internet Services," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 268–281, ISBN: 1581137575. DOI: 10.1145/945445.945471.

[9] J. Regehr, A. Reid, and K. Webb, "Eliminating Stack Overflow by Abstract Interpretation," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 751–778, Nov. 2005, ISSN: 1539-9087. DOI: 10.1145/1113830.1113833.

[10] D. Kästner and C. Ferdinand, "Proving the Absence of Stack Overflows," in *Computer Safety, Reliability, and Security*, A. Bondavalli and F. Di Giandomenico, Eds., Cham: Springer International Publishing, 2014, pp. 202–213, ISBN: 978-3-319-10506-2.

[11] D. Brylow, N. Damgaard, and J. Palsberg, "Static checking of interrupt-driven software," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, 2001, pp. 47–56. DOI: 10.1109/ICSE.2001.919080.

[12] R. Zhang, Y. Du, T. Zhang, Q. Qiu, L. Mao, and J. Niu, "An Improved RTEMS Supporting Real-Time Detection of Stack Overflow," in *Wireless and Satellite Systems*, M. Jia, Q. Guo, and W. Meng, Eds., Cham: Springer International Publishing, 2019, pp. 283–293, ISBN: 978-3-030-19153-5.

[13] J. Regehr, "Random Testing of Interrupt-Driven Software," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05, Jersey City, NJ, USA: Association for Computing Machinery, 2005, pp. 290–298, ISBN: 1595930914. DOI: 10.1145/1086228.1086282.

[14] International Organization for Standardization, "Programming languages – C," International Organization for Standardization, Geneva, CH, Standard, Dec. 1999. [Online]. Available: https://www.iso.org/standard/29237.html.

[15] Debugging Information Format Committee, "DWARF Debugging Information Format, Version 4," Debugging Information Format Committee, Tech. Rep., 2010. [Online]. Available: http://www.dwarfstd.org/doc/DWARF4.pdf.

[16] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101 756, 2020, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2020.101756.

[17] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, Dec. 2018, ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2815038.

[18] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, Oct. 2016, ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2505901.

[19] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," *Computing Research Repository*, vol. abs/1801.02833, 2018. [Online]. Available: http://arxiv.org/abs/1801.02833.

[20] S. Tempel, V. Herdt, and R. Drechsler, *Artifacts for the FDL21 Paper: In-Vivo Stack Overflow Detection and Stack Size Estimation for Low-End Multithreaded Operating Systems using Virtual Prototypes*, Zenodo, Sep. 2021. DOI: 10.5281/zenodo.5091709.

[21] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[22] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C.: USENIX Association, Aug. 2013, pp. 463–478, ISBN: 978-1-931971-03-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson.

[23] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 265–278, ISBN: 9781450302661. DOI: 10.1145/1950365.1950396.

[24] S. Tempel, V. Herdt, and R. Drechsler, "An Effective Methodology for Integrating Concolic Testing with SystemC-based Virtual Prototypes," in *Design, Automation and Test in Europe Conference (DATE). Design, Automation & Test in Europe (DATE-2021)*, Grenoble, France, Feb. 2021.