Secure Context Switching of Masked Software Implementations

Barbara Gigerl
Graz University of Technology
Graz, Austria

Robert Primas
Graz University of Technology
Graz, Austria

Stefan Mangard
Graz University of Technology
Graz, Austria

ABSTRACT

Cryptographic software running on embedded devices requires protection against physical side-channel attacks such as power analysis. Masking is a widely deployed countermeasure against these attacks and is directly implemented on algorithmic level. Many works study the security of masked cryptographic software on CPUs, pointing out potential problems on algorithmic/microarchitecture-level, as well as corresponding solutions, and even show masked software can be implemented efficiently and with strong (formal) security guarantees. However, these works also make the implicit assumption that software is executed directly on the CPU without any abstraction layers in-between, i.e., they focus exclusively on the bare-metal case. Many practical applications, including IoT and automotive/industrial environments, require multitasking embedded OSs on which masked software runs as one out of many concurrent tasks. For such applications, the potential impact of events like context switches on the secure execution of masked software has not been studied so far at all.

In this paper, we provide the first security analysis of masked cryptographic software spanning all three layers (SW, OS, CPU). First, we apply a formal verification approach to identify leaks within the execution of masked software that are caused by the embedded OS itself, rather than on algorithmic or microarchitecture level. After showing that these leaks are primarily caused by context switching, we propose several different strategies to harden a context switching routine against such leakage, ultimately allowing masked software from previous works to remain secure when being executed on embedded OSs. Finally, we present a case study focusing on FreeRTOS, a popular embedded OS for embedded devices, running on a RISC-V core, allowing us to evaluate the practicality and ease of integration of each strategy.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Side-Channel Analysis, Masking, Verification, Embedded OS, RTOS

ACM Reference Format:
Barbara Gigerl, Robert Primas, and Stefan Mangard. 2023. Secure Context Switching of Masked Software Implementations. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579856.3595798

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0098-9/23/07.
https://doi.org/10.1145/3579856.3595798

1 INTRODUCTION

Embedded devices have become omnipresent in IoT, automotive, and industrial applications and often interact with their physical environment. This raises the need for strong cryptographic primitives to preserve private and secure operations. Embedded devices need to be protected against both theoretical and physical attacks. Theoretical security refers to guarantees such as the resistance of cryptography against mathematical attacks, while physical security counteracts adversaries in physical proximity who observe a device's physical properties during computation. In 1999, Kocher et al. [36] presented Differential Power Analysis (DPA), which allows for extracting secrets like cryptographic keys from a device. DPA is performed by observing a device's power consumption, which correlates with the processed data. Since then, masking has become a very popular and well-studied countermeasure to defeat such attacks on algorithmic level [9, 12, 15, 17, 26, 29, 34, 46]. With masking, each secret variable of a cryptographic computation, such as the encryption key, is split into $d + 1$ random shares. Consequently, the power consumption of the device does not correlate with the unshared secret but with the $d + 1$ random shares, which exponentially increases the difficulty of recovering the unshared secret. One particular advantage of masking is that it is provably secure, i.e., it can be proven that an attacker cannot reveal any information about the unshared secret by combining up to d shares.

In the past, many works have pointed out a significant gap between the theoretical and practical security of masked implementations [6, 23, 44], often caused by physical effects such as glitches and transitions. Masking schemes generally assume that independent computations result in independent leakage, which is not necessarily the case in a practical software or hardware implementation. In other words, a masked software implementation that has been formally proven to be d-th order secure in theory might not reach this security level when executed on a CPU. Many works in the past have discussed to which extent the CPU microarchitecture can compromise the security of masked software implementations. Prominent root causes of order-reducing leakage in masking are register or memory overwrites, which leak the Hamming distance between two shares [7, 11, 18, 44]. On top of that, many more such potential problems have been identified that essentially boil down to implementation specifics of the register file, SRAM, load-store logic, data caches, or bypass mechanisms in the CPU pipeline [20, 23, 24, 41]. In order to solve these problems efficiently, works like [23, 24] emphasize that modifications on software level are necessary while additional hardware changes of the CPU are advisable. Eventually, both the CPU hardware and the masked software implementation need to fit together to obtain secure execution that preserves the theoretical security of the masking.

In practice, with the exception of the most basic microcontrollers or IoT devices, embedded devices execute software within an (embedded) OS alongside other tasks which include bus or network communications, and sensor data acquisition and processing. In the case of resource-constrained embedded devices, one often chooses dedicated embedded OSs, including real-time operating systems (RTOS), over fully-fledged operating systems such as Linux. FreeRTOS [2] is a very popular choice for such an embedded OS because it provides a wide range of supported platforms, a large community, and is publicly available (open-source).

So far, works on the practical security of masked software implementations focus on the bare-metal case and therefore assume total control over the execution of software on a CPU [10, 19, 23, 24, 39, 40, 44, 51, 52]. More concretely, they assume that the masked software is not interrupted during execution. The interference of multitasking OSs, especially context switching, that leads to a violation of this assumption has not been considered in previous analysis efforts at all. Still, context switches occur at high frequencies, e.g., due to periodic (timer) interrupts, and in some cases, their occurrence can even be controlled by the attacker. Consider, for example, the following setting, in which an attacker first requests a certain cryptographic operation via a common communication interface and causes an IO interrupt at a later point in time by sending an additional request. The attacker can easily observe repeated executions of the cryptographic operation in which context switches cause additional leakage, allowing to easily mount straight-forward attacks like DPA on the additionally created leakage in the power side-channel [36].

As a countermeasure, one could consider the option of disabling interrupts during the execution of masked software; however, this option is unrealistic in practice due to (1) the starving of other relevant tasks like sensor data acquisition/processing or Bluetooth/UART/MQTT network communication, and (2) the generally strict scheduling requirements of RTOS systems that allows meeting timing constraints [3, 14, 56]. There does exist one work by Balasch et al. [8] from 2015 demonstrating successful DPA attacks on an AES implementation executed by a Linux operating system on an ARM Cortex-A8, and discussing the security of masked software in this setting. The authors show that also the masked version of the AES is not leakage-free. However, it remains unclear whether the empirically found leakage is caused by the CPU microarchitecture, the OS, or even the masking algorithm itself. On top of that, they also do not propose a solution that can reliably prevent the observed leakage.

Contributions. The security of masked software implementations running as a task/process within an embedded OS has not been evaluated so far. It is hence unclear to what extent specific OS features like interrupts, scheduling, and context switches cause leakage in such situations and what corresponding protection mechanisms can be put into place at what cost. We close this gap by providing the first in-depth analysis of masked software executed by an OS on a CPU. The main contributions of this work are as follows:

- We provide the first formal analysis of masked software which runs as a task in an embedded OS on a CPU. Using a toy example, we show that the main problems are caused during context switching by either overwriting shares in memory, or transitions on memory/register file read/write ports (Section 4).
- We propose several possible strategies to solve these problems, resulting in a formally verified context switching routine hardened against side-channel leakage that requires no assumptions on the current location of shares in the register file. This allows masked software, verified for correctness in the bare-metal case, to preserve security when executed on an embedded OS. For each strategy, we provide a comparison of their advantages, disadvantages, and performance overhead (Section 5).
- We present a case study of masked software running as a task in FreeRTOS on a RISC-V CPU. In this case study, we show that the problems identified in our analysis also exist in FreeRTOS and that these problems can be fixed efficiently by the proposed strategies. (Section 6).
- We make the evaluation setup and all software artifacts available on GitHub[1].

2 BACKGROUND

In this section, we first give necessary background information on the masking countermeasure. We briefly introduce Coco, a formal verification tool to prove that the execution of (bare-metal) masked software on a given CPU netlist is secure. For our work, we use Coco as a leakage detection mechanism, as well as to formally verify the security of our countermeasures. Finally, we provide a short introduction to embedded operating systems.

2.1 Masking

Power analysis attacks exploit the fact that the power consumption of a cryptographic device depends on the processed data, such as a secret key [16, 36]. The masking countermeasure breaks this dependency by randomizing sensitive intermediate values processed by the device [15, 26, 31, 42]. Each sensitive variable used in a cryptographic computation is split into $d + 1$ random shares, such that the observation of up to d shares does not reveal any information about the corresponding sensitive value.

In the case of a dth-order Boolean masking scheme, the shares $s_0 \dots s_d$ must satisfy $s = s_0 \oplus \dots \oplus s_d$, where $\oplus$ stands for the exclusive OR (XOR) operation. Hereby, $s_0 \dots s_{d-1}$ are chosen uniformly at random, while $s_d = s_0 \oplus \dots s_{d-1} \oplus s$, which ensures that each share s_i is uniformly distributed and statistically independent of s. For example, a first-order masking scheme ($d = 1$) splits up a sensitive variable s into two parts s_0 and s_1, such that $s = s_0 \oplus s_1$, s_0 is chosen uniformly at random, and $s_1 = s \oplus s_0$.

Throughout the entire implementation, a proper separation of shares and of the output of the component functions needs to be ensured not to violate the d^{th}-order independence, which is commonly expressed in the probing model of Ishai et al. [34]. In the probing model, the attacker has the ability to probe up to d intermediate results of the masked implementation. An implementation is said to be secure if the probing attacker cannot gain any statistical advantage in guessing any secret variable by combining the probed results in an arbitrary manner. While this share separation can be easily ensured for functions which are linear over GF(2^n) – for

[1]https://github.com/barbara-gigerl/sw-masking-rtos

example, the masked calculation of $x \oplus y$ can be performed share-wise ($x_i \oplus y_i$) – the secure implementation of nonlinear functions usually requires the introduction of fresh randomness.

The probing model can directly be applied to masked hardware circuits, in which the attacker can place probes on individual gates and wires, which then allow observing all values at the chosen location for an infinite amount of time. However, the probing model is less suitable for masked software implementations executed by a CPU. For example, the attacker could simply place one probe on the read or write port of the register file and then observe all intermediate values (including shares), which allows breaking masked software of arbitrary protection order. Instead, recent works refer to the time-constrained probing model [23] for masked software implementations, which puts a time restriction of one clock cycle on each probe. More formally, the attacker who possesses d probes can distribute these both spatially and temporally, allowing them to perform measurements at different locations in the same clock cycle, the same location in different clock cycles, or a mix of both. For example, a first-order attacker ($d = 1$) in the time-constrained probing model can only probe register file read or write ports for the duration of one clock cycle.

Besides algorithmic correctness of masking schemes in a respective probing model, the practical security of masked cryptographic algorithms also strongly depends on implementation specifics in hardware [1, 17, 31, 32, 41, 42] and software [7, 13, 20, 54]. We discuss the case of secure software masking in more detail in the following.

2.2 Practical security of masked software

The security of masked software implementations depends on the assumption that independent computations result in independent leakage [7, 15, 26]. However, many works have shown that this property is often violated in practice when executing masked software (bare-metal) on CPUs [7, 23, 24, 39, 41, 44]. The main reason for this is physical side-effects in the CPU, for example, glitches and transitions, which lead to unintended combinations of shares during execution. For example, Coron et al. [18] show that when a share is overwritten by another share of the same sensitive variable, the power consumption correlates with the combination of both, leading to leakage. In practice, this can be observed when overwriting shares stored in a CPU register or the SRAM. Gigerl et al. [23, 24] report that glitches within the control logic used to address the read/write logic of the CPU register file might make leakage-free register file accesses impossible. Additionally, they show that shares of the same sensitive variable must not be read or written consecutively, independently whether they are stored in the register file or memory, due to transitions on the respective read/write ports. As a result, they construct a side-channel hardened version of the RISC-V Ibex core (*secured* Ibex), which allows leakage-free execution of masked software in bare-metal mode as long as a few simple software constraints are followed. In our work, we exclusively work with masked software implementation following these constraints and use the *secured* Ibex core as a reference platform for our experiments.

2.3 Coco

In order to evaluate the security of masked cryptographic software, one can either apply empirical or formal verification methods. Empirical verification involves manually taking power measurements of CPUs during computation, followed by statistical analysis that tries to extract sensitive information such as cryptographic keys [16, 36]. The main downside of this approach is the inability to identify the exact source of leakage in a system, i.e., there is no possibility to determine if a leak was caused by the CPU microarchitecture or the masked software implementation. Alternatively, one can use the recently published tool Coco [23] to formally verify the security of masked software executions in the time-constrained probing model on the gate-level netlist of a CPU. Coco allows to identify concrete gates/wires/registers in the CPU netlist as leakage sources.

In general, Coco takes as input a masked assembly implementation backed up with some annotations and the description of a CPU as a gate-level netlist and then reports whether the execution is leakage-free or not. The annotations (labels) indicate which registers/memory locations contain shares or fresh randomness at the start of the software execution. Internally, the tool then starts by simulating the execution of the software on the CPU hardware in order to obtain an execution trace, which contains a concrete value for each control signal in the CPU. The verifier then propagates the annotated labels through the CPU netlist cycle by cycle while considering the control signals of the execution trace. If Coco finds a gate in a specific cycle that combines all shares of the same sensitive value, the gate in the netlist and the cycle is reported as a leak. Using this information, one can then easily find out whether the leak was caused by the software itself, or micro-architectural side-effects of the CPU. For more details on the internal working mechanisms of Coco, we refer to the original publication [23].

2.4 Embedded operating systems

Embedded systems requiring multitasking make use of an embedded OS, which runs multiple tasks and manages shared resources such as execution time. Some scenarios additionally require real-time capabilities, i.e., that the OS guarantees that specific tasks or events can be handled in a specific amount of time. Such operating systems are called real-time operating systems (RTOS), on which we focus in this work. Events occurring during the execution of an RTOS are often called interrupts, such as the periodic timer interrupt, which happens at specific intervals, or non-periodic interrupts caused by IO operations or other external events. To maintain its real-time capabilities, the OS must react and handle the interrupt appropriately. Therefore, it activates the scheduler to select the next task to be run and performs a *context switch* or task switch. In order to do so, information related to the task in the *task control block (TCB)* needs to be saved, which also contains the working state (context), including general purpose and floating point registers. During a context switch, the TCB needs to be saved and restored from memory. The memory area which contains the TCB is called the *TCB memory slot*, which is often part of the tasks's working stack assigned by the OS on startup. RTOSs are currently being used in all kinds of applications, including smart watches, traffic light systems, and home energy monitoring. FreeRTOS [2] is among the most popular RTOSs, and is built into, e.g., Amazon's AWS IoT, Tesla's electric cars, and Bosch's smart home sensors. Other famous RTOSs include the open-source systems Zephyr [45], RIOT [5], TockOS [37] KataOS [21], but also many closed-source systems like MQX [48] and PikeOS [25].

3 ATTACKER MODEL

For our study, we consider a threat model in which an attacker has physical access to a cryptographic device that runs masked software within an embedded OS on a microprocessor. Examples of such devices are electronic wallets, smart cards, or authentication tokens. The attacker's goal is to leak the cryptographic key stored on the device, which is used by cryptographic software that already features sufficient protection against standard differential power analysis (DPA) using masking countermeasures. The attacker does not need to know specific details about the attacked device, such as the concrete source code; it is sufficient to know which cryptographic operation is implemented. To perform the attack, the attacker (1) connects an oscilloscope to the device such that power/EM traces can be recorded, (2) triggers the targeted cryptographic operation by sending an appropriate request via a communication interface, and (3) interrupts the operation by sending some other request with a specific delay. Consequently, all interaction with the device is purely passive , and, e.g., no direct control of the OS runtime is required.

Given a scenario in which an attacker can force interrupts during the execution of masked software at specific points in time, thereby causing potential masking-related correctness problems, what remains is to record a sufficient amount of such computations for a DPA attack to work. The concrete number of power traces required by the adversary highly depends on the noise level of the attacked system, i.e., a combination of the masked software, the embedded OS, and the microprocessor. Nevertheless, we can look at some previous works that study the impact of unintentional combinations of shares due to overwrites of shares in memory or the register port, which is likely to occur during context switches. For example, as shown by Papagiannopoulos et al. [44], about 50 000 traces are sufficient on an ATMega163 8-bit microcontroller to detect memory overwrites, which are the basis of our first proposed attack. The authors also study a form of transitions on register file read ports, the basis of our second proposed attack, which can be exploited by only about 5 000 traces. In practice, these numbers can be higher, e.g., if there is some variation in the timing of the execution of the masked software and the following interrupt. This essentially has a similar effect on the performance of DPA attacks as algorithmic hiding countermeasures and hence generally do not increase the amount of required measurements by more than a quadratic factor.

4 ANALYZING CONTEXT SWITCHES IN MASKED SOFTWARE

In this section, we identify common problems that could arise when running masked cryptographic software as a task on an embedded OS. As a starting point, we use assembly implementations following all constraints from [23], as well as their *secured* IBEX core for our experiments. The assembly implementations are formally verified for correctness in bare-metal mode using Coco. We then manually insert additional assembly instructions at certain locations to represent a realistic context switch routine. We investigate potential leakage using Coco, which finally reveals two major sources of leakage introduced by context switches. In the following, we provide a more detailed description of our experiment setup and the identified problems.

4.1 Experiment setup

We construct a toy example modeling two tasks to demonstrate the general problem of context switches related to masked software. The first task is executing a masked Keccak S-box (used, e.g., in SHA-3, Shake or Ascon), while the other one is performing unrelated non-cryptographic computations. The toy example is then verified when running on the *secured* IBEX core. In the following, we give more details about the concrete software and hardware setup.

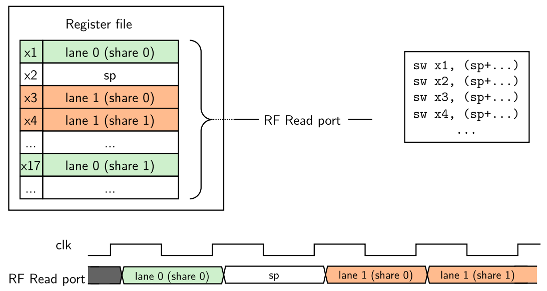
The first task (T_{SBOX}) executes a 1st-order masked Keccak S-box implementation protected by Domain-Oriented Masking (DOM) [31]. In general, the implementation splits up the five 32-bit lanes of the implementation into two shares and uses secure DOM multiplication gadgets to mask non-linear operations. The second task (T_{CNT}) executes a non-cryptographic computation which is unrelated to the first task. We choose a simple function that keeps a counter in a register that is constantly being incremented.

On startup, each task is assigned a specific memory location for the stack and another memory location for the TCB (the TCB memory slot). In practice, the TCB is either stored on the top or at the bottom of the task's working stack. We reserve register r_2 (*sp* on a RISC-V architecture) to store a pointer to this memory area. We then model the effect of real interrupts by manually calling context switching routines from T_{SBOX} and T_{CNT} at certain points in time. This represents a practical scenario where, e.g., an attacker first requests a cryptographic operation via a common communication interface that is then later interrupted by another IO request after a specific amount of time. The context switching routine (context switch) itself is based on an existing implementation included in the FreeRTOS RISC-V port that saves the state of the current task to memory, changes the stack pointer, and then loads the state of the next task from memory. We sketch this function in Appendix A.1 and also further discuss it later in Section 6.1.

We use Coco to investigate the security of our toy example on the *secured* RISC-V IBEX core [23]. Before starting the experiment, we first verify that the 1st-order Keccak S-box runs securely on the RISC-V IBEX core in bare-metal mode. Consequently, any leakage we observe originates from the context switching activity itself and not from issues with the masked software or the micro-architecture of the IBEX core. In the following sections, we show that the security guarantees derived from verifying bare-metal software no longer hold when executing the masked software within a task.

4.2 Transitions on memory/register file read/write port

Whenever a context switch is performed, the register file contents of the current task are stored to memory register by register in a sequence of store instructions. Hence, an attacker probing the register file read port or memory write port for the duration of one specific cycle can observe pairs of two register values of the



```
sw x1, (sp+...)
sw x2, (sp+...)
sw x3, (sp+...)
sw x4, (sp+...)
...
```

Figure 1: Transition on register file read port during context switch

current task. In the second part of the context switch, the register file contents of the next task are loaded from memory, and the current register file contents are overwritten. Again, an attacker probing the register file write port or memory read port can observe pairs of register values of the next task.

If the current task is executing masked software, the register file potentially contains shares of the same secret distributed over several registers. For example, in our toy scenario, the five 32-bit lanes of the Keccak S-box are stored in ten registers, each register containing one share. In Figure 1, we sketch the register file contents of T_{SBOX} at some point during the execution right before a context switch. The timing diagram illustrates the information an attacker can observe by proving the register file read port cycle per cycle. While no critical information can be deduced from the transitions $x_1 \rightarrow x_2$ and $x_2 \rightarrow x_3$, the transition $x_3 \rightarrow x_4$ leaks the Hamming distance between shares 0 and 1 of lane 1, which refers to the unshared value of lane 1.

4.3 Overwriting shares in memory

The exact memory location of the TCB memory slot is defined when the task is created and usually remains unchanged throughout the lifetime of the task on the bottom of the stack. With the general purpose registers being part of the TCB, every context switch during the execution will overwrite the old register contents in memory with the new ones. An attacker probing the respective TCB memory location can therefore observe a transition of the old register value to the new register value. If the memory location previously contained a certain share and is then updated with its counterpart, the attacker can probe the unshared value.

We give an illustration of this scenario using our toy example in Figure 2, in which T_{SBOX} starts execution, is then exchanged by T_{CNT}, until T_{SBOX} resumes. After T_{SBOX}'s second execution, shares stored to memory in the previous context switch might get overwritten by their counterparts. In detail, the following steps occur:

① T_{SBOX} starts execution until it gets interrupted, and the context switch routine is triggered, which saves the register values to the respective TCB memory slot.
② The register file of T_{CNT} is restored, and the task continues execution until the next interrupt.
③ In the context switch, the register values of T_{CNT} are saved to the TCB memory slot of T_{CNT}.
④ The register values of T_{SBOX} are restored from the TCB.
⑤ T_{SBOX} continues the computation. In our case, the Keccak S-Box implementation exchanges registers x_{17} and x_1 (due to implementation-specific reasons), i.e., the locations of the two shares of lane 0 are swapped in the register file. Note that this can indeed be done securely in an implementation, e.g., with the help of intermediate register clearings.
⑥ During the context switch, the registers are saved, and the old TCB contents of T_{SBOX} are overwritten. More precisely, the memory location storing the old content of register x_1 still refers to share 0 of lane 1. The new content of register x_1 is now - after the computation - the other share, which will, however, be stored in the same memory location. Consequently, share 0 is overwritten with share 1 in memory. The attacker can again observe the Hamming Distance between the old and the new register value, which refers to the unshared lane 1.

4.4 Discussion

Whether the stated problems occur in a practical implementation is still determined by many different parameters, including the frequency of context switches (influenced by the timer interrupt frequency and the attacker's ability to trigger such), the exact point at which the context switch occurs and the exact location of shares in the register file. All these parameters make it infeasible to fix these problems by just adapting the masked software implementation because one would need to take into account the behavior of the embedded OS (such as the sequence in which the registers are spilled) and consider a possible context switch after every instruction. In the next section, we hence aim for more general concepts for secure context switch routines (realized either is software or hardware) that allows masked software, verified in bare-metal scenarios, to preserve security when executed on embedded OSs.

A masked software implementation emits leakage if two shares are combined, e.g., by overwrites and transitions, independently of the concrete masking scheme used. In our case, T_{SBOX} is protected by DOM [31] as an example, but for the analysis, it would be irrelevant which masking scheme is used. In general, DOM has been used both in hardware [31, 32, 33, 35, 38], but also in software [23, 24]. Threshold Implementations (TI) [42], Ishai-Sahai-Wagner (ISW) [34], and the Unified Masking Approach (UMA) [30] are other examples of masking schemes which have been applied to both hardware and software implementations. Several works on masked software implementations following no specific schemes exist, which are usually optimized for a concrete cryptographic algorithm [20, 27, 28, 43, 47]. Which scheme to follow depends on the optimization constraints (for example, speed, code size, register sizes, available RNG) of the design. The security of a masked software implementation is, however, not influenced by the concrete technique used because, in any masking scheme, combining two shares will lead to leakage.

5 SCA-SECURE CONTEXT SWITCHING

In this section, we discuss basic strategies to prevent the problems identified in Section 4 and obtain a context switching mechanism that allows masked software, verified in bare-metal scenarios, to

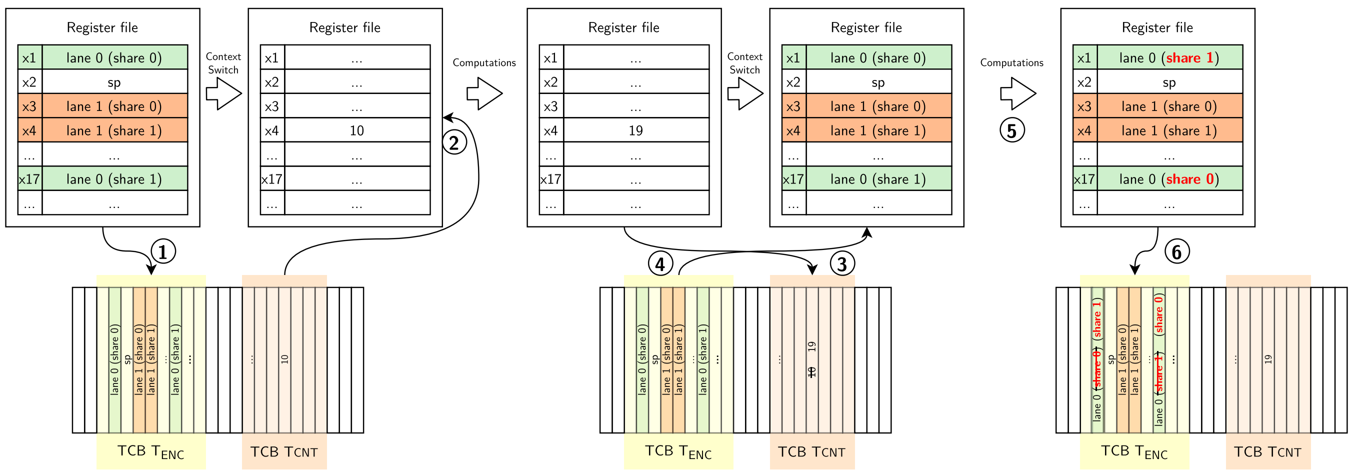


Figure 2: Overwriting shares in memory during context switch

preserve security when executed on embedded OSs. The strategies are not specific to a particular embedded OS implementation but should rather give generic concepts which can be integrated into any embedded OS. We provide an in-depth comparison of the different basic strategies, discuss the overhead in terms of memory and runtime, and evaluate their advantages and disadvantages. We formally prove that each strategy allows leakage-free context switching using Coco by integrating each strategy into the toy example introduced in Section 4. The given basic strategies are divided into two categories, either helping against transitions or memory overwrites. Additionally, we discuss why solving these problems on software level (by increasing the masking order) is neither efficient nor feasible in practice.

In Table 1, we give an overview of the strategies, which we will in the following describe more in detail. The table shows for each strategy which problem is addressed, whether modifications are necessary on OS- or CPU-level, and states the overhead compared to the plain, unprotected context switch, which takes 125 cycles to execute on the *secured* Ibex core. To determine the memory overhead, we compare the size of the (potentially modified) TCB to the original TCB (128 byte).

5.1 Basic strategies against transitions

In Section 4, we identified the problem of transitions on memory/register file read/write ports. In the following, we discuss two strategies to prohibit this problem, dummy operations after every load/store and interleaved context switches. In addition to verification of these strategies in the toy example, we add a second verification scenario to strengthen the proposed security guarantees. In the second scenario, we label all 28 registers as shares of the same native value, perform the hardened context switch with these registers and then check if the execution provides 27th-order security. By that, we can show that the constructed secure context switch is indeed SCA-secure independently of the concrete location of shares in the register file.

5.1.1 Dummy operations after every load/store. The most simple solution to prevent transitions between shares on read/write ports is to insert dummy operations, such as nop instructions, after every load or store in the context switch. This ensures that the read/write port is always pulled to zero between two memory accesses, preventing direct transitions between shares. While this solution is very simple in terms of integration, its effectiveness and runtime overhead is strongly determined by the underlying CPU microarchitecture. On the *secured* Ibex core, it suffices to put a single nop instruction between two memory accesses, yielding a runtime overhead of 46%. However, as shown in works like [24], more complex architectures might require more dummy operations to prevent such leakages.

Instead of using nop instructions, one could try to use instructions of the interrupt handling logic which is executed after storing the register contents to the TCB. While this solution would make the context switching more efficient, the feasibility of integrating this is into an embedded OS is highly dependent on the existing context switching/scheduling logic.

5.1.2 Interleaved context switch. A context switch first stores the TCB of the current task selects the next task, and then loads the TCB of the next task. We alter the sequence of these three events to perform an *interleaved* context switch, which first selects the new task, and then loads/stores the contents of the two involved TCB blocks in an alternating (interleaved) manner. The interleaved context switch essentially uses the load operations as dummy operations mentioned in the previous paragraph. On assembly level, this boils down to alternating store and load instructions, as sketched in Appendix A.2.

While this solution requires no additional runtime or memory overhead, it is very restrictive on the task selection logic, i.e., the scheduler, since all registers used there must not be used during the task's execution. For example, consider a task getting executed, which stores some data into register x_{10}. When it gets interrupted, the scheduler is triggered to select the next task, and if it uses register x_{10} to do so, the task's data in the register will inevitably be overwritten and therefore lost. Therefore, we consider this solution suitable for our toy example where the task selection works in a very simple round-robin fashion but show in Section 6 that it

Table 1: Comparison of basic strategies

Basic strategy	Protection against		Modifications		Overhead	
	Transitions on RW port	TCB memory overwrites	OS	CPU	Memory (TCB)	Runtime (context switch)
No protection	✖	✖	-	-	128 byte	125 cycles
Dummy operations after every load/store	✔	✖	yes	no	128 byte (+0%)	183 cycles (+46%)
Interleaved context switch	✔	✖	yes	no	128 byte (+0%)	125 cycles (+0%)
TCB clearing	✖	✔	yes	no	128 byte (+0%)	183 cycles (+46%)
Rotating TCB memory slots	✖	✔	yes	no	$128\,\text{byte} + \frac{128\,\text{byte}}{\text{Number of tasks}}$	145 cycles (+16%)
Randomness-refreshed loads and stores (SW)	✖	✔	yes	no	132 byte (+3%)	201 cycles (+61%)
Randomness-refreshed loads and stores (HW)	✖	✔	yes	yes	132 byte (+3%)	143 cycles (+14%)

is infeasible for most embedded OS due to the complexity of the scheduling logic.

5.2 Basic strategies against overwrites

In Section 4, we identified the problem of overwriting shares in memory. In the following, we suggest strategies to prohibit this problem.

5.2.1 TCB clearing. The most naive method to protect against TCB memory overwrites is to clear the TCB of a task executing masked software before saving the registers, i.e., overwriting the memory locations with zeros. For each general-purpose register that is saved during the context switch, this requires one additional store operation. The clearing operations can either be executed in one block before storing the register values or alternating with the actual register store operations, as shown in Appendix A.3. If the alternating order is used, it also prevents transitions on the register file read port. However, transitions on the register file write port are not prevented. The runtime overhead of a context switch that clears the TCB is 46 %.

5.2.2 Rotating TCB memory slots. On startup, every task is statically assigned a TCB memory slot which is not changed during execution. In order to prevent overwriting shares in memory, one must ensure that the task executing the masked software implementation does not overwrite its own saved registers, which can be done by dynamically changing the TCB memory slot with every context switch. Since physical memory is limited on such constraint devices, allocating a new TCB memory slot with each and every context switch is not feasible. Instead, we need to make sure that TCB memory slots are reused over time. A TCB memory slot can be reused if it does not store the most recent TCB of any currently suspended task (older copies are fine). Consequently, after a task gets interrupted, it must not use its own current TCB memory slot and no memory slot of any other suspended task. We ensure this by adding one additional TCB memory slot such that there is always at least one TCB memory slot that can be reused, and further using a *rotating* assignment of TCB memory slots.

In Figure 3, we sketch this concept using our toy example. In the beginning (①), the two tasks use TCB memory locations TCB #1 and TCB #2 to store their data. We add TCB #3 to ensure one TCB memory location is always reusable. After T_{SBOX} has been scheduled once (②), it uses the currently unoccupied TCB #3. It cannot use TCB #1 because it belongs to T_{CNT}, which is currently suspended, and it cannot (re-)use TCB #2 because it contains its own old data, and overwriting might lead to leakage. After T_{CNT} has been executed, it uses TCB #2, which previously belonged to T_{SBOX}, and overwrites the old saved TCB of T_{SBOX}, thus clearing all shares stored to memory. In the next step, T_{CNT} uses TCB #1 to store its TCB after execution, and T_{SBOX} uses TCB #3 (③), leading to a rotating assignment of TCB memory slots.

Although this method comes with almost no time overhead, one additional TCB memory location (128 byte) must be reserved in memory such that the tasks do not overwrite each other's contexts. Additionally, there must be at least one other task running which can overwrite the old saved TCB of the task executing the masked software. If this cannot be ensured, one can either insert a dummy task serving this purpose or extend the kernel in a way such that it clears the old context on purpose, i.e., if no other task was scheduled. It is important to note that the overwrite problem persists even though the task executing the masked software implementation is the only one running in the operating system. Assume that only T_{SBOX} is actively running, and T_{CNT} is sleeping. Whenever T_{SBOX} is interrupted, the working registers will be stored in the TCB. Then, the scheduler will be run and decides that no other task should be scheduled, and loads the register values of T_{SBOX} again. That is, the registers of a task will always be stored to memory when an interrupt occurs, independently of whether another task will finally be scheduled or not.

5.2.3 Randomness-refreshed loads and stores. TCB memory overwrites can also be counteracted by not storing the plain task context but by adding 32 bits of randomness to each register before storing it to memory. The same randomness can be used for all registers saved in the context switch but must be renewed with every occurring context switch. The randomness must be removed when restoring the context. A task executing masked software will therefore overwrite its old context protected with a different mask. Randomness-refreshed loads and stores can be either implemented in software, by modifying the context switch routine of the OS, or in hardware, by extending the respective CPU core.

We construct a software implementation of this method using our toy example. We assume that there is a certain memory region supplying fresh randomness upon request, which may be connected to an RNG in practice. When a context switch occurs, we first fetch 32 bits of randomness from the location using a load instruction and store it to one of the general-purpose registers. Next, we add the randomness to every register in the register file using a bitwise XOR before saving the register to memory. The used random value is then stored to the TCB of the respective process. When restoring the registers for the process in the next context switch, the previously used random value needs to first be obtained from the TCB again. After issuing the load instruction, which restores a specific GPR value, the random value needs to be added to the GPR register value again in order to obtain the previous value. We sketch this process in Appendix A.4. The memory overhead of this strategy is around 3% because we need to store the most recently used value of the fresh randomness in the respective TCB, such that it can be fetched before the next load of registers of the respective task. The runtime overhead mostly caused by the additional XOR instructions when storing and restoring the context is 61%. When applying this strategy, at least one register needs to be reserved for storing the randomness, which must not be used in the task's code. Similar to clearing the TCB slot, only tasks executing masked software, such as T_{SBOX}, need to refresh loads and stores in context switches, but all other tasks can stick to the original routine. In practice, the OS usually has a notion about the purpose of each task and can, therefore, easily decide if randomness-refreshing is necessary or not.

Additionally, we provide a hardware implementation of this method, which eliminates most of the runtime overhead by performing the XOR implicitly in the load-store unit of the *secured* Ibex core instead of having to issue a dedicated XOR instruction every time. For this purpose, we extend the core's CSR unit by a 32-bit register which contains the randomness to refresh loads and stores, and a 1-bit register which indicates whether randomness-refreshed loads and stores are enabled. We leave the management of the fresh randomness in memory to the OS, i.e., the OS needs to load fresh randomness from memory to the CSR register itself. In the context switch routine, one, therefore, needs two additional `CSRW` instructions, one for enabling the countermeasure and one for loading the randomness to the respective CSR register. Therefore, the runtime overhead of such a modified context switch is 14%, as shown in Table 1, of which the most accounts to the management of fresh randomness for both tasks.

5.3 Lazy engineering

In 2015, Balasch et al. [7] discuss the "lazy engineering" approach of implementing masked software with a protection order that is higher than theoretically required to compensate for a certain loss in practical security due to micro-architectural side-effects. Assuming that a certain masked software implementation is (bare-metal) d-th order secure, a standard context switch routine can generally reduce the security down to $\lfloor d/2 \rfloor$. For example, a transition on a register file read/write port essentially creates leakage that combines values of registers that are loaded/stored in immediate succession. This can, in our case, cut the number of probes required to observe all

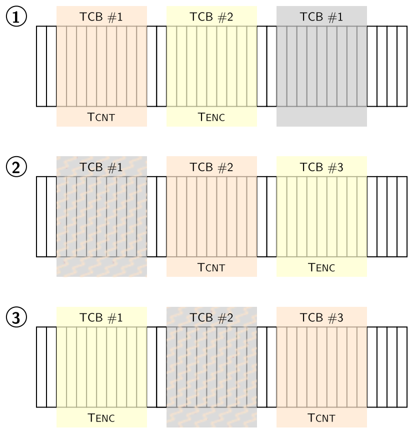


Figure 3: Rotating TCB memory slots

shares in half, for which one compensates with a higher masking order on level of the masked software.

To achieve a first-order secure masked Keccak S-box, we construct a second-order variant, which provides 1st-order security when using the unprotected context switch. However, the 2nd-order implementation requires much more runtime and randomness: While the 1st-order masked Keccak S-Box needs 174 cycles (without context switches) and 160 bits of fresh randomness, the 2nd-order implementation requires 283 cycles (without context switches), which is an increase of 63%, and 480 bit of fresh randomness (+300%).

Therefore, we do not consider this solution feasible in practice due to the exponential overhead for more complex microarchitectures [24]. Especially for higher masking orders, the overhead caused by lazy engineering grows with the masking order, while the overhead of the other suggested solutions is the same independently of the order.

6 CASE STUDY

In this section, we investigate the security of masked software running as a task in FreeRTOS. In Section 6.1, we introduce our evaluation environment. In Section 6.2, we discuss that the problems identified in Section 4 can indeed be found in practice in FreeRTOS using Coco, and how the context switch in FreeRTOS can eventually be fixed against these problems. In Section 6.3, we provide combinations of the basic strategies which defend against both transitions *and* overwrites and evaluate their performance overhead. While a single hardened context switch comes with some overhead, the overall system overhead is rather negligible, given that the frequency of context switches is usually low in practice. To the best of our knowledge, this is the first analysis of masked software within an OS, especially on such a level of detail. This is likely due to the considerable effort of creating a suitable analysis environment, which we describe in the beginning of this section. In our case, for example, this involves porting the entire FreeRTOS to the RISC-V Ibex core, adding peripherals to trigger timer interrupts, and simulating the synthesized processor netlist when executing the OS in a suitable simulator for formally verifying it with Coco. We plan to make our evaluation setup, along with all software artifacts, available in a public repository.

FreeRTOS. FreeRTOS is a popular open-source embedded OS used in many different IoT projects supported by a large community, which makes it the third most used OS in 2019 [4]. It has been ported to different hardware architectures and platforms, including ARM, RISC-V, and x86-32 [49], and targets small single-core microprocessors in embedded systems. In order to provide multitasking, the FreeRTOS kernel uses the scheduler to assign processing time to tasks. The scheduler is usually triggered by a (timer) interrupt or the `yield` system call and selects the next task according to a certain policy that takes into account task priorities and deadlines. For our case study, we use the standard preemptive scheduling policy (`configUSE_PREEMPTION = 1` and `configUSE_TIMESLICING = 0`), which means that the task with the highest priority will be selected by the scheduler [50].

6.1 Evaluation setup

Ultimately, our evaluation environment should allow both formal verification and cycle-accurate performance evaluations. Hence, we need to simulate FreeRTOS, including tasks when running on the *secured* Ibex core in order to obtain a cycle-accurate execution trace. Unfortunately, there exists no exact demo allowing such a simulation, which is why we first need to port FreeRTOS to run on the *secured* Ibex core. We base our port on the existing 32-bit RISC-V Spike simulator demo and make several adaptions, such as changing the addresses of the `mtime` and `mtimecmp` registers. FreeRTOS can only be executed properly in the presence of a periodic timer interrupt. We use a dedicated hardware module for creating these interrupts, which is connected to the *secured* Ibex core. This hardware module provides access to the `mtime` and `mtimecmp` control registers. From a verification perspective, and compared to the bare-metal case, the interrupt signals provide just another set of control signals beside the executed software. Our complete workflow can be sketched as follows:

(1) We synthesize the *secured* Ibex core with Yosys [55] to obtain a gate-level netlist in Verilog format.
(2) We compile FreeRTOS, including all tasks which are later executed.
(3) We wrap the synthesized Ibex netlist into a testbench which includes a timer and a memory model.
(4) We simulate the testbench with Verilator [53], which produces a cycle-accurate execution trace. This execution trace contains a concrete value for each control signal in the netlist and can, therefore, directly be used for performance evaluations.
(5) In order to do formal verification with Coco, we additionally create the respective annotations (labels) indicating the location of shares/fresh randomness at the beginning of the execution and give these annotations, the netlist of the *secured* Ibex, and the execution trace to Coco.

Besides the execution environment, another central part of the evaluation is the tasks that are executed by FreeRTOS. In order to demonstrate that the identified problems occur, we focus on the same scenario as in the previous sections, i.e., that the OS runs one task executing a masked 1st-order Keccak S-box (T_{SBOX}), and one task which increments a counter (T_{CNT}).

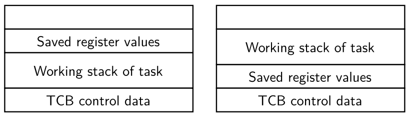


Figure 4: Original (left) vs adapted (right) memory layout of FreeRTOS to support rotating TCB memory slots

In order to perform a meaningful performance evaluation of our solution, we, however, stick to a larger scenario including a complete Ascon round (T_{ENC}) [22], which uses the Keccak's S-box core, running beside T_{CNT}. A complete Ascon round is likely to be interrupted more often than a single Keccak S-box by a periodic timer interrupt or external interrupts. Since the performance overhead of our countermeasures stems purely from the context switching, the results become more significant. Similar to our toy example, both T_{SBOX} and T_{ENC} load the input data (shares) from a predefined memory location, compute the S-box/Ascon round, and then stores the input data back to the memory.

6.2 Hardening the FreeRTOS context switch

In this section, we describe the challenges of integrating the basic strategies into FreeRTOS. Protection against both leakage sources in FreeRTOS is achieved by combining the basic strategies against transitions with those against overwrites.

Preventing transitions. Besides the problem discussed in Section 4, we could identify a second leakage source caused by transitions in the FreeRTOS scheduler. A context switch can generally be divided into three phases: (1) storing the TCB of the current task, (2) selecting the next task, and (3) loading the TCB of the next task. FreeRTOS uses the same code for phases (1) and (3) as we used in our toy example but has a slightly more complex scheduler, which potentially creates another source of transition-induced leakage between shares on the register file write port. For example, our implementation of the Keccak S-Box was interrupted at a point where registers $x20$ and $x24$ each contained a share of the same native value. The scheduling logic (phase (2)) contains a section of code that first overwrites $x20$, and in the next cycle, overwrites $x24$, causing a leaking transition on the register file write port. Whether these leaks occur is, however, highly dependent on both the concrete scheduler logic and the masked software implementation. A generic solution can only be achieved when ensuring that the general purpose register values of the task are not used anymore in the scheduling logic, which is why we clear the registers after storing them to memory.

In Section 5, we discuss interleaved context switches as one possible countermeasure against transition leakage, which requires selecting the new task (phase 2) *before* performing storing and loading register values in an interleaved manner. It is not feasible to integrate this strategy into FreeRTOS because the scheduling logic would inevitably overwrite many unsaved task registers when running it before phase (1). Therefore, we instead focus on dummy operations and clearing registers to defend against transition leakage.

Preventing overwrites. We include all four basic strategies to prevent overwrites of shares in the TCB. Including rotating TCB memory slot requires the most changes, as the original version of

Table 2: Evaluation of variants of SCA-hardened context switching in FreeRTOS

	Runtime in cycles				Number of context switches	Cycles per context switch
	Total	T_{ENC}	T_{CNT}	**Context switching**		
No protection	4149	1359	1221	1569	4	393
Basic strategies						
Dummy operations + clear registers	5810	1366	1564	2880	6	480 (+22%)
TCB clearing	5729	1364	1648	2717	6	453 (+15%)
Rotating TCB memory slots	4203	1353	1154	1696	4	424 (+7%)
Randomness-refreshed loads and stores (SW)	5836	1395	1587	2854	6	475 (+20%)
Randomness-refreshed loads and stores (HW)	4263	1411	1152	1700	4	425 (+8%)
Combined strategies						
Dummy operations + clear registers + TCB clearing	5978	1376	1395	3207	6	534 (+36%)
Dummy operations + clear registers + rotating TCB memory slots	5880	1349	1478	3053	6	508 (+29%)
Dummy operations + clear registers + Randomness-refreshed loads and stores (SW)	7651	1416	1782	4453	8	557 (+41%)
Dummy operations + clear registers + Randomness-refreshed loads and stores (HW)	5940	1410	1476	3054	6	509 (+29%)

FreeRTOS stores the TCB control data and the values of the general-purpose registers separately. That is, the control data is stored on the bottom of the user stack, and the registers on top. We sketch this in the left part of Figure 4. Hence, the memory location where the registers are stored possibly changes with every context switch, depending on the height of the user stack. This makes it impossible to implement TCB memory slot rotation because when a task uses its stack, e.g., during a function call, it potentially overwrites another task's saved registers which were stored there. Instead, we adapt the layout such that the saved register values are also stored below the working stack of the task and can, therefore, not be overwritten by a task's stack usage (c.f. right of Figure 4). Another challenge is constructing a function find to the next free TCB memory slot before saving any general-purpose registers of the task. The function must not use any general-purpose registers, which still contain the task's data, because overwriting them would inevitably destroy the task's state. Thankfully, we can use registers x_3 and x_4, which are never saved during a context switch and are generally unused in FreeRTOS. The original purpose of these registers is to store the thread pointer and global pointer for optimizations, which is, however, not supported by FreeRTOS.

Fewer system changes are needed to implement randomness-refreshed loads and stores, as we simply extend the TCB struct of the OS by a new variable `task_rand` which is updated during the context switch with the used randomness. Also in this case we make use of x_3 and x_4 to load, store and xor the respective random value.

6.3 Discussion

Table 2 shows the overhead of an SCA-hardened context switch when used in FreeRTOS. To measure the performance overhead, we stick to a complete Ascon round (T_{ENC}), scheduled alternating with T_{CNT}. The execution is interrupted by a periodic timer, which frequency can be controlled from software using the `configTICK_RATE_HZ`-define in FreeRTOS. The original configuration of FreeRTOS specifies a timer interrupt every 100 000 cycles which seems plausible considering that in a real system, context switches will not only be triggered by timers but also by many more (non-periodic) external interrupts. As there are no external interrupt sources in our evaluation environment we configure the timer interrupt to occur every 1000 cycles, which however represents an extremely-high-load scenario for the system. Given these numbers, one can then easily extrapolate overheads for scenarios with less frequent context switches.

For each evaluated scenario, we give the total number of cycles needed to compute a full Ascon round, which is the sum of cycles consumed by T_{ENC} and T_{CNT}, and cycles spent on context switching. Note that the number of cycles between two timer interrupts is always constant (1000 cycles), and as the context switching requires more time, less execution time will be available for the tasks, and therefore, more context switches will be necessary in total. We also give the number of cycles required per context switch and the runtime overhead in percent w.r.t. to the basic scenario (no protection).

Which combination to choose depends on the concrete use case. If OS and hardware modifications should be kept minimal and low runtime is not so critical, one should stick to the first option (TCB clearing) because it is very simple to integrate into any OS. If performance is more critical, rotating TCB memory slots are the best option, although they require more OS changes and require at least one other actively running task. Randomness-refreshed loads and stores are more efficient when implemented with hardware support. In software, there is no clear advantage compared to TCB clearing or rotating TCB memory slots. As discussed above, in practice, interrupts are, however, expected to occur much less frequent than in this case study. We argue that therefore, all of the four suggested combinations would be suitable because the amount the total runtime of an Ascon round is increased by applying a secure context switch when interrupted only once is negligible.

Optimizations. Further optimization of the runtime of the individual basic strategies is most likely not possible on software level, since they are already written in Assembly language. However, hardware support for clearing the TCB and all registers could be added and would likely result in a performance gain, although the hardware changes are expected to be much larger than the ones suggested for randomness-refreshed loads and stores. Instead, one could aim for further optimizations on OS level. In fact, the suggested protections must only be applied when a task executing masked software is involved in a context switch. Otherwise, the unprotected (and comparably cheap) context switch can be executed. An additional flag in the task's TCB can be used to distinguish tasks executing masked software from other unrelated tasks.

6.4 Other RTOS

In the following we briefly discuss other open-source RTOS and the possible security implications on masked software running in a task.

Zephyr, RIOT. Zephyr [45] is an RTOS maintained by the Linux Foundation for resource-constrained devices with a strong focus on security. RIOT [5] is similar to Zephyr but comes with different scheduling strategies and supported platforms. The Zephyr and RIOT context switching routines apply a different register order compared to FreeRTOS when storing and loading the register values, which shows why one could never make assumptions about such aspects when designing masked software. We expect both OS are vulnerable to the problems discussed in Section 4, and that our countermeasures can be integrated in a similar way.

TockOS, KataOS. TockOS [37] and the recently announced KataOS [21] are written almost completely in Rust, and both are used for Google's OpenTitan project, which runs the RISC-V Ibex core. The nature of the TockOS context switch suggests the same problems as identified above, which can be solved using the basic strategies except for rotating TCB memory slots. TockOS keeps all processes isolated from each other using a hardware Memory Protection Unit (MPU). Rotating TCB memory slots requires the existence of a common memory region which stores data (register values) of multiple processes, which hurts the principle of isolation, while the other suggested countermeasures are compatible with such isolation techniques.

7 CONCLUSION

In this paper, we provide the first security analysis of context switches for masked cryptographic software. After showing the fundamental problems created by context switches on embedded OSs, we propose several different mitigation strategies in hardware or software. Ultimately, our hardened context switching routines allow masked software from previous works, verified for security in bare-metal execution, to remain secure when being executed on embedded OSs. We present a case study focusing on FreeRTOS, a popular embedded OS for embedded devices, running on a RISC-V core, allowing us to evaluate the practicality, ease of integration, and performance of each strategy. While the runtime of hardened context switches is certainly noticeable, we expect the overall impact on system performance to be rather negligible unless the frequency of context switches is very high.

ACKNOWLEDGMENTS

This work was supported by the FWF SFB project SpyCoDe F8504, and the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". We thank the anonymous reviewers for their valuable suggestions and comments.

REFERENCES

[1] Mehdi-Laurent Akkar and Christophe Giraud. 2001. An Implementation of DES and AES, Secure against Some Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2162)*, Çetin Kaya Koç, David Naccache, and Christof Paar (Eds.). Springer, 309–318.

[2] Inc. Amazon Web Services. 2022. FreeRTOS. https://www.freertos.org/ https://www.freertos.org/. Retrieved on 15/12/2022.

[3] Farhad Andalibi and Paulo Garcia. 2021. Near-Native Interrupt Latency in Real-Time Guests: Handler Emulation Through Memory Map Morphing. In *ICCDE 2021: 7th International Conference on Computing and Data Engineering, Phuket, Thailand, January 15 - 17, 2021*. ACM, 94–98.

[4] Aspencore. 2019. 2019 Embedded Markets Study: Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, Retrieved on 3/11/2022.

[5] Emmanuel Baccelli, Cenk Gündogan, Oliver Hahm, Peter Kietzmann, Martine Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. 2018. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet Things J.* 5, 6 (2018), 4428–4440.

[6] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, and Ingrid Verbauwhede. 2012. Theory and Practice of a Leakage Resilient Masking Scheme. In *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7658)*, Xiaoyun Wang and Kazue Sako (Eds.). Springer, 758–775.

[7] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. 2014. On the Cost of Lazy Engineering for Masked Software Implementations. In *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8968)*. Springer, 64–81.

[8] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. 2015. DPA, Bitslicing and Masking at 1 GHz. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9293)*, Tim Güneysu and Helena Handschuh (Eds.). Springer, 599–619.

[9] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. 2017. Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017.*

[10] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. 2021. Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 2 (2021), 189–228.

[11] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. 2021. Analysis and Comparison of Table-based Arithmetic to Boolean Masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 3 (2021), 275–297.

[12] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. 2017. Private Multiplication over Finite Fields. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 10403)*. Springer, 397–426.

[13] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. 2020. Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations. In *EUROCRYPT (3) (Lecture Notes in Computer Science, Vol. 12107)*. Springer, 311–341.

[14] Bernard Blackham, Yao Shi, and Gernot Heiser. 2012. Improving interrupt response time in a verifiable protected microkernel. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 323–336.

[15] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. 1999. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1666)*, Michael J. Wiener (Ed.). Springer, 398–412.

[16] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. 2002. Template Attacks. In *CHES (Lecture Notes in Computer Science, Vol. 2523)*. Springer, 13–28.

[17] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. 2016. Masking AES with d+1 Shares in Hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9813)*. Springer, 194–212.

[18] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. 2012. Conversion of Security

Proofs from One Leakage Model to Another: A New Issue. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7275)*. Springer, 69–81.

[19] Yann Le Corre, Johann Großschädl, and Daniel Dinu. 2018. Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10815)*, Junfeng Fan and Benedikt Gierlichs (Eds.). Springer, 82–98.

[20] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. 2016. Bitsliced Masking and ARM: Friends or Foes?. In *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10098)*. Springer, 91–109.

[21] AmbiML Developers. 2022. KataOS. https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html.

[22] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. 2016. Ascon v1.2. Submission to the CEASAR Competition. https://ascon.iaik.tugraz.at/files/asconv12.pd. Retrieved on 4/2/2021.

[23] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. 2021. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1469–1468.

[24] Barbara Gigerl, Robert Primas, and Stefan Mangard. 2021. Secure and Efficient Software Masking on Superscalar Pipelined Processors. In *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13091)*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer, 3–32.

[25] SYSGO GMBH. 2022. PikeOS. https://www.sysgo.com/pikeos https://www.sysgo.com/pikeos. Retrieved on 14/12/2022.

[26] Louis Goubin and Jacques Patarin. 1999. DES and Differential Power Analysis (The "Duplication" Method). In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1717)*, Çetin Kaya Koç and Christof Paar (Eds.). Springer, 158–172.

[27] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. 2018. Secure Multiplication for Bitslice Higher-Order Masking: Optimisation and Comparison. In *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10815)*, Junfeng Fan and Benedikt Gierlichs (Eds.). Springer, 3–22.

[28] Dahmun Goudarzi and Matthieu Rivain. 2017. How Fast Can Higher-Order Masking Be in Software?. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.).

[29] Hannes Groß and Stefan Mangard. 2017. Reconciling d+1 Masking in Hardware and Software. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10529)*. Springer, 115–136.

[30] Hannes Groß and Stefan Mangard. 2018. A unified masking approach. *J. Cryptogr. Eng.* 8, 2 (2018), 109–124.

[31] Hannes Groß, Stefan Mangard, and Thomas Korak. 2016. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. ACM, 3.

[32] Hannes Groß, David Schaffenrath, and Stefan Mangard. 2017. Higher-Order Side-Channel Protected Implementations of KECCAK. In *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*. IEEE Computer Society, 205–212.

[33] Michael Gruber, Matthias Probst, Patrick Karl, Thomas Schamberger, Lars Tebelmann, Michael Tempelmeier, and Georg Sigl. 2021. DOMREP-An Orthogonal Countermeasure for Arbitrary Order Side-Channel and Fault Attack Protection. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 4321–4335.

[34] Yuval Ishai, Amit Sahai, and David A. Wagner. 2003. Private Circuits: Securing Hardware against Probing Attacks. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2729)*. Springer, 463–481.

[35] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. 2022. Automated Generation of Masked Hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 1 (2022), 589–629.

[36] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO (Lecture Notes in Computer Science, Vol. 1666)*. Springer, 388–397.

[37] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP'17)*. 234–251.

[38] lowRISC contributors. 2019. AES HWIP Technical Specification. https://opentitan.org/book/hw/ip/aes/index.html https://opentitan.org/book/hw/ip/aes/index.html. Retrieved on 19/4/2023.

[39] Ben Marshall, Dan Page, and James Webb. 2022. MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 1 (2022), 175–220.

[40] David McCann, Elisabeth Oswald, and Carolyn Whitnall. 2017. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 199–216.

[41] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. 2020. On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software. *IACR Cryptol. ePrint Arch.* 2020 (2020), 1297.

[42] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. 2006. Threshold Implementations Against Side-Channel Attacks and Glitches. In *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4307)*. Springer, 529–545.

[43] Elisabeth Oswald and Kai Schramm. 2005. An Efficient Masking Scheme for AES Software Implementations. In *Information Security Applications, 6th International Workshop, WISA 2005, Jeju Island, Korea, August 22-24, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3786)*, JooSeok Song, Taekyoung Kwon, and Moti Yung (Eds.). Springer, 292–305.

[44] Kostas Papagiannopoulos and Nikita Veshchikov. 2017. Mind the Gap: Towards Secure 1st-Order Masking in Software. In *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10348)*. Springer, 282–297.

[45] Zephyr Project. 2022. Zephyr OS. https://www.zephyrproject.org/ https://www.zephyrproject.org/. Retrieved on 14/12/2022.

[46] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. 2015. Consolidating Masking Schemes. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9215)*. Springer, 764–783.

[47] Matthieu Rivain and Emmanuel Prouff. 2010. Provably Secure Higher-Order Masking of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6225)*. Springer, 413–427.

[48] NXP Semiconductors. 2022. MQX Real-Time Operating System (RTOS). https://www.nxp.com/design/software/embedded-software/mqx-software-solutions/mqx-real-time-operating-system-rtos:MQXRTOS. Retrieved on 14/12/2022.

[49] Amazon Web Services. 2022. FreeRTOS Kernel Ports. https://www.freertos.org/RTOS_ports.html https://www.freertos.org/RTOS_ports.html. Retrieved on 5/11/2022.

[50] Amazon Web Services. 2022. FreeRTOS Scheduling. https://www.freertos.org/implementation/a00005.html https://www.freertos.org/implementation/a00005.html. Retrieved on 5/12/2022.

[51] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. 2021. Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 685–699.

[52] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. 2021. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.

[53] Wilson Snyder. 2022. Verilator. https://www.veripool.org/wiki/verilator. Retrieved on February 2nd, 2021.

[54] Junwei Wang, Praveen Kumar Vadnala, Johann Großschädl, and Qiuliang Xu. 2015. Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*, Kaisa Nyberg (Ed.). Springer.

[55] Claire Wolf. 2016. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/. Retrieved on February 2/2/2021.

[56] Peifeng Zhang, Hong Li, and Zhigang Gao. 2009. PIL: A Method to Improve Interrupt Latency in Real-Time Kernels. In *International Conference on Scalable Computing and Communications / Eighth International Conference on Embedded Computing, ScalCom-EmbeddedCom 2009, Dalian, China, September 25-27, 2009*, Keqiu Li, Geyong Min, Yongxin Zhu, Meikang Qiu, and Wenyu Qu (Eds.).

A APPENDIX

A.1 Unprotected context switch

```
task_switch:
  sw x1, (sp)
  sw x2, 4(sp)
  sw x3, 8(sp)
  sw x4, 12(sp)
  # ...
  # Select next task
  # ...
  lw x1, (sp)
  lw x2, 4(sp)
  lw x3, 8(sp)
  ...
  ret
```

A.2 Interleaved context switch

```
task_switch_interleaved:
  mv sp, x30    # Reserve x30, never use in code
  # ...
  # Select next task
  # ...
  sw x1, (x30)
  lw x1, (sp)
  sw x2, 4(x30)
  lw x2, 4(sp)
  sw x3, 8(x30)
  lw x3, 8(sp)
  sw x4, 12(x30)
  lw x4, 12(sp)
  ...
  ret
```

A.3 TCB clearing

```
task_switch_clear_tcb:
  sw x0, (sp)     #x0 is constantly tied to 0
  sw x1, (sp)
  sw x0, 4(sp)
  sw x2, 4(sp)
  sw x0, 8(sp)
  sw x3, 8(sp)
  sw x0, 12(sp)
  sw x4, 12(sp)
  # ...
  # Select next task
  # ...
  lw x1, (sp)
  lw x2, 4(sp)
  lw x3, 8(sp)
  ...
  ret
```

A.4 Randomness-refreshed loads and stores (SW)

```
task_switch_rand_refresh_sw:
  li x30, addr_prng # Reserve x30, never use in code
  lw x30, (x30)     # x30 now contains fresh randomness
  xor x1, x1, x30
  sw x1, (sp)
  xor x2, x2, x30
  sw x2, 4(sp)
  xor x3, x3, x30
  sw x3, 8(sp)
  xor x4, x4, x30
  sw x4, 12(sp)
  # Store x30 to TCB
  # ...
  # Select next task
  # ...
  # Load randomness used in previous store from TCB to
      x30
  lw x1, (sp)
  xor x1, x1, x30
  lw x2, 4(sp)
  xor x2, x2, x30
  lw x3, 8(sp)
  xor x3, x3, x30
  ...
  ret
```

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009