*d*OSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel

Martin Hoffmann, Florian Lukas, Christian Dietrich, Daniel Lohmann
Friedrich–Alexander–Universität (FAU) Erlangen–Nürnberg, Germany
{hoffmann,lukas,dietrich,lohmann}@cs.fau.de

***Abstract*—Because of shrinking structure sizes and operating voltages, computing hardware exhibits an increasing susceptibility against transient hardware faults: Issues previously only known from avionics systems, such as bit flips caused by cosmic radiation, nowadays also affect automotive and other cost-sensitive "ground-level" control systems. For such cost-sensitive systems, many software-based measures have been suggested to harden applications against transient effects. However, all these measures assume that the underlying operating system works reliably in all cases. We present software-based concepts for constructing an operating system that provides a reliable computing base even on unreliable hardware. Our design is based on two pillars: First, strict fault avoidance by static tailoring and elimination of susceptible indirections. Second, reliable fault detection by fine-grained arithmetic encoding of the complete kernel execution path. Compared to an industry-grade off-the-shelf RTOS, our resulting *d*OSEK kernel thereby achieves a robustness improvement by four orders of magnitude. Our results are based on extensive fault-injection campaigns that cover the entire space of single-bit faults in random-access memory and registers.**

I. Introduction

Due to shrinking transistor sizes and operating voltages, transient hardware faults are an emerging challenge for safety-critical real-time systems [8]. Recent functional safety standards, such as the automotive ISO 26262 standard [19], take up this fact and recommend explicit measures against transient hardware faults. These recommendations include both hardware- and software-based measures, whereby the actual choice is still left to the manufacturer. However, in cost-sensitive domains, such as the automotive sector, efficiency in terms of per-unit-price is a key criterion, which limits the use of full hardware redundancy. Here, software-based measures provide a cost-effective and flexible alternative.

There are several approaches proposing process-level redundancy [32], [36], [37] to realize software-based dependability on the application layer. Such concepts can also be applied to automotive real-time software [3]. However, they do not tackle faults *within the operating systems* that coordinate the execution of the replicated instances. Other approaches integrate dependability services into the operating system [10], which simplifies the realization of *triple modular redundancy* (TMR), but does not protect the kernel itself against transient faults. Even the existing ISO-26262–compliant real-time operating systems ensure only strict isolation of the deployed applications against each other (typically by employing hardware-based memory protection [4] or hypervisor [6] technology), but not against faults that occur in the respective kernel structures. In consequence, if a certain safety level has to be reached, these systems still depend on dedicated safeguarded hardware components that act as *reliable computing base* (RCB) [11]. To overcome this hardware limitation, we propose *d*OSEK, a robust real-time operating system that provides a *reliable* computing base, leveraging safety-critical systems even on *unreliable* hardware.

II. System and Fault Model

Our approach aims to tackle transient hardware faults (soft errors) [24], which arise from single-event upsets, for example caused by radiation, electromagnetic noise, or voltage fluctuations. In principle, these effects can have an impact on all parts of a computer system, that is, combinatorial logic, registers, and volatile memory cells [7], [16], [31]. Here, we concentrate on effects that become visible on the hardware–software boundary. Thus, our concrete fault model are single-bit flips that manifest in software-accessible memory and registers.

In this setting, *d*OSEK is supposed to reliably contain and detect any such fault that occurs within the kernel sphere. Thus, in case of a fault, the kernel may throw an exception (*fail-stop*), but never silently violate the spatial or temporal integrity of the application tasks, which thereby get strong guarantees to take care of their own dominion by appropriate measures [32], [36], [37]. In essence, *d*OSEK provides a continuous sphere of redundancy *across* the kernel for a reliable execution environment for the implementation of application-layer dependability measures.

On the hardware side, we still have some requirements: First, we assume that all code and all constant data can be placed in *read-only memory* (ROM) that is considered to be robust against transient faults. Given the even increasing robustness of modern FLASH cells [18], we consider this to be a realistic assumption. Secondly, we assume the presence of standard hardware-based protection mechanisms for isolation in space and time, typically provided as a *memory protection unit* (MPU) and a watchdog timer. These requirements are nowadays provided by even low-cost 32-bit microcontrollers[1] and very common in automotive control units.[2] Here, they define the properties required by *d*OSEK itself to provide its RCB.

III. Overview and Contributions

We present the design and implementation of *d*OSEK, an OSEK/AUTOSAR-conforming [5], [27] *real-time operating*

This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. LO 1719/1-3

[1]For example ARM Cortex-M3–based microcontrollers
[2]For example Infineon TriCore, or Freescale MPC5xx microcontrollers

system (RTOS) that serves as *reliable computing base* (RCB) for safety-critical systems. We developed *d*OSEK from scratch with dependability as the first-class design goal.

We describe our development approach, which is based on two pillars: First, as described in Section IV, we aim for strict *fault avoidance*[3] by static tailoring of the kernel towards the concrete application and hardware platform – without restricting the required RTOS services. We could show in recent work [14] that static tailoring generally leads to a higher inherent robustness of the resulting system due to the reduction of vulnerable run-time state. The dependability-oriented tailoring in *d*OSEK minimizes such vulnerable state even further by also condensing control-flow state within the kernel.

Any fault affecting the remaining indispensable state information is very likely to lead to a crash, timeout or even *silent data corruptions* (SDCs). Therefore, the second step is to constructively re-integrate redundancy in form of dependability measures to eliminate the remaining SDCs. Here, we concentrate – in contrast to others [4], [20], [33] – on reliable *fault detection* and fault containment within the kernel execution path (Section V). We employ arithmetic encoding [12] to realize self-contained data and control-flow error detection, a technique we have previously applied to harden the voter in *CoRed* TMR systems [15], [34]. In this paper, we use it to harden a complete RTOS kernel.

We evaluate our hardened *d*OSEK against ERIKA [1], an industry-grade open source OSEK implementation, which received an official OSEK/VDX certification (Section VI). We present the run-time and memory overhead as well as the results of extensive fault-injection campaigns covering the *complete* fault space of single-bit faults in registers and volatile memory.

IV. Fault-Avoiding Operating-System Development

The general susceptibility of an operating system to errors and SDCs is to a high degree rooted in its basic design and implementation concepts. For instance, we could show in previous work [14] that, without any dependability-oriented measures, a static OSEK-like RTOS (i.e., all resources are allocated at compile time) already exhibits a five times lower number of SDCs than a more dynamic POSIX-like RTOS (i.e., all resources are allocated at run time). So what are the general design principles and implementation concepts that support such *inherent* robustness against transient faults and what are those that impair it?

A. General Considerations

Essentially, a transient fault can lead to an error inside the kernel only if it affects either the kernel's control or data flow. For this, it has to hit a memory cell or register that carries *currently alive* kernel state, such as a global variable (always alive), a return address on the stack (alive during the execution of a system call), or a bit in the status register of the CPU (alive only immediately before a conditional instruction). Intuitively, the more long-living state a kernel maintains, the more prone it is to transient faults. Thus, our first rule of fault-avoiding OS development is: ❶ **Minimize the time spent in system calls and the amount of volatile state, especially of global state that is alive *across* system calls.**

[3]Strictly spoken, we aim to avoid *errors* resulting from hardware faults.

However, no kernel can provide useful services without any run-time state. So, the second point to consider is the containment and, thus, detectability of data and control-flow errors by local sanity checks. Intuitively, bit-flips in pointer variables have a much higher error range than those used in arithmetic operations; hence, they are more likely to lead to SDCs. In a nutshell, any kind of indirection at run time (through data or function pointers, index registers, return addresses, and so on) impairs the inherent robustness of the resulting system. Thus, our second rule of fault-avoiding operating-system development is: ❷ **Avoid indirections in the code and data flow.**

Nevertheless, an important lesson we learned is that the effect of dependability-oriented measures on the actual implementation is difficult to assess in advance. In many cases, additional measures turned out to do more harm than good: For instance, extensive CRC-based consistency checks on the kernel state can help to detect errors early. However, the overhead in time and space also increases the amount of alive kernel state (i.e., the "attack surface" for transient faults), so that the number of SDCs can actually *increase*. Implementation glitches that manifest only on compiler- or ISA-level can easily lead to loopholes for transient faults in algorithms that we considered as robust [15]. So the third rule of fault-avoiding operating-system development is: ❸ **Assess the actual impact of any dependability measure early and often.**

During the development of *d*OSEK, we operationalized these rules by a number of design and implementation principles that we describe in the following sections.

B. Continuous Integration with Fault-Injection Experiments

Our development process comprises detailed code reviewing together with continuous integration and unit testing, as it is state of the art for safety-critical projects. We additionally integrated automated fault-injection campaigns into this process to uncover the (often counter-intuitive) dependability impact of design decisions and implementation patterns (rule ❸). Such a closely connected development cycle can prevent the selection of adverse measures from the very beginning and assist the developer to constantly improve the realized measures for the concrete architecture.

C. Completely Static System Design

Most RTOSs are *dynamic* in the sense that all resource allocation (of tasks, semaphores, events, ...) takes place at run time. These systems are designed to handle a virtually unlimited amount of system objects – even though in hard real-time settings their number is bounded anyway. However, such an approach makes it easy to resemble the established and well-known POSIX system interface, which generally leads to pointer-based data structures, such as linked lists.

The automotive industry took another route when developing the OSEK [27] standard, which defines an RTOS interface that leverages a *fully static* system design. Here, all system objects have to be known already at compile-time: A system configuration file [26] describes all tasks including their

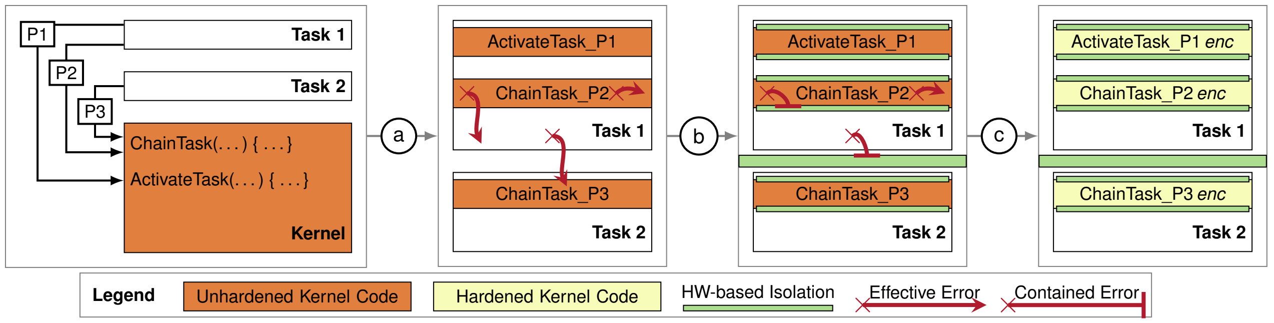


Fig. 1: Conceptual overview of the applied measures. System calls are partially specialized and inlined, with the corresponding parameters propagated as constants (Step a). Tasks are separated from each other – and the inlined OS code – with hardware-based isolation (Step b). Thus, errors are contained and cannot affect external components. Tailored, fine-grained fault-detection is applied to the kernel code itself, enabling full fault containment even within the kernel execution (Step c).

priorities, blocking behavior, potentially taken resources, and alarm usage. This makes it possible to condense all kernel state ahead of time in statically allocated data structures. System objects can be addressed by constant identifiers, which values are assigned at system generation time. This generally leads to lower memory footprints and kernel run time (rule ❶).

The main point of the static system design is, however, that it provides a considerable amount of *a priori* knowledge regarding the *application–kernel* interactions. This facilitates extensive dependability-oriented tailoring of the operating system. In *d*OSEK, we use this information mainly for partial specialization of system calls: Our system generator specializes each system call *per invocation* to embed it into the particular application code (Step a in Figure 1). As system objects are commonly referred to by their constant identifiers, this facilitates an aggressive folding of parameter values into the code. Therefore, less state needs to be passed in volatile registers or on the stack (rule ❶). Our continuous fault-injection experiments (rule ❸) revealed that this leads to significant robustness improvements.

Figure 2 depicts the resulting state of this analysis by the example of a system consisting of three tasks and two alarms: The remaining volatile state variables are subsumed under the blocks *Application*, *Architecture*, and *Kernel*. The architecture-independent minimal *Kernel* state is condensed to two machine words for the current task's priority, its id, and one machine word per task for the task's dynamic priority according to the priority ceiling protocol. Depending on the requirements of the application, the kernel maintains the current state of additional resources: in this case two alarms (three machine words each) and one counter (one machine word). The *Architecture* blocks are related to the dispatching mechanism of the underlying processor. In case of the IA-32, this is reduced to the administration of one stack pointer per task.

D. Avoidance of Indirections

Besides the obviously crucial OS state, we identified different kinds of indirection as a major catalyst for SDCs. These include not only obvious pointers or function calls, as defined by the programming language, but also more unapparent indirections caused by the underlying hardware architecture (e.g., stack pointer or interrupt context). Following rule ❷, we aimed to eliminate as much functionally redundant code paths, data, and indirections, as possible.

1) Data-Flow Indirections: Data structures stored in main memory achieve their dynamic expressiveness from pointers that can address any other structure and substructure in memory. Regular pointers have an unbound pointing-to range. By indirect memory accesses, arbitrary memory locations can be reached. This unbound expressiveness causes pointers to be one of the main sources of SDCs. Therefore the avoidance of data structure indirection is crucial.

We achieved a pointer-less design by allocating all system objects statically as global data structures, with the help of a generator. In occasions where pointers would be used to select one object out of multiple possible candidates an array at a constant address with small indices is preferred (rule ❷). The most frequently used (but far less visible) pointers are the stack pointer and the base pointer. Albeit less obvious, they are significant: A corrupted stack pointer influences all local variables, function arguments and the return address. Here, we eliminated the indirection for local variables by storing them as static variables at fixed, absolute addresses, while keeping isolation in terms of visibility and memory protection. This design decision required extra efforts for the implementation of reentrant functions; in our current implementation we cope with this by uninterruptible kernel execution. Also recursive functions are prohibited, which, however, shall be avoided inside the kernel anyway.

2) Control-Flow Indirections: A function call pushes arguments and the return address onto the stack and jumps to another code block. The return address induces indirection controlling the future control flow and, even worse, is stored near to other indirectly accessed values on the stack. Additionally, all registers that might be modified by the callee have to be stored on the stack. These indirections lead to a significant attack surface for errors – introduced by every single function call.

As already sketched above, *d*OSEK copes with this by partial specialization and aggressive inlining of system calls, which effectively avoids these indirections: Spilling arguments, register values, and especially the return address onto the stack is no longer necessary; argument constraints known to the compiler, especially constant values, can be propagated through the function body. The resulting code is not only faster, but

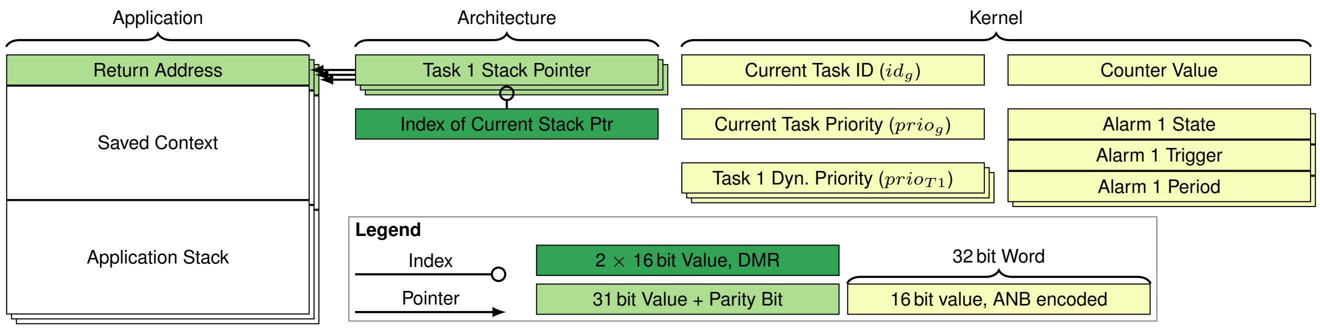


Fig. 2: Overview of the OS data kept in RAM of an example system composed of three tasks and two alarms. Each box represents a 32 bit memory location. All kernel data are hardened using an ANB-Code. The remaining application- and architecture-specific values are safeguarded by *double modular redundancy* (DMR) or parity bits.

especially more robust, as a high amount of memory or register accesses can be eradicated (rule ❶).

The code generator also unrolls all loops over the static system structures. While this, again, increases the code size, it also facilitates compiler optimization and prevents many possible errors, such as endless looping because of a bit-flip in an index.

Further indirections are caused by the use of hardware-based isolation: Traps, as the standard mechanism to cross the *user* ↔ *kernel* boundary, typically induce several implicit indirections via the stack pointer or link register. Where available, *d*OSEK uses architecture-specific instructions that omit this intense stack usage and give the programmer more explicit control.[4]

E. Bidirectional Generative Approach

The aggressive inlining and loop unrolling is not only realized implicitly by the compiler, but it is particularly supported by our system generator. Generating data structures for static operating systems is well known and widely used to exploit static application knowledge for reducing the resource consumption. *d*OSEK proceeds here even further and tailors the kernel code very precisely to the needs of the concrete application as well as the underlying hardware architecture (rule ❶).

1) Application-Oriented Generation: The generator not only implements an architecture specialization for each system call, but also specializes each *system call site* (Figure 1a). This allows exploiting more static information, since the specialized system service can be tailored to each single system call site. All function activations within the system call instance are forcefully inlined. The result is a loop-free, call-free kernel fragment. This fragment could be completely inlined into the application. By using inline traps [9], we were able to even keep up the spatial isolation.

2) Hardware-Oriented Generation: Usually, core kernel functionality and the hardware abstraction are separated into layers. Interaction between those layers does not reveal value in itself, but helps the OS developer to have a clean design. But this interaction leads to indirection in terms of long variable life spans, function calls and unnecessary conditional branches. Therefore, in *d*OSEK, these layers are only logical; they are separated in the generator itself and not in the emitted code. In the resulting *d*OSEK instance, these layers are mangled indistinguishable together to avoid indirection. This is a long-known principle for efficient kernel implementations: “It is the system *design* which is hierarchical, not its implementation.” [13]

V. Fault-Detecting Operating-System Implementation

*d*OSEK’s *fault-detection* strategies split up into two complementary concepts: First, coarse-grained hardware-based fault-detection mechanisms, mainly separating tasks from each other, and from invalid accesses by the kernel. Second, fine-grained software-based concepts that protect the kernel-internal data/control-flows.

A. Integration of Hardware-based Isolation

Hardware-based isolation mechanisms are widely used and a proven dependability measure. Watchdog concepts and MPUs play an important role for safety-related aspects. To a large extent, transient faults can be contained within individual components and reveal a significant amount of possible SDCs, as also shown in previous evaluations [14]. Consequently, *d*OSEK integrates the underlying architecture’s mechanisms into its system design, leveraging a coarse-grained fault detection among tasks and the kernel. With our completely generative approach, all necessary MPU configurations can be derived already at compile time and placed in robust ROM. Besides the obvious separation of tasks from each other, *d*OSEK especially uses the MPU to protect tasks from faults inside the kernel (Figure 1, step b).

B. Safeguarded Kernel Execution

The execution of the *d*OSEK kernel itself is hardened with a fine-grained arithmetic encoding (see also Figure 1, step c). All kernel data structures are safeguarded using a variant of an AN-code [12] capable of detecting both data- and control-flow errors. The code provides a constant common key A, allowing to uncover errors when calculating the remainder, and a variable-specific, compile-time constant signature B_n detecting the mix-up of two encoded values as well as the detection of faulty control flows – the ANB-Code:

[4]For example `sysenter/sysexit` on Intel IA-32 (64) processors.

$$\underset{\text{Encoded value}}{\underset{\uparrow}{n_{enc}}} \;=\; \underset{\text{Key}}{\underset{\nwarrow}{A}} \;\cdot\; \underset{\text{Value}}{\underset{\nwarrow}{n}} \;+\; \underset{\text{Signature}}{\underset{\nwarrow}{B_n}}$$

A particular feature of arithmetic codes is a set of code-preserving arithmetic operations, which allow for computation with the encoded values. Thus, counter and alarm components can be directly realized as fully encoded operations, as indicated in Figure 2. As a result, a continuous sphere of redundancy is spanned, as the corresponding operands remain encoded throughout the entire kernel execution.

1) Encoded Scheduler: In addition to the existing elementary arithmetic operations, *d*OSEK also requires an encoded variant of the mandatory scheduling algorithm. Here, an automated conversion of a plain algorithm to an ANB-encoded variant would be possible [35]. However, such a generic implementation would induce an immense overhead in terms of code and run time. Instead, we realized a distinct operation tailored to the specific demands of the scheduling algorithm, similar to our proven concept of the *Combined Redundancy* (*CoRed*) voter [15], [34].

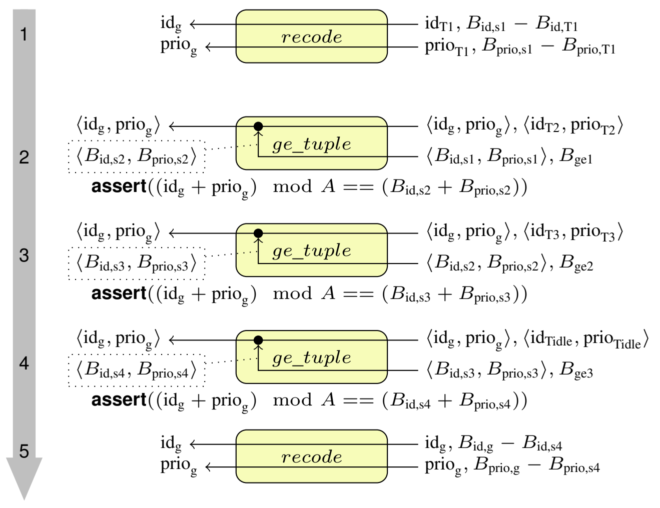
The encoded scheduler is based on a simple prioritized task list. Each tasks' current dynamic priority is stored at a fixed location (see also Figure 2), with the lowest possible value, an encoded zero, representing the suspended state. To determine the highest-priority task, the maximum task priority is searched by comparing all task priorities sequentially.

Thus, the algorithm's complexity in space and time is linear to the constant number of tasks. Figure 3 shows the basic concept for three tasks: The sequence processes a global tuple of ANB-encoded values storing the current highest-priority task id found so far, and the corresponding priority ($\langle \mathrm{id_g}, \mathrm{prio_g} \rangle$, see Figure 2). Sequential compare-and-update operations, based on an encoded greater-equal decision on a tuple of values (ge_tuple), compare the tuples' priority value and update the global values, if necessary. The sequence consists of five steps, as shown in Figure 3:

1) Initialize $\mathrm{prio_g}$ and $\mathrm{id_g}$ to the first task.
2–3) For all further tasks, compare the task's priority to $\mathrm{prio_g}$: If greater or equal, update $\langle \mathrm{id_g}, \mathrm{prio_g} \rangle$.
4) Repeat the last step for the idle task.
5) Recode the results to their original signatures.

The idle task priority is constantly bound to an encoded zero that is representing a suspended state. Thus, if all previous tasks are suspended, the last comparison (in step 4) will choose the idle task halting the system until the next interrupt.

Aside from the actual compare-and-update operation on fully encoded values, the ge_tuple function additionally integrates control-flow error detection. For each step, all signatures of the input operands ($B_{id,s1..s4}, B_{prio,s1..s4}$) and the signature of the operation itself ($B_{ge1..4}$) are merged into the resulting encoded values of the global tuple. Each corresponding signature of a step is then applied in the next operation accordingly. Thus, the dynamic values of the result tuple accumulate the signatures of all preceding operations. As the combination of these compile-time constant signatures is known before run time, interspersed assertions can validate the correctness of each step. Even after the final signature recode



1
$\mathrm{id_g}$
$\mathrm{prio_g}$
recode
$\mathrm{id_{T1}}, B_{\mathrm{id,s1}} - B_{\mathrm{id,T1}}$
$\mathrm{prio_{T1}}, B_{\mathrm{prio,s1}} - B_{\mathrm{prio,T1}}$
2
$\langle \mathrm{id_g}, \mathrm{prio_g} \rangle$
$\langle B_{\mathrm{id,s2}}, B_{\mathrm{prio,s2}} \rangle$
ge_tuple
$\langle \mathrm{id_g}, \mathrm{prio_g} \rangle, \langle \mathrm{id_{T2}}, \mathrm{prio_{T2}} \rangle$
$\langle B_{\mathrm{id,s1}}, B_{\mathrm{prio,s1}} \rangle, B_{\mathrm{ge1}}$
assert$((\mathrm{id_g} + \mathrm{prio_g}) \mod A == (B_{\mathrm{id,s2}} + B_{\mathrm{prio,s2}}))$
3
$\langle \mathrm{id_g}, \mathrm{prio_g} \rangle$
$\langle B_{\mathrm{id,s3}}, B_{\mathrm{prio,s3}} \rangle$
ge_tuple
$\langle \mathrm{id_g}, \mathrm{prio_g} \rangle, \langle \mathrm{id_{T3}}, \mathrm{prio_{T3}} \rangle$
$\langle B_{\mathrm{id,s2}}, B_{\mathrm{prio,s2}} \rangle, B_{\mathrm{ge2}}$
assert$((\mathrm{id_g} + \mathrm{prio_g}) \mod A == (B_{\mathrm{id,s3}} + B_{\mathrm{prio,s3}}))$
4
$\langle \mathrm{id_g}, \mathrm{prio_g} \rangle$
$\langle B_{\mathrm{id,s4}}, B_{\mathrm{prio,s4}} \rangle$
ge_tuple
$\langle \mathrm{id_g}, \mathrm{prio_g} \rangle, \langle \mathrm{id_{Tidle}}, \mathrm{prio_{Tidle}} \rangle$
$\langle B_{\mathrm{id,s3}}, B_{\mathrm{prio,s3}} \rangle, B_{\mathrm{ge3}}$
assert$((\mathrm{id_g} + \mathrm{prio_g}) \mod A == (B_{\mathrm{id,s4}} + B_{\mathrm{prio,s4}}))$
5
$\mathrm{id_g}$
$\mathrm{prio_g}$
recode
$\mathrm{id_g}, B_{\mathrm{id,g}} - B_{\mathrm{id,s4}}$
$\mathrm{prio_g}, B_{\mathrm{prio,g}} - B_{\mathrm{prio,s4}}$

Fig. 3: General sequence of the encoded scheduling operation on the example of three tasks (T1, T2, T3). All operations on signatures B are calculated already at compile time.

operation (step 5) any control-flow error is still detectable by the dynamic signature. Thus, the correctness of the encoded global tuple can be validated at any point in time. In effect, fault detection is ensured, as all operations are performed on encoded values. Further details on the ge_tuple function can be found in Appendix A.

2) Architecture-specific Dependability Measures: The remaining dynamic state highly depends on the underlying architecture. Regarding the currently implemented IA-32 variant, we were able to reduce this run-time state to an array storing the stack pointers of preempted tasks, and an corresponding index variable, as shown in Figure 2. The variables are used within each interrupt entry as well as during the actual dispatch operation. As they are not involved in any arithmetic calculations, but only read and written, we can avoid the overhead of the ANB-encoding in these cases and protect them by DMR or parity checks, respectively.

VI. Evaluation

During the development phase, frequent fault-injection campaigns were necessary to identify weak spots that were not covered by fault avoidance and protection mechanisms yet. For evaluating our progress in reducing *silent data corruptions* (SDCs), we compared various *d*OSEK variants in terms of code size, run time and SDC count against a mature OSEK implementation.

A. ERIKA Enterprise: An Open-Source OSEK

For comparison, we chose ERIKA Enterprise [1], an industry-grade (i.e., formally certified) open-source implementation of the automotive OSEK standard [27].[5]

Similar to *d*OSEK, ERIKA puts a distinct amount of read-only data into ROM sections to exploit the static system design.

[5]In previous work – before ERIKA Enterprise became available – we compared our solution against proprietary commercial implementations, which made it harder than necessary to reproduce our results.

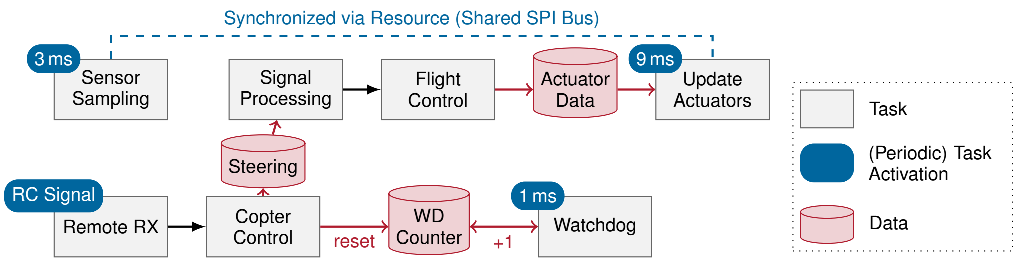


Fig. 4: Simplified representation of the *I4Copter* task and resource constellation used as evaluation scenario.

However, this optimization is mainly done for economizing main memory on small, embedded devices, instead of explicit dependability reasons.

B. *Evaluation Scenario: I4Copter*

The evaluation is based on a realistic system workload scenario considering all essential OS services. Here, we revive a setup, already presented in previous work [14], resembling a real-world safety-critical embedded system in terms of a quadrotor helicopter control application (cf. Figure 4). The scenario consists of eleven tasks, which are activated either periodically or sporadically by one of four interrupts. Inter-task synchronization is done with OSEK resources and a watchdog task, observing the remote control communication. We evaluated several variants of ERIKA and *d*OSEK, all running the same task set:

ERIKA (base) Standard version of ERIKA without sanity checking of system state.

ERIKA (checks) ERIKA with additional sanity checks that result in the call of an error hook. Calling the error hook is counted as a detected error.

***d*OSEK (base)** For the *d*OSEK base version only the indirection avoidance and the generative approach are used against SDCs. Code for spatial isolation is included, whereby the *memory protection unit* (MPU) is disabled.

***d*OSEK (mpu)** Spatial isolation is enabled. Protection faults are counted as detected errors.

***d*OSEK (enc)** The safeguarded kernel execution with encoded operations is enabled, MPU disabled.

***d*OSEK (enc+mpu)** Both, isolation mechanisms, as well as safeguarded kernel execution are in place.

The application flow is augmented with 172 checkpoints. Every OS under test executes the application for three hyper periods, while, at the same time a trace of visited checkpoints is recorded. It is the mission of the systems under test to reproduce this sequence, without corrupting the application state. If the sequence silently diverges in the presence of faults, we record a silent data corruption. The application state (task stacks) is checked for integrity at each checkpoint. To evaluate the fault containment within the kernel execution, we further recorded an SDC in case of violated integrity. Both SDC detection mechanisms were realized externally by the fault-injection framework without influencing the run-time behavior of the systems under test.

The aforementioned external SDC detection and fault injection is implemented with the FAIL* [30] framework. Its elaborate fault-space pruning techniques allow to cover the *entire* space of effective faults, while keeping the total number of experiments manageable. Since FAIL* has the most mature support for IA-32, we choose this architecture as our evaluation platform. The evaluated fault space includes all *single-bit faults* in the *main memory*, in the *general-purpose registers*, the *stack pointer and flags registers*, as well as the *instruction pointer*. This represents the architectural view from the software's perspective. In this evaluation we did not consider lower levels, including caches and pipelines, as they are not simulated by the underlying emulator[6] We focused the evaluation on the generally accepted single-error single-bit assumption. This means that for a certain experiment, a fault occurs at any point in time, but only once and is limited to a single register or memory word. Summarizing all variants, the total effective fault space covers $4.95 \cdot 10^{11}$ single-bit faults (before pruning), with $3.52 \cdot 10^{7}$ actually conducted experiments (after pruning idempotent faults).

C. *Fault-Injection Results*

We can partition the experiment outcomes into three coarse categories. *Benign faults* are mitigated and do not influence the expected activation sequence. Another class of faults leads to *traps or isolation faults* and is handled explicitly by the OS. In the presented results, we focus on the third and most dangerous category: *silent data corruptions*.

All OS variants differ in code size, run time and memory consumption – parameters that directly influence the number of effective injected faults. Thus, to directly compare the robustness, independent of any other non-functional properties, we concentrate on the resulting absolute SDC count, which represents the number of cases in which the RTOS did not provide the expected functional behavior that is any alteration of the execution of the workload in terms of completeness, order, and activation time.

With the help of the simulation-based fault injection framework FAIL*, a robustness evaluation covering the entire effective fault space was done for every variant. Figure 5 shows, on a logarithmic scale, the absolute number of faults that led to an SDC. For each variant, the SDC count is given for three different fault-injection targets: the instruction pointer, general purpose registers including the stack pointer and flags, and the accessed main memory.

1) ERIKA: Most SDCs stem from faults in memory or in general-purpose registers. The memory faults lead to more

[6]Bochs - http://bochs.sourceforge.net/

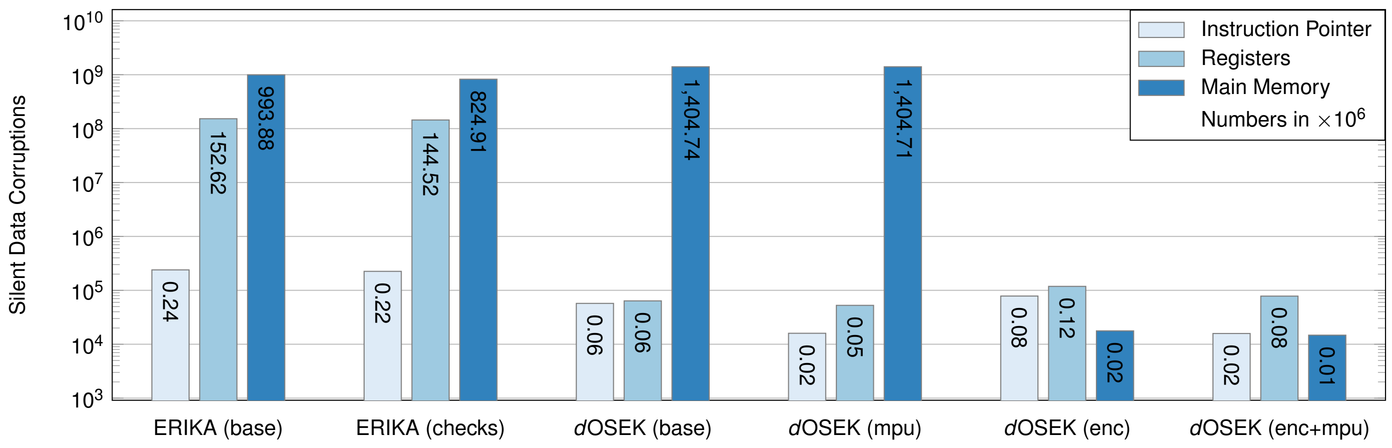


Fig. 5: SDC distribution for the evaluated variants of the *I4Copter* scenario on a logarithmic scale; pruned experiments are factored in. The encoded *d*OSEK system achieves an improvement in the SDC count by four orders of magnitude compared to ERIKA (base). Enabling memory protection *d*OSEK (enc+mpu) further halves the total amount of SDCs.

SDCs, since a comparably high amount of OS data structures lie unprotected in memory while the OS is inactive. Faults in the instruction pointer are a minor problem, as they lead to an invalid-instruction trap in most cases. The sanity checks in ERIKA (checks) are not intended for increasing the robustness but for detecting API usage errors and protocol violations; they still reduce the total SDC count by 15 percent.

2) dOSEK (base, mpu): In the unencoded variants of *d*OSEK, the memory SDC count for *d*OSEK is dominant and almost twice as large as in both ERIKA variants. This originates mostly from the fact that we did not optimize the unencoded scheduler but matched it to the structure of the encoded scheduler as accurately as possible. However, we already see that the SDCs in registers are significantly decreased compared to ERIKA – an effect of the avoidance of indirections. As already observed in previous work [14], the usage of an MPU does not improve the *overall* dependability significantly, when a pointer-less design was applied beforehand. Nevertheless, utilizing the MPU still decreases the instruction pointer errors by 72 percent.

3) dOSEK (enc, enc+mpu): By far, the safeguarded kernel execution of *d*OSEK (enc), especially the encoded scheduler, has the most effect on the SDC rate. It avoids nearly all SDCs in main memory, without increasing the other fault classes significantly. Although the encoding induces additional code, the SDC rate caused by instruction pointer faults is only slightly increased by 37 percent compared to the unprotected *d*OSEK (base) variant. This drawback can be effectively neutralized by additionally enabling the MPU (*d*OSEK enc+mpu). Compared to *d*OSEK (enc), the total SDC count is further halved.

As ERIKA does not provide MPU protection for this architecture yet, we compare only the non-MPU case of *d*OSEK to the best ERIKA variant and achieve an improvement in the SDC count by four orders of magnitude ($10^9 \rightarrow 10^5$).

D. Overhead

Since *d*OSEK relies on a generative approach performing *system call specialization*, it has to be noted that optimizing code size was no primary design goal. Likewise, the run-time overhead for the fully protected kernel, conducting complex arithmetic operations and regular checking, has to be examined. For reasons of clarity, the non-MPU variants of *d*OSEK are left out here, as they are identical in terms of code size and run time and just differ in a globally disabled or enabled MPU.

For the overhead rates (see Table I and Table II) we examined five kinds of system calls in our evaluation scenario. `ActivateTask (AT)` and `TerminateTask (TT)` are system calls that either start another task or terminate the current task, consequently calling the scheduler. The `GetResource (GR)` system call acquires a lock and enters a critical region in the application. As OSEK specifies a priority ceiling protocol, this operation never leads to a waiting state or to rescheduling. The critical region is left by `ReleaseResource (RR)`, which is again a point of rescheduling. Finally, the `SetRelAlarm (SR)` system call interacts with the alarm subsystem and sets up an alarm relative to the current time. Table I also shows the total size of the operating systems for the evaluated scenario.

Regarding *d*OSEK, the code size for each system call can be directly determined, since for each call site a distinct code fragment is generated. To make this measure comparable to ERIKA, which uses classical code reuse, the size of all visited functions during a system call was summed up. This approximates the aggressive inlining approach of *d*OSEK. Observing the run time for *d*OSEK, we have to keep in mind that switching the privileged mode is always done with a trap, even if the MPU protection is not enabled; *d*OSEK has a constant code size and run-time penalty here. The main cause of the high overhead is the encoded scheduler, which requires a high amount of arithmetic operations. Therefore the maximal overhead can be observed with `ActivateTask` (4.2×) and `TerminateTask` (2.4×). On the other hand, the fully protected `GetResource` in *d*OSEK even outperforms ERIKA – a fortunate side-effect of rules ❶ and ❷.

It is important to mention that the overall code overhead for *d*OSEK has to be multiplied by the number of system call sites, while the code size of ERIKA does not further increase with a higher number of system calls. However, the code size of

TABLE I: Code size per system call site in bytes. The total code size in ERIKA is the sum of all executed functions during a system call. *d*OSEK grows for every system call site (Δ / syscall) in the application. The total OS size for the evaluation system includes the remaining system calls not shown here.

Code size in bytes	ActivateTask (7 call sites)		TerminateTask (10 call sites)		GetResource (3 call sites)		ReleaseResource (3 call sites)		SetRelAlarm (1 call site)		*I4Copter* Benchmark
	Δ / syscall	Total	Δ / syscall	Total	Δ / syscall	Total	Δ / syscall	Total	Δ / syscall	Total	Total OS size
ERIKA (base)	–	548	–	933	–	148	–	408	–	314	3,782
ERIKA (checks)	–	661	–	959	–	207	–	389	–	383	3,782
*d*OSEK (mpu)	617	4,319	642	6,416	62	186	601	1,803	118	118	15,385
*d*OSEK (enc+mpu)	2,301	16,107	2,282	22,822	98	294	2,202	6,606	318	318	57,342

both systems grows when the number of system objects (tasks and alarms) increases due to the unrolling of code. The total code-size for the evaluation scenario is 57,342 bytes for *d*OSEK with MPU and encoded operations (worst case), while ERIKA used only 3,782 bytes (best case). On the other hand, caused by the condensed system state, *d*OSEK (enc+mpu) utilizes only 172 bytes of volatile kernel memory, while ERIKA requires 512 bytes.

The run-time costs are measured in executed instructions of the mere system calls, according to the execution trace of the golden run. As shown in Table II, the unencoded version of *d*OSEK compared to ERIKA (base) is increased by 56 percent in the worst case. The remaining results range from a factor of $0.6\times$ to $6.3\times$. Again the encoded scheduling operation makes up the upper bound of the run time in `ActivateTask` ($6.3\times$) and `TerminateTask` ($4.7\times$).

Summing up, *d*OSEK has a code size and run time penalty at each point of rescheduling, which even scales with the number of tasks. However, in a real-world setup, the comparably high run time of the application tasks would typically exceed the number of executed kernel instructions of all variants by multiple orders of magnitude.

TABLE II: Mean and worst-case run time of different system call activations in executed instructions.

Runtime	AT (max)	TT (max)	GR (max)	RR (max)	SR (max)
ERIKA (base)	60 (93)	93 (98)	43 (43)	56 (87)	99 (99)
ERIKA (checks)	61 (95)	95 (100)	47 (47)	59 (89)	100 (100)
*d*OSEK (mpu)	94 (107)	95 (104)	48 (48)	100 (104)	59 (59)
*d*OSEK (enc+mpu)	380 (456)	436 (451)	86 (86)	437 (449)	141 (141)

VII. Discussion and Future Work

Our experimental results show that system tailoring, indirection avoidance and extensive fault detection are the main keys to a robust OS acting as a reliable computing base.

A. Threats to Validity

Our main threat to internal validity is the IA-32 port of ERIKA, which was contributed by the same researchers that work on *d*OSEK. For the porting we had to implement a hardware abstraction layer (1,024 lines of code), which we derived from the corresponding *d*OSEK components and ERIKA templates. The port was carried out by two experienced OS engineers. It was reviewed and revised by the ERIKA maintainers and will become an integral part of the next official ERIKA release. Therefore, we are confident about the quality of our port as a comparison target.

The fact that we choose IA-32 as our evaluation platform exposes another threat to validity: IA-32 is not a commonly used architecture in automotive real-time systems and provides a powerful CISC instruction set that differs significantly from the strict load–store RISC model employed in most microcontrollers. This may impair the transferability of our results. The register-centric characteristics of a RISC-based architecture will clearly lead to a differing result distribution. We are currently working on a *d*OSEK variant for a RISC-based architecture (ARM Cortex-A9) which will allow to evaluate this in more detail. However, as we inject faults in ISA-accessible registers and memory and then test for correct results of complex instruction sequences, we believe that the net effect – an overall robustness improvement – will be similar on other platforms.

In our experiments, we assumed the watchdog timer, the MPU, and the interrupt controller to be free of faults. While the attack surface of most peripherals is relatively small (a few registers that are updated frequently), especially the current MPU state may be long-living and, thus, vulnerable. However, a bit flip inside the MPU that shrinks the accessible memory region does not pose a problem, as it can never lead to a SDC. But even if the memory protection is weakened by the fault, a SDC could only occur in conjunction with a second fault that causes the kernel to access exactly this accidentally extended memory region – a violation of the single fault assumption. The same line of argument holds for the watchdog timer: A fault inside the watchdog timer can only lead to a SDC if it coincidences with a second fault in the same hyper period.

The chosen single-error fault-model exposes an external threat to validity. Even though the single-bit assumption is assumed to cover 95 percent of the overall soft-error rate [21], [23] other researchers came to the conclusion that it can not be considered as realistic for real-world systems [25]. Fact is that "realistic" fault models are still a big topic of research, rarely available, and depend to a high degree on the (often oblique) internals of the hardware architecture. However, our chosen implementation measures, especially ANB-Codes, have shown to work also well in case of multi-bit faults [15]. Also we do not see how our design measures, especially avoidance of indirections, could potentially cause more harm than good with other fault models. Hence, we believe that our qualitative results regarding the reduction of SDCs also hold under more specific fault models. Quantifying these benefits, however, remains a topic of further research.

B. Benefits of the Approach

Our main contribution is the reduction of SDCs for a realistic evaluation scenario by four orders of magnitude ($10^9 \to 10^5$) compared to ERIKA. We see ERIKA as an adequate competitor, since it reveals a comparable number of SDC than our *d*OSEK (base) system. The usage of sanity checks in ERIKA does, against our first intuition, not decrease SDCs significantly. Further, the improvement of the safeguarded kernel execution is almost independent of MPU usage. Therefore, we conclude, that we can gain similar robustness improvements for very low-cost embedded systems that do not offer an MPU.

C. Limitations of the Approach

The current *d*OSEK system provides high reliability: The remaining 108,694 SDCs of the fully safeguarded *d*OSEK (enc+mpu) result from a total of 114,589,564,640 potentially effective bit flips in the evaluation scenario. Nevertheless, one may ask where these remaining SDCs occur and if they could, somehow, be detected as well?

We identified these remaining SDCs to be caused by faults that take place in (a) the unavoidable indirections imposed by the MPU and (b) the short vulnerable interval before the kernel is left, that is, when decoded values have to be used. The latter refers to the few instructions between restoring a decoded register value (for a context switch or a return-from-interrupt) up to the actual return instruction that leaves the kernel. In Appendix B we illustrate these issues on the example of the *d*OSEK context-switch code and a more detailed analysis of the remaining faults. In those architecture-induced and unavoidable intervals, we see the fundamental limitations of our work and purely software-based approaches in general.

However, a reduction of SDCs by a factor of 10,000 may already be good enough. If not, we are convinced that only relatively little hardware support is actually needed to cover the remaining cases as well. This is a subject of further research.

D. Real-Time and Functional Safety

*d*OSEK implements the OSEK RTOS [27] standard and fulfills all of its mandated real-time requirements, such as strict priority-based scheduling and the stack-based priority ceiling protocol. Because of its static design and the purely generative approach with loop unrolling, aggressive inlining, and avoidance of indirections, all operations are inherently bounded and *d*OSEK systems are particularly easy to analyze with respect to their timing properties.

However, predictable and timely task execution is only one aspect of functional safety in the domain of embedded control systems [17]. The system has to show predictable behavior also in the presence of transient hardware faults. By its high robustness, *d*OSEK provides this up to the level of the RTOS. It thereby establishes a *reliable computing base* (RCB) for the implementation of application-specific dependability and recovery mechanisms.

E. Future Work: Cross-Kernel Optimization

Currently, the main disadvantage of our approach is the high ROM overhead. The avoidance of indirections by aggressive inlining is especially expensive for the encoded scheduling operation. With respect to code-size overhead and robustness, we are currently working on whole-system optimizations for the kernel fragments we emit for each call site: The basic idea is that from the scheduling rules mandated by OSEK, one can often constrain or even predict the outcome of a scheduling operation *already at generation time*. For instance, an `ActivateTask()` on a higher-priority run-to-completion task could be replaced by a plain procedure call or even the complete inlining of this task, eradicating the need to embed the expensive scheduling code. Other points of rescheduling may be shortened by only considering tasks that are known to be potentially ready to run, according to the static analysis.

To enable such optimizations, we are currently working on the automatic static analysis of all inter-task interactions to obtain a whole-system model that offers a holistic, cross-kernel control-flow graph (CFG) for each core. First results of applying these techniques reveal promising improvements: For the encoded *I4Copter* scenario, we can thereby cut the code size by half. The run-time overhead induced by encoding is reduced from 450 percent down to 74 percent. Integrating the results of the cross-kernel CFG as run-time assertions, before and after system calls, further reduces the remaining SDCs by 50 percent in our evaluation scenario. Here, we plan to extend these assertions to a recovery mechanism detecting and repairing faulty kernel executions.

VIII. Related Work

While most work from the dependable systems community still assumes the OS itself to be too hard to protect, the topic of RTOS reliability in case of transient faults has recently gained attention: The C^3 μ-kernel tracks system-state transitions at the IPC level to be able to recover system components in case of a fault [33]. Their approach, however, assumes that the fault will be detected immediately, that is, there are no SDCs, and that the recovery functionality itself is part of the RCB. L4/Romain [10] employs full system-call interception to provide transparent thread-level TMR, which mitigates the detection issue, but still requires the μ-kernel to be reliable. The hypervisor approach of Quest-V [22] reduces the *software*-part of the RCB even further – at the price of increasing the *hardware*-part for the required virtualization support. In the end, however, all these approaches depend on the early and reliable detection of faults and their strict containment inside the RCB. Our approach provides exactly that.

The concept of AN-encoding has been known for quite a while [12]; it has been taken up in recent years in compiler- and interpreter-based solutions [29], [35]. Yet, these generic realizations are not practicable for realizing a RCB – not only due their immense run time overhead of a factor of 10^3 up to 10^5, but also due to the specific nature of low-level system software. Thus, following our proven CoRed concept [15], [34], we concentrate the encoded execution to the *minimal necessary points*, to keep the overhead on a bearable level.

IX. Conclusion

Current software-based dependability measures aim to span a continuous sphere of redundancy over the safety-critical applications. Yet, the effectiveness of these measures highly depends on a *reliable computing base* (RCB), in terms of a dependable operating system. *d*OSEK aims to provide this RCB and even extends the sphere of redundancy from the application throughout the kernel execution.

To achieve this, *d*OSEK provides a reliable *fail-stop* behavior avoiding *silent data corruptions* (SDCs) through three main principles: (1) A static system design revealing a-priori application knowledge, which enables fine-grained system tailoring. (2) Fault-avoidance strategies condensing the volatile system state, as well as reducing indirections, as far as possible. (3) Reliable fault detection of the remaining vulnerable state, in terms of coarse-grained isolation and fine-grained arithmetic encoding. By applying these principles rigorously, we achieved an SDC count reduction by four orders of magnitude, compared to an off-the-shelf OSEK operating system. *d*OSEK pays this robustness improvement, in terms of increased run time and code size. However, we are confident to reduce this overhead, and to improve the robustness even further, by more extensive system tailoring utilizing further detailed application knowledge.

Implementation and further details:

`https://www4.cs.fau.de/Research/dOSEK`

REFERENCES

[1] ERIKA Enterprise. http://erika.tuxfamily.org, visited 2014-09-29.

[2] Washington, DC, USA, June 2002. IEEE.

[3] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Experimental evaluation of time-redundant execution for a brake-by-wire application. In *32nd Int. Conf. on Dep. Systems & Networks (DSN '02)* [2], pages 210–215.

[4] C. Aussagues, D. Chabrol, V. David, D. Roux, N. Willey, A. Tornadre, and M. Graniou. PharOS, a multicore OS ready for safety-related automotive systems: Results and future prospects. In *4th Int. Conf. on Embedded Real Time Software and Systems (ERTS2 '10)*, May 2010.

[5] AUTOSAR. Specification of operating system (version 5.1.0). Technical report, Automotive Open System Architecture GbR, February 2013.

[6] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Lessons learned from microkernel verification – specification is the new bottleneck. In *Proceedings of the 7th Conference on Systems Software Verification (SSV '12)*, pages 18–32, November 2012.

[7] Robert C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, 2005.

[8] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *Micro, IEEE*, 23(4):14–19, July 2003.

[9] Daniel Danner, Rainer Müller, Wolfgang Schröder-Preikschat, Wanja Hofer, and Daniel Lohmann. Safer Sloth: Efficient, hardware-tailored memory protection. In *20th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '14)*, pages 37–47, Washington, DC, USA, 2014. IEEE.

[10] Björn Döbel and Hermann Härtig. Who watches the watchmen? – protecting operating system reliability mechanisms. In *8th Int. W'shop on Hot Topics in System Dependability (HotDep '12)*, Berkeley, CA, USA, 2012. USENIX.

[11] Michael Engel and Björn Döbel. The reliable computing base: A paradigm for software-based reliability. In *1st Int. W'shop on Softw.-Based Methods for Robust Emb. Sys. (SOBRES '12)*, LNCS. Gesellschaft für Informatik, September 2012.

[12] Ph. Forin. Vital coded microprocessor principles and application for various transit systems. In *Symp. on Control, Computers, Communication in Transportation (CCCT '89)*, pages 79–84, September 1989.

[13] Arie Nicolaas Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and hierarchy in a family of operating systems. *CACM*, 19(5):266–272, 1976.

[14] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In *17th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '14)*, pages 230–237. IEEE, 2014.

[15] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. A practitioner's guide to software-based soft-error mitigation using AN-codes. In *15th IEEE Int. Symp. on High-Assurance Systems Engineering (HASE '14)*, pages 33–40. IEEE, January 2014.

[16] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. *IEEE Transactions on Electron Devices*, 57(7):1527–1538, July 2010.

[17] IEC 61508-3. *IEC 61508-3: - Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*. International Electrotechnical Commission, Geneva, Switzerland, December 1998.

[18] F. Irom and D.N. Nguyen. Single event effect characterization of high density commercial nand and nor nonvolatile flash memories. *IEEE Transactions on Nuclear Science*, 54(6):2547–2553, Dec 2007.

[19] ISO 26262-9. *ISO 26262-9:2011: Road vehicles – Functional safety – Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*. ISO, Geneva, Switzerland, 2011.

[20] Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *1st Int. W'shop on Microkernels for Embedded Systems*, 2007.

[21] Xin Li, Kai Shen, Michael C Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *2007 USENIX ATC*, pages 1–14, Berkeley, CA, USA, 2007. USENIX.

[22] Ye Li, Richard West, and Eric Missimer. A virtualized separation kernel for mixed criticality systems. In *10th USENIX Int. Conf. on Virtual Execution Environments (VEE '14)*, pages 201–212, New York, NY, USA, 2014. ACM.

[23] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong. Characterization of multi-bit soft error events in advanced SRAMs. In *Intern. Electron Devices Meeting (IEDM '03)*, pages 21.4.1–21.4.4, New York, NY, USA, 2003. IEEE Press.

[24] Shubu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann, San Francisco, CA, USA, 2008.

[25] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In Christoph M. Kirsch and Gernot Heiser, editors, *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2011 (EuroSys '11)*, pages 343–356, New York, NY, USA, April 2011. ACM.

[26] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2014-09-29.

[27] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29.

[28] Ute Schiffel. *Hardware Error Detection Using AN-Codes*. PhD thesis, Technische Universität Dresden, Fakultät Informatik, 2011.

[29] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-encoding: detecting hardware errors in software. In Erwin Schoitsch, editor, *29th Int. Conf. on Comp. Safety, Reliability, and Security (SAFECOMP '10)*, pages 169–182, Heidelberg, Germany, 2010. Springer.

[30] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a versatile fault-injection experiment framework. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *25th Int. Conf. on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, volume 200 of *Lecture Notes in Informatics*, pages 201–210. Gesellschaft für Informatik, March 2012.

[31] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *32nd Int. Conf. on Dep. Systems & Networks (DSN '02)* [2], pages 389–398.

[32] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *37th Int. Conf. on Dep. Systems*

& Networks (DSN '07), pages 297–306, Washington, DC, USA, June 2007. IEEE.

[33] Jiguo Song, J. Wittrock, and G. Parmer. Predictable, efficient system-level fault tolerance in C^3. In *34th IEEE Int. Symp. on Real-Time Sys. (RTSS '13)*, pages 21–32. IEEE, December 2013.

[34] Peter Ulbrich, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Reiner Schmid. Eliminating single points of failure in software-based redundancy. In *9th Eur. Dep. Computing Conf. (EDCC '12)*, pages 49–60, Washington, DC, USA, May 2012. IEEE.

[35] Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In Francesca Saglietti and Norbert Oster, editors, *26th Int. Conf. on Comp. Safety, Reliability, and Security (SAFECOMP '07)*, pages 356–369, Heidelberg, Germany, 2007. Springer.

[36] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime asynchronous fault tolerance via speculation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 145–154, New York, NY, USA, 2012. ACM.

[37] Yun Zhang, JaeW. Lee, NickP. Johnson, and DavidI. August. Daft: Decoupled acyclic fault tolerance. *International Journal of Parallel Programming*, 40(1):118–140, 2012.

APPENDIX A
IMPLEMENTATION: $ge\_tuple()$

The encoded *greater-equal-tuple* operation (Figure 6) is the main component the encoded scheduler is built upon. It compares two encoded tuples, consisting of task ID and dynamic task priority, and overwrites the first tuple with the higher-priority input tuple. During its operation, it recodes the encoded first tuple from the signature pair $\langle B_{\text{id},1}, B_{\text{prio},1}\rangle$ to the new signatures $\langle B_{\text{id},2}, B_{\text{prio},2}\rangle$. The new signatures incorporate all incoming signatures to detect control-flow errors.

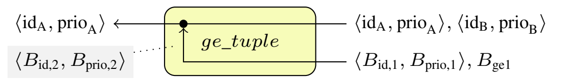

$\langle \text{id}_A, \text{prio}_A \rangle$
$\langle B_{\text{id},2}, B_{\text{prio},2} \rangle$
ge_tuple
$\langle \text{id}_A, \text{prio}_A \rangle, \langle \text{id}_B, \text{prio}_B \rangle$
$\langle B_{\text{id},1}, B_{\text{prio},1} \rangle, B_{\text{ge}1}$

Fig. 6: Basic operation of $ge\_tuple()$

The ge_tuple implementation (Figure 7) is based on the ANB less-or-equal operation as presented by Schiffel [28, p. 68]. Instead of returning an encoded boolean comparison result, the ge_tuple method is extended to change both the compared variable (prio_A) and a corresponding variable (id_A).

```
 1: $\text{diff} \leftarrow (\text{prio}_B - \text{prio}_A)$
 2: $\text{sig}_{\text{cond}} \leftarrow \text{diff} \mod A$
 3: $B_{\text{pos}} \leftarrow B_{\text{prio,B}} - B_{\text{prio},1}$
 4: $B_{\text{neg}} \leftarrow (2^{32} + B_{\text{pos}}) \mod A$

 5: $\text{comp} \leftarrow ((\text{prio}_A - B_{\text{prio},1}) \leq (\text{prio}_B - B_{\text{prio,B}}))$
 6: if comp then
 7:     $\text{prio}_A \leftarrow B_{\text{prio},1} + (\text{prio}_B - B_{\text{prio,B}}) + (B_{\text{ge}1} - B_{\text{pos}})$
 8:     $\text{id}_A \leftarrow B_{\text{id},1} + \text{id}_B + B_{\text{ge}1}$
 9: else
10:     $\text{prio}_A \leftarrow \text{prio}_A + (B_{\text{pos}} - B_{\text{neg}}) + (B_{\text{ge}1} - B_{\text{pos}})$
11:     $\text{id}_A \leftarrow \text{id}_A + (B_{\text{pos}} - B_{\text{neg}}) + B_{\text{id,B}} + B_{\text{ge}1}$
12: end if

13: $\text{prio}_A \leftarrow \text{prio}_A + \text{sig}_{\text{cond}}$
14: $\text{id}_A \leftarrow \text{prio}_A + \text{sig}_{\text{cond}}$

15: $B_{\text{prio},2} \leftarrow B_{\text{prio},1} + B_{\text{ge}1}$
16: $B_{\text{id},2} \leftarrow B_{\text{id},1} + B_{\text{ge}1} + B_{\text{pos}} + B_{\text{id,B}}$
```

Fig. 7: The $ge\_tuple()$ algorithm updates $\langle \text{id}_A, \text{prio}_A \rangle$ with the maximum task priority and ID

The algorithm consists of three steps: It calculates a signature sig_{cond} from the incoming priorities prio_A and prio_B. The actual comparison is unencoded, but changes the result tuple $\langle \text{id}_A, \text{prio}_A \rangle$. If the new priority prio_B is greater or equal than prio_A, we update $\langle \text{id}_A, \text{prio}_A \rangle$ to $\langle \text{id}_B, \text{prio}_B \rangle$, otherwise only the signatures are adjusted.

Line 1–4 For the encoded *if* condition, it is necessary to derive a signature sig_{cond} from the input values, which has a known, fixed constant value for both the positive (B_{pos}) and negative (B_{neg}) comparison result.
This signature is based on the difference of prio_A and prio_B (line 1). As such a subtraction of unsigned 32 bit integers can underflow, the following values can result:

$$\text{diff} = \begin{cases} \text{prio}_B - \text{prio}_A, & \text{prio}_A \leq \text{prio}_B \\ 2^{32} - (\text{prio}_A - \text{prio}_B), & \text{prio}_A > \text{prio}_B \end{cases}$$

The signature sig_{cond} is the remainder of this difference divided by the encoding constant A (line 2). This way, the resulting values only depend on the static signatures, not actual run-time values:

$$\text{sig}_{\text{cond}} = \begin{cases} B_{\text{pos}} = B_{\text{prio,B}} - B_{\text{prio},1}, & \text{prio}_A \leq \text{prio}_B \\ B_{\text{neg}} = (2^{32} + B_{\text{pos}}) \mod A, & \text{prio}_A > \text{prio}_B \end{cases}$$

As these values for B_{pos} and B_{neg} are constants, the calculations in line 3 and 4 are performed by the compiler, not at run-time. To prevent underflows in these values, the ge_tuple method requires that $B_{\text{prio,B}} > B_{\text{prio},1}$.

Line 5–6 The actual branch is chosen by an unencoded comparison of prio_A and prio_B after subtracting their static signatures. The resulting AN-encoded values can be compared directly, as they differ only by the factor A to the unencoded value.

Line 7–8 If the task's priority prio_B is greater or equal than the current maximum prio_A, the new value is stored in prio_A (line 7). The previous value of prio_A must be overwritten in this case, but any previous errors in this value would be propagated and detected through sig_{cond}.
To calculate the new value of prio_A, prio_B is stripped from its own signature. It is encoded with the current signature of id_A ($B_{\text{prio},1}$) and the ge_tuple-signature $B_{\text{ge}1}$ is added. If the maximum in prio_A is replaced, id_A must also be updated to id_B. As before, the new value id_A is the current signature ($B_{\text{id},1}$) and the full encoded value of id_B and the ge_tuple-signature $B_{\text{ge}1}$. The signatures $B_{\text{prio,B}}$ and B_{pos} are included in the new value for id_A, because then the output signatures include all input signatures, as required to detect errors.

Line 10–11 If the task's priority prio_B is less than the current maximum prio_A, the encoded values of prio_A and id_A remain unchanged and only their signatures are adjusted. Both values must subtract the expected value B_{neg} of the condition signature sig_{cond} and add the value of the other possible branch (B_{pos}). The unique working signature $B_{\text{ge}1}$ of this ge_tuple invocation is added as well.
For prio_A, no further adjustments are necessary as the signature of prio_B is subtracted in the other branch. For id_A however, the signature $B_{\text{id,A}}$ has to be added to result in the same static signature as the addition of id_B in the other branch.

Line 13–14 The calculations in the branches use the corresponding constant (B_{pos} or B_{neg}) in place of the dynamic sig_{cond} to derive a common result signature. After the control flow of both branches merges again, the condition signature sig_{cond} is added to both result values (prio_A and id_B). If the wrong branch was taken or a value was corrupted, this addition does not result in the expected value, which allows the error to be detected.

Line 15–16 After the algorithm is finished, the static signatures of prio_A and id_A are changed to include the compared values. These changes are performed at run-time in lines 7–8 or 10–11. The new signatures $B_{\text{prio},2}$ and $B_{\text{id},2}$ for the values for prio_A and id_A are known to the compiler afterwards.

Appendix B
Analysis of the Remaining Faults

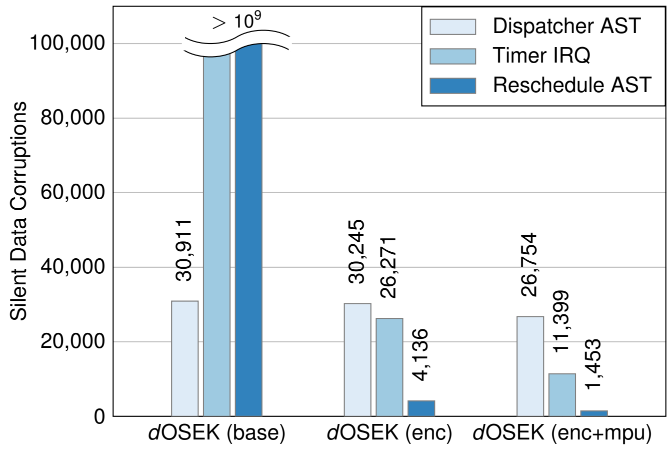


Fig. 8: SDCs plotted against operating-system component.

The remaining SDCs in *d*OSEK mostly stem from errors injected immediately before the kernel is left, that is, when *d*OSEK has to restore and use decoded register values.

Figure 8 shows the three most vulnerable fault locations of this type, which together cover 36 percent of the overall SDC rate of the *d*OSEK (enc+mpu) variant. Listing 1 exemplifies the issues with the *kernel* $\rightarrow$ *user* transition on the example of the dispatching trap handler. Presented is the assembler code as emitted by the compiler.[7]

Line 1–13 The dispatcher loads and decodes the ANB-encoded task id, which was elected by the scheduler beforehand (stored in id_g, see also Figure 1). Starting with line 13, the *decoded* id_g in register `%eax` is susceptible to errors, as indicted by the ragged line.

Line 14–17 The decoded task id is replicated and stored as a double modular redundant index to the current running task (see Figure 1). Any bit flip affecting register `%eax` during line 13 or 14 would corrupt both values and render the DMR stored value ineffective.

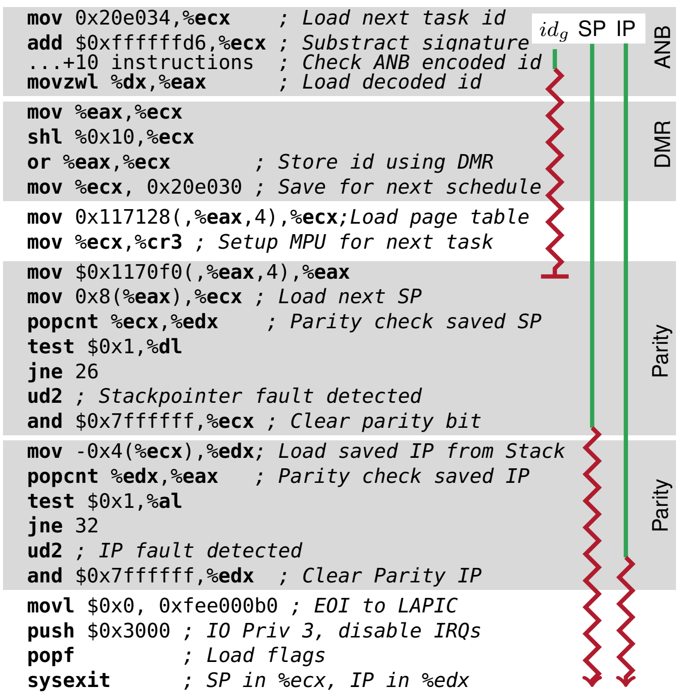
Line 18–19 The page table of the next task is loaded configuring memory protection – still, the plain id is used and vulnerable to faults.

Line 20–32 The decoded id is used to determine the stack pointer (SP) and instruction pointer (IP) of the task to be dispatched. After line 20, the life span of id_g ends and the SP and IP values are loaded. At this point, the parity checks of SP and IP can also detect a corrupted id_g that was used as indirection. Nevertheless, there is a 50 percent probability that a falsely loaded value still provides a valid parity bit by chance.

Line 22–36 End-of-interrupt (EOI) is signaled to the interrupt controller and the cpu flags are configured as needed by the task. The final `sysexit` switches over to the previously set instruction pointer (in `%edx`) and loads the stack pointer to the value provided in register `%ecx`. Again, as indicated by the ragged lines, these values are susceptible to errors which, in effect, would directly influence the control-flow (when affecting `%edx`) or data-flow (`%ecx`).

In theory (i.e., by manual micro-optimization), the remaining error-prone variable life spans (Listing 1) could be shortened by a few instructions. Also, the validity of the task id could be reevaluated right before its usage (e.g., before line 18) and the parity checks may be repeated right before the `sysexit`.

However, the fundamental limitation remains: The error-prone periods cannot be eliminated entirely – at some point we have to decode registers and addresses into the vulnerable format implied by the hardware. At this point, software-based dependability measures reach their limits. Nevertheless, these limits may not be severe: The remaining 108,694 SDCs of the fully safeguarded *d*OSEK (enc+mpu) result from a total of 114,589,564,640 potentially effective bit flips – less than 0.0001 percent.



```
 1: mov 0x20e034,%ecx    ; Load next task id
 2: add $0xffffffd6,%ecx ; Substract signature
 3: ...+10 instructions  ; Check ANB encoded id
13: movzwl %dx,%eax      ; Load decoded id
14: mov %eax,%ecx
15: shl %0x10,%ecx
16: or %eax,%ecx       ; Store id using DMR
17: mov %ecx, 0x20e030 ; Save for next schedule
18: mov 0x117128(,%eax,4),%ecx;Load page table
19: mov %ecx,%cr3 ; Setup MPU for next task
20: mov $0x1170f0(,%eax,4),%eax
21: mov 0x8(%eax),%ecx ; Load next SP
22: popcnt %ecx,%edx    ; Parity check saved SP
23: test $0x1,%dl
24: jne 26
25: ud2 ; Stackpointer fault detected
26: and $0x7ffffff,%ecx ; Clear parity bit
27: mov -0x4(%ecx),%edx; Load saved IP from Stack
28: popcnt %edx,%eax   ; Parity check saved IP
29: test $0x1,%al
30: jne 32
31: ud2 ; IP fault detected
32: and $0x7ffffff,%edx  ; Clear Parity IP
33: movl $0x0, 0xfee000b0 ; EOI to LAPIC
34: push $0x3000 ; IO Priv 3, disable IRQs
35: popf         ; Load flags
36: sysexit      ; SP in %ecx, IP in %edx
```

Listing 1: *d*OSEK dispatching code. The serrated lines indicate periods where the relevant values (task id, stack pointer, instruction pointer) are decoded and prone to errors.

[7]clang/llvm version 3.4 [-O2 -inlinehint-threshold=∞]. The *d*OSEK source code can be found at https://github.com/danceos/dosek