Return-Address Protection in C/C++ Code by Dependability Aspects [*]

Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk

Department of Computer Science 12, TU Dortmund, Germany
e-mail: firstname.lastname@tu-dortmund.de

Abstract: Due to shrinking structure sizes on memory chips, the probability of memory failures, such as spontaneous bit flips, is increasing. Especially in the domain of mass-produced cheap embedded systems, hardware solutions are not affordable. Therefore, there is a need for cost-efficient software-based fault-tolerance mechanisms. In this paper we focus on such a mechanism for the protection of CPU stacks. A baseline assessment conducted with 21 benchmark and test programs shows that the stack is the most fault-susceptible data memory region – even more critical than the OS kernel's scheduler data structure, for instance. Our mechanism, which is based on profiling and a generic aspect-oriented implementation, supports detection and correction of bit flips in return addresses and frame pointers on the stack. It thereby reduces the number of stack-related program failures by 48.7 % and the number of all RAM-related failures by 13.3 % on the average over all benchmarks. The average code size overhead is 3.76 %, and a runtime overhead is only measurable for the subset of short-running benchmarks.

1 Introduction

Errors in main memory are a major cause of failures in today's computer systems [SPW09, NDO11, HSS12]. This problem is expected to worsen in the future [Bau05], as VLSI technologies move to higher chip densities and lower operating voltages, dramatically increasing sensitivity to electromagnetic radiation.

In previous works we have shown that *software-based* fault-tolerance mechanisms that are directed by application-specific knowledge can detect and correct a significant share of memory errors while being efficient in terms of memory consumption and runtime overhead [BSS13, BSS12]. The work presented in this paper follows the same approach, but focuses on CPU stacks. Our contribution is the description of a novel aspect-oriented fault-tolerance mechanism for stack protection and its thorough evaluation.

As virtually all high-level programming languages support the nested invocation of reusable code fragments, e.g., functions, methods, or procedures, today's CPUs have a built-in awareness of a call stack and special instructions for its manipulation. For instance, x86 CPUs have a `call` and a `ret` instruction as well as a stack pointer register `ESP`. Whenever

[*]This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. SP 968/5-2.

a function is called, the `call` instruction decrements the stack pointer, saves the current instruction pointer (`EIP`) – the *return address* – on the stack, and loads the address of the target function into the instruction pointer register. Typically being the last instruction of a called function, the `ret` instruction reads the return address from the stack, increments the stack pointer, and loads the address into the instruction pointer. Besides saving return addresses, the stack on x86 is also used for function call parameters and local variables. This leads to a stack-memory layout that consists of so-called stack frames – one for each active function. On x86, the register `EBP` is used to reference parameters and variables in the stack frame of the current function. `EBP` (also called *base pointer* or *frame pointer*) contains a copy of the `ESP` register created at the beginning of a function execution. To be able to restore the previous `EBP` register value (directly before returning with `ret`), it is saved on the stack as well. Thereby, all stack frames are linked. Being a dynamic linked data structure that stores code addresses, it is no surprise that bit flips can cause program misbehavior or even crashes.

Other CPU types than x86, especially RISC CPUs, have dedicated CPU registers for saving the return address, and also for passing parameters and holding local variables. However, these registers only help the leaf function in the function call tree. Other functions need to use an in-memory stack as on x86. Therefore, these architectures can be expected to be less susceptible to memory errors, but failures caused by bit flips in the stack still exist.

The remaining sections of this article are organized as follows: After a brief discussion of related work in Section 2, we will present an initial dependability assessment with one unmodified benchmark program in Section 3. After this motivation, Section 4 will explain our approach that combines *Profiling* with *Dependability Aspects*. The evaluation of the approach is presented in Section 5. The paper closes with a discussion of our results and the conclusions in Sections 6 and 7.

2 Related Work

Much research has been done in the area of security. Stack frames are a common target of security attacks that manipulate the stack to hijack the path of execution by overwriting a return address. *StackShield*[1] and *Return Address Defender* [CH01], for example, are compiler extensions that save a copy of the return address on a redundant stack. The necessary CPU instructions are inserted after a function is called and before it returns. Both compiler extensions compare the copy to the original return address and raise an exception on mismatch, but do not recover from such events transparently. *StackGuard* [CPM+98] and the GCC's *stack-smashing protector*[2] insert a special marker value (*canary*) between local variables and the return address on the stack. If a local variable overflows due to a security attack, the canary will be corrupted with certainty. Before a function returns, the canary is checked and when it is invalid, the program terminates. Such an indirect return address protection is not effective against any bit errors in the return address itself,

[1] `http://www.angelfire.com/sk/stackshield`

[2] `http://www.research.ibm.com/trl/projects/security/ssp/`

as only the canary is checked. *Libverify* [BST00] is a binary translation library for Linux applications. It relocates the original functions to the heap at runtime and incorporates protection code, which doubles the code size due to the relocation. In many embedded scenarios, such a solution cannot be afforded. Thus, to protect efficiently against hardware memory errors, none of these security solutions are applicable.

Aspect-oriented programming (see Section 4.1), in particular AspectC++, has been used to implement fault-tolerance measures. Afonso has described several aspect-oriented idioms for improving fault-tolerance in embedded OS code [ASMT07], but does not address stacks. Alexandersson et al. have implemented control-flow checking with AspectC++ that detects illegal control-flow transitions [AK11]. However, the validation takes place *after* a function has returned, so that errors cannot be prevented upfront. Our approach differs in that we implement a proactive recovery, which prevents failures *before* they happen.

3 Scenario and Problem Analysis

In this section we outline the application scenario we use to analyze the impact of bit flips in the CPU stack on a system's dependability. The results motivate our approach and are used to identify requirements. Finally, the idea behind the approach is sketched.

3.1 Baseline Dependability Assessment

In this study, we use the *embedded Configurable operating system (eCos)* [Mas02], implemented in C/C++, as a realistic scenario to analyze the fault susceptibility of various benchmarks bundled with eCos itself. In previous work [BSS13] we analyzed this scenario using fault-injection (FI) experiments and a single-bit flip fault model. In the following, we repeat a similar analysis, and subsequently concentrate on faults on the application per-thread and kernel stacks[3].

In order to determine the target system's susceptibility to faults in data memory (data, BSS, and stack segments), we injected 8-bit burst faults using FAIL* [SHK$^+$12], i.e., we consider program runs in which all bits in a single byte flip at some point in time. Our analysis in [BSS13] showed that the results are similar to those obtained with the common single-bit flip model (at least for the eCos benchmark set), but require eight times less experiments.

Table 1 shows the top five most susceptible symbols of the MUTEX1 benchmark, i.e., the memory areas that most often lead to a program failure when hit by a fault. It seems not surprising that the system's stack ranks first (42.2 % of all program failures originate in faults injected in the stack memory), as a large memory area naturally offers a big target surface, but in fact only 69.0 % of this stack are used at all. For the remainder of this paper, we ignore the other symbols, the issues that lead to failures there, and possible remedies; some of these we already addressed in previous work [BSS12, BSS13].

[3]Most of the benchmarks are multi-threaded, and the eCos kernel has its own stacks.

Symbol	Address	Size	#Failures	(%)
stack	1107936	10224	2377760	(42.2 %)
thread_obj	1107584	352	984232	(17.5 %)
Cyg_Scheduler::scheduler	1120256	132	445920	(7.9 %)
cvar1	1107536	8	221888	(3.9 %)
m1	1107516	12	176448	(3.1 %)

Table 1: Fault-injection results: Top 5 fault-susceptible symbols for the unmodified MUTEX1 benchmark.

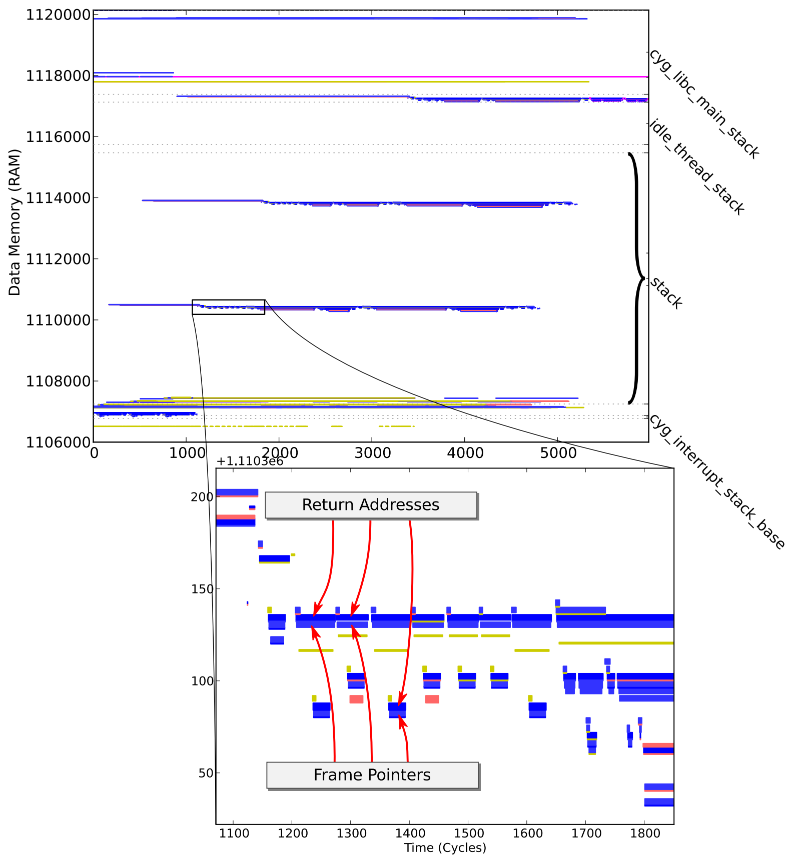


Figure 1: Results from the FI campaign with the unmodified MUTEX1 benchmark: Each point denotes the outcome of an independent run after injecting a burst bit-flip at a specific time and data-memory coordinate. Injections in *white* areas have no observable effect. *Blue* marks illegal memory accesses and jumps. CPU exceptions are colored *red* and timeouts *yellow* respectively. *Magenta* data points show benchmark runs that finish, but yield wrong output (silent data corruption).

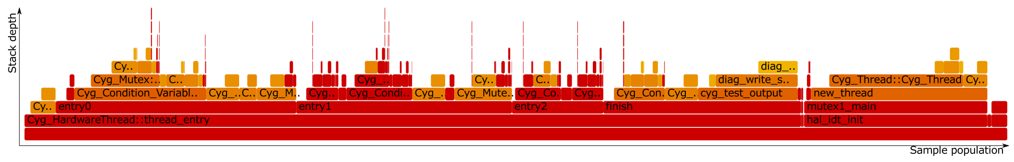


Figure 2: MUTEX1 benchmark call stack histogram in form of a *flame graph* (modified version of `https://github.com/brendangregg/FlameGraph`): Each bar represents a function's stack frame, and the x-axis denotes the function's runtime in the same stack context (but does *not* show the passing of time from left to right – it is actually ordered alphabetically). Lighter colors show a higher number of calls per runtime: Dark-red functions are long-running and only called once, orange to yellow ones run shorter and/or are called multiple times.

A closer look at an excerpt of the stack area of the MUTEX1 FI results in Figure 1 reveals more details. The larger diagram covers the the complete benchmark runtime in its horizontal dimension (5,986 CPU emulator cycles), and all of data memory (20,111 bytes) in its vertical dimension. The zoomed-in section reveals a primary reason the stack is so susceptible to faults: Memory corruptions in return addresses and frame pointers (cf. Section 1) lead to a crash with certainty, if the function returns to its caller after the FI. A more thorough manual analysis exposes that this is by far not the only, but the most homogeneous and widespread reason for crashes after faults in the stack memory.

3.2 A Protection Scheme Draft

An obvious software-based protection scheme for this problem is a compiler-based approach that stores redundant copies of return address and frame pointer directly after entering any function, and compares (and restores) them right before returning to the caller. Figure 2 illustrates that it may be too naïve to protect *all* functions, though: Some are called very few times and execute for a long time (darker colors), others are called very often and/or run very shortly (brighter). Intuitively, the latter sort should be considered to be run without additional protective code, as it would quickly worsen the runtime (and static code size) overhead – even beyond the break-even point where the additional exposure to faults outweighs all gains from being protected.[4]

This directly results in the requirement that the application of the protection scheme needs to be *configurable* regarding the subset of functions that is protected. The following sections describe an approach and implementation that meets this requirement.

[4] A quantitative analysis of this effect is beyond the scope of this paper.

4 A Dependability Aspect for Return-Address Protection

In the previous section, we found that return addresses of C/C++ functions are highly susceptible to memory errors. However, only the long-running functions contribute significantly to the total system's dependability. In this section, we describe a methodology to identify the most important functions (Section 4.2), and, afterwards, selectively apply *error detection (EDM)* and *error-recovery mechanisms (ERM)*. For both steps, we adopt *aspect-oriented programming*, which is briefly introduced in the following.

4.1 Aspect-oriented Programming

The idea behind *aspect-oriented programming (AOP)* [KLM$^+$97] is a modular implementation of *crosscutting concerns*, which affect various source-code locations. Aspect-oriented programming languages usually support features to encapsulate a crosscutting concern into a single module – an *aspect*. A programmer specifies pieces of *advice* that define "where" and "when" desired actions shall be invoked via *match expressions*. The control flow of the program is intercepted at the specified points and is transferred to the aspect that carries out the advice code. The loose coupling between aspects and the remaining software modules makes AOP a suitable solution for implementing crosscutting dependability measures, as already shown in Section 2.

Aspect-oriented programming tools are, for instance, available for Java (AspectJ [KHH$^+$01]) and C++ (AspectC++ [SL07]). AspectC++ has been developed by our group over the last ten years and constitutes a source-to-source compiler that *weaves* the aspects into the desired source-code locations. The advice code itself is inlined into the existing code, so that runtime costs are comparable to a manual (scattered) implementation of a particular concern [LST$^+$06]. This property is of the utmost importance, because dependability mechanisms have to be very efficient to be robust on their own respect.

4.2 Identifying Critical Return Addresses

Those functions that execute *longer* than others are promising candidates for return-address protection from a probabilistic perspective. The return addresses of long-running functions are much more likely to be hit by random bit-flip events than those of short functions. To identify the longest-running functions, we found AOP an excellent tool. *Every* function call and return, as described via the wildcard match expression `"% ...::%(...)"`, can be intercepted by an aspect that records the runtime of a particular function run. The per-function aggregate of these runtimes can be interpreted as a *criticality metric*.

Figure 3 shows the implementation of a profiling aspect that evaluates the criticality metric. Line 2 defines those functions that are considered for profiling by the `pointcut` keyword. In this example, the function `main` with arbitrary arguments `(...)` and `int` return type is specified (`main` serves exemplarily as a placeholder). When all functions should be

```
 1  aspect Profiling {
 2    pointcut profile() = "int main(...)"; // functions to profile (textual list with wildcards)
 3
 4    advice execution(profile()) : around() { // intercept the control flow at those functions
 5      ClockCycles start = rdtsc(); // save current clock time (using RDTSC CPU instruction)
 6      tjp->proceed(); // continue the intercepted function's execution
 7      Collector<JoinPoint::JPID>::chain.data->add(rdtsc() - start); // add elapsed time
 8      Collector<JoinPoint::JPID>::signature = JoinPoint::signature(); // the function's name
 9    }
10  };
```

Figure 3: An implementation of the profiling aspect written in AspectC++.

profiled, the aforementioned match expression could be used. The around advice in Line 4 replaces the `execution` behavior of the specified functions. When entering the function, the CPU clock time is determined with the x86 `RDTSC` CPU instruction. Afterwards, the original function is continued by the statement `tjp->proceed()`[5]. When the original function finishes, the advice code rechecks the clock time and collects the elapsed time in a C++ template-based container data structure `Collector`. The `Collector` class is not shown here, but a similar implementation is provided by the official AspectC++ examples[6]. The key to the function/runtime mapping is the unique identifier `JoinPoint::JPID`. AspectC++ assigns such an identifier to each source-code location where advice code is inserted. Moreover, AspectC++'s `JoinPoint` API exposes the intercepted function's signature (`JoinPoint::signature`), which is also stored in the `Collector`. On termination of the profiled application, the `Collector` can be queried to report on the longest-running functions.

The aspect in Figure 3 wittingly does not profile *every* function. The reason is that some functions are *inlined* by the C++ compiler, and, therefore, have no return address. Thus, the profiling aspect has to be restricted to *non-inline* functions only. For each application, the pointcut `profile()` must be set to match exactly the non-inline functions of the particular application binary. We used the `nm` program from the GNU Binutils package to automatically generate the `profile`() pointcut expression from an application binary file.

Figure 4 summarizes our approach to identify (and protect) the most critical functions of an application. The non-inline functions of an application binary are determined (①) and passed to the profiling aspect in terms of a pointcut expression. The profiling aspect (②) evaluates the functions' runtimes and generates a list of the longest-running ones. This list is a subset of the non-inline functions, and, again, is represented as a pointcut expression that is handed over to the return-address protection aspect (③) described in the following section.

The decision, whether a function runs *long enough* to be considered as critical, is configurable inside the profiling aspect. In our case, we chose those functions that had a runtime accounting for at least one thousandth of the sum of all functions' runtimes. Note that in

[5]`tjp` (this join point) provides access to context information of the intercepted function.
[6]AspectC++ and the examples are freely available at `http://www.aspectc.org/`.

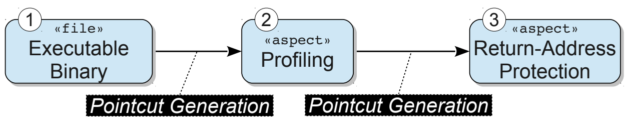


Figure 4: Process of applying the dependability aspect for return-address protection. The functions' signatures of the target application (①) are determined, and, afterwards, profiled to assess their runtimes (②). The most critical (longest-running) ones become protected (③).

general, the sum of all functions' runtimes is larger than the application's pure runtime. Additionally, we constrained that these functions must execute for at least 50 clock cycles on the average, to get rid of short but often-called functions. Other parameters are possible but have not been evaluated, yet.

4.3 Redundant Return Addresses

The previous section described our approach to retrieve a pointcut expression that reflects those functions of an application that are worth to be protected. Such a pointcut is the basis for a dependability aspect that implements the return-address protection. As usual for error detection (EDM) and error-recovery mechanisms (ERM), we apply *redundancy* to detect and correct potential bit errors in the return addresses. In essence, immediately after a function is entered, redundancy for its return address is created, and, right before the function leaves, the real return address is checked and optionally repaired. We implemented two different mechanisms:

EDM: A joined two's complement checksum is used to detect errors in both the return address and the previous frame pointer. On checksum mismatch, the running thread is aborted (fail-stop behavior).

ERM: Two copies of the return address and the previous frame pointer are created, yielding triple-modular redundancy (TMR) with majority voting. Errors are transparently repaired.

Figure 5 shows the implementation of the protection aspect for the EDM variant (the slightly larger ERM variant is not shown). The target functions are defined by the pointcut `critical_functions()` in Line 20: `"int main(...)"` again has to be regarded as a placeholder for the generated list of critical functions. The following execution advice intercepts those functions, and, at first, constructs an object holding the redundancy on the stack. At this time, the C++ constructor of the class `RedundantReturnAddress` is invoked, which computes a checksum over the return and frame addresses. Afterwards, the intercepted function is continued via `tjp->proceed()`. When the continued function finishes, the advice scope quits, and the redundancy object `rra` is destroyed. The C++ destructor of that object validates the checksum.

```
template<unsigned int JPID>
class RedundantReturnAddress {
  volatile unsigned int checksum; // maintain a checksum of return address + frame pointer
  public:
  inline RedundantReturnAddress() :  // constructor: initialize the checksum
    checksum(  (*(unsigned int *)__builtin_frame_address(0)) // yields frame/base pointer
             + ( (unsigned int)  __builtin_return_address(0)) // yields return address
             + JPID + 1) {} // add in joinpoint-specific ID (+ 1 avoids adding zero)

  inline ~RedundantReturnAddress() { // destructor: validate the checksum
    unsigned int checksum = (*((volatile unsigned int *)__builtin_frame_address(0)))
                          + (*(((volatile unsigned int *)__builtin_frame_address(0))+1))
                          + JPID + 1;
    if(checksum != this->checksum)
      signal_error(); // only error detection: signal_error() must not return
  }
};

aspect ReturnAddressProtection {
  pointcut critical_functions() = "int main(...)"; // textual list of functions to protect

  advice execution(critical_functions()) : around() { // intercept the control flow
    RedundantReturnAddress<JoinPoint::JPID> rra; // allocate object and call its constructor
    tjp->proceed(); // continue the intercepted function's execution
  } // rra's destructor is called here implicitly, which triggers return-address validation
};
```

Figure 5: EDM variant of the return-address protection aspect written in AspectC++.

In summary, the aspect ensures that the EDM/ERM is applied to the proper functions. The EDM/ERM itself is implemented in the exchangeable class `RedundantReturnAddress`. The constructor and destructor are declared as `inline` to permit access to the current return and frame addresses, which are obtained through the GCC-instrinsic[7] functions `__builtin_return_address(unsigned int)` and `__builtin_frame_address(unsigned int)` respectively. The arguments of these builtin functions specify the stack depth to look into, and a value of `0` takes the current stack frame. Note that the destructor obtains the return address through `__builtin_frame_address` plus offset to force a reload of the return address. These builtin functions provide a convenient interface to the low-level call-stack details, which would otherwise require platform-dependent assembly code to access.

The remaining detail, which has not been discussed yet, is the template parameter `JPID` of the class `RedundantReturnAddress`. The unique `JPID`, provided by AspectC++, allows differentiation between the protected functions. The redundancy can be encoded in a function-specific way, for example, by simply adding the `JPID` compile-time constant to the checksum. Thus, a checksum is valid only for the function that created the checksum. Otherwise, a corrupt stack frame, for example containing the return address plus valid checksum of *another* protected function, would remain undetected. In other words, the `JPID`-specific encoding of redundancy allows an *acceptance test*: For example, an empty

[7] `http://gcc.gnu.org/onlinedocs/gcc/Return-Address.html`

stack frame consisting solely of zeros can be detected, which would otherwise pass the checksum validation (EDM) and majority voting (ERM) respectively.

5 Evaluation

In this section, we quantitatively evaluate the EDM and ERM effectiveness in a set of benchmarks with fault-injection (FI) experiments, and measure the induced static and dynamic overhead. This allows us to predict the suitability for yet unknown scenarios, and to draw conclusions on the overall methodology.

5.1 Evaluation Setup & Fault Model

We evaluated both *Detection* (EDM) and *Correction* (ERM) variants on eCos 3.0 with a set of 21 benchmark and test programs that is bundled with eCos itself. Table 2 briefly describes each benchmark; including the baseline variant, this set totals at 63 variant/benchmark combinations. All binaries were compiled for i386 with the GNU C++ compiler (GCC, eCosCentric GNU tools 4.3.2-sw, optimization level -O2); eCos was set up with its default configuration, *grub* startup, and the *bitmap* scheduler variant. We disabled both serial and VGA output, as it is very time-consuming and would completely mask out any EDM/ERM runtime overhead.

We again chose a uniformly-distributed burst bit-flip model in data memory, which flips *all eight* bits at a memory address at once. The restriction to data memory (data, BSS, and stack segments) is reasonable for low-cost embedded systems where read-only data and code (text segment) are stored in far less susceptible (EEP)ROM or Flash, and global objects and the program stack are kept in non-ECC RAM.

Bochs, the IA-32 (x86) emulator back end that the FAIL* experimentation framework [SHK+12] currently provides, was configured to simulate a modern 2.666 GHz x86 CPU. It simulates the CPU on the instruction level with a simplistic timing model of one instruction per cycle (with the exception of the `HLT` instruction, which spans multiple cycles until the next interrupt), and does not provide any insights on caching and pipelining effects. Therefore the results obtained from injecting memory errors in this emulator are very pessimistic: We expect that a contemporary cache hierarchy would mask many main-memory bit flips, for example, the return addresses of short-running functions could be fully kept in a cache.

5.2 Effectiveness: Error Detection & Correction

Figure 6 gives a qualitative impression of the FI campaign results: The Detection variant (Figure 6a) detects almost all faults in return addresses and frame pointers that led to crashes

Benchmark	Description / Test. domain *(#thr.)*	Benchmark	Description / Test. domain *(#thr.)*
BIN_SEM1	Binary semaphore functionality *(2)*	MUTEX1	Basic mutex functionality *(3)*
BIN_SEM2	Dining philosophers *(15)*	MUTEX2	Mutex release functionality *(4)*
BIN_SEM3	Binary semaphore timeout *(2)*	MUTEX3	Mutex priority inheritance *(7)*
CLOCK1	Kernel Real Time Clock (RTC) *(1)*	RELEASE	Thread release() *(2)*
CLOCKCNV	Kernel RTC converter subsystem *(1)*	SCHED1	Basic scheduler functions *(2)*
CLOCKTRUTH	Kernel RTC accuracy *(1)*	SYNC2	Different locking mechanisms *(4)*
CNT_SEM1	Counting semaphore functionality *(2)*	SYNC3	Priorities and prio. inheritance *(3)*
EXCEPT1	Exception functionality *(1)*	THREAD0	Thread constructors/destructors *(1)*
FLAG1	Flag functionality *(3)*	THREAD1	Basic thread functions *(2)*
KILL	Thread kill() and reinitalize() *(3)*	THREAD2	Scheduler and thread priorities *(3)*
MQUEUE1	Message queues *(2)*		

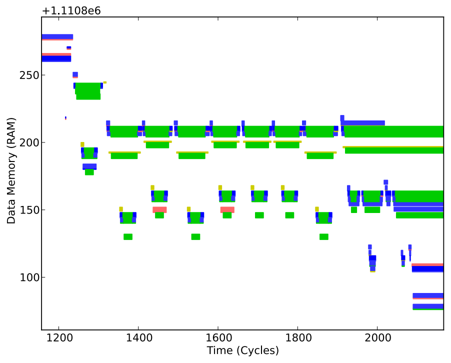
Table 2: eCos kernel test benchmarks with the number of running threads in parentheses.

Symbol	Address	Size	#Failures	(%)
stack	1109600	10224	1805912	(29.4 %)
thread_obj	1109248	352	1347976	(22.0 %)
Cyg_Scheduler::scheduler	1121920	132	629024	(10.2 %)
cvar1	1109200	8	307360	(5.0 %)
m1	1109180	12	243464	(4.0 %)

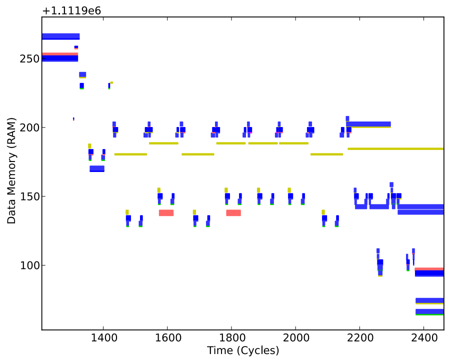
Table 3: MUTEX1 top 5 susceptible symbols in the Correction variant: The number of crashes resulting from faults on the main stack are reduced by 24.0 % in comparison to the baseline variant (cf. Table 1) for this particular benchmark. Note the increase in failures for thread_obj by 37.0 %, resulting from longer exposure by the (pathologically high) 36.7 % runtime overhead for this tiny benchmark (cf. Figure 8).

before (cf. Figure 1) and fail-stops; the Correction variant (6b) additionally transparently corrects these faults. Only very short susceptible timeframes between call and replica creation, and between check/repair and function return remain. The top five susceptible symbols list for the MUTEX1 benchmark in the Correction variant (Table 3) still ranks the main application stack first, but with a 24.0 % reduction in the number of benchmark failures after FI in the stack.

The FI campaign results for the remaining benchmarks is shown in Figure 7 – differentiated into the observed experiment outcomes. On average, the Detection variant reduces stack failures by 53.8 %: especially the long-running benchmarks profit from the EDM. The introduced redundancy increases the attack surface, though: The green “error detected” bars exceed the baseline in all cases (+46.4 % on average). The Correction variant is similarly effective, on the average 45.5 % stack failures are completely masked (another 3.2 % are detected) – and the increased attack surface pays off even for the shortest-running benchmarks.



(a) Detection variant: *Green* results denote experiments where the fault was successfully detected (fail-stop). In the baseline variant (cf. Figure 1) all of these led to crashes.



(b) Correction variant: Almost all return address and frame-pointer related crashes are masked; only very short susceptible timeframes between call and replica creation, and between check/repair and function return remain.

Figure 6: MUTEX1 benchmark, close-up of the same stack area as shown in Figure 1, for both Detection and Correction variants.

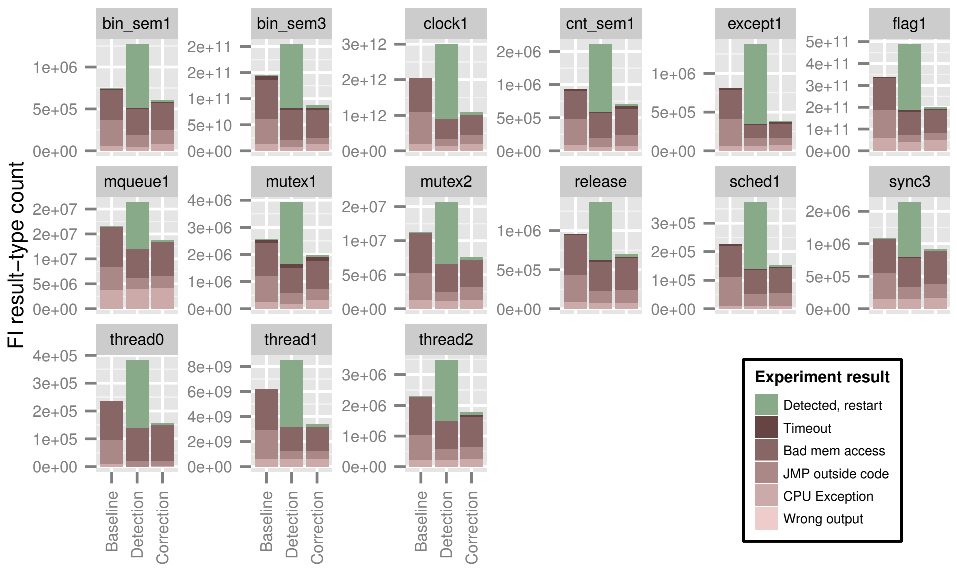


Figure 7: Absolute failure type (and detection) counts from FI experiment campaign (BIN_SEM2, CLOCKCNV, CLOCKTRUTH, KILL, MUTEX3 and SYNC2 benchmarks omitted due to their extremely long runtime): Both EDM/ERM variants improve resiliency to faults on the stack(s) substantially.

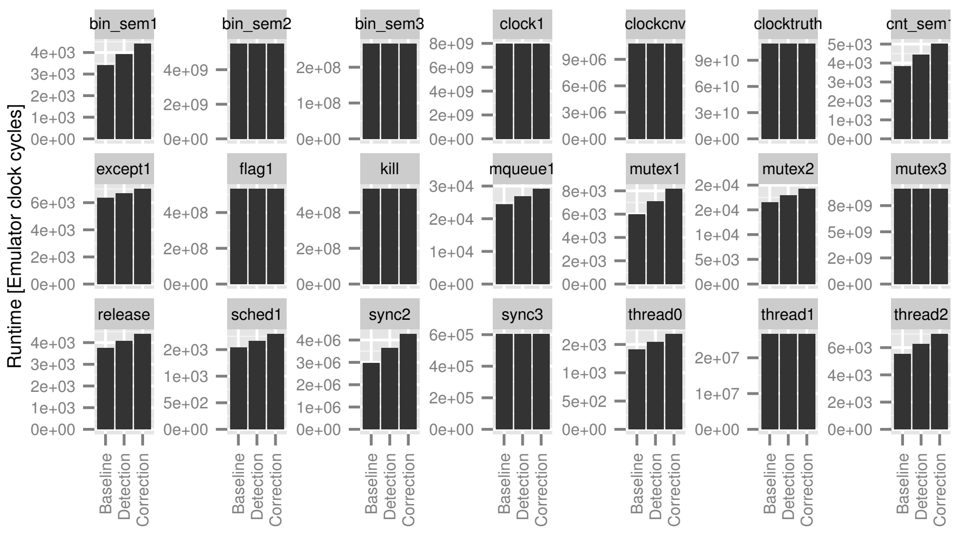


Figure 8: Benchmark runtime overhead: Only the short-running benchmarks exhibit measurable slowdown.

5.3 Efficiency: Static and Runtime Overhead

The static overhead of both protection variants is pleasingly low: both data and BSS sections stay at their original size, only the text segment grows. For the Detection variant, the total code size overhead is 1.24 % (ranging from 0.28 % in the THREAD1 benchmark to 2.41 % in MUTEX2); for Correction, the overhead totals at 3.76 % (THREAD1: 0.81 %; MUTEX2: 7.14 %).

The runtime overhead[8] for each benchmark is shown in Figure 8: The very short-running benchmarks ($< 10^5$ cycles) exhibit a measurable overhead of +10.5 % (Detection) respectively +21.0 % (Correction). The CPU-bound SYNC2 benchmark even adds +22.6 %/+44.1 %. All longer-running benchmarks spend most of their time in a CPU sleep mode (`HLT` instruction, waiting for the next timer interrupt); in these cases, the runtime overhead is completely masked (0.0 %).

6 Discussion

The evaluation shows that the return-address protection aspect comes at different levels of overhead, depending on the benchmark. The most important property is the runtime overhead, which directly relates to the effectiveness of our approach. The Detection variant

[8]Measured in emulator cycles: We showed in [BSS13] that at least for this set of small benchmarks the runtime overhead results are consistent with measurements on a real contemporary x86 machine.

consistently shows little overhead and improves the net resilience for all benchmarks (total failures reduced by 14.8 %). The Correction variant reduces the total amount of failures only by 13.3 % due to the slightly higher runtime overhead for proactive recovery.

On the average, only 9 functions have been protected per benchmark. This decision completely determines the overhead and the effectiveness of our approach. We further plan to investigate the trade-offs between protecting more or less functions. Such an opportunity demonstrates the *flexibility* of our methodology. By means of aspect-oriented programming, our solution is fine-grainedly configurable by the user and can be tailored to the specific use case. In comparison, a similar compiler-implemented mechanism would not offer the same degree of flexibility. However, a convenient interface to the compiler is beneficial, such as the GCC-intrinsic function `__builtin_return_address`. The low-level compiler could expose even more knowledge to upper software layers, for example, more fine-grained means to control specific types of optimizations. Then, crosscutting dependability measures, such as the return-address protection aspect, could be implemented more easily.

7 Conclusions and Future Work

In this study, we have presented an aspect-oriented approach to detect and correct memory errors in stacks of C/C++ programs. We have shown that return addresses and stored frame pointers are highly sensitive and deserve dedicated protection mechanisms. The total code size overhead of our solution is only 3.76 %, and a runtime overhead is only measurable for a subset of short-running benchmarks. At the same time, the number of stack-related program failures could be reduced by 48.7 %, which corresponds to a total failure reduction of 13.3 %. Thus, a significant net resiliency gain could be achieved.

Compared to hardware-based memory protection, software bugs are also detected. Consider a parallel thread or an interrupt handler that corrupts a stack frame – the proposed mechanism will detect and possibly correct such errors. The same applies to security issues of C/C++ programs (buffer overflows), which could be prevented by a similar dependability aspect. In future work, we plan to investigate how our approach compares to the existing compiler-based security solutions.

References

[AK11] Ruben Alexandersson and Johan Karlsson. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In *41st IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '11)*, pages 303–314. IEEE, June 2011.

[ASMT07] Francisco Afonso, Carlos Silva, Sergio Montenegro, and Adriano Tavares. Applying aspects to a real-time embedded operating system. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '07, New York, NY, USA, 2007. ACM.

[Bau05] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of*

Computers, 22(3):258–266, May 2005.

[BSS12] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Protecting the Dynamic Dispatch in C++ by Dependability Aspects. In *1st GI W'shop on SW-Based Methods for Robust Embedded Sys. (SOBRES '12)*, Lecture Notes in Informatics, pages 521–535. German Society of Informatics, September 2012.

[BSS13] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative Software-based Memory Error Detection and Correction for Operating System Data Structures. In *43nd IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '13)*. IEEE, June 2013.

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, Berkeley, CA, USA, 2000. USENIX Association.

[CH01] Tzi-Cker Chiueh and Fu-Hau Hsu. RAD: a compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing Systems*, 2001.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th Conf. on USENIX Sec. Symp. (SSYM'98)*, Berkeley, CA, USA, 1998. USENIX Association.

[HSS12] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. In *17th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '12)*, pages 111–122, New York, NY, USA, 2012. ACM.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *15th Eur. Conf. on OOP (ECOOP '01)*, volume 2072 of *LNCS*, pages 327–353. Springer, June 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, volume 1241 of *LNCS*. Springer, June 1997.

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *EuroSys 2006 Conference (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM.

[Mas02] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[NDO11] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs. In *ACM SIGOPS/EuroSys Int. Conf. on Computer Systems 2011 (EuroSys '11)*, pages 343–356, New York, NY, USA, April 2011. ACM.

[SHK+12] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a Versatile Fault-Injection Experiment Framework. In *25th Int. Conf. on Arch. of Comp. Sys. (ARCS '12), Workshop Proceedings*, volume 200 of *Lecture Notes in Informatics*, pages 201–210. German Society of Informatics, 2012.

[SL07] Olaf Spinczyk and Daniel Lohmann. The Design and Implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[SPW09] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *Proc. of the 11th int. joint conf. on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 193–204. ACM, 2009.