Gradual Software-Based Memory Protection*

Michael Stilkerich Daniel Lohmann Wolfgang Schröder-Preikschat
{stilkerich,lohmann,wosch}@cs.fau.de
Friedrich-Alexander University Erlangen-Nuremberg, Germany

ABSTRACT

Software-based memory protection (MP) provides not only spatial isolation of different applications, but also additional means to detect programming errors within an application. However, this luxury comes at the cost of extra runtime checks that add overhead to the application. In this paper, we present the idea of *gradual* software-based MP, where we only add a subset of runtime checks to a program to gain configurability with respect to the tradeoff between dependability and cost. To support the selection, we analyze the importance of different classes of safety checks and research which types of checks could be implemented at no cost by a hardware protection unit. After examining the relative frequency of different runtime checks in a large Java class library, we expect these ideas to be applicable to the majority of runtime checks in an embedded application.

1. INTRODUCTION

Electronic support functions in cars have rapidly developed in the past decade [2]. A modern mid-class car is equipped with about 80 electronic control units (ECUs), which communicate with each other through up to five different bus systems. This development is problematic in several aspects: the multitude of ECUs and wires that connect the ECUs with each other are costly, especially with the ever increasing copper price; the wires with about 50 kg noticeably contribute to the weight of the car; the connectors that attach the wires to the ECUs are known to be fault-prone and a major cause of hardware defects. To address these issues, the automotive industry is currently consolidating the number of ECUs in a car by replacing multiple ECUs with fewer, but more powerful microcontrollers, where multiple applications that formerly ran on a dedicated ECU now share a common microcontroller.

The coexistence of multiple applications on a microcontroller introduces the requirement to the underlying system software to enable the isolation of the applications with respect to different aspects. One of the key aspects is memory protection (MP), which is one of the points in that AUTOSAR OS [1] improves over its predecessor OSEK OS [13] in that it mandates write-protection in some scalability classes. The protection model of AUTOSAR is region-based and requires the presence of a memory protection unit (MPU) on the target microcontroller. On controllers without an MPU, MP is not supported.

Besides hardware-based MP there is software-based MP, where safety is constructively ensured by the executing program itself. Normally this is achieved by writing software in a type-safe programming language or employing runtime safety checks. Past research has shown that software-based protection based on the use of a type-safe high-level language [3, 5, 12] is superior in terms of overhead compared to lower level approaches [7], as it enables high-level static analyses that can validate many safety checks at compile time. Type-safe languages not only provide software-based MP, which increases the dependability of the resulting software product and precludes or detects many common programming errors [14] (e.g., uninitialized variables, off-by-one errors), it often also supports modern programming paradigms that increase productivity in software development [15], an important factor if costs matter.

Both of these approaches have advantages and disadvantages compared to each other. Memory protection is not a one or the other decision and should be a configurable property of the system software.

We have developed a system that allows choosing the type of MP, a fundamental architectural property, without the need to change the application. In this paper, we give an overview on this system (Section 2) and present some ideas (Sections 3–4) in the context of software-based MP based on observations we made in our research. In Section 5, we discuss, how these ideas could be applied in practice, and in Section 6, we give some early numbers to get an impression of the practicability of our approach.

2. MEMORY PROTECTION AT OPTION

Our protection model is designed according to the needs of the domain of embedded systems and similar to that of AUTOSAR OS with some extensions. For hardware-based MP, we require the presence of an MPU that allows restricting memory access to regions of the address space. Memory management units that manage the memory at the finer granularity of pages are found on only very few microcontrollers.

*This work was partly supported by the DFG under grant no. SCHR 603/4 and SCHR 603/7-1

IIDS 2010 Paris, France

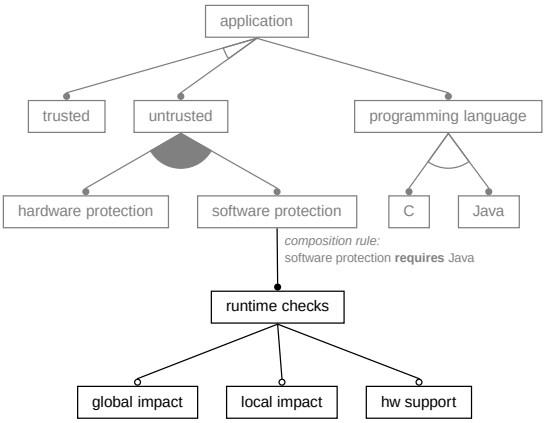


Figure 1: Memory Protection Variants

The AUTOSAR-OS protection model distinguishes trusted and non-trusted applications. An application comprises a number of tasks and ISRs. The data that belongs to an application is the stacks of its tasks and ISRs plus the private data segment of the application. Trusted applications run with MP features disabled and thus become part of the trusted computing base, whereas non-trusted applications only have write access to their own data. Read protection additionally restricts read accesses of an application to its own data and is optional in AUTOSAR.

We combine the Multi-JVM KESO with the operating system family CiAO to create a system that allows mixing applications isolated through different MP mechanisms. The available configuration spectrum is depicted in the form of a feature diagram in Figure 1.

2.1 The CiAO OS Family

CiAO [9] (CiAO is Aspect-Oriented) is a family of operating systems for embedded applications that has been designed and developed to be highly configurable by aspect-oriented programming (AOP) [6]. The implementation language of CiAO is AspectC++. CiAO's system design allows configuring even fundamental and highly crosscutting OS policies, among them hardware-based MP which we presented in a previous paper [10]. The primary target platform for CiAO is the Infineon TriCore, an architecture of 32-bit microcontrollers mostly used in the automotive industry that also serves as a reference platform for AUTOSAR. CiAO provides an API as defined by AUTOSAR OS.

2.2 The KESO Multi-JVM

To provide besides MPU-based MP the option of software-based MP, we run KESO on top of CiAO. KESO [17] is a Multi-JVM, that is, it allows tasks being isolated in different protection domains, each of which appears as a JVM of its own from the application's point of view. This isolation is constructively ensured by preventing any shared, global data among the different domains. Initially, this is established by providing each of these domains with an own set of global data (i.e., the static fields of classes in Java), and later on sustained by preventing object references from being passed to other domain through the available IDC mechanisms.

The main objective of KESO is to provide software-based MP tailored towards the domain of embedded systems. KESO does not support all aspects of the Java language and the Java virtual machine (JVM) and does not provide the full Java standard class library. In particular, KESO requires static applications and does support neither dynamic class loading nor Java reflection. The class library provided by KESO provides access to the system services of an OSEK/VDX or AUTOSAR OS, which is presumed as infrastructure software, and a safe and lightweight mechanism to access device registers from Java code without affecting the type-safety of the program. KESO supports optional garbage collection for applications that want to use dynamic memory allocation.

Part of KESO is an ahead-of-time Java Compiler that generates ISO C code from Java bytecode. KESO uses whole-program static analyses to perform many runtime checks at compile time. Where static analyses fail to validate the check ahead of time, an appropriate runtime check is emitted to the generated C code. Upon failure of such a runtime check, an exception will be raised. KESO does—for reasons of efficiency—not support fully-fledged Java exception handling. Instead, an exception is considered a fatal error in the program's execution, which can (as configured) result in either the invocation of a user defined exception handling routine, the termination of the control flow that caused the exception, or a reset or halt of the complete system (fail-stop-behavior). This is similar to the error hook routine found in OSEK/VDX or AUTOSAR operating systems.

2.3 Gradual Software-Based MP

One of KESO's features is that it allows disabling the emission of runtime checks based on criteria such as the type of the check and the bytecode instruction that causes the check, which enables the generation of a partially checked, *gradually safe* program. At the extreme side, all checks can be omitted, resulting in a program that does not suffer any overhead penalties but still benefits from increased dependability due to the checks successfully performed ahead-of-time compared to a program written in an unsafe low-level language. In the feature diagram (Figure 1), this corresponds to the sub-features of software-based memory protection (printed in black). We will discuss these features in the remainder of this paper. Our discussion is based on the following observations:

- The concrete microcontroller used in a mass product is usually the cheapest suited one from a line of functionally equivalent microcontrollers that scale in resources and price. However, even after choosing the smallest model from the line that still fulfills the requirements, the available resources will never be utilized by 100%. Individual runtime checks need very little resources, but each check contributes to the dependability of a program. The spare resources on such a microcontroller can therefore be used to increase the dependability by adding a selected set of runtime checks to the application.
- The consequences of an omitted runtime check that would have failed are not the same for all checks. Depending on the type of check and the checked instruction, we can group the checks into those that only have a local impact on the current application and those with global impact. We highlight these differences using the example of the two most common types of runtime checks in Section 3.
- Some microcontrollers have an MPU available that is not used by the system software. Such an MPU can be used to perform certain checks in hardware at no cost, rendering these runtime checks unnecessary in software.

We examine which types of checks can be shifted to hardware in Section 4.

- Bugs are not equally distributed over the codebase. According to [11], 80% of the bugs are within 20% of the software modules. Software modules containing bugs are likely to contain further bugs, so runtime checks should preferably be added to such modules to help finding these bugs. We propose some approaches for applying gradual software-based MP in practice in Section 5.

2.4 Transferability to Other Type-Safe Languages

Note that while Java may not seem the most appealing language to embedded-system programmers, KESO dispenses with most Java features that lead to the common sense that Java may be unsuited for embedded systems. We instead focus on the Java language as an example of a type-safe language because we have the implementation at hand and emphasize, that other type-safe languages, particularly those derived from C [3, 5, 12], employ runtime checks very similar to those of Java and that the ideas presented in this paper can easily be transferred to these.

3. IMPACT CLASSIFICATION

A self-evident criterion to classify the safety-value of runtime checks is the severity of the impact that the absence of such a check may cause to the system. In this section, we attempt to create a classification based on this criterion for the two most common types of runtime checks, null checks and array-bounds checks.

We split safety checks into the following two groups, which we will refer to as *impact classes* of a certain check:

Local Impact The omission of a check with the impact classification *local* may result in a malfunctioning of the application (i.e., the protection domain) in the context of which the check would normally have raised an exception. This defect will, however, not affect other applications on the same system.

Global Impact The omission of a check with the impact classification *global* may result in a malfunctioning of applications other than the one in the context of which the check would normally have raised an exception. The consequences of the defect may not be contained within the faulty application.

Omitting all checks of impact class local would still retain a degree of protection comparable to the one provided by hardware-based MP with the help of a memory protection unit (MPU), which is also the MP level required by AUTOSAR OS.

Table 1 summarizes the results that we will elaborate on in the remainder of this section, grouped by the type of runtime check. The first column contains a general description of the operation that causes the check; column two lists the Java bytecode instructions [8] that correspond to these operations in the JVM; column three shows the impact class of the check; the last column shows if the check may be performed by the hardware (see Section 4).

In the following, we separately consider load and store operations. We assume static knowledge of the address space layout of the target platform.

For load operations, the data type of the loaded value makes the difference between local and global impact. While the erroneous load of a primitive data value will only affect the computations of the application that loaded that value, interpreting a random value as a reference introduces wild pointers and breaks the soundness of the type system. Subsequent operations on that wild reference may affect data of other applications.

OPERATION	BYTECODE INSTRUCTIONS	IMPACT CLASS	HW
Null Check			
Instance Field load			
reference type	`getfield`	global	✔
primitive type	`getfield`	local	✔
store	`putfield`	local/global	✔
Array Operations load			
reference type	`aaload`	global	✔
primitive type	`[bcdfils]aload`	local	✔
store	`[abcdfils]astore`	local/global	✔
get length	`arraylength`	local	✔
Method Invocation			
dynamic binding	`invokeinterface` `invokevirtual`	global	✖
static binding	`invokespecial`	global	➥ local
Array-Bounds Check			
null array	*analogue to missing null check*		✖
valid array			
store	`[abcdfils]astore`	global	✖
load	*analogue to missing null check*		✖
Arithmetic Error Checks			
division by zero	`[il]div, [il]rem`	local	✔

Table 1: Impact Classification

For store operations, we are interested in the memory regions that could be modified by such a store. In a system that constructively isolates applications by means of software-based MP, the data of different applications is not necessarily physically separated from each other, but may, for example, be allocated from a common heap. It often is, however, the case, that portions of the address space are not being used, particularly on MCUs with a large address width. If an illegal store happens at an address that belongs to an unused portion of the address space, it does not corrupt data of the application (or, cause other defects, such as a reconfiguration of a hardware device if device registers where mapped to the affected address) and we consider the impact being of class local. On the other hand, if the store affects a used portion of the address space, the effect depends on the type of data stored there. Giving an answer to this question—if possible at all—would be a very tedious task, so we assume a global impact in such cases.

3.1 Null Checks

There are no wild pointers in type-safe languages. References or pointer values always point to an existing object, with the exception of one special `null` reference. This special value introduces the need to null-check operations on an object reference. If such a check is omitted, an illegal access will take place at a particular offset from the null reference, which is normally address 0. In contrast to unsafe languages, the absence of wild pointers therefore puts us in position to predict the affected address regions within a certain range. For the individual operations that require a null check, the impact is as follows:

Instance field stores happen at *fixed* offsets from the base address (i.e., the address 0 in the case of an invalid reference). These offsets are *statically known* for each individual null check. It is therefore exactly possible to foretell the memory slot that might be affected by omitting a particular null check. The maximum offset of an instance field depends on the numbers and types of members of the used Java classes, but is usually small in the area of approx. 20 bytes.

Array component stores use an offset, which may *vary* in different iterations of the check and whose range may *not be determinable at compile time*. The largest possible offset is the size of the array's datatype (statically known for each individual site) times the maximum array index. While the Java Virtual Machine specifies array indexes of 32-bit signed data type, KESO provides the option to the developer to choose a smaller data type such as unsigned 16-bit or 8-bit, since this is sufficient for most embedded applications.

Instance method invocations are the third type of operation that requires a null check. Since KESO normally assumes the `this` reference to be null checked at the call site, it does not generate null checks for operations on the `this` reference within instance methods. Consequently, the omission of a null check on a method invocation's call site can generally have the union of effects of all the other null checks and thus belongs to the impact class global. A special case is method calls with dynamic binding, where the address of the actually called candidate is determined at runtime from the actual type of the object the method is called on. In the case of a null reference, an unknown, possibly random, value would be interpreted as the type and an arbitrary address would be called, resulting in a loss of control flow integrity. However, if the type read were a predictable value (e.g., 0), one could benefit from this knowledge by pointing this slot of the dispatch table to an error handler to detect the error, trading a few bytes of RAM for the saved execution time.

3.2 Array-Bounds Checks

Array-bounds checks consist of a null check preceding the bounds check itself. We have already discussed the null check portion. The impact of an omitted bounds check is mostly the same as omitting the associated null check, with one exception: a missing bounds check on an array *store* operation on a *valid* array will certainly affect valid application data, since the access will not happen relative to the null address but to the address of the array which lies within the application data. The bounds check of an array store operation is thus more important than the null reference portion and should not be omitted. KESO splits array-bounds checks into a null check and a bounds check which allows them to be individually omitted.

3.3 Summary

To summarize, based on the knowledge of the address space layout, some checks show to be less important than others. The checks marked as local in Table 1 can be omitted without losing the MP properties that would be provided by an MPU. For load operations, this decision is independent of the actual address space layout and only depends on the data type loaded. For store operations, one has to check whether a used portion of the address space could be affected by an illegal store to decide whether the check is of local or global impact. For this reason, the impact class of runtime checks caused by store operations is denoted as *local/global* in Table 1. Telling the actual impact class is easily possible for store operations with a fixed, known offset from the base address; for array operations, where the operation is parameterized with an index value that might not be known at compile time, the prediction has to take into account the value range that this index may take. KESO allows system designers to choose a small data type as needed by the application to limit this range.

Figure 2: Infineon TC1796 Address Space

4. HARDWARE-SUPPORTED SOFTWARE-BASED MP

Many larger microcontrollers come with a trap system that can detect and signal certain error conditions in the execution of a program. In this section, we analyze which of the common runtime checks we considered could principally be performed in hardware. The last column of Table 1 summarizes the results of these considerations, based on the example of a TC1796 platform. A simple case of a runtime check that is usually also performed by the hardware is the detection of a division by zero (given that the instruction set provides a division instruction).

Many 32-bit microcontrollers come with an MPU, which is often left unused. Normally, an MPU is used to restrict memory accesses of different applications to data regions that belong to this application. In order to use this feature, the applications have to be structured in a way that allows identifying for each piece of data in the system the application that this piece of data belongs to, and to physically group data belonging to the same application in memory (since an MPU only supports a small number of regions). In addition to these structural requirements, the MPU needs to be reconfigured whenever the execution changes the context to a different protection domain. This includes the use of system services and context switches, which can pose a significant overhead especially for applications that frequently communicate with other applications or the kernel; to give an example, in CiAO the temporal overhead to the AUTOSAR service `ActivateTask()` is 258% [10] when MPU-based MP is enabled. For these reasons, an MPU is often left unused.

We found that a *statically configured* MPU could be used to support software-based MP by shifting some of the runtime checks from software to hardware, saving both the costs of code memory and execution time without sacrificing safety. 32-bit microcontrollers are good examples of MCUs, where large parts of the address space are often unused. Figure 2 shows used (black) and unused (white) portions of the address space of the TC1796 MCU; more than 75% of the address space is reserved. This still does not imply that the remaining 25% are actually being used; in fact, normally only a few MiB of the four GiB address space actually contain application code, application data, or memory-mapped device registers that are actually being used by the particular application.

On any Tricore derivate, the first eight bytes of the address space are generally reserved and the MPU provides a designated trap (memory protection NULL trap) for accesses to this region. This is similar to many other architectures and is because the address 0 is commonly used as a reserved address value. In addition, the entire lower two GiB of the

address space are reserved on the TC1796 MCU, since these are used on processors of the line that are equipped with a memory management unit (MMU). Consequently, without any MPU configuration, memory accesses to the lower two GiB of the address space would trap. Statically configuring the MPU so that only the union of regions actually being used by the applications in the system reduces the accessible address space to the necessary minimum.

MPU-based protection cannot perform semantically rich checks such as array-bounds checks, however, it is suited to perform a large portion of the null checks in hardware. The last column of Table 1 summarizes how MPU-based protection can support software-based protection: null checks for field accesses with a fixed, known and small offset from the address 0 are likely to be completely shiftable to hardware. For array operations, it depends on whether the widest region that may be affected is within an unused portion of the address space. In the case of KESO running on the TC1796 platform, if the application does not require arrays of more than 65536 elements, the null reference portion of the array checks can entirely be performed in hardware. This support is particularly valuable for operations loading reference values, where we found that an omission may result in a global impact. For *statically bound* method invocations, the MPU cannot directly detect the error at the call-site, however, with the presence of bounds checks for array store operations and null checks for dynamically bound method invocations the impact of the missing check can be reduced to local.

5. BUG HIDEOUTS

In this section, we propose approaches that could be used for the decision which checks to include and which to omit with respect to the expected cost-benefit ratio. The approaches can also be combined. In any approach, if the used microcontroller platform permits checks to be shifted to the hardware as discussed in Section 4, this should always be the first step as it eliminates the costs of these checks without impairing the safety of the system. The considerations of the following approaches would then be reduced to the remaining checks.

Instruction Based. One possible approach is to choose the more important checks based on our findings of Section 3. By omitting checks with local impact, a safety level similar to that provided by MPU-based MP can be retained.

By Developer. Another approach is to let the developer partition the application with respect to the likeliness of different modules of the software to contain bugs, and to include runtime checks in those modules. The criteria a developer uses to identify those modules could be based on

- **Quality Assurance (QA).** In some industries, the pressure to reduce the time-to-market may prevent all modules of a software system to undergo the QA process. Modules that went through the QA process are less likely to contain bugs than a module that has not been reviewed.
- **Case studies.** There exist various case studies [4] that found bugs not to be equally spread across the program, but to concentrate in few software modules. One of the reasons for this is the fact that not all software modules are equally complex. An example for a fault-prone class of software is device-drivers [16].

Feedback Approach A different approach that also builds on the finding that bugs concentrate in few modules of a software product [4] uses existing knowledge to identify bug-prone modules. This knowledge could be generated by feedback from the testing phase, or from bugs found in previous generations of the product. This approach can be refined over several generations. To return to the use case of the automotive industry, the process could be executed as follows: For the first generation of the product, feedback from code reviews or results of the testing department are used to identify modules where a high number of bugs have been found; or, for the first generation of the product, all safety checks are enabled, possibly accepting the need for a larger variant of the microcontroller line for this generation. In subsequent generations, reports from garages form the basis for the identification of the buggy modules; safety checks in well working modules could be reduced while the number of safety checks in modules that caused problems in the wild could be increased. This could lead to a reduction of the number of safety checks that might allow for a smaller microcontroller derivate in subsequent generations.

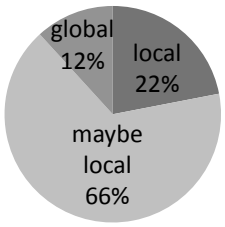
6. PRACTICABILITY

We do not yet have an embedded Java application at hand that could provide a representative measure of the practicability of the presented approaches, which depends on the frequency with which the different type of checks appear in the code base. To get an impression of the relative frequency of the different checked instructions we considered before, we have analyzed the bytecode of the Java class library GNU classpath, version 0.12. We are well aware of the fact that the code characteristics of this library are different from those of a typical embedded application, but still believe that the classpath library can give at least an impression of the proportions.

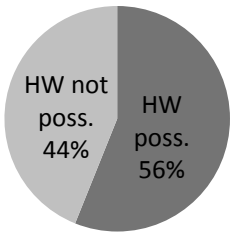
Figure 3a shows the absolute numbers of the operations that we considered in Section 3 and the check types that each of these operations require. Noticeable is the high number of dynamically bound method invocations as opposed to the very few statically bound method calls. This can be explained as follows: in Java, most instance method invocations are dynamically bound, except for some special method invocations (e.g., the invocation of the constructor of the parent class). When KESO compiles Java bytecode to C code, it determines the candidates available for each virtual method and, if there is only a single candidate, generates a statically bound call instead of the dynamically bound one. Embedded applications normally make very little use of actual virtual function calls, so that KESO is normally able to statically bind most method invocations. For this reason, we have not included the null checks caused by method invocations in Figure 3b–d. Figure 3b shows the portions of local and global null checks in GNU classpath. 22% of the null checks stem from operations that load primitive values (local impact), opposing 12% checks that are caused by operations loading reference values (global impact). The majority of 66% is caused by store operations, the impact of which can only be told with knowledge of the address space layout and the respective offsets. For the Tricore 1796b microcontroller in combination with 16-bit array indexes, all of these checks belong to the impact class local, wherefore only 12% of the null checks need to be retained to ensure application isolation as provided by MPU-based isolation. Figure 3c shows the amount of the checks that could be supported by hardware, again using the example of the

operation	freq	checks needed
field operations		
load primitive	2380	null
load reference	1878	null
store	645	null
array operations		
load size	1720	null
load primitive	2599	null + bounds
load reference	1714	null + bounds
store	19571	null + bounds
method calls		
dyn. binding	24752	null
stat. binding	217	null

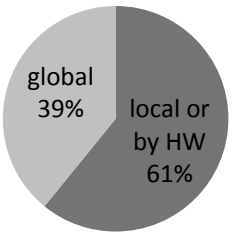
(a) Frequency of Operations



(b) null checks: Impact Class



(c) null+bounds checks: HW-supportable



(d) null+bounds checks: overall

Figure 3: Distribution of Runtime Checks in GNU Classpath 0.12

TC1796 microcontroller; basically, all null checks could be performed in hardware, whereas the bounds checks could not. Figure 3d finally shows that 61% of all the null and bounds checks are either of local impact or could be performed by the hardware.

7. CONCLUSION

In this paper, we presented the idea of applying software-based MP gradually to spend spare resources to increase the dependability of embedded software. The approach is based on selecting a subset of the runtime checks based on different criteria. We found that not all checks provide the same degree of safety, and created a classification for the two most common types of runtime checks, finding a subclass of checks that is sufficient to provide a degree of spatial isolation as required by the AUTOSAR-OS standard. We also examined which runtime checks could be performed by a statically configured MPU at no cost. We proposed approaches to apply these ideas in practice, and discussed our expectation that a significant portion of the runtime checks either could be performed by hardware or is of an impact class that allows its omission while retaining spatial application isolation.

8. REFERENCES

[1] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[2] M. Broy. Challenges in automotive software engineering. In *28th Int. Conf. on Software Engineering (ICSE '06)*, pages 33–42, New York, NY, USA, 2006. ACM.

[3] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In R. D. Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 520–535. Springer, 2007.

[4] A. Endres. An analysis of errors and their causes in system programs. In *Proceedings of the International Conference on Reliable Software*, pages 327–336, New York, NY, USA, 1975. ACM.

[5] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *2002 USENIX TC*, pages 275–288, Berkeley, CA, USA, 2002. USENIX.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. 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*, pages 220–242. Springer, June 1997.

[7] R. Kumar, E. Kohler, and M. Srivastava. Harbor: Software-based memory protection for sensor nodes. In *IPSN '07: 6st Int. Conf. on Information Processing in Sensor Networks*, pages 340–349, New York, NY, USA, 2007. ACM.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. AW, second edition, 1999.

[9] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX TC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[10] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat. Configurable memory protection by aspects. In *4th W'shop on Progr. Lang. and OSes (PLOS '07)*, pages 1–5, New York, NY, USA, Oct. 2007. ACM.

[11] S. McConnell. *Code Complete*. MS Press, second edition, 2004.

[12] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *POPL '02: 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 128–139, New York, NY, USA, 2002. ACM.

[13] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, Feb. 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2009-09-09.

[14] G. Phipps. Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.*, 29(4):345–358, 1999.

[15] E. Quinn and C. Christiansen. Java Pays – Positively. IDC Bulletin W16212, May 1998.

[16] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comp. Syst.*, 23(1):77–110, Feb. 2005.

[17] C. Wawersich, M. Stilkerich, and W. Schröder-Preikschat. An OSEK/VDX-based multi-JVM for automotive appliances. In *Embedded System Design: Topics, Techniques and Trends*, IFIP International Federation for Information Processing, pages 85–96, Boston, 2007. Springer.