The Reliable Computing Base – A Paradigm for Software-based Reliability

Michael Engel

Computer Science 12
TU Dortmund
Otto-Hahn-Str. 16
44221 Dortmund, Germany
michael.engel@tu-dortmund.de

Björn Döbel

Operating Systems
TU Dresden
Nöthnitzer Str. 46
01187 Dresden, Germany
doebel@os.inf.tu-dresden.de

Abstract: For embedded systems, the use of software-based error detection and correction approaches is an attractive means in order to reduce often inconvenient overheads in hardware. To ensure that such a software-based fault-tolerance approach is effective, it must be guaranteed that a certain amount of hardware and software components in a system can be trusted to provide correct service in the presence of errors. In analogy with the Trusted Computing Base (TCB) in security research, we call these components the Reliable Computing Base (RCB). Similar to the TCB, it is also desirable to reduce the size of the RCB, so the overhead in redundant hardware resources can be reduced. In this position paper, we describe approaches for informal as well as formal definitions of the RCB, the related metrics and approaches for RCB minimization.

1 Introduction

The International Technology Roadmap for Semiconductors [ITR] predicts that the frequency of permanent as well as transient faults in future semiconductors will increase significantly. This effect is expected to be a result of a combination of effects – shrinking feature sizes to increase the number of transistors available on a chip, lowered supply volatages to reduce the energy consumption, and a steadily increasing cost pressure.

This poses significant problems especially for small, cost-sensitive embedded systems. Semiconductors used in these systems often cannot afford additional error-correction methods like large ECC-protected memories or redundant execution on separate hardware units. Previous publications, like our results on reliability annotations [ESHM11], have shown that the overhead for error correction can be significantly reduced by avoiding the correction of errors that are classified as non-fatal. This, in turn, requires the introduction of *software-based* fault-tolerance approaches, since on the hardware level, no information on the semantics of a given data object or machine operation is available. To provide a more precise classifica-

tion of data and instructions, source-code based annotations and transformations have already been shown to significantly reduce the error correction overhead. For more precise analyses that take fine-grained resources (like single processor registers) into account, however, methods are required that provide more information on the application context in order to close the semantic gap between instruction and data bits on the machine level and application behavior [EM12].

However, one important fact is often overlooked when discussing software-based fault-tolerance methods. Since these, like the application software on top, are also implemented in software, the system designer has to provide methods to ensure error-free operation of the software components implementing fault-tolarance.

One approach to provide resilience for the fault-tolerant software components is to ensure that these components are being executed solely on reliable hardware, e.g., by explicitly protecting only a subset of the hardware components of a system against errors. In order to make this a feasible approach, we have to solve two problems:

1. How can we determine the fault-tolerance software components that are required to be correctly executing?
2. How can we ensure that the hardware components used by these software components are reliable?

In order to capture the reliability properties and requirements of a system's hardware and software components, in this paper, we introduce the concept of the *Reliable Computing Base* (RCB). Inspired by similar definitions from the field of security research, the RCB intends to encompass those components of a system that are expected to be reliable. These RCB components, then, are capable of reliably correcting errors in components outside of the RCB.

This position paper intends to be the first step in defining the RCB concept. A first informal definition is coined and possible important parameters that will aid in the determination of a more formal definition of the RCB are identified. One important topic is the minimization of the RCB, an approach to reduce the error correction overhead. Here, we try to build a bridge from the idea of RCB minimization to the increasing occurrence of heterogeneity in MPSoC architectures and memory hierarchies, which we expect to provide interesting possibilites for minimizing the RCB.

2 Related Work: The Trusted Computing Base (TCB)

The major inspiration for our definition of the Reliable Computing Base is the Trusted Computing Base (TCB) concept from security research. Similar to problems in reliability, software methods to ensure the security of a system have to

handle unexpected effects that result in a deviation of a system's expected behavior.

However, there exist several important differences between security and dependability problems. Security problems usually arise due to bugs in the program code. Based on the well-known fact that the number of software bugs per lines of code is mostly constant, the major driving force in security research is to reduce the *code size* of the TCB, thus reducing the number of possible errors in the security-critical code.

In dependability, however, the situation is different. The probability that a transient or permanent hardware fault affects a software component of a system does not depend on bugs in the software, but rather on code properties describing the semantics of the executed operations and related data objects. Error models, in turn, describe the *probability* of an error affecting a given component.

One important common fact of software-based fault-tolerance methods and TCB-based security approaches, though, is that both have to rely on properties of the underlying hardware to protect them. For security, hardware features like privileged CPU states and protection of memory accesses by an MMU or MPU assure that unprivileged code cannot access security-critical information, code, and hardware resources. For dependability, the critical components of the fault-tolerance software infrastructure have to be protected, e.g., by additional hardware, in order to guarantee their error-free execution.

In the remainder of this section, we will take a look at common definitions for the TCB and approaches to minimize its size. The following sections, then, coin our preliminary RCB definition and contrast it with the TCB concept.

2.1 TCB Definitions

Despite (or, rather, due to) being a research topic for several decades now, we were unable to find a single concise definition of the TCB in literature. Thus, we will discuss the most relevant definitions below.

The first definition of the term Trusted Computing Base was coined by John M. Rushby [Rus81]. He describes the TCB as *"a combination of a kernel and trusted processes, which are permitted to bypass a system's security policies"*.

A more precise definition was coined by Butler Lampson et al. [LABW91]. They define the TCB of a computer system as

> *«...a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.»*

This second definition includes an important extension to the previous definition by explicitly mentioning the *size* of the software and hardware components.

The Trusted Computer System Evaluation Criteria (the infamous “Orange Book”) [52085] define the TCB as follows:

> *«The heart of a trusted computer system is the Trusted Computing Base (TCB) which contains all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based. The bounds of the TCB equate to the "security perimeter" referenced in some computer security literature. In the interest of understandable and maintainable protection, a TCB should be as simple as possible consistent with the functions it has to perform. Thus, the TCB includes hardware, firmware, and software critical to protection and must be designed and implemented such that system elements excluded from it need not be trusted to maintain protection.»*

This definition strengthens the fact that the TCB cannot be restricted to software components alone. Rather, it includes firmware and hardware components as well. Also, the definition states clearly that components excluded from the TCB need not be trusted.

Concerning the applicability of the TCB concept to embedded systems, the report continues:

> *«For general-purpose systems, the TCB will include key elements of the operating system and may include all of the operating system. For embedded systems, the security policy may deal with objects in a way that is meaningful at the application level rather than at the operating system level.»*

This definition fits well with our basic assumption that the inclusion of application semantics is a key element to reduce the error correction overhead.

However, a major difference is that the report implies that an application-specific protection policy can take place at application level:

> *«Thus, the protection policy may be enforced in the application software rather than in the underlying operating system.»*

Here, we observe a significant difference between our ideas of TCB and RCB definitions. Software-based reliability approaches tend to apply generic as well as application-specific error correction methods. Generic methods include common approaches like checkpointing and recovery or triple modular redundancy (TMR), which restore a system state to a correct previous state, using application semantics mostly to decide if an error is to be handled at all. In contrast, application-specific error-correction methods apply a more fine-grained application semantics. For example, a corrupt data value in a program may not be restored to the precise original value in case of an error, but rather replaced by a default value inside the context-defined value range.

2.2 TCB Minimization

An important research topic in security research is the question how to minimize the TCB. The underlying assumption is based on the fact that bugs in software are unavoidable and the number of bugs per lines of code (LoC) is mostly constant [Hat95]. Thus, by reducing the amount of code running in privileged mode the system depends on is a central objective [SPHH06].

This objective was one of the driving forces in the development of second-generation microkernels, like Liedtke's L4 system [Lie96] and subsequent developments such as TU Dresden's Fiasco and UNSW's OKL4 [HL10] systems.

Informal Procedures to minimize TCB are described in the "Orange Book" [52085]. Formal approaches to verify the correctness of the code in the TCB, like [HTS02], have struggled for the last decade due to the complexity of the models and proofs involved. One successful result is the verified seL4 kernel [KAE+10], which accounts to about 8,700 lines of C code and 600 lines of assembler code. However, even this effort has to rely on the correctness of several hardware and software components, notably the correctness of the C compiler, the assembly code, the hardware, and kernel initialization.

The protection of the code in the TCB from unprivileged accesses is accounted for by hardware mechanisms like privileged execution modes and memory management units. Thus, reducing the work to – manually or automatically – find security-related bugs is the major driving force in TCB minimization. Automatic approaches, while already proven possible, are still an open research topic.

2.3 Application-specific TCB

If the set of applications on a system is known in advance, which is usually the case in embedded systems, methods to minimize the TCB can benefit from this fact. An application-specific TCB is determined by analyzing the set of services an application requires from the underlying system software. If this set of components that an application needs to trust in order to operate securely is known, the remaining unused TCB components can be removed. For example, an application that does not require file system access will not require the related security functionality inside the TCB. This approach facilitates the minimization of the TCB especially for well-structured, componentized systems, like operating systems based on a microkernel.

3 The Reliable Computing Base

3.1 Informal Approach

In this section, we intend to give a first, informal definition of the Reliable Computing Base, mainly derived from the TCB definition by Lampson et al. [LABW91].

Our first definition of the Reliable Computing Base is:

> *"The Reliable Computing Base (RCB) is a subset of software and hardware components that ensures the operation of software-based fault-tolerance methods and that we distinguish from a much larger amount of components that can be affected by faults without affecting the program's desired results."*

Here, it is important to describe the deviations from the original TCB definition in detail.

The RCB definition uses the term *"subset of ... components"*. In contrast, the TCB definition uses the *"small amount"*. Reducing the amount of code, measured in LoC, is the important goal for TCB minimization, based on the assumption of a constant number of bugs per LoC [Hat95]. For the RCB, the vulnerability of code is much more important than code size itself and we expect it to adhere to more complex metrics than LoC.

The *"software and hardware components"* are relevant for the TCB as well as the RCB. For the TCB, often hardware protection ensures the security of critical software components. For RCB, the relevant hardware components should be be especially protected from errors in order to guarantee reliable execution of the code inside the RCB.

The RCB definition uses the wording *"ensures the operation"*, whereas the TCB definition uses *"that security depends on"*. The RCB definition reflects our assumption that software-based reliability can effectively be staged. Obviously, some core software components have to execute reliably in any case. However, these components might also implement only a small core that uses further software-based methods to ensure the dependability of the remaining fault-tolerance methods. We give an example of this approach we termed *"dependability bootstrapping"* in sect. 5.4. Here, analyzing the tradeoffs in execution time, code size, and energy for different bootstrapping strategies is an interesting open topic.

In addition, the RCB definition describes *"software-based fault-tolerance methods"*. This first approach to a RCB definition is coined for software-based fault-tolerance approaches. An open question is whether the RCB definition is also sufficiently general to apply it to hardware-based dependability approaches.

The *"much larger amount that can tolerate errors"* in the RCB definition refers to the fact that it is actually possible to correct errors affecting a large part of the code and data corrected by software-based methods. Otherwise, the overhead

for software-based approaches would be inacceptable if these could only cover a relatively small amount of code and data.

The term *"desired results"* leaves room for interpretation. Depending on the application, anything from 100% perfect results to only avoiding system and application crashes is included here. This expresses the basic assumption that not all errors show identical criticality. The system designer should thus be enabled to differentiate handling for different classes of error impacts. Here, the TCB definition uses *"without affecting security"*. This definition seems similarly open-ended, since it is not further described what is included in "security".

While this first, informal definition gives a rough idea of our RCB concept, a more formal definition of the RCB will help in building automated analysis tools to determine and, possibly, minimize the RCB. In the following section, we present existing approaches for dependability metrics from the literature which seem useful for a future formal RCB definition.

4 RCB-related Metrics

In order to assess the vulnerability of hardware and software components to errors, previous research has already produced a number of metrics that seem useful in defining the RCB in a more formal way. This more formal definition, then, can lead the way to providing automatic code analysis and optimization approaches that minimize the RCB.

The concepts we will describe include Mukherjee's ACE (Architecturally Correct Execution) Analysis [MWE+03] and the related Architectural Vulnerability Factor (AVF). Based on these metrics, we will describe Sridharan's Program Vulnerability Factor (PVF) [SK08].

4.1 ACE Analysis

ACE analysis is a technique that provides an early reliability estimate for microprocessors [MWE+03]. It is based on the concept of "Architecturally Correct Execution". An Architecturally Correct Execution (ACE) instruction is an instruction whose results may potentially affect the output of the program. By coupling data from abstract performance models with low level design details, ACE analysis identifies and rules out transient faults that will not cause incorrect execution. However, there are some restrictions to ACE analysis. One important restriction is that, while many transient faults are analyzable, some cannot be covered. As a result, ACE analysis is conservative and provides a worst-case lower bound for the reliability of a processor design in order to ensure that a given design will meet reliability goals.

4.2 Architectural Vulnerability Factor

Based on the definition of Architecturally Correct Execution, Mukherjee defines the Architectural Vulnerability Factor (AVF) metrics as follows [MWE+03]: *"The AVF is the fraction of time an ACE instruction is present in a logic device"*. For a given hardware component H, an execution cycle n, and the number of ACE bits in H (B_H), the AVF over a period of N cycles is defined as (see also [SK10]):

$$AVF_H = \frac{\sum_{n=0}^{N}(\text{ACE bits in } H \text{ at cycle } n)}{B_H \times N} \tag{1}$$

Using the AVF metric, we can determine the vulnerability of different hardware components of a CPU. Simple examples from [MWE+03] include:

- Commited error in program counter: AVF $\approx$ 100%
- Error in branch predictor: AVF $\approx$ 0%

These results are unsurprising. If a program counter value changes unexpectedly, in almost any cases the control flow of the currently executing program changes to a completely different context, so in almost any case the program is affected. In contrast, the branch predictor only optimizes prefetching. If a prediction fails, the only result is a possibly increased execution time[1].

More complex examples are given for a specific implementation of an IA64-like architecture in [MWE+03]:

- AVF for the instruction queue: 28%
- AVF for execution units: 9%

4.3 Program Vulnerability Factor

Based on the definition of the AVF, Sridharan et al. defined the Program Vulnerability Factor (PVF) [SK08], which describes the influence of a hardware component's AVF on the execution of a given piece of code using that component.

The PVF of an architectural bit is defined as the fraction of time (in the number of instructions) that the bit is architecturally correctly executed. Accordingly, the PVF of an entire software resource is the fraction of bit-instructions in the resource that are ACE. For a particular software resource R with size B_R, the PVF over I instructions can be expressed as:

$$PVF_R = \frac{\sum_{I}(\text{ACE architecture-bits in } R)}{B_R \times I} \tag{2}$$

[1] However, in timing-critical systems, such effects will have to considered in worst-case execution time (WCET) analyses.

4.4 Confining the RCB

Given the source and machine code of a software-based fault-tolerance system, we expect to be able to employ the AVF and PVF metrics to determine the overall metric for the vulnerability of the RCB. Ideally, the PVF for all RCB components should be zero. Accordingly, it has to be known which hardware components the fault-tolerance software components are mapped to, so we can calculate the related AVF. Thus, confining the RCB includes *mapping* of code to hardware and software components as an important step.

Static and dynamic mapping approaches to ensure that dependability-critical code only executes on appropriate hardware components is an important future step in RCB-related research. Here, we expect to benefit from our previous experience with mapping applications to MPSoCs [BPS+10] and scratchpad mapping approaches.

5 RCB Minimization

5.1 What should be minimized?

As mentioned above, in security research, TCB minimization tries to reduce the amount of code. Consequentially, for minimizing the RCB, the PVF of the fault-tolerance software components should be minimized. It is an as yet open question what ideally should be reduced to achieve RCB minimization. Possible relevant parameters include the code size, the number of data accesses, the number of executed instructions, and the complexity of the underlying execution platform.

5.2 Embracing Heterogeneity

In the following paragraphs, we would like to propose two interesting design points for minimization, both of which are based on the heterogeneity of the underlying hardware platform.

One basic idea for RCB minimization is to embrace the heterogeneity present in MPSoC cores and memory hierarchies. A possible approach would be to constrain the execution of fault-tolerance methods to simple cores, which we expect to offer a significantly lower AVF. However, heterogeneity makes a designer's life harder, so the question is how much heterogeneity we are willing to afford. One useful approach are single-ISA heterogeneous multi-core architectures [KTR+04], which implement one instruction set architecture on a set of microarchitectures with different performance, energy, and reliability characteristics. This reduces the development overhead significantly, since only one compiler toolchain has to be supported and only one set of ISA-visible resources needs to be evaluated for an

analysis of the RCB.

Single-ISA heterogeneous architectures have left the research labs and have recently become commercially available. Examples include the ARM big.LITTLE architecture featuring Cortex A15 and Cortex A7 cores [ARM11] and NVidia's Tegra 3 MPSoC.

5.3 Heterogeneous CPU Cores

Multiple heterogeneous architectures have been proposed, especially for embedded systems. One prominent example is Texas Instruments' OMAP3 architecture, which integrates an ARM Cortex A8 RISC core and a C64x VLIW DSP core. However, such an architecture is comparatively complex to program for; a similar situation arises with the integration of RISC CPU and GPGPU cores. The already discussed *single-ISA* heterogeneous multicores implement the same instruction set (or a subset thereof), but differ significantly in their microarchitectural complexity and, thus, their performance, energy consumption, and architectural vulnearbility.

Accordingly, ARM Inc.'s big.LITTLE architecture [ARM11] uses two ARM processor cores that differ significantly in complexity. The simpler Cortex M4 core, shown on the left hand side of Fig. 1, has a short, non-superscalar pipeline, whereas the more complex Cortex A15 core, depicted on the right hand side of Fig. 1, features a rather long, complex, superscalar pipeline[2].

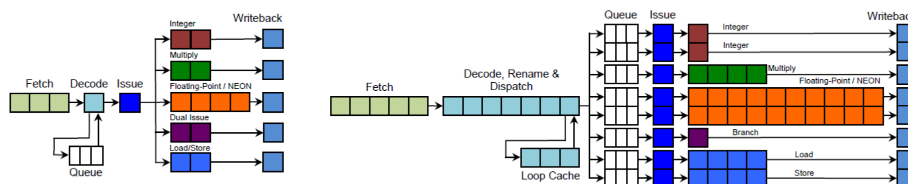
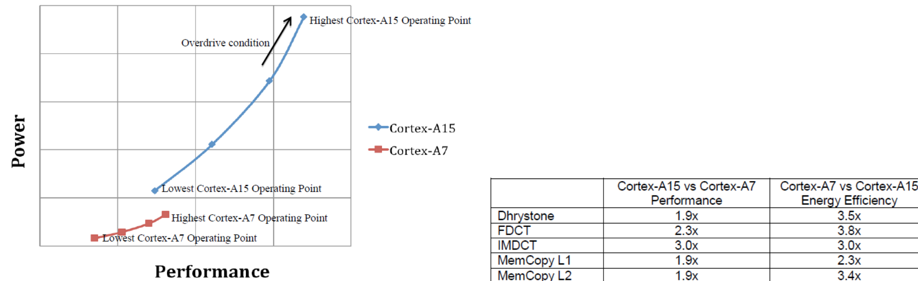


Figure 1: ARM big.LITTLE Cores: Cortex A7 (left) and A15 (right)

It is reasonable to expect that the vulnerability of the smaller Cortex M4 core in this architecture will be far lower than the vulnerability of the Cortex A15 core. However, both cores also differ significantly in their performance and energy consumption, as shown in Fig. 2. Both cores support the 16-bit THUMB instruction set. The Cortex A15, in addition, also supports 32-bit ARM instructions.

Thus, an open research question is how to map and schedule fault tolerance methods to the available cores. Parameters supporting these decisions will have to be determined using AVF analyses for the different processors and the related PVF analysis for the fault-tolerance methods.

[2]All figures in this paragraph are taken from ARM's whitepaper [ARM11]



	Cortex-A15 vs Cortex-A7 Performance	Cortex-A7 vs Cortex-A15 Energy Efficiency
Dhrystone	1.9x	3.5x
FDCT	2.3x	3.8x
IMDCT	3.0x	3.0x
MemCopy L1	1.9x	2.3x
MemCopy L2	1.9x	3.4x

Figure 2: ARM big.LITTLE Cores: Energy vs. Performance

5.4 Heterogeneous Memory Hierarchy

Another approach that embraces architectural heterogeneity is concerned with the memory hierarchy of a system. Here, the idea is to ensure that fault-tolerance software methods are exclusively executed in small memories that are protected from errors by hardware methods like ECC. Such memories could, e.g., be especially locked instruction or data cache lines or small, fast scratchpad memories. The rest of a system's memory, i.e., the vast amount of dynamic RAM, can stay unprotected if the software fault-tolerance methods can ensure the correctness of the data outside the protected memory.

One approach to ensure the validity of code and data is to constrain the execution of code and the currently accessed data to reliable scratchpad memories only. Accordingly, overlay techniques have to ensure that code and data copies into these memories are protected by software (algorithmic protection by checksumming, etc.).

In turn, this approach could also be used to further minimize the RCB. If the minimal code base to correct errors affecting fault-tolerance methods residing in unprotected memory could be determined and, in turn, this minimal code base would always remain in protected memory, this could form a sort of reliability bootstrap, requiring only a very small amount of code to permanently reside in protected memory. In turn, more space remains free for the actual program's execution. Here, we expect to be able to analyze and optimize the tradeoffs between performance and RCB size.

Scratchpad optimizations are a well-explored area of research for objectives like energy [PFV+07] or WCET [FK09] minimization. Based on overlay methods previously developed by TU Dortmund's DAES group [VM06], we expect to be able to provide approaches to minimize the RCB by ensuring execution of code in protected memories only and correcting upcoming errors in other memories using correction methods statically mapped to the reliable scratchpad memory.

6 Conclusions and Future Work

This paper attempts to define the concept of the Reliable Computing Base, which we consider an important paradigm for the design and implementation of software-based fault-tolerance approaches. We outlined the similarities with and differences from the TCB concept used in security research and proposed metrics and examples for minimizing the RCB.

Our next steps intend to make the RCB a useful, well-defined paradigm. This includes a solid formal definition of the RCB based on the presented metrics and an evaluation of different optimization methods, including the discussed exploitation of heterogeneous hardware structures.

Based on a formal definition, we would like to develop automatic methods to determine the RCB of a given system and devise automatic approaches to analyze parameters useful to minimize the RCB.

These future analyses require the provision of a detailed hardware and software component model in order to calculate reliable values for the AVF and PVF metrics. Depending on the granularity of the models, the definition of the RCB can also take place at different granularity levels. We expect the RCB to be a useful concept that spans the range from simple, single-core microprocessor systems to networks many-core machines.

7 Acknowledgments

This work was supported by the German Research Foundation (DFG) Priority Programme SPP1500 under grants no. MA-943/10 and HA-2461/8-1. The authors would like to thank the participants of the SPP1500 mini workshop "Dependability Analysis and Evaluation Techniques" for their input to an internal presentation of the ideas underlying this paper.

References

[52085] DoD 5200.28-STD. *Trusted Computer System Evaluation Criteria.* DoD Computer Security Center, December 1985.

[ARM11] ARM, Inc. Big.LITTLE Processing with ARM Cortex™-A15 & Cortex-A7. *ARM Whitepaper*, 2011.

[BPS+10] Christos Baloukas, Lazaros Papadopoulos, Dimitrios Soudris, Sander Stuijk, Olivera Jovanovic, Florian Schmoll, Daniel Cordes, Robert Pyka, Arindam Mallik, Stylianos Mamagkakis, François Capman, Séverin Collet, Nikolaos Mitas, and Dimitrios Kritharidis. Mapping Embedded Applications on MP-SoCs: The MNEMEE Approach. In *Proceedings of the 2010 IEEE Annual*

Symposium on VLSI, ISVLSI '10, pages 512–517, Washington, DC, USA, sep 2010. IEEE Computer Society.

[EM12] Michael Engel and Peter Marwedel. Semantic Gaps in Software-Based Reliability. In *Proceedings of the 4th Workshop on Design for Reliability (DFR'12)*, Paris, France, January 2012. HiPEAC.

[ESHM11] Michael Engel, Florian Schmoll, Andreas Heinig, and Peter Marwedel. Unreliable yet Useful – Reliability Annotations for Data in Cyber-Physical Systems. In *Proceedings of the 2011 Workshop on Software Language Engineering for Cyber-physical Systems (WS4C)*, Berlin / Germany, October 2011.

[FK09] Heiko Falk and Jan C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *The 46th Design Automation Conference (DAC)*, pages 732–737, San Francisco / USA, jul 2009.

[Hat95] Les Hatton. Computer Programming Languages and Safety-Related Systems. In *Proceedings of 3rd Safety-Critical Systems Symposium.* Springer-Verlag, January 1995.

[HL10] Gernot Heiser and Ben Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the First ACM Asia-Pacific Workshop on Systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM.

[HTS02] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel: the VFiasco project. In Gilles Muller and Eric Jul, editors, *ACM SIGOPS European Workshop*, pages 165–169. ACM, 2002.

[ITR] ITRS. Intl. Technology Roadmap for Semiconductors, `http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_ExecSum.pdf`.

[KAE+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, June 2010.

[KTR+04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *SIGARCH Comput. Archit. News*, 32(2):64–, March 2004.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. In *Proceedings of the Thirteenth ACM symposium on Operating Systems Principles*, SOSP '91, pages 165–182, New York, NY, USA, 1991. ACM.

[Lie96] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, September 1996.

[MWE+03] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, Washington, DC, USA, 2003. IEEE Computer Society.

[PFV+07] Robert Pyka, Christoph Faßbach, Manish Verma, Heiko Falk, and Peter Marwedel. Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications. In *10th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 41–50, Nice/France, April 2007.

[Rus81] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[SK08] Vilas Sridharan and David R. Kaeli. Quantifying software vulnerability. In *Proceedings of the 2008 Workshop on Radiation effects and fault tolerance in nanometer technologies*, WREFT '08, pages 323–328, New York, NY, USA, 2008. ACM.

[SK10] Vilas Sridharan and David R. Kaeli. Using hardware vulnerability factors to enhance AVF analysis. *SIGARCH Comput. Archit. News*, 38(3):461–472, June 2010.

[SPHH06] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, April 2006.

[VM06] Manish Verma and Peter Marwedel. Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. *IEEE TVLSI*, 14(8), 2006.