Stay Strong, Stay Safe – Enhancing Reliability of a Secure Operating System

Dirk Vogt*, Björn Döbel, and Adam Lackorzynski
Technische Universität Dresden
Department of Computer Science
01062 Dresden, Germany
{dvogt,doebel,adam}@os.inf.tu-dresden.de

ABSTRACT

Current research in operating systems focuses either on security or on reliability. However, modern embedded platforms demand solutions that suit both kinds of requirements. In this paper, we present *L4ReAnimator*, a framework that allows restarting crashed applications and reestablishing lost communication channels on top of the Fiasco.OC microkernel. It therefore effectively combines the already existing capability-based security architecture of Fiasco.OC with reliability features at a reasonable cost.

1. INTRODUCTION

Research in embedded systems and hardware indicates that future systems will be much more susceptible to errors than today's. Reasons—explained for instance in [1]—are smaller hardware structure sizes leading to a higher impact of radiation to transistor state, temperature-induced problems due to over-heating of some areas of the chip, higher alterations of transistor aging, and production-induced component faults. Therefore, hardware and software mechanisms need to be found that handle these errors in order to keep the systems functional.

While some errors, such as permanently non-functional hardware, at least partially need solutions at the hardware level, there is a large class of errors that are transient and can be fixed by restarting a failed component. This class of errors includes single event upsets at the hardware level as well as programming errors such as memory leaks.

Current research in operating systems either focuses on security or on reliability. Microkernels such as seL4 [8] and Fiasco.OC [9] implement capability-based systems providing fine-grained access control and improved flexibility. seL4 provides a certain amount of reliability by using a formally verified OS kernel. However, this verification does explicitly not take into account errors in user-level applications or errors in the underlying hardware. On the other hand, systems such as Minix 3 [7] and CuriOS [3] provide applications with efficient restartability in the situation of a crash.

In this paper we present L4ReAnimator, an extension to the L4 Runtime Environment (L4Re) running on top of Fiasco.OC. L4ReAnimator provides a framework to semi-transparently reintegrate crashed applications into a running system.

We give a short overview of Fiasco.OC and L4Re, our target operating system and discuss the problems with integrating restartability into a capability-based operating system in Section 2. We present the *L4ReAnimator* framework in Section 3, show that the required effort and performance overhead is reasonable in Section 4 and compare our solution to existing approaches to building reliable operating systems in Section 5.

2. CAPABILITIES IN L4RE

In this section we give an overview on Fiasco.OC and its capability system. Detailed information on the underlying design can be found in [9]. Due to space limitations, we only examine those features necessary for providing restartability here.

2.1 L4Re Overview

Our operating system platform comprises the Fiasco.OC microkernel and the L4Re user-level runtime environment. The system is organized as a set of interacting objects. The kernel provides spatial isolation between objects in form of *tasks*. The basic unit of execution is a *thread*. Objects interact by calling functions of other objects similar to the idea of object-oriented programming. This *invocation* is the only system call present in Fiasco.OC.

An example for the use of such objects is the kernel-provided *IPC gate*, an object that allows establishing a communication channel between threads. System-level components such as device drivers and protocol stacks implemented on top of Fiasco.OC use these objects to provide their services to other applications in terms of client-server relationships.

In order to maintain absolute control over object relationships, there are no globally accessible objects in L4Re. Instead, the microkernel manages a per-task table of *capabilities* referencing objects. Each task can denote the objects it has access to by their *capability slot* number in this table. Keeping the capability space local to the task prevents tasks from obtaining knowledge about the rest of the system by simply guessing global names and allows for a fine-grained setup of access rights to objects.

To ease capability management at the user-level, a task's

*Now at Vrije Universiteit Amsterdam

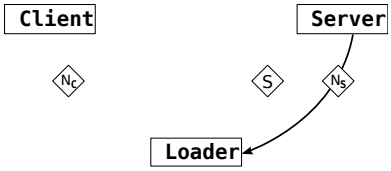

Figure 1: Session start up
The server creates a service management capability (S) and registers it in its name space

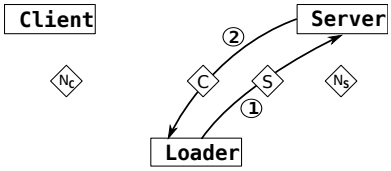

Figure 2: Session initialization
The loader initiates a session using the S capability (1). The server creates and returns a new session capability C (2).

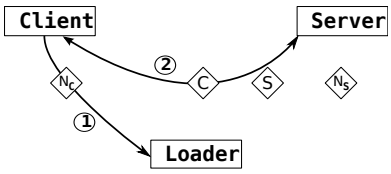

Figure 3: Session use
The client queries its name space for a service capability (1) and gets C mapped into its capability table. Thereafter, client and server use C for communication (2).

base set of capabilities is organized in a name space containing mappings from task-local names to remote object names. These mappings are maintained by a task's name space manager, which is usually identical to the task's creator.

An advanced feature of L4Re name spaces are *session capabilities*. These represent a dynamically created client-server communication channel. This dynamism allows servers to implement multiple services within a single task and to provide explicit per-client communication channels. Sessions are not created directly by the client, but by its name space manager. This allows for client implementations that are completely agnostic of how service capabilities are obtained and in turn for server implementations to be replaced without the need of modifying any clients.

2.2 Example Problem

Figures 1-3 depict the initiation of a client-server session. In this scenario, client and server are started by a binary loader that also acts as both tasks' name space manager by providing them with name service capabilities N_C and N_S respectively. Upon start up, the server creates a service management capability S and registers it in its name space (by invoking N_S) as shown in Figure 1. For the client to use a service it needs to be able to query the corresponding session capability in its name space. To provide such a capability, the name space manager in Figure 2 invokes a session open call through the service management capability. In response, the server creates a new session capability C and returns it

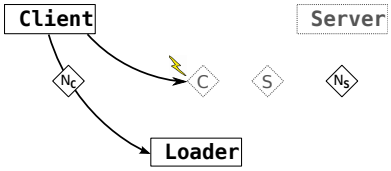

Figure 4: Crash
After a crash, the session and service capabilities get destroyed and client and loader possess dangling references to these capabilities.

to the loader. Later on, the client will query its name space for the required service as depicted in Figure 3. The loader then maps the C capability to the client and thereafter, client and server use C as a direct communication channel for the duration of the client session.

Figure 4 shows the situation after the server is terminated, for instance because an underlying error detection mechanism detected a soft error. Both, the service management capability S as well as the session capability C are destroyed. The first step in recovery is to restart the server. However, this does not suffice, because both, the loader and the client still have dangling capabilities to the old server instance's objects and therefore cannot continue normal operation, because future invocations of these missing capabilities will fail.

The system is lacking a mean to detect such *capability faults* and handle them properly. In the next section we introduce *L4ReAnimator*, a framework that steps in for help under these circumstances and takes care of re-establishing lost communication channels.

3. CAPABILITY FAULT HANDLING

In this section we review the requirements posed on a restart mechanism: fault containment, reintegration of restarted components, persistence of session state, and transparency. As we aim at retrofitting an existing system with such a mechanism, we consider the tradeoffs between the effort involved in implementing a feature and the respective benefit. Thereafter, we describe the concrete implementation of these features in *L4ReAnimator*, a restartability framework for L4Re.

L4ReAnimator focuses on the process of restarting failed applications. It is orthogonal to other approaches that cover error detection and we believe it can be combined with techniques such as software-encoded processing [14].

3.1 Restartability Requirements

Analyzing several existing fault-tolerant systems, we found that all of them can be assessed by how far they go in reaching four properties and how much effort these properties place on the system's users. We discuss the properties here and postpone discussion of the differences between our solution and existing systems to Section 5.

Fault containment aims at limitting propagation of errors throughout the system. Fiasco.OC provides fault containment by (1) using capability-based communication channels, and (2) separating tasks' address spaces. The former leads to explicit communication channels that can be interposed to add fault detection and checking if necessary. The latter leads to containment of memory access errors to a single task.

As we will explain in Section 3.2.3, the remaining problem is resource sharing between tasks, which is supported by Fiasco.OC and needs to be treated carefully with respect to containment.

Once a crashed component is restarted, it needs to be *reintegrated* into the running system. In terms of a capability system, this means, that all other applications that were using a capability for a service provided by the crashed component need to have these capabilities replaced with ones referring to objects within the new component instance. Additionally, Fiasco.OC supports sharing memory between tasks, so previously established memory mappings including the crashed component need to be updated as well.

Server applications usually keep a certain amount of client-related state. When restarting the server, this state needs to be rescued in order to transparently continue serving the client. This requirement is called *persistence*. We have implemented a checkpoint/restore facility that leverages L4Re's memory management infrastructure to store a checkpoint of a task's address space and the state of its threads and is able to recover from such a checkpoint upon application restart. This approach is similar to the transparent user-level checkpointing done by libckpt [10]. The mechanism is orthogonal to the main problem of recovering lost capabilities and we omit further details due to space limitations.

Another commonly mentioned requirement for a restartability mechanism is *transparency*. No application but the restarted one should be bothered to take part in the recovery process. The major disadvantage of a transparent mechanism is that the whole system needs to be properly designed to support it. When retrofitting an existing system with the ability to restart components, this would mean a huge effort of modifying existing components. On the contrary side, completely dismissing transparency would also mean a huge effort, because all clients needed to be modified in order to be aware of potential service faults and of the necessary steps to be taken upon encountering a fault situation.

In the rest of this section, we present *L4ReAnimator*, a framework that supports reintegration of L4Re applications into a running system using a *semi-transparent* restart mechanism. This means that the burden of implementing recovery code is placed on the server programmer, because she is best aware of the steps required to re-establish a client connection. This code can be provided to clients through a library, which in combination with the framework provided by *L4ReAnimator* allows clients to remain completely unaware of the recovery process.

3.2 Capability Fault Handling in L4Re

The *L4ReAnimator* framework consists of the components shown in Figure 5. As in every basic Fiasco.OC setup, the kernel maintains a set of kernel objects that are referenced by kernel-protected capabilities. An L4Re task includes a capability index table into whose slots the kernel maps capability references. The slot number is then used by the L4Re task to denote this capability. It is up to the user-level task to manage the slots in this table and decide, where capability are mapped to.

L4ReAnimator adds additional infrastructure to this setup, which we are going to describe in detail in the next subsections: (a) we add means to detect the loss of a capability, which we call a *capability fault*, (b) we provide functions containing the recovery code that is run upon a fault, which

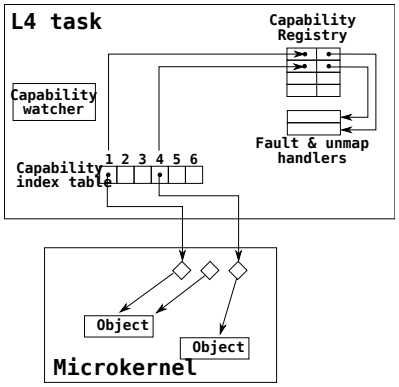

Figure 5: *L4ReAnimator* Architecture

we call *capability fault handlers* and functions which contain additional cleanup code that needs to be executed when a capability is lost, which we call *unmap actions*, and (c) a *capability registry* that maintains a list of mappings between active capabilities and the respective capability fault handlers and unmap actions.

3.2.1 Detecting Capability Faults

In order for clients to continue using a service capability beyond termination and restart of the service provider, we need a mechanism to notify clients about such situations. One approach to this problem is to implement a notification service that clients can subscribe to and that upon a restart is used to notify interested clients of this situation, so that they can adapt to this problem [7]. This has the disadvantage, that clients need to pay the performance overhead involved in reestablishing a channel even if they don't need this service right now. Therefore, we'd prefer to handle such situations lazily.

When a capability disappears, an application will be in one of two situations:

1. The application is currently not in the process of invoking the capability. In this case re-establishment of the capability mapping is postponed until the application invokes the capability again. This invocation will result in an error notifying the application that a non-existing capability has been invoked.

2. The application is currently blocked on a capability invocation. In this case the kernel will report an error indicating that the invocation was cancelled.

We detect both situations by hooking into L4Re's system call bindings. If we detect one of the two error conditions and L4ReAnimator is enabled in this application, a capability fault is raised. All this is done by the L4ReAnimator framework without client interaction.

3.2.2 Handling Capability Faults

Once a capability fault is raised using the previously described mechanism, the *capability registry* is used to look up a *capability fault handler* for the capability that caused the fault. The fault handler is a function that is executed to re-establish a lost capability mapping. In order to do so, the fault handler needs to know about the type of the underlying capability and about the protocol that is used for re-establishment.

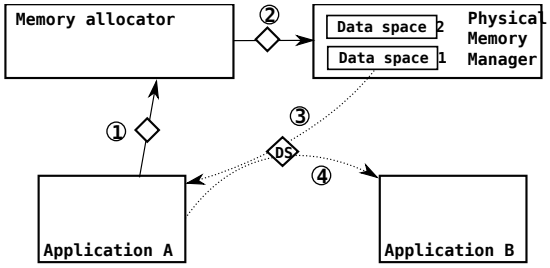

Figure 6: L4Re Memory management

Because of the specific knowledge involved, writing capability fault handlers is up to the implement or of a service. Our approach is to let server programmers write capability fault handlers for their applications and distribute them to clients among with their server-specific libraries. The *L4ReAnimator* framework provides the infrastructure to make using these fault handlers transparent to the client. The server library therefore only needs to register capability fault handlers whenever it uses L4Re's methods to allocate a capability slot. Thereafter, the framework makes sure that this handler is called upon a fault on this capability.

3.2.3 Reintegrating Shared Resources

In addition to communicating via capabilities, L4Re allows applications to share resources. This allows implementation of shared-memory communication channels such as in the scenario depicted in Figure 6.

Establishing such a channel takes place in three phases. First, the communication initiator (application A) requests the creation of a new shared memory area from its memory allocator (1). The allocator therefore creates a data space (an abstract memory container) from a provider such as the physical memory manager (2), in this case data space 1. The data space is represented by a capability (DS) that is mapped to application A in reply to its memory request (3). In order to establish a shared-memory channel to application B, A will map this capability to B as well (4). Now both applications attach the DS capability to their local address space manager and as soon as they touch the respective memory regions, a page fault will be raised and using the DS capability be routed to the physical memory manager that will then establish a mapping of physical pages.

If application A now terminates, all the capabilities it mapped to other applications are invalidated. This includes the mapping of DS to application B, so once B tries to raise a page fault in the shared memory channel thereafter, it will detect that the capability is invalid and use a capability fault handler to reestablish access to the channel. However, even though A crashed, the resources belonging to data space 1 are still present, because they belong to the memory manager which did not crash. This means that all existing page mappings from the memory manager to application B still exist and B may read and write these regions at will.

This may lead to a situation where B never realizes that A has been restarted: If the whole data space was mapped to B before A crashed, it will never again raise a page fault on this memory region, therefore never invoke the invalidated DS capability and thus never detect that the communication channel needs to be updated. So, even if A is successfully restarted and creates a new shared-memory channel (e.g., data space 2), B will never be able to use it.

The problem can be solved in three ways:

1. The physical memory manager mapped the DS capability to exactly one client: application A, which thereafter mapped it to B. The Fiasco.OC kernel provides a mechanism to detect whether a capability does still have child mappings. If A crashes, the DS mapping vanishes, the physical memory manager can use periodic checking of all its capabilities to detect such a situation and withdraw all resource mappings related to the dangling capability.
2. Application B can employ periodic checking of all the capabilities it knows to be valid and thereby detect whether one of these capabilities has become invalid. It can then free the related resources from its address space.
3. The Fiasco.OC kernel can be extended to send resource revocation notifications to either the creator of a capability (the physical memory manager), or the users of a capability (e.g., application B) and these can handle such messages by revoking corresponding resource mappings.

We currently use the second approach to solve this problem as it places the overhead of periodic checking on the users of a resource. Thereby clients can chose whatever checking period seems suitable for their purposes and the resource manager does not become a central bottleneck because of one client's reliability requirements.

L4ReAnimator supports this solution by running a *capability watcher thread* within each address space. Resource-related capabilities can be registered with this thread and it will then periodically check their validity and raises a capability fault upon an error. The capability registry allows that, in addition to a capability fault handler, for each capability slot a *resource unmap handler* can be registered, which is used to revoke all resource mappings related to a capability once it becomes invalid.

3.3 Capability Fault Handlers

One of the early experiences we already made using *L4ReAnimator* is that in many client-server scenarios there is no need to implement a protocol-specific capability fault handler, but a generic one suits the application's needs. A common scenario in an L4Re application is to obtain a capability using a name space query. Such capabilities can be reintegrated by simply re-querying the name space manager for the corresponding name. L4ReAnimator therefore provides a generic *name space capability fault handler* that can be registered upon a successful name space query.

Another common operation is opening a client session. In this case, we need to keep track of the service capability that is used to perform the session open call and of the parameters used for establishing the session. Again, this can be done generically by tracking session open calls and registering a respective *session capability fault handler* after successful session establishment.

Additionally, we also implemented several handlers for more complex protocols. One example is a shared-memory consumer-producer buffer, which is implemented using a shared memory buffer (represented by a data space capability) and two capabilities for signalling (one for the consumer, one for the producer). This can be solved by registering the same capability fault handler for all three capabilities

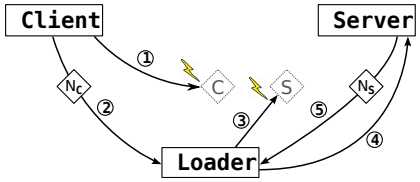

Figure 7: Re-establishing a server session

because depending on which capability is invoked first, reintegration of the buffer needs to take place. Furthermore, capability watching is used to detect if the shared memory buffer disappears, and an unmap action for the shared buffer is needed that unmaps the respective memory pages.

3.4 Recursive Reanimation

The framework presented so far allows us to resolve the situation initially described in Section 2.2 and Figure 7. During session setup, client and loader register capability fault handlers for capability C. The client obtains C through a name space lookup and therefore registers a name space capability fault handler. The loader obtains C by invoking a session open call through the service capability S and therefore registers a session capability fault handler for C.

Once the client invokes the session capability C and detects that it has become invalid (1), a capability fault is raised and the respective name space capability fault handler is called that retries to look up the name for this capability in the name space N_C (2). The loader being also the client's name space manager will try to map C again and detect C's invalidity as well. This will raise another capability fault, this time invoking the loader's session capability fault handler, retrying to open the client session using capability S (3). However, this will only succeed after the server has been restarted (4) and registered a new service capability through its name space (5). Once this is done, the session is re-opened, a new session capability C is created and client and server can use this capability to communicate.

This recursive approach to reintegrating components into the system fits well into the hierarchical resource management policies traditionally used in L4-based microkernel systems [5].

3.5 Generalization to Other Systems

Although our implementation focuses on Fiasco.OC and L4Re as a target system, we believe our ideas can be applied to other capability systems such as seL4 [8], or Genode [4]. These systems share the common concept of using capabilities for referencing objects. Hence, they also share the need for reintegrating a component after restart by reviving existing communication channels.

In order to reuse L4ReAnimator on an arbitrary capability kernel, two features need to be provided: (1) invocation of an invalid capability needs to raise an error, and (2) user-level needs to be able to check the validity of a capability. (1) is necessary for lazily re-establishing a communication channel, (2) is required for reintegrating shared resources, which can also be achieved using kernel-provided resource revocation notifications.

4. EVALUATION

We deem the effort for implementing L4ReAnimator to be reasonable. The whole C++ implementation of capability registry, capability watcher thread, hooks within the L4Re system call bindings, and the generic capability fault handlers comprises about 1,600 lines of code.

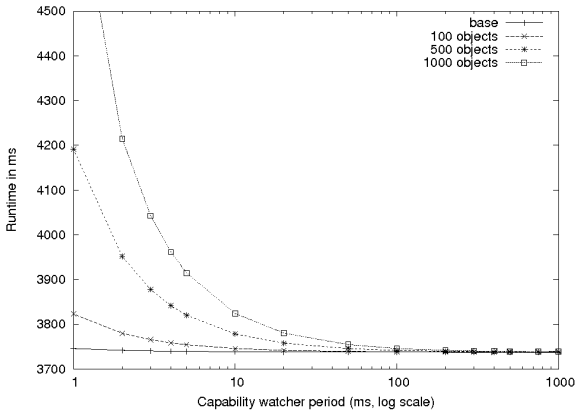

Figure 8: Capability watching overhead

We furthermore performed microbenchmarks to evaluate L4ReAnimator's influence on performance. IPC between tasks is a common operation in a microkernel-based operating system and we found that the system call instrumentation L4ReAnimator employs to detect invalidated capabilities leads to an 8.66% overhead for a CPU-local IPC operation on an AMD Phenom X3 clocked at 2.11GHz.

Additionally, we measured the overhead imposed by the capability watcher mechanism, which occurs due to the watcher thread periodically waking up and checking all watch objects. Figure 8 presents the results for a CPU-intensive application (matrix multiplication) and shows that this overhead grows with the number of watched objects. Even for 1,000 watched objects the overhead becomes negligible with a high-enough wake-up period. The choice of the right watch period constitutes a trade-off decision between the involved overhead and the error detection and recovery latency.

5. RELATED WORK

The idea of restarting an application for fixing transient errors is nothing new and has already been adapted to operating systems components. In this section we analyze existing solutions and compare them to our design.

The *BirliX* operating system [6] architecture is a distributed system comprising of objects. Each object inherits from a primary type that also provides generic means of checkpointing and recovering an object. Objects interact through RPC via communication channels identified by globally unique IDs. This enables re-connecting objects after a crash at the cost of sacrificing the security advantages of having only task-local name spaces as provided by modern capability systems. Our work combines object-level restartability with an existing capability-based access control mechanism in order to achieve security and fault tolerance.

Nooks [13] is an extension to the Linux kernel putting device drivers into dedicated lightweight protection domains that are separate from the rest of the kernel. A layer of wrapper functions intercepts all calls between driver domains and the kernel and maintains replicas of all data shared between those domains. Further additions employ *shadow drivers [12]* that log all interaction between a driver and its clients. Upon driver failure, the shadow driver manager initiates a restart. During recovery, the shadow driver handles all client requests to the driver. Additionally, the shadow driver's log is used to reset the device to the state it was in before the crash.

While it has been shown that Nooks provides restartability at a reasonable performance cost, it does explicitly not target malicious software and the effort involved in implementing the wrapper layer and shadow drivers is non-negligible. Our work explicitly targets a secure operating system and tries to keep the effort for providing restartability as low as possible by providing a generic framework for programmers to plug their restartable applications into.

Minix 3 [7] is a microkernel-based operating system explicitly designed for supporting restartability of its components. A reincarnation server keeps track of the system state and detects crashed components at termination or using a heart beat mechanism. A data storage server enables components to store their state across instantiations. Recovery of a crashed application is performed by the reincarnation server, which also notifies interested clients of this situation. The burden of re-establishing communication channels to the crashed components is then completely upon these clients, which eases implementation of the recovery process, but hardens life for the client. However, most of the recovery work can be hidden in libraries. Our work shares Minix' idea that fully transparent application restart can only be achieved with a large effort and that less transparent solutions exhibit a better cost-benefit ratio.

EROS [11] is similar to the operating system used in this work in that it uses capabilities to enforce access control at the object level. EROS also takes into account fault tolerance by incorporating a mechanism to create checkpoints at runtime. These checkpoints always include the whole running system. This eases reinstantiation, because one does not need to care about re-establishing capability mappings for single components. Our approach provides a more fine-grained level of restartability, by allowing to restart and reintegrate single objects.

The *CuriOS* microkernel [2, 3] achieves fully transparent restart of applications, but does not incorporate any access control mechanism. Transparency for clients of a crashed component is implemented by always locating the failed component at the same memory address, so that clients can keep using references to objects over multiple instances. Kernel-provided server-state regions are used to keep client-related server state independent from restarted servers and by controlling when these regions are accessible by the server, CuriOS prevents wild overwrites caused by programming bugs. The price paid for these features is that all applications need to be rewritten to adhere to CuriOS application model. While being reasonable for designing a new operating system, this effort appears to be too high when retrofitting an existing system with restartability. We chose to reduce this effort drastically by giving up on the full-transparency requirement in *L4ReAnimator*.

6. CONCLUSION

In this paper we presented L4ReAnimator, a generic framework for providing restartable applications within the L4Re runtime environment. In order to keep the effort for enhancing the existing system small, we dismissed the goal of providing a mechanism that is fully transparent to the client. Instead, we use semi-transparent restart, a mechanism that requires service implementers to write and deploy reintegration code in form of capability fault handlers. For clients, L4ReAnimator provides a generic framework that allows them to use service-provided fault handlers without further modifications to the client.

Using L4ReAnimator we enhanced the capability-based L4Re operating system with the ability to reintegrate restarted components into a running system at a reasonable cost.

Acknowledgements

This research was sponsored by the EU FP7 project eMuCo.

7. REFERENCES

[1] Borkar, S. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro 25* (2005), 10–16.

[2] David, F. M., and Campbell, R. H. Building a self-healing operating system. In *DASC '07: Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 3–10.

[3] David, F. M., Chan, E., Carlyle, J. C., and Campbell, R. H. Curios: Improving reliability through operating system structure. In *Usenix Symposium on Operating Systems Design and Implementation* (2008), R. Draves and R. van Renesse, Eds., USENIX Association, pp. 59–72.

[4] Feske, N., and Helmuth, C. Design of the Bastei OS architecture. Tech. Rep. TUD-FI06-07-Dezember-2006, TU Dresden, 2006.

[5] Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K., Uhlig, V., Tidswell, J., Deller, L., and Reuther, L. The SawMill multiserver approach. In *ACM SIGOPS European Workshop 9/00* (2000).

[6] Härtig, H., Kühnhauser, W. E., Lux, W., and Reck, W. Operating System(s) on Top of Persistent Object Systems — The BirliX Approach —. In *Proceedings of 25th Hawaii International Conference on Systems Sciences* (1992), vol. I, IEEE Press, pp. 790–799.

[7] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. *Reorganizing UNIX for Reliability*, vol. 4186/2006 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 81–94.

[8] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, Oct. 2009), ACM, pp. 207–220.

[9] Lackorzynski, A., and Warg, A. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (Nuremberg, Germany, 2009), ACM, pp. 25–30.

[10] Plank, J. S., Beck, M., Kingsley, G., and Li, K. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference* (January 1995), pp. 213–223.

[11] Shapiro, J. S., Smith, J. M., and Farber, D. J. Eros: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles* (New York, NY, USA, 1999), ACM, pp. 170–185.

[12] Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. Recovering Device Drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004).

[13] Swift, M. M., Bershard, B. N., and Levy, H. M. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)* (Bolton Landing, NY, Oct. 2003), pp. 207–222.

[14] Wappler, U., and Fetzer, C. Software encoded processing: Building dependable systems with commodity hardware. In *Lecture Notes in Computer Science on Computer Safety, Reliability and Security (SafeComp 2007)* (2007).