We Crashed, Now What?*

Cristiano Giuffrida
Department of Computer Science
Vrije Universiteit, Amsterdam
giuffrida@cs.vu.nl

Lorenzo Cavallaro
Department of Computer Science
Vrije Universiteit, Amsterdam
sullivan@cs.vu.nl

Andrew S. Tanenbaum
Department of Computer Science
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

Abstract

We present an in-depth analysis of the crash-recovery problem and propose a novel approach to recover from otherwise fatal operating system (OS) crashes. We show how an unconventional, but careful, OS design, aided by automatic compiler-based code instrumentation, offers a practical solution towards the survivability of the entire system. Current results are encouraging and show that our approach is able to recover even the most critical OS subsystems without exposing the failure to user applications or hampering the scalability of the system.

1 Introduction

As a result of their ever-growing complexity, modern operating systems are constantly plagued by bugs. Empirical studies have determined that the size of the Linux kernel almost doubles between major releases, and so do the number of bugs [1]. Unfortunately, a bug in the OS code can potentially lead to a crash during normal operation, resulting in serious consequences ranging from data loss for PC users to disruption of service for high-availability systems. In commercial systems, downtime directly translates to revenue loss [11]. In mission-critical systems, disruption of service can lead to catastrophic consequences, including an electricity blackout affecting millions of people [12]. Unfortunately, crash recovery has proven to be an extremely challenging problem, especially when dealing with many classes of failures and protecting a large fraction of the OS.

By nature, a crash is an unpredictable event that occurs when the system is in an arbitrary state and can produce a number of undesirable effects a recovery solution must deal with. First, a crash can result in data inconsistencies. Consider, for example, some code adding an element to a linked list. The operation is clearly not atomic and a crash occurring during an update can leave the list in an invalid state. Second, part of the state can get corrupted as a result of a memory error. Consider, for example, a buffer overflow that overwrites critical data. Finally, errors and inconsistencies can easily propagate to different subsystems than the one where the bug originated.

The majority of the approaches to crash recovery described in the literature attempt to incorporate some form of explicit recovery support in the OS. The recovery infrastructure is typically tailored to specific components, including drivers [5, 14], OS extensions in general [18], and file systems [13]. The obvious benefit of an ad-hoc recovery infrastructure is the ability to drastically reduce the complexity one has to deal with at recovery time.

The problem with these approaches is their limited scope. Generalizing these techniques to the entire OS to build a high-coverage crash recovery infrastructure would introduce significant complexity and directly expose a considerable portion of the recovery code to the OS programmer. Experience with this model goes way back, with the famous quote from Tom Van Vleck admitting that half the code he was writing in the Multics operating system was solely dedicated to recovery [16]. Since the number of software bugs is roughly linear with the number of lines of code [10], introducing a lot of recovery code creates a vicious circle. In addition, these approaches suffer from poor maintainability and expandability properties. When interfaces between subsystems change, the recovery code must be adapted accordingly or potentially be rewritten from scratch.

In a completely different direction, researchers have looked at strategies to build more generic recovery infrastructures to transparently protect a large fraction of an OS. Successful approaches have used instrumentation techniques [8] or kernel replication combined with forms of live migration in face of a crash [3]. The key benefits are backward compatibility with most existing operating systems, low exposures of the recovery infrastructure to the OS programmer, and potentials to overcome the maintainability issues depicted earlier.

*This work has been supported by The European Research Council under grant ERC Advanced Grant 227874.

The main problem with transparent techniques is the evident tradeoff between the number of recovery scenarios considered and the complexity of the recovery infrastructure. For instance, best-effort strategies, such as the one proposed in [3], offer low complexity but are not generally concerned with state inconsistencies that can easily trigger another crash. On the other hand, pure instrumentation-based approaches drastically reduce the number of assumptions on the nature of a crash but incur very high complexity. As expected, with more complexity comes a higher overhead and poor scalability due to potential interrequest dependency tracking in the kernel.

In this paper, we explore a different approach to crash recovery that attempts to mitigate the weaknesses of both models and offers a potential solution to the general problem. Our first contribution is a clear definition of crash recovery with a logical decomposition of the problem into individual subproblems. Our second contribution is a novel approach to high-coverage crash recovery for operating systems that provides appealing reliability, scalability, and maintainability properties. Our guiding principle is to leverage a carefully planned OS design, which does not expose the recovery infrastructure to the OS programmer, but drastically reduces the complexity of the problem space considered and allows effective crash recovery using automatic instrumentation techniques in a nonintrusive way.

The key novelty of our work revolves around combining the benefits of both design-based and instrumentation-based approaches. When compared to common design-based techniques, our approach shows considerably higher coverage without significantly increasing the complexity of the recovery infrastructure or imposing more burden on the programmer. When compared to pure instrumentation-based techniques, our approach uses a lightweight instrumentation recovery infrastructure greatly reducing its overall complexity, without, at the same time, hampering the scalability properties of the system. A prototype and preliminary results confirm the viability of our approach.

2 The Crash-Recovery Problem

Despite much attention dedicated to the problem of crash recovery in the literature [2, 3, 5, 8, 13, 14, 18], very little effort has been put into clearly identifying all the subproblems associated to define a roadmap a solution should follow to be comprehensive and practical. In the following, we propose a possible breakdown of the problem space into common subproblems, which we believe every dependable crash recovery solution should carefully consider.

Crash detection. A crash is a fatal condition that occurs when a piece of software stops performing the activities it has been designed for. A recovery solution can either detect such conditions proactively (i.e., before crashes actually occur), or reactively using hardware-based or software-based techniques. A detection mechanism should isolate crashes properly not to let them interfere with the recovery process itself.

State transfer. To be able to survive a crash, a recovery solution must establish an appropriate execution context where the system can resume operation. To avoid service disruption, the state of the OS must be transferred from the faulty execution context to the new execution context.

State consistency. When the old execution context is interrupted prematurely by a crash, the transfered state could reflect an arbitrary point in execution and be unsuitable to resume operation in a deterministic way. A crash recovery solution should guarantee that no state inconsistencies are present and allow for deterministic execution from a stable state in the new execution context.

State dependency tracking. Depending on the architecture of the OS, multiple execution contexts (e.g. kernel threads) may be running at the same time and create complex state dependencies among each other. A crash recovery solution should guarantee these dependencies are consistent and ensure that all the OS subsystems have a coherent view of the global state when the recovery procedure completes.

State corruption. A consistent state with respect to a stable execution point is not guaranteed to be error-free. Arbitrary data corruption might have occurred before the crash. A crash recovery solution should attempt to limit, detect, and possibly recover from any form of corruption.

Restart. A crash recovery solution must define a safe execution point in the new context to resume operation. In addition, a dependable restart strategy should attempt to avoid further contingent crashes. For example, deterministic replay of the original execution under the same conditions would inevitably reproduce the same crash.

3 Our approach

Our approach is to engineer the OS to dramatically reduce the complexity of crash recovery. While the idea of leveraging OS support to simplify recovery has been investigated before [2, 5, 13, 14, 18], we differ in that our design scales up to nearly the entire OS and neither failures nor recovery code are directly exposed to the programmer.

First, we have broken down the OS into a collection of separate components with well-defined interfaces. Each component retains a certain amount of private state that is never explicitly shared with other components. To enforce the strongest level of isolation possible, we segregate each component in a user-space process with its own private address space protected by the MMU. This property leads to a microkernel-based multiserver design with most of the OS running in user space.

Our goal is to push this approach to the extreme, moving all the critical OS subsystems to user space, including drivers, file systems, networking, process management, virtual memory management, and even scheduling. This is advantageous for at least two reasons. First, since our design aims to provide component-agnostic crash recovery for *all* the stateful user-space components, pushing more subsystems to user space increases the coverage of our recovery technique. In addition, this approach dramatically reduces the size and the complexity of the (micro) kernel, which only provides interprocess communication (IPC) and basic low-level resource management. The very limited size of the resulting microkernel could make it practical to verify the code using formal techniques and guarantee that the microkernel itself is free of programming errors [6].

While the reliability properties of microkernel-based architectures have been long known and studied, we believe little effort has been put into building a high-coverage crash recovery infrastructure that solves many of the aforementioned state management problems while imposing only minimal burden on the OS programmer. We believe a careful design and the support of emerging compiler-based instrumentation techniques can help to fill the gap, as our approach demonstrates.

The second organizing principle we rely on is an abstract communication mechanism to provide failure-resilient IPC to OS components. In our design, components communicate only via message passing. Each component is assigned a virtual identifier that is stable even in the event of a crash. When a component crashes, the system allows a new instance to take over and resume normal execution after recovery. The new instance is a separate process automatically inheriting the same virtual identifier, so that all the pending IPC messages are transparently rerouted by the kernel. Transparency of addressing is essential to avoid exposing failures to client components, as also recognized in prior work [2].

A key assumption in our design is the programming model adopted for OS components. Every component strictly adheres to a pure event-driven model with well-defined properties. The model revolves around a simple task loop: receive a message, process it, and go back to the top of the loop to wait for another message. The task loop is always single threaded but short-lived to achieve the maximum throughput possible. Note that this design does not hamper the ability to efficiently implement support for application-level threads, nor does it rule out different OS components running on different cores, a subject for future research.

While processing a message, a component may need to contact the kernel or another component to fulfill the request. In our model, messages originated while processing a request are pushed to the end of the task loop by design. A component will typically receive a message, store information about the pending request in its local state, send a message to a dependent component required to complete the request, and go back to the top of the task loop to serve another request. When the dependent component sends a reply, the client can continue processing the original request.

Our design is inspired by continuation-based programming [9] and effectively treats each OS subsystem as a state machine. This approach provides extremely appealing properties for state management. First, at the top of the task loop, every component is in a stable and coherent state that unambiguously identifies its persistent data and state of execution. Second, transitions from state to state are well-defined and explicit in the programming model. Finally, state transitions are always coherent across components. Since request-originated messages are pushed to the end of the task loop, a dependent component will only receive a message and have the opportunity to transition to a new state when the client has already transitioned to a new stable state itself. We only allow messages with idempotent semantics to originate in the middle of the task loop. These messages do not give rise to state transitions in dependent components, thereby preserving the consistency of the global state of the system.

In our model, the task loop represents the recovery window. Our ultimate goal is to support transparent and component-agnostic recovery from crashes occurring anywhere during the execution of the task loop. When the recovery procedure completes, execution must restart at the top of the loop with the same state the component had upon reception of the last message before the crash. We want to resume execution as though the crash never occurred and the last run of the task loop was never even started. By design, the top of the loop is a global safe execution point, with both the component and the rest of the system in a coherent state.

To be able to restore the last stable state of the component, we need a mechanism to roll back all the local changes that occurred within the last run of the task loop. To avoid exposing the details of the recovery infrastructure to the OS programmer, we employ *automatic compiler-based instrumentation* techniques to track every change to the state of the component transparently. Our instrumentation infrastructure is lightweight and used only to track local state changes in a component-agnostic and dependency-agnostic way (see below).

The recovery procedure is initiated when a crash is detected in one of the OS components. As in prior work, our failure model assumes one failure at a time and trusts the recovery and instrumentation code. It is worth noting that, however, our model can also be used to survive multiple concurrent failures, but the approach becomes best-effort, since assumptions normally made

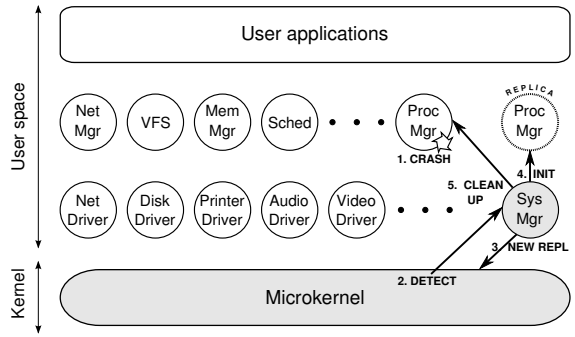

Figure 1: The crash recovery process in our OS model.

during the recovery process may no longer hold (e.g. the system manager tries to fork a new copy of a dead driver without knowing that the process manager has also crashed). More sophisticated dependency resolution policies to tackle this problem are part of our future work.

To avoid increasing the complexity of the kernel, the recovery code is executed entirely in user space. Part of the recovery code is encapsulated in the *system manager*, a separate OS component that coordinates the entire recovery process. In detail, the system manager: (i) selects a valid replica of the component as a new process; (ii) informs the kernel of the new process to rebind the virtual identifier, and make the new process runnable; (iii) sends an initialization message to the new process requesting to start in face of a crash; (iv) cleans up the old process and creates another replica if necessary. Replicas are only created beforehand for critical OS components that are themselves part of the process creation protocol. For the others, replicas are created on demand to minimize resource consumption.

The remaining part of the recovery code is isolated in a generic library all the OS components are linked against. The library hides all the details of the initialization protocol with the system manager, transparently executing the component's startup code on a fresh start and the recovery code after a crash. The code in the recovery library restores the component to a coherent state and jumps to the beginning of the task loop right after, to resume normal execution. The careful reader may have already noted that such a deterministic strategy would lead to the component constantly crashing in case of nontransient failures, e.g., as a consequence of an attacker exploiting memory corruption vulnerabilities. While a more comprehensive analysis is subject of our future work, we advocate the adoption of per-component policies to solve this problem and improve the quality of the recovery process, as better detailed in the next section.

Figure 1 depicts the architecture of the system and the steps of the recovery process.

4 Practical Crash Recovery

In this section, we revisit the problem space analyzed earlier and discuss the benefits of our approach.

Crash detection. We adopt a uniform mechanism based on runtime exceptions to detect crashes in OS components. CPU and MMU exceptions are triggered directly by the hardware when unexpected conditions occur. Software exceptions can be raised by the component itself (e.g. library calls like `exit()` or `panic()` or assertion failures), or eventually be reported to the kernel by another trusted OS component (e.g. feedback from the scheduler to detect a component stuck in an infinite loop). All the hardware and software exceptions are intercepted by the kernel, which informs the system manager to initiate recovery. Thanks to address-space isolation, crashes cannot propagate and the recovery process can always be activated correctly. We focus specifically on crashes in our approach, and do not attempt to catch byzantine failures, e.g., random or malicious behavior. Actually, we believe this problem is orthogonal to our current work and can be addressed with other strategies, for example, using automatically inserted assertions to immediately catch an invalid state of a component [18].

State transfer. State transfer is entirely implemented in the recovery library that performs interaddress space copy from the dead instance to inherit the state. Both the data and the heap are blindly copied from the old instance in a protected manner, using a capability-like design.

State consistency. The recovery library uses information generated by the instrumentation code to restore the last stable state correctly. A number of implementation strategies are possible here. To date, we have experimented with deferred writes replication at the object-level, as discussed in more detail in the next section. We point out that the library can transparently revert all the changes occurred in the last run of the loop and thus guarantee that the component state is consistent. This approach requires lightweight instrumentation code at the cost of only being able to revert the state of a component to the beginning of the last recovery window. For example, a component may enter a tainted state and crash a number of recovery windows later, possibly propagating the taint further across the components it is interacting with. Should this happen, our framework is only able to restore the last stable state of the component that may be as well remain tainted. To address this problem, it is possible to use component-specific recovery procedures to recover from a tainted state, or use assertions to immediately catch invalid states and not to let them spread to other components or across recovery windows. Using larger recovery windows that spam across several events is not a pain-free solution, since such an approach would require high-overhead techniques (e.g. checkpointing) to

consistently record the global state along with any inter-component dependencies.

State dependency tracking. Our design requires no explicit dependency analysis to enforce global coherence of the state. When restoring the last stable state of the component, the rest of the system is always consistent since transitions to a new stable state are always coherent across components. The absence of explicit dependency analysis is crucial to keep the programming model simple and preserve the scalability properties of the system.

State corruption. The design of techniques to prevent, detect, or recover from state corruption depends on the threat model. While an extensive analysis is outside the scope of this paper, we point out the potential of our approach in this respect. First, address-space isolation helps confine the potential propagation of the error by design. Second, corruption management mechanisms can be effectively integrated in our lightweight instrumentation infrastructure on a local scale. Examples include interaddress space replication, checksumming, or instrumented runtime checks [4, 17]. The evaluation of these techniques is part of our future work.

Restart. Our approach implements a *soft* rollback to resume execution at the top of the last task loop using only local information. The ability to deterministically resume execution at a fixed stable execution point makes it practical to implement several policy-based restart strategies and avoid subsequent crashes in the next run. For instance, if we only aim to address transient failures (e.g. hardware faults), the recovery library can simply replay execution. In other cases, an option is to immediately reply to the message with a valid error code to perform request-oriented recovery similarly to [8]. An alternative is to let the OS programmer register callbacks to check the local state for consistency and decide whether to reply to the message, replay execution, or select a different strategy to process the request (if any). We remark how the programming model exposed to the programmer is effective and functional, thanks to the localized and stable state the callback has to deal with. Consider, for example, the unbearable alternative of having the programmer implement consistency checks for the entire system at an arbitrary execution point. The comparison and evaluation of different restart strategies is part of our future work.

5 Current Results

We have prototyped our ideas on MINIX 3, a microkernel multiserver POSIX-conformant OS that runs on commodity x86 hardware [15]. We have restructured every OS process to fit our event-driven model and prototyped the crash recovery infrastructure in user-space. We added support for our generic crash detection mechanism in the microkernel with minimal effort.

Preliminary results showed that our approach is able to restart even the most critical OS components flawlessly during normal system operation, keeping the system fully functional and without exposing the failure to user processes. For instance, our approach can successfully restart the process manager (PM), which stores and manages the most critical information about all the running processes—both regular and OS-related—in the system. Our preliminary experiments showed that the global state of PM was always correctly restored upon restart and no information was ever lost. We point out that failures were conveniently induced, but simulating more realistic scenarios via fault injection is part of our ongoing work.

In our prototype, the instrumentation code is implemented as a series of compiler passes using the LLVM compiler framework [7]. A first pass is used to instrument code and store information about the state objects of each OS component. To this end, we record information about global variables, keep track of dynamic memory allocation, and allocate a *shadow* region of memory to store the last stable state. The second pass is used to insert the code to keep the shadow region consistent and up to date. Several implementation strategies are possible for this purpose.

To date, we evaluated the practicality of our approach with a basic strategy that associates a flag with each state object, marks the flag as dirty whenever an object is updated, and copies all the dirty objects to the shadow region at the end of the task loop. We use alias analysis to determine the set of objects to flag as dirty at every occurrence of a store instruction in the code. We remark that our lightweight instrumentation introduces only a fixed shadow region and flags that are used to commit changes at the end of each task loop without requiring any additional log or journal.

We point out that this basic strategy is implemented at a coarse level of granularity and may incur significant overhead depending on the activity performed. The evaluation of more efficient finer-grained strategies is part of our ongoing work. However, our focus here is not on the overhead—which we are optimistic about significantly reducing—but rather on the scalability properties that are *innate* in our model. Unfortunately, scalability is easily undermined when interrequest dependencies across different kernel subsystems must be tracked as in [8], but, as described earlier, our model does not require such a complication and is aimed to build a dependable, as well as scalable, crash recovery solution.

To demonstrate the scalability properties of our design, we evaluated our approach with postmark—reproducing a I/O intensive workload, and a POSIX test suite—reproducing a system-call intensive workload.

Figure 2 shows the relative runtime overhead induced

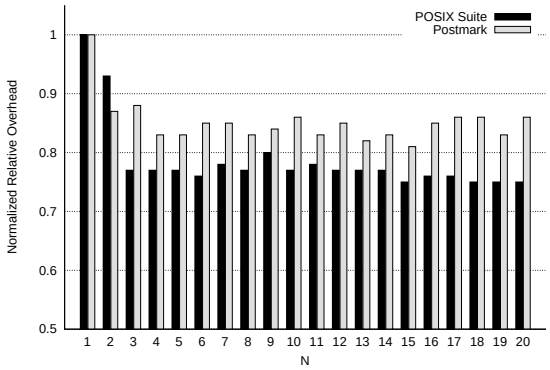

Figure 2: Scalability impact under parallel workloads.

by our instrumentation code compared to the baseline system, under different parallel workloads generated by N concurrent instances of the same benchmark. The distribution plotted is normalized against the relative overhead of a sequential run of the benchmark. As depicted in the figure, the relative overhead is constant, indicating no impact of our approach on scalability, regardless of the particular benchmark and workload intensity considered. This demonstrates the benefit of our approach, which does not require explicit dependency analysis to ensure global consistency of the state at recovery time. In contrast, other approaches *exclusively* relying on instrumentation techniques [8] need to track all the reads and writes and their contextual dependencies. We believe the resulting nonnegligible scalability impact is ill-suited to high-availability applications. Poor scalability easily translates to significant reduction in terms of system availability under intensive parallel workloads, which in turn results in a less dependable system.

6 Conclusions

In this paper, we discussed many of the issues associated with crash recovery and argued that new solutions with better dependability properties are needed. To this end, we presented a novel approach that employs an event-driven OS design to dramatically simplify state management. The result is a well-defined recovery window with frequent stable execution points where the state is globally consistent. To recover from arbitrary crashes, we revert the state of the faulty component to the last stable state. Thanks to the design adopted, this operation does not require expensive checkpointing nor complex dependency analysis, but can be implemented via nonintrusive instrumentation techniques, as our prototype demonstrates.

References

[1] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An Empirical Study of Operating Systems Errors. In *Proc. of the 18th ACM Symp. on Oper. Systems Prin.* (2001).

[2] DAVID, F. M., CHAN, E. M., CARLYLE, J. C., AND CAMPBELL, R. H. CuriOS: Improving Reliability through Operating System Structure. In *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation* (2008).

[3] DEPOUTOVITCH, A., AND STUMM, M. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proc. of the Fifth ACM European Conf. on Computer Systems* (2010), ACM.

[4] DHURJATI, D., KOWSHIK, S., AND ADVE, V. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. *SIGPLAN Not.* (2006).

[5] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Failure Resilience for Device Drivers. In *Proc. of the 37th Annual IEEE/IFIP Int'l Conf. on Dep. Systems and Networks* (2007).

[6] 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. of the 22nd ACM Symp. on Oper. Systems Prin.* (2009), ACM, pp. 207–220.

[7] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization* (2004).

[8] LENHARTH, A., ADVE, V. S., AND KING, S. T. Recovery Domains: an Organizing Principle for Recoverable Operating Systems. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2009), ACM.

[9] MILNE, R., AND STRACHEY, C. *A Theory of Programming Language Semantics*. Halsted Press, 1977.

[10] OSTRAND, T. J., AND WEYUKER, E. J. The Distribution of Faults in a Large Industrial Software System. *ACM SIGSOFT Softw. Eng. Notes* (2002).

[11] PATTERSON, D. A. A Simple Way to Estimate the Cost of Downtime. In *Proc. of the 16th USENIX Systems Admin. Conf.* (2002).

[12] POULSEN, K. Software Bug Contributed to Blackout. *Security Focus* (2004).

[13] SUNDARARAMAN, S., SUBRAMANIAN, S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SWIFT, M. M. Membrane: Operating System Support for Restartable File Systems. In *Proc. of the 8th USENIX Conf. on File and Storage Technologies* (2010).

[14] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering Device Drivers. *ACM Trans. Comput. Syst.* (2006).

[15] TANENBAUM, A. S., AND WOODHULL, A. S. *Operating Systems Design and Implementation*, 3 ed. Prentice Hall, 2006.

[16] VLECK, T. V. Unix and Multics, 1995.

[17] YOUNAN, Y., PHILIPPAERTS, P., CAVALLARO, L., SEKAR, R., PIESSENS, F., AND JOOSEN, W. PAriCheck: an Efficient Pointer Arithmetic Checker for C Programs. In *Proc. of the 5th ACM Symp. on Information, Computer and Communications Security* (2010).

[18] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. SafeDrive: Safe and Recoverable Extensions using Language-based Techniques. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Impl.* (2006).