Summary

Many real-world systems need to run 24/7 for years and never go down. Consider, for example, industrial control systems and e-banking servers. Unfortunately, software updates—a necessary evil to cope with the fast-paced evolution of modern software—are on a collision course with our growing need for nonstop operation, with traditional update practices resorting to a full system restart to deploy new versions or even small security patches. The update-without-downtime problem has recently fostered much research on live update—the ability to update running software on the fly. Prior live update solutions, however, offer no practical and general way to update operating systems (OSes) or long-running applications, requiring extensive manual effort for nontrivial updates and offering poor system support to detect and recover from update-time errors. These factors have significantly lowered the usability and dependability guarantees of prior techniques compared to those of regular software updates, ultimately discouraging widespread adoption of live update.

This dissertation introduces new techniques to implement safe and automatic live update for the entire software stack, with support for several possible classes of updates—ranging from security patches to complex version updates—and generic systems software written in C. At the heart of our live update mechanisms lies a novel process-level update strategy, which starts the new version as a new independent process and automatically transfers the state of the old process to the new one. When an update becomes available, our framework allows the new version to start up and connect to the old version to request all the information from the old execution state it needs (e.g., data structures, even if they have changed between versions). When all the necessary data have been transferred over and the old state completely remapped into the new version, our framework terminates the old version and allows the new version to atomically resume execution in a way completely transparent to the users.
Our automated state transfer strategy is empowered by a new instrumentation pass in the LLVM compiler framework that maintains metadata about all the program data structures in memory. This allows our framework to inspect the state of two different versions and seamlessly remap data structures between them, even in face of complex state and memory layout transformations. To ensure a safe update process, our framework can also detect common update-time errors using three different mechanisms: (i) run-time error detection—detecting crashes and other abnormal run-time events using hardware/software exceptions—(ii) invariants-based detection—detecting state corruption from violations of statically extracted program state invariants—(iii) time-traveling state transfer-based detection—detecting memory errors from state differences between distinct process versions known to be equivalent by construction. When an error is detected during the update process, our framework automatically rolls back the update, terminating the new version and allowing the old version to resume execution normally, similar to an aborted atomic transaction. This process-level update prevents update-time errors in the new version from propagating back to the old version, allowing for safe error recovery at update time. This is in stark contrast with prior solutions, which typically patch the running program in place, so if the update fails, there is no fallback to a working version.

At the OS level, we demonstrate the effectiveness of our techniques in PROTEOS, a new research OS designed with live update in mind. In PROTEOS, process-level updates are a first-class abstraction implemented on a multiserver OS architecture based on MINIX 3. PROTEOS combines our live update techniques with a rigorous event-driven programming model adopted in the individual OS components, allowing updates to happen only in predictable and controllable system states. At the application level, we demonstrate the effectiveness of our techniques in Mutable Checkpoint-Restart (MCR), a new live update framework for generic long-running C programs. MCR extends our techniques to allow legacy user programs to support safe and automatic live update with little manual effort.

In conclusion, this dissertation presents evidence that a major paradigm shift is necessary in the design of live update systems. Unlike existing approaches, our live update techniques allow OS components and long-running application programs to be updated without patching running programs in place and potentially endangering their execution should the update fail for any reason. We demonstrate that this new paradigm is amenable to effectively automating and safeguarding the live update process, while reducing the implementation burden to the bare minimum. Our experience with live updating major components of the entire software stack shows important limitations in prior solutions and confirms our hypothesis that safe and automatic live update is a realistic option if careful software design and adequate system support are available. We see our work as the first important step towards truly practical, general, and trustworthy live update systems for the real world.