Research Statement *Junfeng Yang*

My goal is to make software systems reliable and secure. Today's software systems are large, complex, and plagued with errors, some of which have caused critical system failures. My research has been focused on creating effective tools to improve the reliability and security of real software systems.

My approach combines fundamental systems insights and formal program analysis techniques. Both are crucial: formal techniques provide strength over ad hoc approaches, giving my research an unusual depth; constructing the tools and applying them to real systems distills the essence of the formal techniques, measures the actual effectiveness of the tools, produces practical impact, and exposes new research problems and directions that matter most. This rare combination has resulted in advances to software systems research, a best paper award, and improved reliability of widely deployed open-source and commercial software systems.

An example of this approach is my work on thoroughly checking storage systems. These systems, including file systems and databases, are often trusted with the only copy of data, so their unrecoverable data-loss errors are among the worst errors. Unfortunately, building correct storage systems is incredibly difficult because developers must anticipate all possible failure scenarios and correctly recover from all of them. Yet, there is a lack of effective tools to help developers detect storage system errors.

To address this challenge, we created EXPLODE, a lightweight *model checker* for finding serious errors in real storage systems. EXPLODE employs a novel *in situ* architecture that interlaces the mechanism required for model checking within the checked system, enabling users to check *live* systems and drastically reduce the manual effort to model-check a system from months to minutes. This architecture made model checking so easy that we applied EXPLODE to many popular storage systems, and found serious errors in *every* system. My research along this line won an OSDI best paper award, spurred a body of new work on storage system reliability and another on model checking real systems, led to numerous patches to the Linux kernel, and was praised as "a very valuable service" by file system developers.

Driven by the cloud computing trend, many storage systems (and other software systems) are now distributed. we thus created MODIST, a model checker to extend the in situ architecture to distributed systems. To mitigate the *state-space explosion* problem, a key challenge facing model checking, my collaborators and I invented a *sound* algorithm to avoid checking redundant states, speeding up MODIST by up to 10^5 times. MODIST is also a practical success: it found serious *protocol* errors in real systems such as Berkeley DB replication and Microsoft SQL Azure, and is now being transferred to Microsoft product groups.

The massive number of services powered by cloud computing and the rise of multicore hardware have caused multithreaded programs to become increasingly pervasive and critical. Yet, these programs are extremely difficult to get right; a key reason is that different runs of a multithreaded program may show different behaviors, depending on how the threads interleave. This *nondeterminism* makes it difficult to write, test, debug, and verify multithreaded programs, resulting in many difficult-to-debug "heisenbugs" in widespread multithreaded programs.

To eliminate heisenbugs and problems caused by nondeterminism, we created TERN, a compiler and runtime system to make threads deterministic. It works by memoizing past thread interleavings, or *schedules*, and reusing them on future inputs if possible. By reusing schedules, we improve understandability, predictability, and repeatability (even across different inputs) of multithreaded programs. We subsequently created PEREGRINE to *efficiently* make threads deterministic even if there are data races. The insight is that although most multithreaded programs have races, these races occur rarely, so we only have to enforce expensive memory access orders for the "racy" portions of an execution. PEREGRINE also largely removed the manual annotations required by TERN, using novel program analysis algorithms. Our results show that PEREGRINE is efficient and deterministic, can frequently reuse schedules, and is easy to use.

FUTURE PLAN. TERN and PEREGRINE are just two examples in this fertile research direction of deterministic execution and reliable parallel programs; the bulk of work still lies ahead. In the shorter term, I plan to extend the ideas and approaches of reusing schedules to increase confidence in testing, optimize deterministic replay and replication, and tolerate *unknown* concurrency errors (*e.g.*, by running diversified schedules at program replicas).

In the intermediate term, I plan to work on optimizing and verifying parallel programs. One idea is an efficient operating system scheduler that optimizes thread scheduling and placement based on memoized schedules, as these schedules can effectively "predict" the future. Another promising idea is to *verify* a parallel program according only to a small set of schedules, then enforce these schedules at runtime. By focusing only on a small set of schedules, not all possible ones, we improve precision and simplify verification; by enforcing these schedules at runtime, we guarantee soundness of the verification.

In the longer term, I will continue creating systems and algorithms to make reliable software systems. If successful, this research will result in drastically improved software reliability and security, benefiting every computer user.