Research Statement

Daniel Fryer

My research interests encompass systems software, including the operating system, all layers of the storage stack, and application middleware. I aim to increase the dependability, security and efficiency of the software that forms the foundation of most of today's applications. My approach is to transform real-world software by introducing practical formalisms into the development process, making system software more amenable to verification, maintenance and optimization.

Most of my work so far has been focused on protecting data from corruption due to software bugs. Another project I am involved in aims to define a suitable language for annotating persistent data structures, making them more amenable to inspection and manipulation by external tools. The projects I detail below all present opportunities for future work, touching on systems software, software engineering and programming language design.

Research Contributions

The security and integrity of our applications depends on the security and integrity of stored data. When persistent state becomes corrupted, whether by a bug, hardware failure, or an attack, recovery is time-consuming and expensive. Most computer users have experienced the loss of all or part of an important document due to buggy software! Corruption can occurs silently, so that even backups and snapshots may contain corrupt data, which is not detected until someone attempts to access that data. My thesis aims to prevent corruption on the *write path*, before corruption is visible to other threads or future operations, creating a firewall around trustworthy state.

Run-time Consistency Checking

To address the threat of silent corruption, I built the *Recon* system to protect file system metadata from corruption due to bugs in the file system or operating system. File system metadata is important to protect, because a small corruption of metadata can cause widespread data loss or exposure.

Recon is novel in three ways. First, Recon operates on unmodified file systems. Recon's independence from the file system code allows us to guarantee that I/O cannot bypass Recon (for instance, Recon can run in a hypervisor.) Rather than relying on hooks in file system code, it uses knowledge of the file system's format to perform introspection on the I/O being performed by the file system. Second, this checking is done at run time, instead of while the disk is offline, as in tools like fsck and CHKDISK. Because Recon interposes between the file system software and the disk, Recon prevents corrupt writes from reaching the disk. In order to do this efficiently, the global consistency properties checked offline by fsck must be transformed into local invariants, which apply to changes about to be made. These local invariants take advantage of the assumption that the prior disk state was consistent, as well as precise knowledge of what was changed by the write. This transformation is the third novel aspect of Recon, demonstrating that online checking can be significantly more efficient than offline checking.

Evaluating a system designed to protect against rare or unknown bugs requires measuring its ability to detect the symptoms of bugs, regardless of cause. We took a two-pronged approach when evaluating Recon's corruption-detection abilities: fault injection, to ensure that it caught all injected corruptions which violated the consistency of the file system, and a review of known data-corrupting bugs in the past, showing that Recon would have been effective against those bugs.

We published our evaluation of Recon at USENIX FAST 2012 [3], where it was awarded best paper. The work on Recon developed into two parallel projects, one continuing to explore run-time monitoring, and the other exploring ways to specify data structures, their relationships, and their invariants.

Atomicity and Durability Invariants

Many real-world bugs which cause corrupt storage systems result from mistakes in the implementation or use of *transaction mechanisms* [4], which are used to ensure consistency after a crash. To address this, I identified the atomicity and durability invariants that govern the correct functioning of journaling and shadow paging systems. These invariants concern not only the contents of the I/O requests, but also their ordering with respect to each other and with respect to write barriers and flush requests, which force the underlying block storage to make an asynchronous write durable.

These invariants were integrated into the Recon run-time checking framework, protecting the atomicity, consistency and durability of the Ext4 and Btrfs file systems. In addition to the invariants, we identified ways in which file system design impacts run-time verification and offered suggestions for future storage system designs. This work was published at FAST 2014 [1], and was also published in extended form in the ACM Transactions on Storage [2].

Runtime Integrity Checking for Applications using NVRAM

Like file systems, applications also have integrity constraints. Some of these are enforced by language runtimes (e.g. type safety) or middleware (e.g., database integrity constraints), while others are only enforced haphazardly. Inconsistent in-memory data structures are especially problematic in light of the recent development of *persistent memory.* These memory devices are persistent (like Flash) but byte-addressable (like DRAM), and programmed with ordinary load/store instructions. Without the "gateway" of I/O operations separating persistent and volatile data, it is more likely that persistent data structures will become corrupted by bugs.

I have been developing *Ingot* to protect application data structures in NVRAM. Ingot presents a logical framework for transforming global invariants into local invariants. At run time, Ingot uses a combination of an *undo log*, together with metadata attached to persistent memory allocation, in order to determine which invariants need to be checked. To evaluate Ingot, I have implemented invariant checking in Redis, a popular key-value server, and am preparing the results for publication.

Declarative Specifications

Many of the details of a storage system's data structures are not found in header files defining structures or classes, but in the code that manipulates these structures. This implicit, *imperative* specification obscures details necessary for interpreting the data. Various languages exist for declaratively specifying data structures and generating code which serializes or deserializes data, but they have not been widely adopted within storage systems because they lack generality and the ability to describe already-existing data structures. The goal of the *Spiffy* project is to fully specify these data structures in a *declarative* manner. Spiffy is designed to accommodate existing storage systems, because it takes a long time before a new system is considered trustworthy.

One application of these specifications is to generate code which accesses file system metadata. For example, a system which preferentially caches metadata must distinguish metadata from data. We can implement this caching layer using the combination of a template, which implements the caching logic, and code generated from annotated file systems. The benefits of this arrangement are that each file system only has to be annotated once, and each application template only has to be written once, instead of writing low-level storage applications from the ground up. Other possible applications include storage management, VM introspection, and forensic tools. I collaborate with the first author of the *Spiffy* project, and this work is currently in submission. Earlier work on this project was published as a short paper in RV 2014 [5].

Future Directions

NVRAM and Scalability New nonvolatile memories promise to significantly accelerate storage, coming within an order of magnitude of DRAM. This will expose scalability bottlenecks in systems designed with the assumption that I/O is slow, including databases and file systems. One of my ongoing projects is to identify and remove these bottlenecks in anticipation of high performance first-tier storage. I have done this with an existing research file system designed for NVRAM, and am investigating how to apply lessons from this to open-source storage systems currently in use, including file systems, conventional databases, and NoSQL stores.

Rack-Scale Storage Another trend related to fast first-tier storage is the idea of *disaggregated storage* in datacenter design. Traditional storage has latencies that dominate network latency, making it practical to deploy centralized storage servers, connected using Ethernet or Fibre Channel. As storage latencies drop, it becomes more and more important to co-locate storage and compute in order to support big data applications. At the same time, partitioning this storage between individual compute nodes is inefficient with respect to shifting workloads and hotspots. Disaggregated storage answers this by decoupling the storage hardware from the compute nodes using a flexible fabric designed for low-latency communications within a compute rack. While at Microsoft Research I investigated some options for this architecture in an ongoing project.

Monitoring distributed storage systems New storage systems are being built for distributed environments (e.g. cloud services). Because there are so many moving parts in a distributed environment it is especially important to program defensively. Enforcing invariants on distributed systems is an extensive area of future work, enabling us to detect violations of these properties before a critical failure causes data loss or unavailability. Narrowing the scope of acceptable behaviour complements other security measures such as intrusion or anomaly detection.

Limits of efficient on-line integrity checking Another avenue of future work is finding limits on what invariants can be enforced efficiently at run time. For instance, enforcing the integrity of reference counts is relatively easy in the Ingot system, whereas enforcing the integrity of garbage collection is infeasible. I would like to collaborate with researchers who are more familiar with logic and formal methods in order to develop a cohesive way of explaining what properties can be efficiently verified while a data structure is being updated.

Integrating declarative and imperative code Much like integrity constraints in databases, applications could benefit from more convenient integration of invariants and constraints with the code they use to manipulate data. The languages commonly used in application programming may have some of the features we need to express an invariant (e.g. types, variables, operators) while lacking others (e.g. quantifiers). On the other hand, a language like Datalog may have all of the logical features we need, but requires us to explicitly load facts about the application state into its environment. I intend to investigate how we can compile assertions written in a high-level logic language into the application code without this overhead.

References

[1] Daniel Fryer, Dia Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the integrity of transactional mechanisms. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 295–308, Berkeley, CA, USA, 2014. USENIX Association.

[2] Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the integrity of transactional mechanisms. *Trans. Storage*, 10(4):17:1–17:23, October 2014.

[3] Daniel Fryer, Jack Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, February 2012.

[4] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, February 2013.

[5] Kuei Sun, Daniel Fryer, Dai Qin, Angela Demke Brown, and Ashvin Goel. Robust consistency checking for modern filesystems. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 85–91, 2014.