Beyond Storage APIs: Provable Semantics for Storage Stacks

Ramnatthan Alagappan Vijay Chidambaram Thanumalayan Sankaranarayana Pillai
Aws Albarghouthi Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison

Abstract

Applications are deployed upon deep, diverse storage stacks that are constructed on-demand. Although many storage stacks share a common API to allow portability, application behavior differs in subtle ways depending upon unspecified properties of the underlying storage stack. Currently, there is no way to test whether an application will behave correctly on a given storage stack: corruption or data loss could occur at any point in the application lifetime.

We argue that we require an expressive language for specifying the complex storage guarantees required by different applications. The same language can be used to write a high-level specification capturing the design of different storage-stack layers. Given the required guarantees, and the storage-stack specifications, we can prove that stacks constructed dynamically (by composing different storage-stack layers) provide the guarantees required by the application.

1 Introduction

Modern applications are deployed in a variety of environments [2, 13, 15, 24, 31]: on laptops, on mobile phones, on tablets, and on private and public clouds. Each environment involves a different storage stack: for example, the laptop might use the `btrfs` file system on top of a SATA drive [10], while the mobile phone might use `F2FS` on top of an SSD [11, 26]. With the advent of software-defined storage, we believe applications will soon be able to request and automatically obtain customized storage stacks [20, 25, 32]. Amazon EC2 already does this at a coarse level by allowing users to specify the storage they require for each virtual machine [1]. The day is not far off when storage stacks will be constructed on the fly, mixing and matching different layers like block re-mappers, logical volume managers, and file systems [39, 40].

Users would like their applications to run on different environments without modification [2]. In addition to reducing development effort (and bugs), application portability avoids *vendor lock-in* [7,14], where an application is tied to a particular stack because it uses features unique to that stack. To achieve application portability, many vendors strive to provide API compatibility with popular vendors like AWS [9, 12]. For example, OpenStack uses the same API as Amazon's cloud services to allow users like Intel, Yahoo, and Walmart to easily port their applications from services like Amazon EC2 to OpenStack Nova [9].

Unfortunately, while a common API guarantees that applications will execute on different stacks, it does not guarantee that they will do so correctly. Most API specifications simply describe what operations are offered by the storage stack. The specifications do not describe the semantics offered by the system: for example, whether two operations are persisted in order or whether an operation is persisted atomically. Yet, recent work has shown that application correctness hinges on storage-stack semantics [34, 35, 43]. For example, LevelDB [22] required the rename of a file to be persisted before the unlink of another file. If the storage stack does not order these operations, it results in data corruption [8, 34].

Testing whether an application will behave correctly on a storage stack is challenging for two reasons. First, the guarantees that an application requires from storage are not well-specified; if the developer has only been testing on one platform, they may not even realize that their application depends on certain features of the platform. Our work on application crash vulnerabilities suggests that the required guarantees are complex, and cannot be expressed in simple binary checks or numeric limits [34]. Second, modern storage stacks are composed of many layers (*e.g.*, the Windows IO stack has 18 stackable layers [38]). Each layer builds upon the guarantees given by lower layers to provide guarantees to higher layers. To identify the guarantees given by a dynamically composed stack, we have to examine the guarantees given by each layer in the stack.

We tackle each challenge by borrowing techniques from the programming languages community. First, we propose to specify complex storage guarantees in a formal language (such as Isar [42]). We suggest that the same language could be used to specify the high-level design of each layer of the storage stack. Second, proof assistants (such as Isabelle [33]) can be used to prove that the stack provides the guarantees required by the application. Just as a statement could be proved given a collections of axioms and theorems, we propose that guarantees required by applications could be proved given the guarantees offered by each storage-stack layer.

We believe such verification will be essential for software-defined storage in clouds and datacenters. When storage stacks are constructed on the fly, the correspond-

#	Setup	Consequence	Root Cause
1	Android 4.4 Micro SD Card	All applications using the Micro SD card for implementing virtual IPC to micro controller affected	`O_DIRECT` flag for direct I/O operations on the removable storage not implemented in Android 4.4. It was available in previous versions.
2	Android 4.2.x FAT32 SD Card	Errors not allowing sqlite to create temporary files	sdcard daemon emulating FAT32 FS cannot do fstat of an open but unlinked file. Also, there is no support for ext family of file systems on SD Cards.
3	Android 2.x 4.x Internal SD Card	Cannot open more than 1024 files at a time across Apps	Device internal SD cards are limited to only 1024 open file descriptors. Occurs only with internal SD card on few device types.
4	Google AppEngine Cloud Storage Dev Server	Local deployment of applications that use long filenames fail	Occurs only on some filesystems including btrfs and NTFS because of the limit in path length. Not yet reproduced on Google Cloud Storage deployments.
5	MySql over NFS/FreeBSD	Frequent restarts of MySql	Occurs only when NFS server is running on FreeBSD and the client also runs FreeBSD. `fsync` calls on FreeBSD return `ENOLCK` even when the flush was successful. MySql considers this as a fatal error.
6	Git on DropBox directory	Inconsistent commit histories	The order in which updates are synchronized to DropBox is different from local file-system updates. Specifically, commit metadata is synchronized before the user data.
7	Sqlite on DropBox directory	Corrupt Sqlite database	The order in which updates are synchronized to DropBox is different from local file-system updates. Specifically, the log file is unlinked before synchronizing the database file completely.
8	Git repository backed up by rsync	Corrupt and unusable repository	rsync synchronizes the git index and the added source files before the corresponding object files.

Table 1: **Portability Bugs**. *The tables lists bugs that occur when the storage stack doesn't provide features or guarantees required by the application. For each bug, the table shows the environment, the consequence, and the underlying root cause. The root causes vary from easily detectable issues like `O_DIRECT` not being supported, to deeper issues like ordering guarantees not provided by the stack.*

ing high-level specifications for different layers can be retrieved, and the guarantees of the resulting stack can then be compared with application requirements. Such checking can be used to construct the optimal storage stack (in terms of resource utilization or other metrics) that will satisfy the given application requirements.

In the rest of the paper, we first present a small study of bugs that occur when applications are ported to different storage stacks (§2). We then describe in detail the challenges in verifying that a storage stack provides required application guarantees (§3). We describe our experience in specifying the design of a simple two-layer storage stack in Isar, and using Isabelle to prove that the `put` operation in a simple key-value store is atomic (§4). Finally, we describe remaining challenges in realizing this vision (§5), discuss related work (§6), and conclude (§7).

2 Portability Bugs

We now present a small selection of *portability bugs*: bugs that occur when applications are ported to different storage stacks. To find these bugs, we searched the public bug databases of projects deployed on different storage stacks. For example, Android is installed on a variety of mobile phones and tablets; thus, we expected to find portability bugs in Android. We also investigated applications run on cloud platforms like Google AppEngine, and in distributed settings like NFS. In addition to examining public bugs, we also performed experiments that revealed bugs on widely-used environments that loosely coupled local storage with cloud or remote storage.

Table 1 lists eight application bugs that are caused by the storage stack failing to provide a guarantee required by the application. For each bug, we specify the setup in which it occurs, its consequence, and the root cause.

The first five bugs are listed in the public bug databases on Android, Google App Engine, and MySQL. They result from the stack not supporting certain operations (#1, #2), placing unexpected limits on resources (#3, #4), or returning unexpected error codes (#5).

The last three bugs were revealed in our experiments. Our work on application crash vulnerabilities [34, 35] demonstrated that applications such as Git [30] and SQLite [36] require ordering guarantees from the storage stack. For example, SQLite requires that its journal writes are persisted before checkpoint writes. SQLite orders these writes using `fsync()`. On a local file-system, this works perfectly: the writes are persisted in the required order on local storage. If the local storage is then synced to a remote location or cloud storage (as is widely done with Dropbox [6]), the order in which the files are uploaded matters: a network outage could result in the remote location seeing inconsistent application state. Dropbox transfers files roughly based on file size [4, 5], while rsync seemingly transfers files sorted by name [3]. Based on this, we conducted the following experiment: run the application inside a folder synced using Dropbox or rsync; perform an application operation such as inserting into SQLite database; emulate network outage while Dropbox or rsync is in the middle of the sync process; inspect application state in remote location. All three experiments

resulted in application inconsistency and/or data corruption. Similar bugs would be caused for any application that requires ordering from the stack [34].

3 Avoiding Portability Bugs

To avoid portability bugs, we need to verify that the stack provides the storage guarantees required by the application. The first challenge is in identifying the required storage guarantees. The developers may be unaware that they are depending upon guarantees from the storage stack: hundreds of Linux applications (written for ext3) depended on data written to a file being persisted before its rename, and lost data when ext4 no longer provided this guarantee [18]. Tools like ALICE [34] can help the developer identify storage guarantees required by the application. Assuming that the application developer is willing to identify and describe the storage guarantees required, there remain two challenges: specifying the guarantees, and checking that it is provided by the stack.

Specifying Storage Guarantees. For some portability bugs, specifying the required guarantee is simple: for example, one could have a binary check for `O_DIRECT` support. Specifying a minimum limit on resources like path names or file descriptors is also straightforward. In other cases, the storage guarantees required are complex. For example, Mercurial and LevelDB require that a file append (*e.g.* "XYZ") results in a prefix of the append data being persisted (*e.g.,* "X" or "XY") [34]. Many guarantees are of the form *"if not A then B else C should hold"*, and cannot be easily expressed in the form of binary checks or numerical limits. Specifying storage guarantees thus requires a rich, expressive language.

Computing and Verifying Stack Guarantees. Modern storage stacks are comprised of several layers [38]. The storage media at the bottom provides some basic guarantees. The layer above the media builds on these guarantees to provide more guarantees to the layer immediately above; thus, guarantees are built up over the stack and the top-most layer (*e.g.,* a database) provides guarantees to the application (*e.g.,* atomic transactions). Software-defined storage can dynamically add or remove stack layers like block re-mappers; calculating how the addition or removal of a layer from the stack affects storage guarantees requires understanding how that layer works (at least at a high level) along with the layers above and below. Therefore, we cannot statically assign guarantees to layers – it must be dynamically calculated.

Thus, verifying correct application behavior on a storage stack is a hard problem, requiring more sophisticated techniques than simple API compatibility tests.

4 Verifying Storage Guarantees

We now describe how we envision a system that can prove that a storage stack has certain guarantees. We start by specifying the guarantees required by the application at the topmost layer of the storage stack, the working logic of each layer, and the guarantees provided by the lowermost layer. Using the guarantees of the lowermost layer and the specification of each higher layer, we can progressively calculate the guarantee provided at each layer. Finally, we can verify whether the guarantees at the topmost layer satisfy the application's requirements. Figure 1 provides a high-level overview of the system.

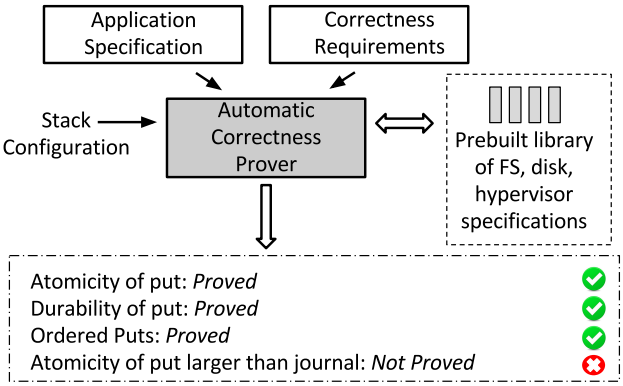

Figure 1: **System Architecture**. *The figure shows a typical workflow – the application specification, correctness requirements, and storage stack configuration are obtained as input, appropriate specifications from the prebuilt library are selected, and finally, the required guarantees are proved/disproved.*

Specifying guarantees at the topmost layer and the working logic of all layers is not straightforward. Subsection 4.1 explains how we use the Isar formal proof language [42] for solving this, and why we think this can be performed with reasonable effort in the future. Subsection 4.2 explains how we use the Isabelle proof assistant [33] for computing and verifying the guarantees for each layer. Subsection 4.3 describes our experience in using Isar and Isabelle to prove that a simple key-value store operating on raw block storage provides atomic `put()` operations. We have made the Isabelle scripts described in this section available online [37].

4.1 Specifying Storage-Stack Layers

The working logic specification of each layer essentially describes how an operation exposed by that layer works. For example, a file-system layer will expose how it performs operations like `rename` or `unlink` and a block layer will expose how it reorders write requests. Isar allows expressing specifications in terms of functions, and these specifications can then be used by proof-assistant tools. We have found the Isar language to be sufficient for specifying the guarantees and the working logic of storage stacks we have investigated so far.

While writing specifications is made possible by Isar, the specifications can be complex; we found specifying the working logic of non-trivial storage layers to be a difficult process. However, in the future, we envision a library

of specifications of widely used filesystems, hypervisors, and storage devices: users can simply plug-in the specifications, instead of writing them anew.

Users can provide a specification of the application, and then provide correctness requirements in terms of application-level operations (*e.g.,* transactions must be atomic). Given the application specification, Isabelle can translate correctness requirements into storage guarantees. Alternatively, the required guarantees can be automatically obtained by using tools like ALICE [34].

4.2 Computing Storage Guarantees

Proof assistants like Isabelle can be used to generate proofs for statements in the context of axioms and theorems. We treat the storage guarantees required by the application as statements to be proved; the storage guarantees of the stack provide the axioms using which the statement should be proved. For multi-layers stacks, the guarantees given by each layer function as axioms for the layer above; the guarantees given by the storage media form the axioms for the lower-most layer.

When Isabelle is unable to prove the given statement, its response (which sometimes includes a counter-example) can be parsed to identify which layer in the storage stack is preventing the high-level property from being provided. Currently, we write machine-checked proofs by hand in Isabelle; in section 5, we discuss opportunities and challenges in constructing proofs in an automatic way.

4.3 Example: Key-Value Store

Consider a key-value store that runs directly on top of block storage. The block storage provides 4K-block atomic writes and *in-order* block operations. We want to verify that the key-value store provides atomic `put()` operations in the presence of system crashes.

We first write a specification for the key-value store operations. For example, the specification describes how the `put()` operation is performed. The key-value store in our example writes small key-value pairs to a single block directly, but uses journaling for large key-value pairs. A dedicated portion of the disk is used as a journal, the update is written initially to this area, and then checkpointed to the actual disk location. If a crash happens, the key-value store tries to recover from the journal.

We then specify the guarantees provided by the block storage: 4K-block writes are atomic and ordered. This forms the set of axioms on top of which Isabelle builds proofs. Finally, we specify the application requirements using Isar: `put()` must be performed atomically, even in the case of a crash. This forms the correctness requirements that we will try to prove using Isabelle.

We break the proof into two parts: one for small key-value pairs that fit into a single block and the other for large key-value pairs that do not fit into a single block. For the first part, we prove that if the key-value pair is smaller than a block, it will be written atomically; Listing 1 shows the corresponding logical statement expressed in Isabelle. For the reader, this statement might be intuitive, since the underlying block layer provides atomic 4K-block writes. Nonetheless, proving this seemingly simple statement requires lot of effort.

```
theorem atomic_only_for_one_block:
assumes A0: "disk.length > index"
assumes A1: "disk != NULL"
shows
"key.length + value.length  <= block_size
==>
isatomicupdate disk (kv_put disk key value
    index)"
```

Listing 1: **Atomicity theorem for update that spans one block.** *The listing shows the pseudocode theorem for atomicity of put() operation.* `isatomicupdate` *is a function that checks if two disks differ only by one block. Note that this listing is pseudocode and not exact Isabelle/Isar syntax.*

For the second part of the proof, we proved that an update done using the journaling technique is always atomic with one exception: the update being larger than the journal. The atomicity guarantee of a `put()` operation that updates two blocks can be logically expressed as follows: *(final[i]=initial[i]* $\wedge$ *final[j]=initial[j])* $\vee$ *(final[i]=key* $\wedge$ *final[j]=value)*, where *initial* is the state of the disk before the `put()` started, *final* is any disk state resulting from a crash while performing the `put()`, and *i* and *j* are the indexes updated. The above logical statement should hold for all possible *final* disk states. We also proved that the update is durable if the checkpoint is complete. We discovered that the proof effort required significantly varies depending on how the requirements are specified. As we gain experience with Isabelle proofs, we will be able to prove guarantees with considerably lesser effort.

This simple example illustrates a few key features about our system. First, given the specification of different stack layers, our system can find dependencies *across* layers, thus finding corner cases that may not be obvious to a human. Second, the system can be *incrementally* extended: the specification of each layer does not need to know how the layer below works. A new file system requires only one new layer. We believe this supports our vision of building a library of high-level specifications that can be used to verify guarantees of different storage stacks.

5 Challenges

We discuss the challenges that remain in realizing our vision of achieving application portability across diverse storage stacks.

Obtaining Specifications. We assume that the developers of the storage-stack layers will provide specifications along with the source code. Figuring out the specifica-

tions without developer support will be hard; we take heart from the fact there are only a few choices available for each layer. For example, most stacks use one among a few popular file systems. Thus, a modest amount of manual work can create a library that can reused by a large number of storage stacks.

Interplay Between Layers. When a layer uses functionality from a lower layer, Isabelle can use the guarantees given by the lower layer to prove guarantees at the upper layer. However, this is currently done by directly using lower-level functions in the specification of the upper layer. Modifying this so that the upper layer uses a generic intermediate layer (like VFS), with parameters that select different lower layers, remains a significant challenge.

Automatically Proving Guarantees. Given the storage-stack specifications and the application's correctness requirements, we can use Isabelle to automatically figure out what guarantees need to hold across layers for the requirement to be proved. Isabelle still requires the user to specify different strategies (*e.g.,* induction) to try and prove the goal. For the proofs we have developed, Isabelle required (non-trivial) guidance in terms of what strategies to employ. In the future, we would like to model specifications and requirements in a fragment of first-order logic, and use SMT solvers (automated theorem provers) such as Z3 [19] to automatically prove guarantees. Unlike Isabelle, using Z3 limits what we could prove automatically – we plan to carefully investigate this. The advantage of using Z3 is that it allows automatic verification without user involvement for dynamically changing stacks.

Proofs Without Layer Specifications. We currently assume that all layer specifications are publicly available. However, companies may be hesitant to reveal their layer specification publicly. Thus, we would require a way to prove that a stack provides certain guarantees, without having the specifications of the layers that comprise the stack. Akin to zero-knowledge proofs [21], we would need each layer to verify that they provide certain guarantees, while hiding the details of how exactly they provide each guarantee. Each layer only needs to know the guarantees associated with the operations exported by a lower layer, and not how the operations are implemented. Hence, we would still be able to prove the guarantees provided by the stack overall. Achieving this would allow application correctness to be verified on stacks composed of layers developed by many different companies.

6 Related Work

Recent work has applied verification techniques to build verified operating system kernels [29], end-to-end security [23] and in-kernel interpreters [41]. Various efforts in the past have modelled and verified file-system implementations specifically. Keller et al. observe that modern file systems are modular; this facilitates specifying each module (or on-disk data structure) formally and then generating code and correctness proofs [28]. Arkoudas et al. formally specify a simple abstract file system and prove the implementation correct by establishing a simulation relation between the abstract specification and the implementation [16]. Similarly, Kang et al. show how a flash-based file system can be formally verified [27]. While such work focuses on verifying the file system in isolation, we aim to verify end-to-end application correctness on different storage stacks (of which file systems are only one component). Our methodology matches closely with a recent work to verify network-wide invariants of SDN applications by Ball et al. [17].

7 Conclusion

To truly realize the potential of software-defined storage, infrastructure providers must construct customized storage stacks on demand that satisfy customer requirements while optimizing metrics such as utilization. A key part of satisfying the customer is verifying that the application will execute correctly on the constructed stack. In this paper, we have shown why this is challenging: applications depend on subtle guarantees of the underlying storage stack, that are more complex than the simple functionalities defined by typical API documentation. Logically specifying these guarantees, and verifying whether different layers can provide these guarantees, is a complex and unsolved problem. While we have taken the first steps in solving this problem, significant challenges remain. We hope future research tackles these challenges, thus making applications truly portable across diverse storage stacks.

Acknowledgments

We thank the anonymous reviewers and ADSL lab members for their insightful comments and feedback. This material is based upon work supported by the NSF grants CNS-1421033, CNS-1319405, and CNS-1218405 as well as generous donations from Cisco, EMC, Facebook, Google, Huawei, IBM, Microsoft, NetApp, Samsung, Seagate, and VMWare. Vijay Chidambaram is supported by a Microsoft Research PhD Fellowship. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] Amazon Instance Storage. `http://docs.aws.amazon.com/AWSEC-2/latest/UserGuide/InstanceStorage.html`.

[2] Application portability in action: A demonstration of cloud foundry core. http://www.centurylinkcloud.com/blog/post/portability-in-action-a-demonstration-of-cloud-foundry-core.

[3] Debian Bug report logs. rsync: should not reorder the file names on the command line. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=640492.

[4] Dropbox Datastore API. http://www.dropbox.com/developers/blo-g/78/datastore-api-growth.

[5] Dropbox Sync. https://www.dropbox.com/en/help/9.

[6] Git on Dropbox. http://stackoverflow.com/questions/1960799/using-git-and-dropbox-together-effectively.

[7] How to Avoid Cloud Vendor Lock-In. http://www.linuxinsider.com/story/79417.html.

[8] LevelDB - Issue 189: Possible bug: fsync() required after calling rename(). https://code.google.com/p/leveldb/issues/detail?id=189.

[9] OpenStack EC2 Compatibility API. http://docs.openstack.org/admin-guide-cloud/content/instance-mgmt-ec2compat.html.

[10] openSUSE 13.2 To Use Btrfs By Default. http://www.phoronix.com/scan.php?page=news_item&px=MTYzNjA.

[11] Review: In its second generation, the Moto X becomes a true flagship. http://arstechnica.com/gadgets/2014/09/review-in-its-second-generation-the-moto-x-becomes-a-true-flagship/3/.

[12] Swift api feature comparison. https://wiki.openstack.org/wiki/Swift/APIFeatureComparison.

[13] Titanium Mobile Development Environment. http://www.appcelerator.com/titanium/.

[14] Watch out for that cloud lock-in. http://www.zdnet.com/article/watch-out-for-that-cloud-lock-in/.

[15] Xamarin. http://xamarin.com.

[16] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *Formal Methods and Software Engineering*, pages 373–390. Springer, 2004.

[17] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 282–293, New York, NY, USA, 2014. ACM.

[18] Jonathan Corbet. That massive filesystem thread. http://lwn.net/Articles/326471/, March 2009.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[20] EMC. Rethink Storage: Transform the Data Center with EMC ViPR Software-Defined Storage. ttp://www.emc.com/collateral/white-papers/h11749-transform-data-center-with-vipr-software-defined-storage-wp.pdf.

[21] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[22] Google. LevelDB. https://code.google.com/p/leveldb/, 2011.

[23] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[24] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[25] InfoStor. Emerging Trends in Software Defined Storage. http://www.infostor.com/storage-management/virtualization/emerging-trends-in-software-defined-storage-1.html.

[26] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/o stack optimization for smartphones. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 309–320, Berkeley, CA, USA, 2013. USENIX Association.

[27] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in alloy. In *Abstract state machines, B and Z*, pages 294–308. Springer, 2008.

[28] Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too! *ACM SIGOPS Operating Systems Review*, 48(1):58–64, 2014.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[30] Linus Torvalds. Git. http://git-scm.com/, 2005.

[31] James Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 257–273, Seattle, WA, April 2014. USENIX Association.

[32] NetApp. NetApp Software-Defined Storage. http://www.netapp.com/us/technology/software-defined-storage/.

[33] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[34] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[35] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Joo young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-Level Consistency. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep '13)*, Farmington, PA, November 2013.

[36] SQLite. SQLite transactional SQL database engine. `http://www.sqlite.org/`.

[37] The ADvanced Systems Laboratory (ADSL). Verified Storage Stacks. `http://research.cs.wisc.edu/adsl/Software/verifiedstoragestacks/`.

[38] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[39] VMWare. Software-Defined Storage (SDS) and Storage Virtualization. `http://www.vmware.com/software-defined-datacenter/storage`.

[40] VMWare. The VMware Perspective on Software-Defined Storage. `http://www.vmware.com/files/pdf/solutions/VMware-Perspective-on-software-defined-storage-white-paper.pdf`.

[41] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: a trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 33–47. USENIX Association, 2014.

[42] Markus Wenzel. Isar–a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–183. Springer, 1999.

[43] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.