File Systems Deserve Verification Too!

Gabriele Keller[1 2] Toby Murray[1 2] Sidney Amani[1 2] Liam O'Connor[1 2] Zilin Chen[1 2]
Leonid Ryzhyk[1 2 3] Gerwin Klein[1 2] Gernot Heiser[1 2]

[1]NICTA *, Sydney, Australia
[2] University of New South Wales, Australia
[3]University of Toronto, Canada

Abstract

File systems are too important, and current ones are too buggy, to remain unverified. Yet the most successful verification methods for functional correctness remain too expensive for current file system implementations — we need verified correctness but at reasonable cost. This paper presents our vision and ongoing work to achieve this goal for a new high-performance flash file system, called BilbyFs. BilbyFs is carefully designed to be highly modular, so it can be verified against a high-level functional specification one component at a time. This modular implementation is captured in a set of domain specific languages from which we produce the design-level specification, as well as its optimised C implementation. Importantly, we also automatically generate the proof linking these two artefacts. The combination of these features dramatically reduces verification effort. Verified file systems are now within reach for the first time.

1. Introduction

File systems are a critical part of any operating system (OS). For example, the Linux kernel source tree presently contains 49 different file systems (not counting variants and pseudo file systems), plus a few for use in specialised systems (e.g. romfs or promfs). Together these account for a significant fraction of the total code base of the kernel: after drivers (57%) and architecture-specific code (18%), file systems comprise 7% of the source lines of code (SLOC) of the Linux kernel, version 3.10.

The advent of new classes of storage devices, as well as the desire for more (and more specialised) functionality, such as different reliability-performance trade-offs, drives the growth in the number of file systems. New usage scenarios, e.g. resulting from the widespread adoption of virtualisation, are creating pressure to change internal APIs in the storage stack [Jannen et al. 2013], which will require massive re-writing of file-system code.

The ongoing development of file systems is not only costly, it is also a significant hazard to the dependability of the OS. Like much other system code, file-system code is complex, full of tricky corner cases, and is necessarily bloated by lots of boring error handling code. As a result, file systems are a significant source of kernel bugs [Yang et al. 2006, Palix et al. 2011]. Even mature file systems with supposedly stable code bases, such as Ext3, still experience frequent bug fixes [Lu et al. 2013].

The use of static analysis for finding bugs in systems code has become popular in recent years [Ball et al. 2010, Bessey et al. 2010]. While very useful in many cases, this approach provides little promise for file systems code, as the majority of file-system bugs are semantic faults [Lu et al. 2013].

We postulate that file systems are too important to be left at the whim of the traditional design-code-debug-redesign cycle performed on C code — they deserve and need better. We furthermore claim that it is possible to build real-world file system implementations that are provably correct. Specifically, we observe that:

- Real file systems are highly modular.
- The storage stack of contemporary operating systems requires and defines many intermediate abstraction levels.
- For each module, the translation between abstraction levels is logically straightforward, but practically obscured by error handling with tricky corner cases.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.

PLOS '13 November 03-06 2013, Farmington, PA, USA.

http://dx.doi.org/10.1145/2525528.2525530

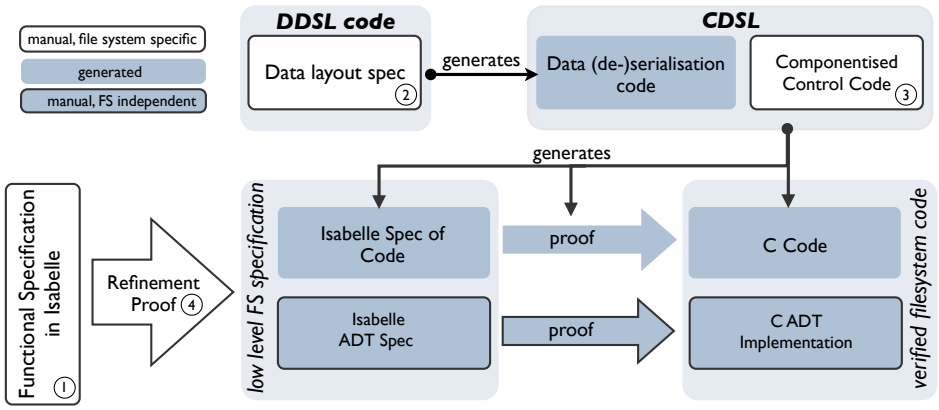

Figure 1. Framework Components

We can summarise the above observation that (given a design provided by experts) the implementation of file systems is, conceptually, not overly complex, yet full of pitfalls. **This is as good a case for automation as there ever was!** Furthermore, we observe that, if coding can be automated, i.e. the code generated, then it should also be possible to prove the correctness of the generation. Compared to the generation of optimised machine code from an arbitrary C source [Leroy 2009], generating file system code is an easier problem. Although the proof will have to work on richer semantic structures, the input is more constrained in its structure and type system.

In short, our claim is:

- Given an appropriate design, the implementation of file systems is logically straightforward.
- The salient algorithms and on-media data structures can be easily expressed in a (collection of) appropriate high-level, domain-specific languages (DSLs).
- It is straightforward to produce C code from such DSLs, without impacting overall performance.
- It is also possible to generate *correctness proofs* with the C code.

We propose a concrete approach for constructively validating these claims. Specifically, we present the prototype of a framework which lets us *specify the high-level control logic as well as the on-medium data structures* of a file system in a modular fashion. From this, the framework generates the modules' C implementation, including all error-handling and serialisation/de-serialisation code. The framework *also generates formal specifications of module functionality as well as proofs of the correctness of the generated implementation.*

This provides the inputs to a verification of the overall file-system functionality by a *manual refinement proof* from a high-level specification to the module specifications. This step is important, because it is the connection to the top-level specification that makes sure the generated lower-level specifications are not merely trivial or defective.

While formal verification has been shown to be cost-competitive with traditionally-engineered high-assurance code [Klein et al. 2009], we expect dramatically increased verification productivity (in terms of cost per verified line of C), achieving *verified file systems whose life-cycle cost is significantly less than that of traditionally-engineered ones.*

We have so far completed the design for the first versions of two domain-specific languages, one for the high-level control logic and the other for specifying the on-medium data structures. To convince ourselves that the languages' expressiveness is sufficient, we have used these DSLs to specify an initial flash file system called BilbyFs. We have implemented the front-ends of the code and proof generators, as well as C code generation from the control DSL, and have formal semantics for both languages, some mechanised in Isabelle/HOL.

In Section 2 we give an overview of our framework, followed by a discussion of our approach to a modular design in Section 3. Section 4 introduces two domain-specific languages which play a central role in the approach. We discuss the current status of the project in Section 5 in more detail.

2. Overview

File systems code can generally be classified as one of the following: (A) code for serialising and de-serialising between in-memory data structures and flat on-medium representations; (B) code to create and maintain the in-memory data structures to store the file system information; and (C) code implementing the actual file-system logic (short *control code*). A large fraction of this control code deals with errors; [Saha et al. 2011] showed that file systems have among the highest density of error-handling code in Linux.

Code of different categories differs significantly in nature and structure, and we treat them separately, resulting in simpler, specialised tools. Specifically we are using two different, domain-specific languages for class (A) and (C). Class

(B) represents commonly used data structures, such as lists, finite maps and buffers, most of which are not specific to a particular file system and can be verified independently. Even if some of these data structures are non-standard and specific to the file system, their verification will still be orthogonal to the verification of control code and data layout.

Figure 1 shows the main components of our framework. Boxes represent code or specifications while block arrows represent the proofs which connect them. Framed boxes indicate components that the file system implementor must provide. Of these, white boxes are specific to the file system being built, and shaded boxes are reusable between different file systems. Shaded boxes without frame represent components which are automatically generated and slim arrows show the generation of artefacts (specifications, proofs and code).

As we can see in the diagram, the file system implementor must provide four distinct components: ① A high level specification of file system functionality, ② a specification of the in-program and on-medium data layout in our data-description language, DDSL, ③ the implementation of the control code in our control-description language, CDSL, and ④ the proof that the generated Isabelle low level specification corresponds to the functional specification.

Note that steps ② and ③ alone guarantee the absence of many programming errors common in file systems. However, step ④ is what guarantees the functional correctness of the resulting file system.

The framework uses the DDSL description to generate (1) a high-level Isabelle specification of the data structures and the serialisation and de-serialisation functions, (2) the corresponding CDSL code and (3) the proof showing that the latter is a valid implementation of the former. Both the generated and hand-written CDSL code combined are then used to generate (1) a low-level Isabelle specification of the file-system control code including the DDSL-generated (de-)serialisation code, (2) the C code implementing the specification and (3) the correctness proof for this implementation with respect to the low-level specification.

For now, we assume that we have suitable implementations and correctness proofs for the abstract data structures used in the file system, such as lists, arrays, finite maps and buffers, with operations to search, alter, and iterate over these structures without using arbitrary loops or recursion in the client. The design of CDSL depends on iterators and collection-oriented operations, as the control language intentionally does not allow recursion or arbitrary loops. A Turing-incomplete language is key to keeping the code and proof generation tractable.

3. Modular file system design and verification

Modularity generally has many benefits, including fostering re-use and reducing life-cycle cost. This applies to verification just as well as to programming. File systems are naturally amenable to modular decomposition — all Linux file systems exhibit a similar structural composition [Lu et al. 2013], aimed at supporting a separation of concerns.

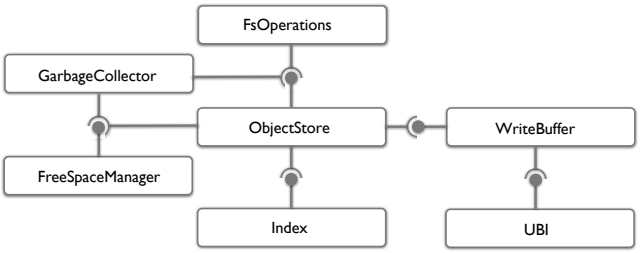

Figure 2. Modular decomposition of BilbyFs

To drive the design and implementation of our framework, we designed a modular flash file system. BilbyFs is meant to be on par with mainstream flash file systems such as JFFS2 and UBIFS. It is built on top of the flash abstraction layer UBI and designed to live underneath the virtual file system layer with which it interacts through the virtual file system switch (VFS).

The design of BilbyFs is highly modular. To allow compositional verification we make sure that components communicate through well-defined interfaces. For the initial prototype of BilbyFs, we trade design flexibility for verification simplicity by avoiding complex component interactions involving callbacks. It remains to be seen how this trade-off will change over time, especially in later design iterations of the file system when we will have to reconcile this restriction with the need to implement asynchronous I/O to achieve acceptable performance.

Figure 2 shows the design of BilbyFs. The FsOperations component implements the interface expected by the VFS, as a client of the ObjectStore component. ObjectStore provides a uniform API for storing arbitrary objects on flash, which FsOperations uses to implement file system objects such as inodes, directory entries and data blocks. This decomposition confines the key file system logic to FsOperations, and makes that component independent of the on-medium representation.

ObjectStore relies on three more components: the Index, which stores the on-medium address of each object, the free space manager (FreeSpaceManager), which keeps track of free space, and the write buffer (WriteBuffer), which allows multiple objects to be atomically written to disk. WriteBuffer's implementation itself relies on the UBI layer to provide reliable block read, write and erase operations on flash. Finally, the garbage collector (GarbageCollector), which in turn depends on the free space manager, maximises the space available for the ObjectStore component. Normally GarbageCollector runs concurrently with file system operations. Since this first version of our verification technology does not yet support reasoning about concurrency, the current design invokes the garbage collector between file system operations.

In its current design, BilbyFs does not store the index on flash. Instead, similarly to JFFS2, it scans the medium at mount time to rebuild the index in memory. This restriction limits the maximum size of the physical medium for which this design is likely to be practical to a few gigabytes. Addressing this limitation is possible without radically changing our approach, and is an obvious future extension, but will increase the size of the implementation and the manual proof effort.

Each file system component is implemented in CDSL, from which its low-level Isabelle specification is generated. This specification is very similar to the input CDSL code, so while the manual proof (④ in Figure 1) must show that these machine-generated specifications implement the high-level specification of file system functionality, the automatic synthesis of these specifications does not create any additional problems for the manual verification.

4. Code and Proof Generation

4.1 Control Code Language CDSL

CDSL is a simple, monomorphic, first-order, functional language with linear types [Wadler 1990], and a Haskell-like syntax. We kept all the restrictions of a strongly-typed, purely functional language: no implicit side effects or destructive updates and no untyped expressions, while omitting most features which make functional languages powerful: higher-order functions, partial application, recursion.

Most systems code, and in particular file-system code, uses only limited forms of repetition, iterating over data structures. In our framework, such iteration is encapsulated by a single control structure, which visually resembles a `for` loop. This structure is parameterised by *iteration schema*, loop patterns which are known to terminate, and can be supplied by separately verified ADTs. Thus, we can get away without recursion in the language.

Furthermore, linear type systems enforce that values are used exactly once, making it impossible to write functions which create, copy or discard linear values. Such operations can only be performed by primitives in the language.

The restrictive nature of CDSL and its strong type system facilitate generation of efficient code, and, more importantly, proofs: the more we know about the code, the stronger the assertions we can provide. For example, the linear type system enables safe disposal of resources and limits the sharing of references to statically well-behaved cases.

We ascribe two semantic interpretations to the same control-code language. The first, *value semantics*, views CDSL as a purely functional language with linear types, with no notion of a mutable store. This semantics is useful for our high-level functional correctness proof, as it is simpler to reason about immutable, constant values than a mutable heap that changes over time.

The second semantics, *update semantics*, is imperative in nature: Values of linear type are represented by locations in an abstract store where the data can be found. Updates to the value — primitive functions which, in the value semantics, appear to consume an old linear value and return a new linear value — are now free to *destructively* update the old value in the store. We can safely do that, because the type system ensures that only one reference to the value exists: destroying the old value will not result in unexpected behaviour elsewhere in the system. The update semantics is useful because it is close to the semantics of the C code generated by the CDSL compiler.

Our type system characterises a subset of programs for which the two semantics coincide. We expect to prove the following *generic* refinement theorem: for any type-correct program p, p evaluated under the *update* semantics will be a refinement of p evaluated under the *value* semantics.

We adopt existing Isabelle/HOL frameworks for giving semantics to the generated C code [Tuch et al. 2007, Greenaway et al. 2012]. The update semantics is relatively close to the typed-heap model of C produced by recent versions of Greenaway et al.'s framework [Greenaway et al. 2012]. We hope to exploit this similarity to simplify the generation of proofs showing the correspondence between a program's update semantics and the semantics of its generated C code. Combined with the generic refinement theorem, these will show that the generated C code correctly implements the program's value semantics.

4.2 Data Description Language DDSL

DDSL specifies *physical* (on-medium) data layouts and synthesises correct CDSL code for *serialising* in-memory data structures to physical representation and *de-serialising* back. There are numerous data-description tools [McCann and Chandra 2000, Back 2002, Fisher et al. 2010] that perform similar tasks, some of them, like those in the PADS family, have been the subject of pen-and-paper correctness proofs. Our ambition goes well beyond: our tool will not only produce code, but also mechanically checked Isabelle specifications and correctness proofs for the generated CDSL code.

We base DDSL on PADS [Fisher and Walker 2011], extended with support for bitfields and tagged unions, inspired by Cock [2008]. While PADS data specifications can contain arbitrary user-provided constraints on values, we limit constraints to built-in boolean expressions defined in DDSL, as we generate both Isabelle and CDSL code from the DDSL specification.

The generated CDSL code includes datatype declarations for in-program data representation as well as corresponding serialisation and de-serialisation functions, including error checking.

As a simple example, the following DDSL snippet describes physical data consisting of one byte "`a`" directly followed by a little-endian 32-bit word "`b`":

```
data SObj = { a :: Pu8, b :: Ple32 }
```

From this the DDSL compiler generates structure and function definitions in CDSL, which for this example comprise about 20 source lines of code (SLOC), and whose prototypes are given here (`Buf` is a buffer ADT defined elsewhere):

```
SObj = {a : #U8, b : #U32}
sobj_new : (#U8, #U32) -> .SObj + ()
sobj_free : .SObj -> ()
sobj_deser : (*Buf!, #U32) -> (.SObj, #U32) + ()
sobj_ser : (*SObj!, .Buf, #U32)
        -> (.Buf, #U32) + (.Buf, #U32)
```

The first line declares a type, `SObj`, to be a record containing two (unboxed) integers, of size 8- and 32-bits respectively. The second and third lines declare functions for creating and destroying `SObj`s respectively. A preceding dot, as in `.SObj`, denotes a value of linear type, `#` a value of unboxed type, and `*` a read-only (shareable) value. The trailing `!` with a preceding `*`, as in `*Buf!`, indicates a "deep" read-only value (i.e. one whose fields are also read-only if it is a record). The `+`s indicate those functions that can potentially throw an error and are discussed later in Section 4.3. The fourth line declares the deserialisation function `sobj_deser`, which takes a buffer `pb` to deserialise from the given index `pos`, and on success, returns a newly-allocated `SObj`. Lastly, the serialisation function `sobj_ser` is declared to take a reference `p` to a read-only `SObj` to be serialised, and a buffer `pb` to serialise to at the given index `pos`.

DDSL will also generate Isabelle specifications which capture the correctness of the generated CDSL code in a *parsing* relation, which connects the physical and in-program data representations. This relation forms the centrepiece of the formal DDSL semantics. In this example, a parsing relation $\mathsf{SObjR}\ p\ pb\ pos$ would be generated, which relates `SObj` structures p to their corresponding serialised representation in a buffer pb at position pos.

The generated correctness lemmas for the CDSL functions would assert that they establish the parsing relation between the physical and in-program data representations. The correctness lemmas for the `SObj` example would include:

$$\text{case } (\mathsf{sobj\_deser}\ pb\ pos) \text{ of } \mathsf{OK}\ (p, pos') \Rightarrow \mathsf{SObjR}\ p\ pb\ pos$$
$$\text{case } (\mathsf{sobj\_ser}\ p\ pb\ pos) \text{ of } \mathsf{OK}\ (pb', pos') \Rightarrow \mathsf{SObjR}\ p\ pb'\ pos$$

The first says that that, if successful, `sobj_deser` $pb\ pos$ gives a resulting `SObj` that corresponds to the contents in the given position pos in the given buffer pb. The second says that, if successful, `sobj_ser` $p\ pb\ pos$ gives a buffer pb' and index pos' such that the contents at position pos in pb' corresponds to the given `SObj` p.

The generated code, and thus its proofs, are relatively straightforward. The problems here (e.g. of reasoning about bitfield operations [Cock 2008, Cohen et al. 2009]) are relatively well understood and we expect this part of the project to yield early results.

4.3 Error Handling and Effects

As mentioned earlier, a large part of the control code of file systems is concerned with error handling. Often, a transactional style is employed for error cases, where functions return an integer code that indicates whether they have executed successfully or encountered an error. If the code indicates failure, the caller must *unwind* its previous actions – in particular, releasing any allocated space. A large amount of file system bugs arise from failure to appropriately handle error cases [Lu et al. 2013].

To solve this problem, we make it impossible to ignore the possibility of errors being returned from the function, by modelling functions that can fail as returning a *sum type* of the form $(\tau_0, \ldots, \tau_n) + (\tau'_0, \ldots, \tau'_m)$, which describes a function returning a collection of values each of type τ_i respectively in case of success, and in case of failure an error code (not explicitly listed in the type) and a collection of values each of type τ'_i respectively.

The type ensures that the caller *must* handle the error case in order to get the return value. For instance, recall that the function `sobj_new` takes an 8- and a 32-bit integer as its arguments, and returns an `SObj`. If it fails, it only returns an error value, so its return type is given as `.SObj + ()`. To access the `SObj`, the caller must handle the potential error. In contrast, the function `sobj_free` returns nothing and the type indicates that it always succeeds, so no error handling code is necessary. The serialisation and deserialisation functions can each yield errors, for instance when called with a buffer that is too short.

To gain an intuition of how this works, consider the following CDSL code, which tests the `SObj` serialisation and deserialisation operations.

```
sobj_example (buf : .Buf) : (.Buf) + (.Buf) = {
  so <- sobj_new(0, 666)
    handle (err) { fail (err,buf) };
  buf, new_pos <- let! (so) sobj_ser(so, buf, 0)
    handle (err, buf, idx) {
      sobj_free(so);
      fail (err,buf)
    };
  so2, new_pos' <- let! (buf) sobj_deser(buf, 0)
    handle (err) {
      sobj_free(so);
      fail (err,buf)
    };
  let! (so,so2)
    assert (not(so.a == so2.a && so.b == so2.b &&
                new_pos == new_pos'));
  sobj_free(so2); sobj_free(so);
  return (buf)
}
```

This code creates an object and checks that, after serialising and deserialising it, its contents are identical to the original. The function has type `.Buf -> .Buf + .Buf`, as

it takes a linearly-typed buffer as its sole argument, which it returns on both success and error; in case of error it also yields an error value. The code can be read as a sequence of instructions, with left-arrows similar to assignments, possibly to multiple values at the same time if a function returns more than one result, as for example `sobj_ser`. Every function that could potentially fail has an associated handler. Here, the linear type system ensures that we free all objects that we have allocated. Omitting any of the `free` statements in this example would be a type error.

The error handling code is fairly verbose at the moment. This can be alleviated easily in the future by adding syntactic sugar for common cases.

The `let!`-statements, taken from Wadler [1990], allow a value of linear type to be temporarily shared in a read-only context. They make for instance the assertion condition in the example well-typed, but they do not need to be present in the generated C code.

Aside from side effects which only exist in the update semantics of CDSL, such as destructive updates to the file system state, we also have effects relating to the outside environment, e.g. the OS and storage medium. These effects are present in the value semantics as well, as we model the outside world as a (collection of) linear values.

5. Summary

As alluded to in Section 1, we have the completed the design of first versions of DDSL and CDSL and used them to specify the initial design of BilbyFs, based on a hand-written C prototype of the file system. While we do not expect major changes to the languages at this point, more complicated case studies planned for the future may lead to adjustments. We are also planning to add more syntactic sugar to allow for more concise expression of common patterns and idioms.

For DDSL we have a pen-and-paper formal specification of the static semantics and have implemented the CDSL code generation. We are currently working on the generation of Isabelle/HOL specifications and correctness proofs. DDSL specifications are compact: small DDSL specifications encompass a large amount of CDSL code, usually around 8 times as much. Since the generated code is simple, we expect proof generation to be relatively straightforward.

We have formally defined both the value (in Isabelle/HOL) and update (with pen-and-paper) semantics of CDSL, and proved type soundness, as well as the refinement between update and value semantics for an earlier, more complicated, design of the language. These should port easily to the current, simpler CDSL design. These are important steps for showing the correctness of the generated C code.

We are currently generating (unverified) C code from CDSL, to get some idea of how efficient the verified code will be and where the trade-offs are.

The manual top-level proof of functional correctness (part ④ in Figure 1) for BilbyFs, which makes the generated lower-level specifications trustworthy, is in progress as well. This proof currently uses manually-written lower-level Isabelle specifications of the file system components. These will soon be automatically produced by the CDSL compiler, via e.g. CDSL's value semantics.

Based on our experience so far, we are confident that the completely automatic generation of the code of a realistic file system and the automated verification of the correctness of its modules, is feasible.

References

G. Back. DataScript – a specification and scripting language for binary data. In *Conference on Generative Programming and Component Engineering*, pages 66–77. Springer, 2002.

T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 35–42, Lugano, Switzerland, 2010. FMCAD Inc.

A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, Feb. 2010.

D. Cock. Bitfields and tagged unions in C: Verification through automatic generation. In B. Beckert and G. Klein, editors, *Proceedings of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, Aug. 2008.

E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. In *Proceedings of the 4th Systems Software Verification*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 85–103. Springer, 2009.

K. Fisher and D. Walker. The PADS project: An overview. In *International Conference on Database Theory*, pages 11–17. ACM, 2011.

K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. *Journal of the ACM*, 57:10:1–10:51, Jan. 2010.

D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In L. Beringer and A. Felty, editors, *3rd International Conference on Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 99–115, Princeton, New Jersey, Aug. 2012. Springer. ISBN 978-3-642-32346-1.

W. Jannen, C.-C. Tsai, and D. E. Porter. Virtualize storage, not disks. In *Workshop on Hot Topics in Operating Systems*, pages 1–7, Santa Ana Pueblo, NM, USA, May 2013.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct. 2009. ACM. doi: 10.1145/1629575.1629596.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *USENIX Conference on File and Storage Technologies*, San Jose, CA, USA, Feb. 2013.

P. J. McCann and S. Chandra. PacketTypes: abstract specification of network protocol messages. In *ACM Conference on Communications*, pages 321–333, Stockholm, Sweden, 2000.

N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–318, Newport Beach, CA, USA, 2011.

S. Saha, J. Lawall, and G. Muller. An approach to improving the structure of error-handling code in the Linux kernel. In *Conference on Language, Compiler and Tool Support for Embedded Systems (LCTES)*, pages 41–50, Chicago, IL, USA, Apr. 2011.

H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, Jan. 2007. ACM.

P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24:393–423, 2006.