Implementing a Verified On-Disk Hash Table

Stephanie Wang

Abstract

As more and more software is written every day, so too are bugs. Software verification is a way of using formal mathematical methods to prove that a program has no bugs. However, large-scale verified software systems are often hard to use or integrate for a programmer with no background in verification. In this project, I present a verified key-value store implemented as an on-disk hash table on the verified file system FSCQ. This is done by layering abstract representations on top of one another. In this case, the lowest abstraction layer is an array, or a raw disk, and the highest is a map, or a key-value store. This project serves as a case study for implementing commonly used data structures on disk.

1 Introduction

It's becoming clearer than ever that the current methods for proving software correctness in the wild are not enough. Most programmers rely on code reviews and on writing good test cases, but these methods are rarely exhaustive. Nuanced bugs can and do often slip through cracks, sometimes with disastrous results.

Formal software verification can ensure that a software system has no bugs, but that means little if its interface or limitations make it unusable to another programmer. For example, the verified file system FSCQ is powerful in its expressiveness regarding disk operations and crashes, but programmers are unlikely to want to deal with the raw disk interface that FSCQ exposes [1]. In order to make verified software a viable option in everyday systems, it's necessary to build and verify easily composable systems. In this project, I propose to build and verify a key-value store on top of FSCQ.

There are a couple reasons for choosing a key-value store. The first is that it can be built with a simple interface that's familiar to many programmers without experience in verification. Also, it's a widely studied system with many applications in different contexts.

While this may be the umpteenth key-value store in existence, it's also one of the first examples of a certified on-disk data structure. To reach performance competitive with a non-verified implementation, it's necessary to implement the key-value store as an on-disk hash table. This makes the project a prime case study for verifying on-disk data structures.

In Section 3, we will discuss how to verify a single implementation of a key-value store. We'll give an append-only log in Section 3.1 and a hash table in Section 3.2 as examples.

2 Previous Work

Formal software verification is growing fast. There have been several exciting projects where teams of programmers and verifiers have succeeded in building and verifying large and critical systems. These systems have varied from tools like a C compiler [4] to application infrastructure like an operating system [3].

One ongoing project is FSCQ, a verified filesystem that also provides a type of logic for proving program correctness in the presence of disk crashes [1]. This logic is called “Crash Hoare logic". Hoare logic is a long-established set of formal proof methods that allows one to define a program's correctness in terms of pre- and post-conditions [2]. To prove a program's correctness, one must prove that if the pre-condition holds before entering the program, the post-condition must hold after exiting the program. A program in this case is really an abstraction; it can represent a single statement or a 1000-line Python script. Hoare logic allows one to easily compose programs like these into larger programs, as long as each post-condition fulfills the pre-conditions for the following program.

The “Crash" in Crash Hoare logic comes from having a crash condition in addition to the pre- and post-conditions [1]. To prove a crash condition for a program, one must prove that if the pre-condition holds, the crash condition will hold after any given step of the program, since a crash could happen at any point.

FSCQ provides several automation tactics to help step through a program built on the write-ahead log layer `MemLog`. These tactics are critical for proving the described pre-, post-, and crash-conditions.

Another useful FSCQ proof technique is isolation logic. This is the idea of isolating different parts of the disk, or any other array, away from each other. In this way, one can specify a change in one part of the disk while leaving the rest of the disk unspecified. Under isolation logic, the unspecified area of the disk remains unchanged. Then, one can isolate proof efforts to the changed part of disk.

Another concept used in verifying FSCQ, as well as many other verified systems, is the idea of abstraction layers. This is similar to the same idea in normal software, where complex systems abstract away lower-level systems that are used in the implementation. The difference when writing verified software is that one also has to prove that the low-level implementation truly implements the high-level abstraction.

This project requires combining these verification techniques to build a top-level abstraction of a key-value store that can be easily used by other programmers. The lowest-level abstraction layer can be thought of as an array, or raw disk blocks, while the top-level abstraction is a map of keys to values.

3 Design

The key-value store will be implemented as a layer on top of the write-ahead log layer (`MemLog`) provided by FSCQ [1]. `MemLog` provides atomic transaction semantics for operations like `read`, `write`, and `commit`.

Efficient hash tables can be complex to represent and verify in Coq because of the various methods of collision resolution, such as open addressing or chaining. For this reason, we chose to implement a key-value store layer in several iterations before attempting a fully reliable and efficient hash table. Reliable in this case means that as long as there is room on the disk, a key and value can always be put successfully.

Each iteration on the key-value store layer requires three parts. The first is an abstract representation of a key-value store, denoted as a property *rep* of a list of key-value pairs. We chose to start with a list abstraction rather than a Coq map like FMap, even though the latter matches a key-value store more closely, to make proofs more feasible. This is because Coq proofs often depend on induction and are therefore more suited to list objects than map objects. Each version of *rep* specifies the existence of some raw disk and a relationship between the list of key-value pairs and the contents of this existential disk. This representation must be strong enough that it specifies both directions of the relationship. In other words, there must be some way of translating a key-value store into the contents of a disk, as well as some way of translating the same disk contents back to the same key-value store.

The second part is implementations for the `get` and `put` operations. These are often quite short and simple. A `get` takes in a key and returns a value. A `put` takes in a key and a value, and returns a boolean `ok`, which shows whether the `put` was successful or not.

Finally, bringing the two together are theorems that state `get` and `put` correctness in terms of the abstract representation *rep*.

There will be two main correctness theorems, one each for `gets` and `puts`. In general, each of the correctness theorems requires that the representation holds for the disk before the operation. They also require some post-condition regarding the return value of the operation or the state of the disk.

Although the Crash Hoare logic is essential for reasoning about disk crashes, most of the crash-conditions for the theorems specified in this project are relatively simple. The crash-conditions for a general Hoare theorem are used to specify the possible disk states after a disk crash. A `get` is read-only and a `put` requires only one commit. Then, we only need to account for at most two possible disk states after a crash, either the original disk, or the disk after a `put`. These two states will always match either a pre- or post-condition, so all proofs for the crash-conditions can be solved almost entirely automatically. So, for the purposes of simplifying the discussion of the `get` and `put` correctness theorems, we will only discuss the pre- and post-conditions.

For each iteration, the `get` correctness theorem states that as long as the defined representation holds for the disk, then any `get` will retrieve the most recent value called by a `put` at the requested key. For a hash table, which doesn't generally store older values for a key, the most recently put value can just be the value at the key in the current store. A `get` should be read-only, so isolation logic is sufficient to prove that the representation still holds for resulting, unchanged disk. Formally, for any key-value list l, key k, and value v:

$$\frac{rep\ \ l}{is\_last\_put(l, k, v)} \qquad \texttt{get(k)}$$

The `put` correctness theorem states that as long as the defined representation holds for the disk and there is room, then after any successful `put`, the representation holds for the updated disk. In other words, the key-value store is updated with the new key and value, according to what "updated" means in the context of the defined representation. In this case, an immediately subsequent `get` to the same key should return the same value. If the `put` is unsuccessful, then the disk should remain the same. Formally, for any key-value list l, key k, and value v:

$$\frac{rep\ \ l \bigwedge length\ \ l < maxlen}{ok = true \bigwedge rep\ \ update(l,\ \ key,\ \ value)} \qquad \texttt{put(k, v)}$$

If the `put` is not successful (if $ok = false$), then like the `get` operation, the disk remains unchanged. Again, isolation logic is enough to prove this.

All theorems are proven by unfolding the `get` and `put` operations and then using small-step semantics on the underlying disk operations. Proving these theorems shows that the on-disk data structure and operations actually implement the *rep* abstraction.

As mentioned above, the representation *rep* is an abstraction to a list of key-value pairs, but the more logical abstraction would be a map that associates keys to values. Although directly proving that a map abstraction holds on a raw disk is difficult, this could be feasible using refinement. Refinement proves equivalence between an abstract model and an implementation of the model by proving that each abstract step, along with its pre- and post-conditions, has an equivalent step in the implementation. In this case, we could refine a map abstraction to the list representation that we work with in this project.

Next, we'll discuss examples of key-value store representations, specifically an append-only log and a hash table with no collision resolution. For each implementation, we'll discuss the representation *rep*, the `get` and `put` operations, and the correctness theorems.

3.1 Append-Only Log

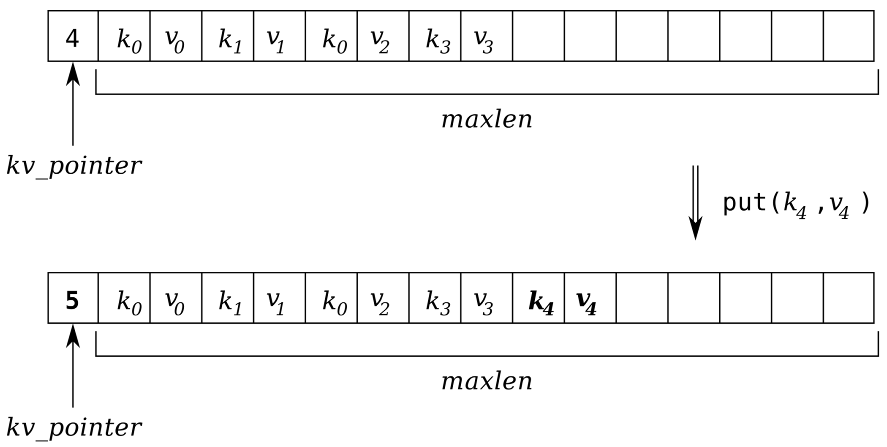
An append-only log is probably the easiest, and least efficient, representation of a key-value store to start with. For this representation, a log of all (*key*, *value*) pairs is stored on disk (Figure 1a). The log represents the history of all `put` calls, with the most recent last.

There are five parts relating the append-only log representation to the disk contents (Figure 1b). The first states that there is a constant index, *kv_pointer*, on disk that stores the current length of the log. The second states that the keys are written at every other index in an array on disk. The third states that the values are written at the other indices. The fourth states that the length of the total disk is equal to `maxlen`. Finally, the fifth states that the log of key-value pairs is a contiguous prefix of the disk.

If any one of these parts were missing, the representation would not be strong enough to prove correctness for `get` and `put`. For example, if the fifth part regarding the disk prefix were excluded, then it'd be possible to have all the correct key-value pairs in their correct log order on disk, but with holes in the disk. This would completely invalidate the log length stored at *kv_pointer*.

A `get` returns the last value in the log that matches the desired key. This is implemented by first reading the length of the log from the *kv_pointer* block. Then, the operation loops through pairs of blocks from 1 to the length of the log. For each pair (*key*, *value*)in the list, if *key* matches the desired key, then the value passed to the next iteration of the loop is updated to be *value*.

A `put` is an append of a (*key*, *value*) pair to the end of the list, which increments the length of the list by 1 (Figure 1a). This is implemented by first reading the length of the log from the *kv_pointer* block. Then, *key* is written at $2 \times kv\_pointer$ and *value* is written at $2 \times kv\_pointer + 1$. To update the length of the log, the value at *kv_pointer* is incremented by 1.


4
k_0 v_0 k_1 v_1 k_0 v_2 k_3 v_3
maxlen
kv_pointer
put(k_4 ,v_4)
5
k_0 v_0 k_1 v_1 k_0 v_2 k_3 v_3 k_4 v_4
maxlen
kv_pointer

(a) An example of the disk contents corresponding to an append-only log representation. *kv_pointer* is an index on disk that contains the number of pairs currently in the log. When a `put` is successful, the new key and value are written at the value in *kv_pointer* and the value in *kv_pointer* is incremented.

```
Definition rep l := (exists diskl,
  kv_pointer |-> addr2valu $ (length l) *
  array kBase (map (fun e => addr2valu (fst e)) diskl) $2 *
  array vBase (map snd diskl) $2 *
  [[ length diskl = wordToNat maxlen ]] *
  [[ list_prefix l diskl ]])%pred.
```

(b) The Coq definition for the append-only log representation. The first three lines define where the length of the log, the keys, and the values can be found on disk. The fourth line limits the length of the log to *maxlen*. The final line ensures that the log is a contiguous prefix of the actual disk.

Figure 1: The on-disk contents and representation definition for an append-only log.

The idea of an “update" for this representation is relatively simple. If l is the log of key-value pairs before a call to `put`(*key*, *value*), then l appended with (*key*, *value*) will be the log afterwards.

Once all of the above details were specified properly, proving `get` and `put` was straightforward, so we will not discuss the details of proving the above theorems. This is in part due to the very direct relationship between the key-value log and the disk structure itself. For more interesting and much more efficient examples, we'll next examine a key-value store with hashing.

3.2 Hash Table

To ease the verification process, there are several limitations on the first iteration of a hash table layer. The first is that there is no collision resolution method. So unless the desired key matches the key already at the hashed index, a key collision will result in an unsuccessful `put`. The second is that the hash table has a fixed size. The only specification for the hash function h used is that it takes in a key and outputs a disk address.

The representation for this layer is significantly more complex than that of the append-only log. A few properties carry over. These are the properties specifying that keys and values are found at alternating indices on disk, allowing us to treat the disk as a list of pairs, as well as the property that the length of the disk is `maxlen`.

The remaining four properties relate a list containing all key-pairs currently in the store and the contents of the disk (Figure 2):

(1) For any pair (*key*, *value*) in the list, (*key*, *value*) is written at the correct index on disk. Specifically, if the disk were treated as a list of pairs, then (*key*, *value*) would be the h(*key*)-th pair.

The next two properties consider the existence of a second list l' of key-value pairs. (2) If the disk were to be filtered for all nonempty blocks, then the result would be some l'. (3) l' is a permutation of l, the key-value store.

Finally, (4) there are no duplicate keys on the disk.

The first property relates the list contents to the disk contents; the remaining three deal with the opposite relation, as well as the remaining disk contents. Again, although having all four of these properties may seem overly complex, all properties are necessary to prove correctness of `get` and `put`. For example, without the property preventing duplicate keys, there could be multiple instances of some (*key*, *value*) pair on disk. As long as the h(*key*) index on disk really did contain (*key*, *value*), this would still be a valid disk under the representation. However, it would be impossible to prove property (1) after a different value were put at the same *key*. This is because the other copies of (*key*, *value*) would still be on the disk and would then, by properties (2) and (3), also be in l. But after the `put`, the h(*key*)-th pair on disk would no longer contain *value*, so property (1) no longer holds.

The `get` and `put` operations remain simple. For a call to `get`(*key*), the disk is read at index $2 \times h(key)$. If there is already a key there and it matches *key*, then the value at the next disk block is read out and returned.

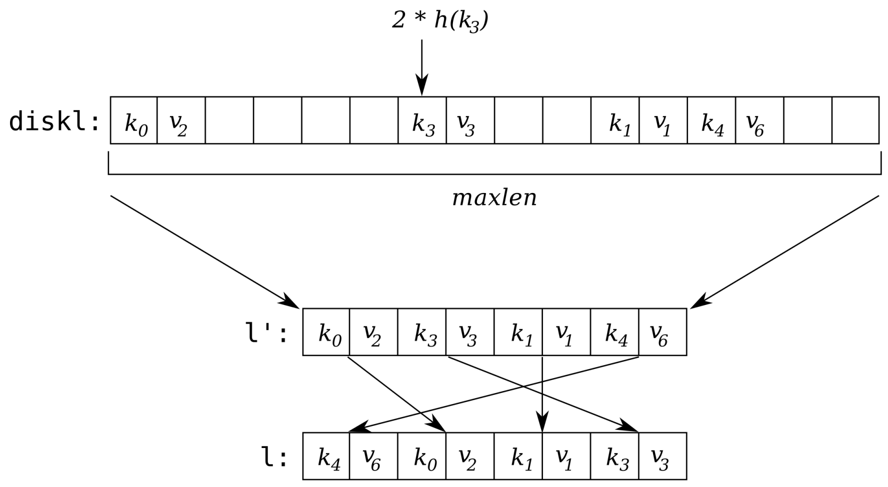
The `put` operation works similarly. For a call to `put`(*key*, *value*), the disk is read at index $2 \times h(key)$. If there is already a key there and the key does not match *key*, then the `put` fails and returns $false$. Otherwise, *key* and *value* are written at that index and the next, respectively, and `put` returns *true* for success.

For a hash table, we no longer have to worry about the most recently put value because old values are never stored. Instead, a `get` must guarantee that whatever value is returned is actually the right value in the current key-value list l. And if no value is returned, it must be true that the desired key does not appear in l with any value.

The concept of an “update" to the key-value store in this case is similar to the idea of an upsert. If the key to update is already in the key-value list l, then the corresponding value in the tuple is updated. Otherwise, the new key-value pair is inserted into the list. A key-value update to the disk contents just writes the key and value at the hash of the key.

This representation is much harder to prove correct than the append-only log because the properties above are, for the most part, disjoint. This means we have to prove the properties individually even though they may depend on each other. For instance, after a `put`, all properties must be proven for the updated key-value store and disk. Property (1) can be difficult to reason about without also proving what key-value pairs are in the list l. But the contents of l follow from properties (2) and (3), which we are also trying to prove.

The `get` correctness theorem is easier to prove than the `put` correctness theorem because there is no disk state



(a) An example of the disk contents *diskl* corresponding to a hash table representation. l' is the disk contents after filtering out all empty entries. l is the actual key-value store, which is some permutation of l'. Any key-value pair in l can be found at the hash of the key on disk.

```
Definition hash_rep' l diskl := exists l',
  hash_rep l diskl /\
  filter_empty diskl = l' /\
  Permutation l' l /\
  no_dup diskl.
```

(b) The Coq definition for the subset of the hash table representation that defines the relationship between the disk contents on *diskl* and the key-value pairs in the store *l*. The first property, *hash_rep* says that any key-value pair in *l* can be found at the hash of the key on disk. The next two properties consider some other key-value list l' that has all the non-empty entries on *diskl* and is a permutation of the actual store, *l*. The final property says that there are no duplicate keys on *diskl*.

Figure 2: The on-disk contents and representation definition for a hash table.

change during a `get`. To prove `get`, it is enough to prove a couple lemmas that consider nonempty disk entries. These lemmas translate properties (1)-(4), which may consider the whole disk or key-value store list at once, into implications about individual entries on disk.

The `put` correctness theorem is much more difficult to prove for the reasons discussed above. There are three cases here for the desired *key* and *value*: (1) the *key* collides with another key that is already written at the same index. (2) there is already a value for the same *key*. (3) the index $h(key)$ is empty on disk.

There is little to prove for case 1, since nothing has changed on disk. This can be proven just using proof automation.

The second case when there is already a value for the same key at the proper index is slightly harder, but there is still no addition of a new key, just an update to the value. Therefore, properties like property (1), which states that the index $h(key)$ must contain the correct key and value, are easy to prove because only the value at the key has changed since the original disk state.

The third case is the hardest because thus far, the properties stated only consider keys that are in the key-value list. Nothing has been proven about the absence of keys. For instance, here is one lemma that was necessary to prove the third case:

Lemma 3.1. *If diskl at $h(key)$ is empty, then for any key' such that $h(key) = h(key')$ there does not exist a value such that $(key', value)$ is in the current key-value store.*

3.3 Future Goals

So far, I have proven everything except the third case for a `put`, as described in Section 3.2. Once this hash table is fully proven, the next steps will be to add a collision resolution method. We believe that the easiest extension from the current hash table representation would be open addressing with linear probing. While this would make the hash table more reliable in the event of a key collision, this type of collision resolution can over time accrue unnecessary overhead for certain `get` and `put` call. This problem could be reduced by extending the current hash table representation with chaining.

Verifying the hash table representations extended with some collision resolution method would closely follow the verification steps described in this section. We suspect that the first obstacle will be to properly specify the representation itself. For instance, trying to represent something like open addressing with linear probing will require an extra property describing the relationship between the entries themselves, since certain entries will appear in certain indices on disk based on what entries were there first.

Verifying a hash table with chaining will be difficult to do directly from the current representation. This is because chaining will probably require linked lists of blocks on disk, or else we'd only be able to support a bounded number of hash collisions. To implement variable-length chaining, we would need to define an intermediate representation that abstracts away some set of disk blocks into a linked list.

Finally, to reach performance and space efficiency competitive with existing key-value stores, it will probably be necessary to eventually implement resizable hash tables. Again, verifying a resizable hash table will probably closely follow the verification process for the fixed-size hash table described here, with an additional constraint that the size is within a certain bound of the number of entries currently in the store. The `get` implementation could just be a call to the `get` for a fixed-size hash table. For the `put` (and eventually `delete`) operation, a call to the fixed-size `put` may be followed by a resize operation. The challenge here will be in proving that the abstraction still holds on the disk after any `put`, even while many key-value entries may change disk location.

In the (distant) future, the verified key-value store could also be made concurrent. The current key-value layer is being built on top of the write-ahead log `MemLog`, a layer in FSCQ that guarantees transaction atomicity but has no concurrency control. If a concurrency layer were to be built into or on top of `MemLog`, a serializability guarantee like snapshot isolation could easily be the key-value store, as well as any other systems built on top of `MemLog`.

It may even be possible to transform this key-value store into a replicated hash table by integrating with Verdi [5]. Verdi is a framework that transforms state machines into formally verified distributed systems with verified consistency and fault-tolerance properties. Verdi already has an example of a verified key-value store called *vard*. However, *vard* is not verified to the filesystem level and does not seem to support multi-operation transactions, which would probably be relatively easy to add to an on-disk hash table. By composing Verdi and FSCQ together in this way, we could build a key-value store with verified transactional properties from FSCQ, as well as verified consistency and fault-tolerance proprerties from Verdi. Still, it is impractical to reason about which key-value store operations should be transformed into distributed Verdi log operations until a fully-featured key-value store is built and verified. In addition, using the Verdi framework would not be enough to produce the highly desirable distributed hash table. To build such a system, we'd likely have to modify the key-value store code directly.

4 Discussion

Above all, the lesson I learned from this project was that formalization is hard. A hash table is a data structure that many programmers have learned about and understand at a programmatic level, but trying to specify its behavior in formal language is a very different story, often surprisingly so. For instance, the representation we'd written for a hash table with no collision resolution in Section 3.2 was incorrect for several weeks before we even noticed after trying to prove a false lemma. Even now, it's impossible to tell whether the representation I'm trying to prove is strong enough (although I have high hopes)!

In the same vein, my biggest mistake was to jump into proving things before I'd thought about the representation's specification enough. The equivalent key-value store in unverified software would probably be simple enough that any mistakes in the code could be ironed out simply by going back and fixing an offending line or two. Here, though, the proofs are much more complex than the code and may often require a stronger or weaker specification. Although I have no way of telling for sure, fixing the specification after attempting an impossible proof is probably much more time-costly than taking more time to think about the specification before attempting a possible proof.

Automation was limited in this project. FSCQ's provided automation was essential for the project, but I had little idea of how to automate any of the manual proofs that I did. In addition, the couple times FSCQ automation failed, it was difficult for me to determine the next step. My only choice was to break down the automation tactic into its individual steps and try each one individually. But, although I knew at a high level what the automation tactic did, I had little idea what each individual step was responsible for.

One thing I'd love to see from Coq is a better way to explore theorems and specifications without breaking everything. Ideally, a verifier should probably specify once and never have to revise once starting the proofs, but I believe this is rarely the case in practical situations. My own process was to iterate on the abstract representation, lemmas, and proofs in turn until the correctness theorems could be fully solved. Often, however, one small change in a representation could break all of the parts that did work. This could be as simple as renaming a hypothesis used in a proof, or as complex as completely rewriting a lemma, followed by all dependent lemmas.

I'd also love a richer theorem search engine than the one currently included in Coq, which matches queried Coq terms for any and all theorems that mention them. This could often lead to too many search results, with no way to narrow them down other than to query for additional Coq terms. I often had a feeling that I was proving something that had already been proven before in a more general form. In this situation, I had no way of finding the more general form besides browsing the Coq documentation.

5 Conclusion

The overhead of formally verifying software systems is still quite high for the average programmer. One way around this is to build certified systems that can be integrated with other systems with minimal verification effort. A verified filesystem like FSCQ is a prime example of such a system.

The raw disk that FSCQ is built on can be repurposed for other efficient on-disk data structures. In this project, we work towards building and verifying a key-value store as an on-disk hash table using the disk interface provided by FSCQ. The resulting system will have a simple and familiar enough interface that any programmer should be able to use it.

This ongoing project also serves as an interesting case study for verifying and even just thinking about on-disk data structures. As evidenced by this project, even a simple data structure can have a complex representation with several properties that must somehow be bound together. For the sake of an easier proof effort, it's essential to express the representation through as few properties as possible. And as in all formal verification, the most important goal appears to be for the specification to strike the right balance between strong and specific versus weak and general.

References

[1] Using Crash Hoare Logic for Certifying the FSCQ File System. Submitted to *ACM Symposium on Operating Systems Principles (SOSP)*, Cambridge, Massachusetts, 2015.

[2] C.A.R. Hoare, An axiomatic basis for computer programming. In *Communications of the ACM* 12 (10), pages 576-580, 1969.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, 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 (SOSP)*, pages 207âĂŞ220, Big Sky, MT, Oct. 2009.

[4] X. Leroy, A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363âĂŞ446, Dec. 2009

[5] J.R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M.D. Ernst, and T. Anderson. *Verdi: A Framework for Formally Verifying Distributed System Implementations*. Conditionally accepted to PLDI 2015.