DOI:10.1145/3052937

Article development led by acmqueue
queue.acm.org

The fuzzer is for those edge cases that your testing did not catch.

BY ROBERT GUO

MongoDB's JavaScript Fuzzer

AS MONGODB BECOMES more feature-rich and complex with time, the need to develop more sophisticated methods for finding bugs grows as well. Three years ago, MongDB added a home-grown JavaScript fuzzer to its toolkit, and it is now our most prolific bug-finding tool, responsible for detecting almost 200 bugs over the course of two release cycles. These bugs span a range of MongoDB components from sharding to the storage engine, with symptoms ranging from deadlocks to data inconsistency. The fuzzer runs as part of the continuous integration (CI) system, where it frequently catches bugs in newly committed code.

Fuzzing, or fuzz testing, is a technique for generating *randomized, unexpected, and invalid input* to a program to trigger untested code paths. Fuzzing was originally developed in the 1980s and has since proven to be effective at ensuring the stability of a wide range of systems, from file systems[15] to distributed clusters[10] to browsers.[16] As people have attempted to make fuzzing more effective, two philosophies have emerged: smart and dumb fuzzing. As the state of the art evolves, the techniques that are used to implement fuzzers are being partitioned into categories, chief among them being *generational* and *mutational*.[7] In many popular fuzzing tools, smart fuzzing corresponds to generational techniques, and dumb fuzzing to mutational techniques, but this is not an intrinsic relationship. Indeed, in our case at MongoDB, the situation is precisely reversed.

Smart Fuzzing

A smart fuzzer is one that has a good understanding of the valid input surface of the program being tested. With this understanding, a smart fuzzer can avoid getting hung up on input validation and focus on testing a program's behavior. Testing that a program properly validates its input is important but isn't the goal of fuzz testing.

Many fuzzers rely on an explicit grammar to generate tests, and it is that grammar that makes those tests smart. But MongoDB's command language

Figure 1. AST of one command in the corpus.

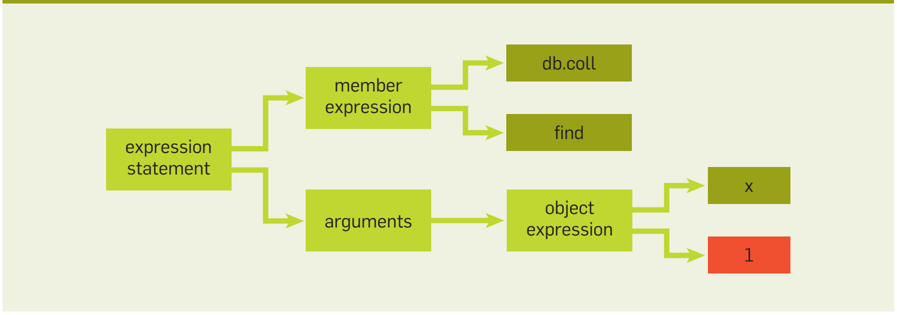


Figure 2. Node replaced with placeholder node.

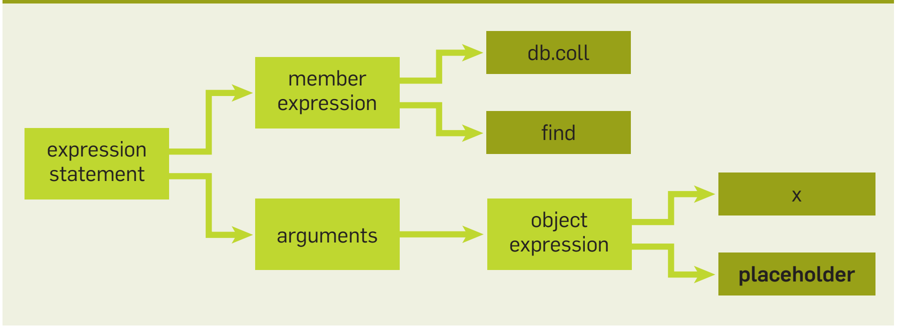
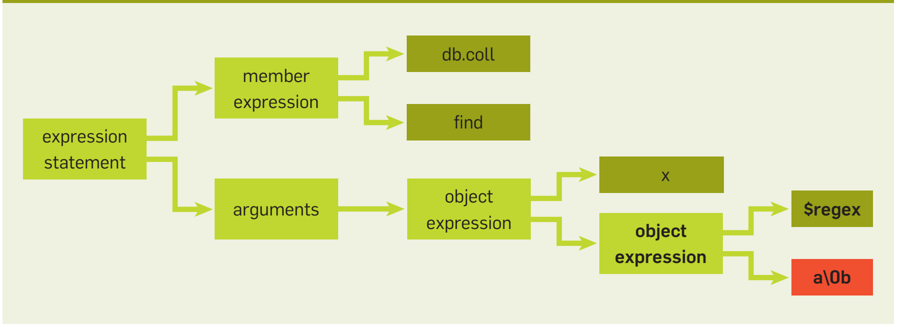


Figure 3. Placeholder replaced with another ObjectExpression.



is young, and we did not want to delay our fuzzer's delivery by taking the time to distill a formal grammar. Instead, we borrow our knowledge of the MongoDB command grammar from our corpus of existing JavaScript integration tests,[18] mutating them randomly to create novel test cases. Thus, our *mutational* strategy results in a *smart* fuzzer.

These JavaScript integration tests have been a mainstay of our testing for many years. Our CI system, Evergreen,[8] invokes a test runner that feeds each test file to a mongo shell, which executes the commands within the test file against MongoDB servers, shard routers, and other components to be tested. When the fuzzer runs, it takes in a random subset of these JS tests and converts them to an AST (abstract syntax tree) of the form understood by JavaScript interpreters. It then wreaks (controlled) havoc on the tree by selectively replacing nodes, shuffling them around, and replacing their values. This way we generate commands with parameters that wouldn't be encountered during normal testing but preserve the overall structure of valid JavaScript objects.

For example, the code `db.coll.find({x:1})` finds a document in collection `coll` with a field `x` having the value `1`, as shown in Figure 1.

To begin fuzzing that AST, the fuzzer first traverses the tree to mark nodes that should be replaced. In this case, assume it has decided to replace the value of the `ObjectExpression`,[14] a `1`. This node is then replaced with a placeholder node, as shown in Figure 2.

As the fuzzer traverses the tree, it also picks up values that it thinks are interesting, which are usually primitive values such as strings and numbers. These values are harvested and used to construct the final values of the placeholder nodes.

In this example, the placeholder is replaced with another `ObjectExpression` containing the key and value that it harvested from elsewhere in the corpus, as shown in Figure 3.

When this tree is converted into code, it becomes a new test case:

```
db.coll.find(x:{$regex:'ab'}}
```

This replaces the original test case of finding a document whose field `x` has value `1` with a new test case that finds documents matching the regular expression `a\0b`.

A test very much like this one was actually run by our fuzzer, and it turned out that MongoDB did not properly handle regular expressions strings containing null bytes, so this test case caused the server to crash.[11]

Lessons From Experience

AST > Regular expressions

Using an abstract syntax tree is a great strategy for fuzz testing. Previously, we had tried fuzzing using a regex-based approach. This involved stringifying the tests and finding specific tokens to replace or shuffle. Maintaining those regexes became a nightmare after a while, and it's very easy to introduce subtle mistakes that cause the mutations to become less effective. Syntax trees, on the other hand, are designed to represent all the information you need to know about the code, which is a superset of what can be deduced from using regexes. Additionally, ASTs are very difficult to get wrong: all the fuzzer is doing is manipulating properties in an object.

Open source libraries that turn code into ASTs for most languages are available; we used acorn.js.[1]

Heuristic > Random

When implementing the mutational aspect of a fuzzer, noting which types of mutations are provoking the most bugs can yield benefits. The initial implementation randomly chose which nodes to replace, but modified `ObjectExpressions` contributed to finding more new bugs, so we tweaked the probabilities to make more mutations happen on ObjectExpressions.

Dumb Fuzzing

Smart, AST-based mutation gives the MongoDB fuzzer a familiarity with the input format, but it also guarantees blind spots, because the corpus is a finite list harvested from human-written tests. The school of dumb fuzzing proposes an answer to this shortcoming, advocating fuzzers that generate input randomly, without regard to validity, thereby covering areas the developer may have overlooked.

This is a bit of a balancing act. With no knowledge of the target program at all, the best a fuzzer could do would be to feed in a random stream of 0s and 1s. That would generally do nothing but trigger input validation code at some intervening layer before reaching the program under test. Triggering only input validation code is the hallmark of a bad fuzzer.

To put some dumb in our fuzzer without resorting to random binary, values are generated from a seed list. Since our test inputs are JavaScript objects consisting of MongoDB commands and primitive values, the seed list is composed of MongoDB commands and primitive types that we know from experience are edge cases. These seed values are kept in a file, and JavaScript objects are generated using them as the keys and values. Here's an excerpt:

```
var defaultTokens =
{ primitives: ['Infinity',
'-Infinity', 'NaN', '-NaN',
'ab','AB','000','000000'],
commands: ['all',
'bitsAllClear'] // etc. }
```

These values are drawn from our experience with testing MongoDB, but as far as the fuzzer is concerned they are just nodes of an AST, and it composes the test input from them without regard to what would be valid. Thus, our generational method produces dumbness.

It Doesn't Work Like This

We are trying to balance coverage with validation avoidance. To generate test input that has a chance of passing input validation, we could start with a template of a valid JavaScript object. The letters in this template represent placeholders:

```
{a:X, b:Y, c:Z}
```

We could then replace the capital letters with seed primitive values:

```
{a: 4294967296, b: 'ab', c:
NumberDecimal(-NaN)}
```

and replace the lowercase letters with seed MongoDB command parameters:

```
{create: 4294967296,
$add: 'ab', $max:
NumberDecimal(-NaN)}
```

This is not a valid MongoDB command, however. Even filling in a well-formatted template from a list of valid MongoDB primitives, this generated input still triggers only the validation code.

Hybrid Fuzzing

Mutational fuzzing leaves blind spots, and generational fuzzing on its own won't test interesting logic at all. When combined, however, both techniques become much more powerful. This is how our fuzzer actually works.

As it mutates existing tests, every once in a while, instead of pulling a replacement from the corpus, it generates an AST node from its list of seeds. This generational substitution reduces blind spots by producing a value not present in the corpus, while the mutational basis means the resulting command retains the structure of valid input, making it likely to pass validation. Only after it is deep in the stack does the program realize that something has gone horribly wrong. Mission accomplished.

Here is an example of hybrid fuzzing in action, using a simplified version of a test that actually exposed a bug. The fuzzer starts with the following corpus, the first line of which becomes the AST shown in Figure 4:

```
db.test.insert({x:1});
db.test.update({some:
"object"}, ...);
```

The `ObjectExpression` is converted into a placeholder node, in the same manner as mutational fuzzing.

Then the fuzzer decides that instead of replacing the placeholder node with a value from elsewhere in the corpus, it will replace it with a generated object—in this case, a `newExpression` with a large `NumberLong` as the argument, shown in Figure 5.

This yields the following test:

```
db.test.insert({a:
    new
    Number-
    Long("9223372036854775808")});
db.test.update({}, {$inc: {a: 13.0}});
```

The result is that a large 64-bit integer is inserted into MongoDB, and then its value is updated. When the ac-

Figure 4. AST of one command in the corpus. (checking on this)

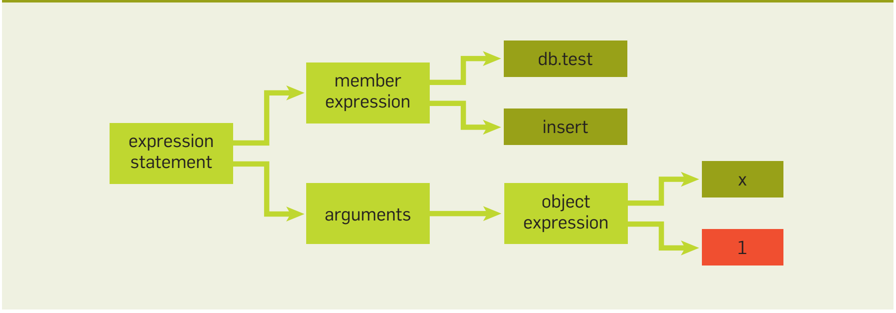
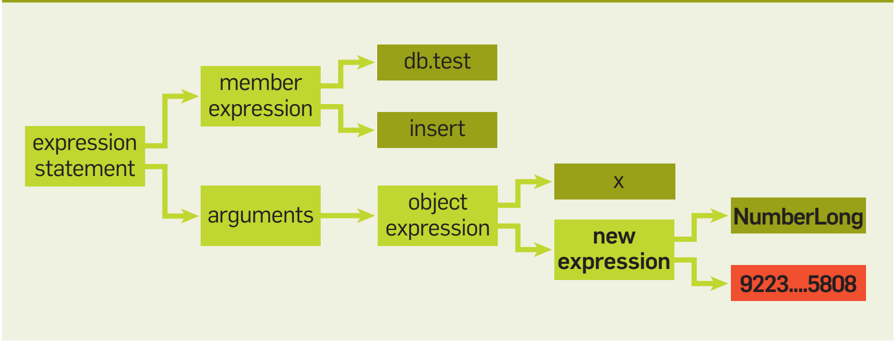


Figure 5. Placeholder replaced with a generated object.



tual test ran, it turned out that the new value would still be a large number, but not the correct one. The bug was that MongoDB stored the integer as a double internally, which has only 53 bits of precision.[13] The fuzzer was able to find this by generating the large `NumberLong`, which did not appear in any test.

The combination of mutational fuzzing with the edge cases we seed to the generational fuzzer is an order of magnitude more powerful than writing tests for these edge cases explicitly. In fact, a significant portion of the bugs the fuzzer found were triggered by values generated in this way.

An Unbridled Fuzzer Creates Too Much Noise

Ultimately, fuzz testing is a game of random numbers. Random numbers make the fuzzer powerful but can cause unforeseen problems. We needed to take some steps to ensure the fuzzer does not blow itself up. Take the following block of code, which resembles something that would be present in one of MongoDB's JavaScript tests:

```
while(coll.count() < 654321)
    assert(coll.update({a:1},
    {$set: {...}}))
```

This code does a large number of updates to a document stored in MongoDB. If we were to put it through the mutational and generational fuzzing steps described previously, the fuzzer could produce this possible test case:

```
while(true)    assert(coll.up-
date({}, {$set: {"a.654321" : 1}}))
```

The new code now tests something completely different. It tries to set the 654321st element in an array stored in all documents in some MongoDB collection.

This is an interesting test case. Using the `$set` operator with such a large array may not be something we thought of testing explicitly and could trigger a bug (in fact, it does).[12] But the interaction between the fuzzed true condition and the residual while loop is going to hang the test!—unless, that is, the assert call in the while loop fails, which could happen if the line defining `coll` in the original test (not shown here) is mutated or deleted by the fuzzer, leaving `coll` undefined. If the assert call failed, it would be caught by the Mongo shell and cause it to terminate.

Neither the hang nor the assertion failure, however, are caused by bugs in MongoDB. They are just by-products of a randomly generated test case, and they represent two classes of noise that must be filtered out of fuzz testing: branch logic and assertion failures.

Branch logic. To guard against accidental hangs, our fuzzer simply takes out all branching logic via AST manipulation. In addition to while loops, we remove `try/catch`, `break`, and `continue` statements, `do/while`, `for`, `for/in`, and `for/of` loops. These language structures are defined in a static list.

Assertion failures. For the assertion failures, every single line of generated test code is wrapped with a try/catch statement. All the logic will still be executed, but no client-side errors will propagate up and cause a failure.

After passing through this sanitizing phase, our earlier example now looks like this:

```
try {
    assert(coll.update({},
    {$set:  {"a.654321"  :  1}}))
} catch  {}
```

So How *Does* the Fuzzer Catch Bugs?

Wrapping everything in a try/catch block keeps fuzzer-generated noise from overwhelming us with false positives, but it also prevents any bugs from surfacing through the client-side assertions our typical tests rely on. Indeed, a fuzzer has to rely on other mechanisms to detect the errors it provokes.

Tools for generic errors. The first set of tools are ones we are using anyway, for finding segmentation faults, memory leaks, and undefined behavior. Even without a fuzz tester, we would still be using these language runtime tools,[3] such as LLVM's address sanitizer[4] and undefined behavior sanitizer,[5] but they become far more useful when a fuzzer is bombarding the test target with all its random input.

These tools are good for generic coding errors, but they don't validate that a program is behaving as expected by end users. To catch issues with business logic, our fuzzer relies on assertions within the testing target that check for conditions it should not be in.

Assertions within the system under test. Many applications make liberal use of asserts to guard against illegal conditions, but fuzz testing *relies* on them to catch application logic errors. It wreaks havoc in your codebase and assumes you have instrumented your application's components such that havoc is noticed.

For example, when acting as a secondary in a MongoDB replica set, mongod has an assertion to halt if it fails to write an operation.[9] If a primary node logs a write for its secondaries, they had *better* be able to perform the write as well, or we will wind up with serious data loss when failovers happen. Since these assertions are fatal errors, the testing framework immediately notices when fuzz tests trigger them.

The limitation of randomized testing. This is really the only way that assertions can be used to catch errors provoked by randomly generated tests. Assertions in the target program can be oblivious to the tests being run; indeed, they must hold true under all circumstances (including when the program is being run by a user). In contrast, assertions within tests must be specific to the test scenario. We have already shown, however, that fuzzer-generated tests, by their nature, must not include fatal assertions. So under truly random conditions, a fuzzer will trigger *no tailored assertions*. This is a limitation of all randomized testing techniques, and it is why any good testing framework must not rely solely on randomized testing.

Triaging a Fuzzer Failure

Tests that perform random code execution and rely on target system assertions have some downsides: the problems they find have no predefined purpose; many of the operations within them might be innocuous noise; and the errors they produce are often convoluted. Failures observed at a particular line of the test might rely on a state set up by previous operations, so parts of the codebase that may be unrelated have to be examined and understood.

Thus, fuzzer failures require triage to find the smallest set of operations that trigger the problem. This can take significant human intervention, as with the known issue[17] where calling `cursor.explain()`[6] with concurrent clients causes a segmentation fault. The test that provoked this issue used a dozen clients performing different operations concurrently, so beside understanding which state the operations in the test set up, log messages from all the client and server threads had to be inspected manually and correlated with each other.

All this work is typical of triaging a fuzzer test failure, so we built a set of features that help developers sift through the chaos. These are specific to testing a MongoDB cluster across the network using JavaScript but can be used as inspiration for all fuzzing projects.

We are only interested in the lines of code that send commands to a MongoDB server, so the first step is to isolate those. Using our trusty AST manipulator, we add a print statement after every line of fuzzer code to record the time it takes to run. Lines that take a nontrivial amount of time to run typically run a command and communicate with the mongodb server. With those timers in place, our fuzz tests look like this:

```
var $startTime = Date.now();
try {
    // a fuzzer generated
    line of code
} catch (e) {
}
var $endTime = Date.now();
print('Top-level statement 0
completed in',
    $endTime - $startTime,
    'ms');

var $startTime = Date.now();
try {
    // a fuzzer generated
    line of code
} catch (e) {
}
var $endTime = Date.now();
print('Top-level statement 1
completed in',
    $endTime - $startTime,
    'ms');

// etc.
```

When we get a failure, we find the last statement that completed successfully from the log messages, and the next actual command that runs is where the triage begins.

This technique would be sufficient for identifying the trivial bugs that can cause the server to crash with one or two lines of test code. More complicated bugs require programmatic assistance to find exactly which lines of test code are causing the problem. We bisect our way toward that with a breadth-first binary search over each fuzzer-generated file. Our script recursively generates new tests containing each half of the failed code until any further removal no longer causes the test to fail.

The binary search script is not a cure-all, though. Some bugs do not reproduce consistently, or cause hangs, and require a different set of tools. The particular tools will depend entirely on your product, but one simple way to identify hangs is to use a timer. We record the runtime of a test suite, and if it takes an order of magnitude longer than the average runtime, we assume it has hung, attach a debugger, and generate a core dump.

Through the use of timers, print statements, and binary search script, we are able to triage the majority of our failures quickly and correctly. There is no panacea for debugging—every problem is new, and most require a bit of trial and error to get right. We are continuously investing in this area to speed up and simplify failure isolation.

Running the Fuzzer in the CI System

Fuzz testing is traditionally done in dedicated clusters that run periodically on select commits, but we decided to include it as a test suite in our CI framework, Evergreen. This saved us the effort of building out a new automated testing environment and saved us from dedicating resources to determine in which commit the bug was introduced.

When a fuzzer is invoked periodically, finding the offending commit requires using a tool such as git-bisect.[2] With our approach of a mutational fuzzer that runs in a CI framework, we always include newly committed tests in the corpus. Every time the fuzzer runs, we pick 150 sets of a few dozen files from the corpus at random and run each one through the fuzzer to generate 150 fuzzed files. Each set of corpus files always includes new logic added to the codebase, which means the fuzzed tests are likely testing new code as well. This is a simple and elegant way for the fuzzer to "understand" changes to the codebase without the need for significant work to parse source files or read code coverage data.

When a fuzz test causes a failure, the downstream effect is the same as any other kind of test failure, only with the extra requirement of triage.

The Fuzzer: Your Best Friend

Overall, the fuzzer has turned out to be one of the most rewarding tools in the MongoDB test infrastructure. Building off our existing suite of JavaScript tests, we were able to increase our coverage significantly with relatively little effort. Getting everything right takes time, but to get a basic barebones system started, all you need is a set of existing tests as the corpus, a syntax-tree parsing for the language of your choice, and a way to add the framework to a CI system. The bottom line is that no matter how much effort is put into testing a feature, there will inevitably be that one edge case that was not handled. In those face-palm moments, the fuzzer is there for you.

References

1. Acorn; https://github.com/ternjs/acorn.
2. Chacon, S., Straub, B. Git-bisect; https://git-scm.com/book/en/v2.
3. Clang 3.8 Documentation. Using Clang as a compiler; http://releases.llvm.org/3.8.0/tools/clang/docs/index.html#using-clang-as-a-compiler.
4. Clang 3.8 Documentation. AddressSanitizer; http://releases.llvm.org/3.8.0/tools/clang/docs/AddressSanitizer.html.
5. Clang 3.8 Documentation. UndefinedBehaviorSanitizer; http://releases.llvm.org/3.8.0/tools/clang/docs/UndefinedBehaviorSanitizer.html.
6. Cursor.explain(). MongoDB Documentation; https://docs.mongodb.com/manual/reference/method/cursor.explain/.
7. Déjà vu Security. Generation fuzzing. Peach Fuzzer, 2014; http://community.peachfuzzer.com/GenerationMutationFuzzing.html.
8. Erf, K. Evergreen continuous integration: Why we reinvented the wheel. MongoDB Engineering Journal 2016; https://engineering.mongodb.com/post/evergreen-continuous-integration-why-we-reinvented-the-wheel/.
9. GitHub. MongoDB; https://github.com/mongodb/mongo/blob/f5c9d27ca6f0f4e1e2673c64b84b628ac29493ec/src/mongo/db/repl/sync_tail.cpp#L1042.
10. Godefroid, P., Levin, M.Y., Molnar, D. SAGE: Whitebox fuzzing for security testing. *Commun. ACM 55*, 3 (Mar. 2012, 40-44; http://courses.cs.washington.edu/courses/cse484/14au/reading/sage-cacm-2012.pdf.
11. Guo, R. Mongos segfault when invoking .explain() on certain operations. MongoDB, 2016; https://jira.mongodb.org/browse/SERVER-22767.
12. Guo, R. $push to a large array fasserts on secondaries. MongoDB, 2016; https://jira.mongodb.org/browse/SERVER-22635.
13. Kamsky, A. Update considers a change in numerical type to be a noop. MongoDB, 2016; https://jira.mongodb.org/browse/SERVER-16801.
14. McCloskey, B., et al. Parser API. Mozilla Developer Network, 2015; https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API#Expressions.
15. Nossum, V., Casasnovas, Q. Filesystem fuzzing with American Fuzzy Lop. Oracle Linux and VM Development—Ksplice Team, 2016; https://events.linuxfoundation.org/sites/events/files/slides/AFL filesystem fuzzing, Vault 2016_0.pdf.
16. Ruderman, J. Introducing jsfunfuzz. Indistinguishable from Jesse, 2007; https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.
17. Siu, I. Explain("executionStats") can attempt to access a collection after it has been dropped. MongoDB, 2016; https://jira.mongodb.org/browse/SERVER-24755.
18. Storch, D. MongoDB, jstests. GitHub, 2016; https://github.com/mongodb/mongo/tree/r3.3.12/jstests.

Robert Guo is a software engineer on the MongoDB server team, focusing on data consistency and correctness. He is currently working on MongoDB's JavaScript fuzzer.