**School of Computer Science and Engineering**

**Faculty of Engineering**

**The University of New South Wales**

# Property Based Testing of C Code from Haskell

by

Alexander Hodges

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Computer Engineering

Submitted: April 2019

Supervisor: Christine Rizkallah

Student ID: z5117819

# Abstract

Property based testing uses randomised test data generated from a property to test code, which is useful for finding bugs in a given function. C is used in many popular libraries and also for the development of low level performance critical applications. However as C is an unsafe language, it is prone to hard crashes. When a program hard crashes the testing framework is unable to find the minimal failing case and so hinders the diagnosis of the failure. This project links C with property based testing in a manner in which it is easy to specify the properties and determine the input which caused the program to crash, while being able to use arbitrary C structures and functions.

# Contents

**1 Introduction** **1**

1.1 Why we like C . . . . . . . . . . 2

1.2 Trying C Based Property Testing . . . . . . . . . . 3

1.3 Summary . . . . . . . . . . 6

**2 Background** **7**

2.1 Literature Review . . . . . . . . . . 7

2.1.1 Why not unit tests? . . . . . . . . . . 7

2.1.2 QuickCheck . . . . . . . . . . 8

2.1.3 Quick Fuzz . . . . . . . . . . 10

2.1.4 Theft . . . . . . . . . . 11

2.1.5 The GHC Runtime System . . . . . . . . . . 14

2.2 Conclusion . . . . . . . . . . 14

**3 Solution and Results** **15**

3.0.1 Initial Fork Investigation . . . . . . . . . . 15

3.1 Initial Implementation . . . . . . . . . . 15

3.1.1 Benchmarks . . . . . . . . . . 18

3.1.2 Result Analysis . . . . . . . . . . 21

3.2 New Design . . . . . . . . . . 21

3.2.1 QuickCheck State . . . . . 22

3.2.2 Successful Case . . . . . 23

3.2.3 Crash Case . . . . . 25

3.3 Results . . . . . 26

3.3.1 Linux Desktop Results . . . . . 26

3.3.2 MacOS Results . . . . . 27

3.3.3 Analysis . . . . . 27

3.4 Limitations . . . . . 27

3.4.1 Priority Queue Example . . . . . 27

3.4.2 Multithreading . . . . . 29

3.5 Future Work . . . . . 29

3.5.1 Deep Integration with QuickCheck . . . . . 29

**4 Conclusion** **30**

**Bibliography** **31**

# Chapter 1

# Introduction

When developers write programs they want to be sure that the program does what they are expecting. One way a developer can convince themselves that a program is correct is through testing the program. Testing works by either manually or automatically specifying inputs to the piece of code the programmer wants to test and verifying that the code produces the correct result. One way of generating this test data is through property based testing. With this form of testing the programmer writes a set of properties for each function they want to test. The testing framework will then automatically generate random data and ensure that the function satisfies the specified properties. If the testing framework finds a failing case then the framework will try to shrink the input to find the smallest failing case. Shrinking is done by breaking down the generated input into smaller parts and running the function against this smaller data and seeing if the program still fails the property.

For an example of property based testing, suppose we have written a `merge` function which given two lists of integers called `xs` and `ys` should combine then in such a way that the returned list is also sorted. We also need a `sort` function to specify our property. The example we will look at is written in Haskell [M+10] and uses the QuickCheck [CH00] framework to specify the property. Firstly we define the type signatures of the `merge` and `sort` functions:

```
merge :: [Int] -> [Int] -> [Int]
sort :: [Int] -> [Int]
```

The first line states that the `merge` function must be given two lists of integers and will return another list of integers. The second line states that the `sort` function should be given a list of integers and should return another list of integers. With these two functions we can now define our property:

```
prop_merge_sort :: [Int] -> [Int] -> Bool
prop_merge_sort xs ys =
    (sort (xs ++ ys) == merge (sort xs) (sort ys))
```

The property states that appending the two lists `xs` and `ys` followed by sorting then should produce the same list as sorting each list individually and then running our `merge` function on these two now sorted lists. This property also has the advantage of specifying the complete functional correctness of the `merge` function, meaning that any program which passes this property is totally correct.

## 1.1 Why we like C

From the example above we have seen that we can property test Haskell code, however there are many programs which are written in languages other than Haskell which also need to be tested. C is an imperative language which has been used to write many popular libraries; it is also useful for writing low level performant code. However, as C is a low level language, memory management is up to the programmer [HB03]. Managing memory manually along with the unsafe nature of C means that any one of the bugs in a C program could result in a segmentation fault. A segmentation fault occurs when the operating system (OS) detects the program is trying to access memory which it shouldn't; this usually results in the program being terminated. For this reason it is beneficial to test C code before it is run in production.

## 1.2 Trying C Based Property Testing

To test out how property based testing in C works, we will look at a custom 'fast array' library in C that implements the `my_reverse` function. The goal is to write a property for this function which tests to ensure that the `my_reverse` acts like a normal reverse function.

As this program is written in C we will start by using a C based property testing framework called Theft [Sco17]. We will look at Theft in greater depth in Section 2.1.4. Below is a Haskell version of the property which would be used to test a Haskell version of the function followed by the Theft property to test the C code.

```
prop_reverse_same :: [Int] -> Property
prop_reverse_same xs = my_reverse xs == reverse xs
```

Figure 1.1: Pure Haskell Property

Figure 1.1 states that giving the custom `my_reverse` function a list of integers should return the same list as the one returned by the built in `reverse` function.

```
  static enum theft_trial_res prop_reverse(struct theft *t,
                                            void *arg1) {
    // Extract array from argument
    int* orig = (int*) arg1;

    // Allocate the memory for the copy of the array
    int* copy = malloc(sizeof(int) * ARRAY_SIZE);

    // Copy the array
    for(int i = 0; i < ARRAY_SIZE; i++) {
      copy[i] = orig[i];
    }

    // Perform the in place reverse
    my_reverse(copy, ARRAY_SIZE);

    // Verify it has been reversed successfully
    for(int i = 0; i < ARRAY_SIZE; i++) {
        if(orig[i] != copy[ARRAY_SIZE - i - 1]) {
            return THEFT_TRIAL_FAIL;
        }
    }
    return THEFT_TRIAL_PASS;
}
```

Figure 1.2: Theft Reverse Property

Figure 1.2 shows a Theft property for testing the `my_reverse` function. The C function in Figure 1.2 firstly creates a copy of the input array. The copy needs to be done due to the `my_reverse` function performing an in place reverse of the list. The `my_reverse` function is called on the copy of the array and then compared to the original input. The comparison is done in the second `for` loop and ensures that the copy is a revered version of the original input. To reduce the complexity of the code a static compile time constant `ARRAY_SIZE` was used to specify the size of the array. The static array size limitation is not present in the Haskell version of the code.

This function, however, is just the property. Theft, as we will see in more detail in Section 2.1.4, requires both a function to allocate the list data structure (including setting the random values) as well as code to tell Theft how to run the actual test. As we noted in Section 1.1 C is an unsafe language. When writing the Theft testing code the lack of safety in C can result in the testing code itself crashing and needing to be

debugged. Compared to the pure Haskell property the Theft code is more verbose and more difficult to get correct.

Ideally, we want a solution like the pure Haskell property. To do this, Haskell can be linked to C via a foreign function interface (FFI) [Jon01]. By using the FFI to make C functions available from Haskell code we can then use QuickCheck to test these functions. Below is the result of combining QuickCheck with the FFI:

```
foreign import ccall unsafe "fastlist.h my_reverse"
    my_reverse :: Ptr CInt -> CInt -> IO ()

prop_reverse_same :: [CInt] -> Property
prop_reverse_same xs = ioProperty $ do
  p <- newArray xs
  my_reverse p (fromIntegral $ length xs)
  xs' <- peekArray (length xs) p
  pure (xs' == reverse xs)
```

Figure 1.3: C FFI Based Property

Figure 1.3 is more complicated than the pure Haskell implementation, due to the C code not having a guarantee of purity as well as having to pass Haskell data structures to the C code. The lack of purity causes the resulting Haskell type of the function to be wrapped in the IO monad. The type signature for the C function can be seen on the first line with the call to `foreign import ccall unsafe`. The `ioProperty` function is needed to be able to do IO operations inside a property. Finally the `newArray` and `peekArray` functions allow the programmer to create and read a C compatible array from and into a Haskell style list respectively.

This unlike the Theft example is the entire code needed to test the `my_reverse` function, however by running the `prop_reverse_same` with QuickCheck it immediately finds a case which causes the function to access an invalid memory location. The OS then sends the segmentation violation (SIGSEGV) signal, terminating the process. The termination results in the following message:

```
*TestList> quickCheck prop_reverse_same
cabal: repl failed for FastList-0.1.0.0. The build process segfaulted (i.e.
SIGSEGV).
```

This error message is not very helpful as it fails tell us what input caused the program to crash. To be able to identify the cause of the bug, the testing framework must provide the case that caused the program to crash. Without the failing test case, we would not be able to reproduce the bug and so it would remain present in the program until the programmer is able to determine how to reproduce it. As we will see in Section 2.1.4, Theft is able to work around this problem by using the `fork` system call.

## 1.3 Summary

Property based testing is one of the ways a developer can gain more confidence in their program. We have seen in the above example that writing these properties in the Haskell language is much simpler than using a C based approach, and we were able to link Haskell to C via the FFI. However, the problem of dealing with unexpected crashes needs some further insight, as we need to determine how to extract the failing case. Moving forward we now have the following goal to keep in mind as we look at other projects and our design:

*To easily write properties for C code which are tested automatically and to provide helpful information on failure.*

# Chapter 2

# Background

In this chapter we will investigate various tools and methodologies around the property based testing area and testing in general.

## 2.1 Literature Review

### 2.1.1 Why not unit tests?

Unit tests are another form of testing that a programmer can use to ensure the program behaves as expected [CL02]. They work by specifying the inputs to the program and the expected output. A simple example of a unit test would be testing the reverse of the list `[1,2,3]` is `[3,2,1]` an example using the `my_reverse` from Section 1.2 is shown below:

```
test_one :: [Int] -> Bool
test_one xs = [3,2,1] == (my_reverse [1,2,3])
```

The unit testing framework will then run all the tests and report on any which didn't produce the expected output. There are some disadvantages to this method, however. Firstly when testing multiple functions that interact, the developer has to write a

polynomial number of tests [Hug16] to ensure that each interaction of the function is tested. An example of this would be if a calculator applications was being developed the programmer would need to write tests for both the addition and multiplication parts. The programmer would also need to write tests which combine the addition and multiplication operations in various ways to ensure that they still behave as expected.

Another downside to unit testing can be seen when there is a change to how the function works, the unit tests will need to be reviewed and potentially re-written to conform to the new functionality. Returning to the calculator example, if there was a change in how the addition function worked the developer would need to not only change the addition tests but also the tests which combine addition with multiplication. By using properties a change in the functionality would only need to change the related properties and the testing framework will generate the new tests.

Finally, as the developer needs to write each test individually, there is a chance that the developer misses an edge case that they did not consider. For the `my_reverse` example it could be that an input list size of 100 could break by allocating memory incorrectly. A property based testing framework is able to generate hundreds of test cases for each run, which reduces the chance a subtle bug like this would be able to slip through.

### 2.1.2 QuickCheck

QuickCheck is a property based testing framework originally written for Haskell [CH00], the QuickCheck program has now been ported to many more languages [Hug16]. Haskell, as a purely functional language is well suited to property based testing as an input to the function will always produce the same output [CH00].

To support random data generation, the QuickCheck tool must be able to generate an arbitrary value for the input of the function. In Haskell this is done using the `Arbitrary` type class. For a type to be instance of the type class, the programmer must specify how to randomly generate a value for that type. QuickCheck comes with instances of these type classes for a large selection of the built in Haskell data types

which means that for functions which use the built in types, only the property itself needs to be specified and QuickCheck will handle the rest.

After specifying how to generate the test data (if needed), the next step is to actually specify the properties. As seen in the introduction, these properties can be quite short while still stating a large amount of information.

```
my_prop :: [Int] -> [Int] -> Bool
my_prop xs ys = (sort (xs ++ ys) == merge (sort xs) (sort ys))
```

Taking a deeper look into this property, the first line is the type signature which says this property is a function which takes two lists (with elements that can be ordered) and returns a property. As the example only uses lists, there is no need to specify how to generate the data. QuickCheck will use the value generator to produce a series of lists which it will then pass to the property to check if it holds. The left hand side of the `==` says to combine the two lists together and then sort the result. The right hand side states that each individual list should be sorted before being passed to the `merge` function; once sorted they are passed to the function and the resulting list is checked to ensure it equals the left hand side.

Now that the property has been specified, it can now be tested using the QuickCheck library. When the `quickCheck` function from QuickCheck is invoked it will use the provided data generation method to produce test data; it will then feed that into the property and check to see that it holds. If the property doesn't hold then QuickCheck will try to determine what about the specific part of the input caused the failure; this is done through a process called shrinking. The shrinking process gets the original input that caused the failure case and takes a sub-part of it. The sub-part is then tested against the property and if it fails again QuickCheck will shrink the sub-part. The shrinking process finishes when no sub-parts cause the property to not hold. When using the included types in Haskell the shrinking process is automatic. Once the smallest input has been found it will report it to the user along with whichever property failed.

To see shrinking in action suppose we have a wrongly implemented `merge` function:

```
merge xs ys = (sort xs) ++ ys
```

Running the `my_prop` through QuickCheck quickly results in it finding a failing case and shrinking it. Both cases can be seen below:

```
Failed:
[-1,2,0,-5,-4]
[4,2,-2]
```

(a) Failing Case

```
*** Failed! Falsifiable (after 6 tests and 6 shrinks):
[0]
[-1]
```

(b) Shrunk Case

**Summary**

Using the QuickCheck library along with Haskell allows for the simple specification of properties and shrinking of failing cases. However, as seen in the introduction it can not detect when a segmentation fault occurs due to the OS killing the process. There are some potential workarounds to this problem which we will investigate in Section 2.1.3 and Section 2.1.4

### 2.1.3 Quick Fuzz

QuickFuzz is another Haskell based testing framework. Unlike QuickCheck it does not test properties but uses a technique called *fuzzing* which tests a whole program against invalid inputs. QuickFuzz has a particular focus on fuzzing file formats and uses a GIF image file as an example. Of interest is the fact that QuickFuzz is designed to detect crashes. QuickFuzz internally uses the QuickCheck framework for generating the random test data and shrinking any failing cases.

To be able to specify how the file is laid out, the user must provide a data type which represents the structure of the file. This data structure can then be used by QuickCheck to randomly generate values which represent the file type. For the data to be randomly generated by QuickCheck the `Arbitrary` instances need to be defined. To automate this the MegaDeTH tool is used to derive these instances automatically [GCB16].

Once QuickCheck has generated the data it must then be written to a file which can then be read by the program. This requires the user to create an `encode` function which takes an instance of the data structure and creates a file of that type from it. Finally, QuickFuzz can the run the program using the user provided command line. As the program is run in a separate OS process any crashes can be detected by QuickFuzz. If the program crashes QuickFuzz will then use QuickCheck's shrinking functionality to try and find the smallest possible failing case.

**Summary**

QuickFuzz provides a potential solution to being able to detect program crashes, however this method has some limitations. As QuickFuzz tests a complete program this limits the granularity to the whole program. If the user wanted finer grained testing, such as testing an individual function then they would need to use another testing framework. We want our framework to be able to test any function without having to write an entire program around it.

### 2.1.4 Theft

As mentioned in Section 1.2 Theft is a property based testing framework that uses the C language to specify properties. To test programs using Theft the user needs to specify at least a function to allocate the input data and the property that the user wants to test with.

The first function, the allocation function is similar to the `Arbitrary` instance from

QuickCheck. The allocation function must create a random instance of the input data. The Theft library provides the `theft_random_bits` function to generate random bits from the internal Theft seed. This seed needs to be used to ensure that when Theft runs the test again the same failure are observed as well as enabling Theft to shrink the failing input. Unlike QuickCheck the allocation function must also allocate the memory for the input data due to Theft not knowing anything about the structure of the data.

Below we have the allocation function for the list example introduced in Section 1.2 followed by the implementation of the `Arbitrary` instance for the Haskell version.

```
enum theft_alloc_res list_alloc_cb(struct theft *t, void *env,
                                   void **instance) {
    int* ret = malloc(sizeof(int) * ARRAY_SIZE);
    for(int i = 0; i < ARRAY_SIZE; i++) {
        ret[i] = theft_random_bits(t, 32);
    }
    *instance = ret;
    return THEFT_ALLOC_OK;
}
```

```
instance Arbitrary Int where
  arbitrary = do
    max <- getSize
    choose (-max, max)
  shrink num = [0, 1, -1] ++ [-(num-1)..(num-1)]
```

The Haskell implementation requires two `Arbitrary` instances to be able to generate a random list of integers. Once we have defined the instance for integers however, we get the list instance for free as QuickCheck's `Arbitrary` instance for lists works on any value of a list's containing type. We can also see that the Haskell instance includes a shrink function which eases debugging as it allows QuickCheck to find the smallest value which causes the crash. The larger number of helper functions provided by QuickCheck along with us not having to do memory management allow us to write the instance in Haskell more succinctly.

The other function the user needs to specify is the actual property. The property function takes two arguments; the first is a struct which holds the Theft internal state and the second argument is a pointer (`void *`) which points to the randomly generated input data from the `alloc` function. As the input data is passed as a void pointer Theft needs not know anything about the layout of the data structure it is all handled by the functions written by the user. The passing of the void pointer also highlights the issue with using C as the language for writing the properties: the user must deal with C's lack of type safety. If these pointer operations are mishandled it could result in a segmentation fault in the testing code itself, which would require painful debugging.

To run the property tests the developer must also tell Theft how to run it. The configuration exposes the `fork` option which tells Theft to run the test in another OS process. The `fork` system call creates a copy of the process called the child process. If the child process gets killed by the OS such as through a segmentation fault, the parent process can detect this. Once Theft has detected the crash it can then apply shrinking, just like QuickCheck to determine the smallest input which causes the program to crash.

Theft also has other optional functions that can be specified, these include free, hash, shrink and print. The shrink function must be specified to be able to tell Theft how to shrink the input data. The limited type system of C means that if the user wants shrinking support, this function must always be provided. This contrasts to QuickCheck which can automatically shrink data in most cases as QuickCheck provides a variety of `Arbitrary` instances as part of the implementation.

**Summary**

Theft provides a basis for inspiration on how to solve the problem of property based testing of C code. However, as seen in Section 1.2, specifying these properties in C is a very verbose process when compared to Haskell's QuickCheck.

### 2.1.5 The GHC Runtime System

The Haskell language uses a runtime system along with lightweight threads to handle multitasking [MPJS09]. The lightweight threads are transparent to the operating system, so there can be multiple lightweight threads assigned to a single OS thread. While using lightweight threads makes thread creation cheap, allowing for potentially millions of threads, this creates an issue with the `fork` system call. When the `fork` is called from Haskell, only the caller's lightweight thread is copied to the child process, meaning that any shared variables accessed from the child are now invalid [oG02]. This problem isn't faced by Theft due to the C language's lack of runtime, but will need to be considered for our design due to our use of Haskell specifications.

## 2.2 Conclusion

We have seen that property based testing is a helpful aid to finding bugs in programs along with various techniques to be able to detect and diagnose crashes. However, we have not found a solution which completely matches our goal of being able to easily specify properties of a program, and test them in a manner which can detect crashes. The next chapter will outline my progress into developing a solution which meets our goal.

# Chapter 3

# Solution and Results

### 3.0.1 Initial Fork Investigation

Before any implementation could be started the `fork` system call would need to be tested to ensure that it worked with the GHC runtime as we discussed in Section 2.1.5. The `fork` system call wrapper in GHC came with a warning [oG02] concerning the use of shared variables not being passed to the child process. The initial hypothesis was that as QuickCheck wasn't multi-threaded it wouldn't create any separate threads To investigate this I built a simple program which would call `fork` and then run a QuickCheck test in the child process, this was a success and evolved into the first implementation.

## 3.1 Initial Implementation

For the first implementation we will use a simple design which works in a similar way to Theft. This design uses the `fork` system call to create a child process. The child process can then run the unsafe property in a safe environment. If the C code were to cause a crash the parent process will be unaffected as only the child process will be killed.

For this design the tester needs to provide the QuickCheck property written in Haskell and the C code that the Haskell property will call via the FFI. The implementation, called `quickFork` has the following type:

```
quickFork :: Arbitrary a => Testable b => (a -> b) -> a ->
    Property
```

The developer will pass the function `a -> b` which represents the property that they want to test. The second `a` value is generated by QuickCheck and the returned `Property` is the final result of the running the function that was provided. As the `fork` system call is made inside the custom property generated by `quickFork`, the `fork` system call is completely transparent to QuickCheck. By making QuickCheck unaware of the spawning of new child processes we can get the automatic shrinking and data generation for free.

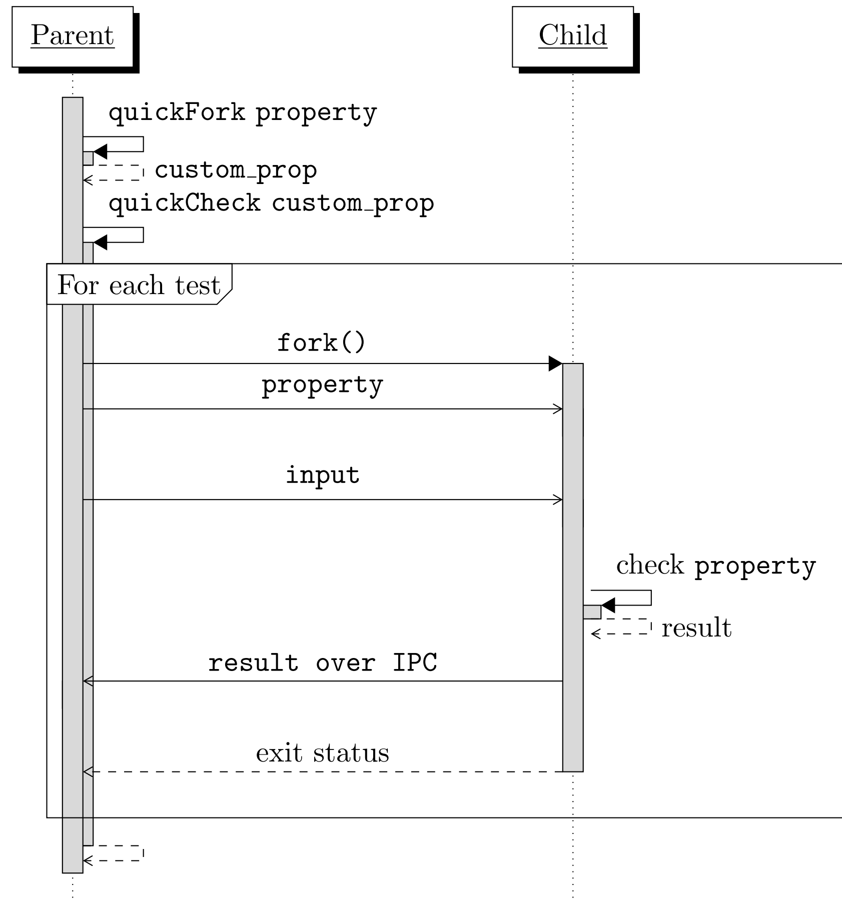



Figure 3.1: Property Verification

Figure 3.1 shows the steps that are taken to safely verify the property. Firstly the original property that the developer provides is passed to the `quickFork` function, this wraps the given property to call the `fork` system call. Next QuickCheck is called, passing in the custom property returned by `quickFork`. QuickCheck will then generate the first test data and pass it to the custom property. The custom property will then call `fork` and test the property against the data QuickCheck generated. If the child doesn't crash it will send a pass or fail message to the parent over a pipe depending on if the property held or not. If the child crashes the parent process detects this by checking the exit code of the child, if it is non 0 the parent will treat that as a failure too. The parent will then return either `True` or `False` as the result of the custom property to QuickCheck. If the custom property fails QuickCheck can begin shrinking

the test data otherwise QuickCheck will generate another test input and run through the process in the *For each test* of Figure 3.1.

### 3.1.1 Benchmarks

To test the performance impact four types tests were ran on both a Linux based desktop machine, which has a Intel 4790k processor along with 16GB of ram and a 2017 13 inch MacBook Pro running MacOS 10.14.6, both machines used GHC 8.6.5. The test revolved around a `my_reverse` function written in C and was compared with `reverse` from the Haskell base package. The reverse C function is shown below, it takes in an array of integers called `l` and the size of the array as `size`:

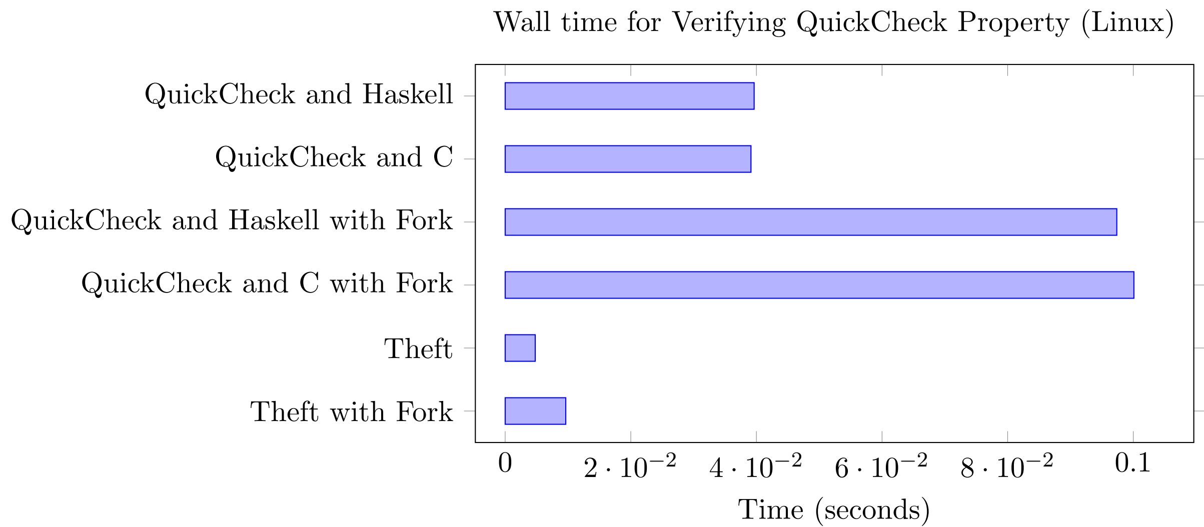
```
void my_reverse(int* l, int size) {
    if(size <= 1)
        return;
    int t;
    int front = 0;
    int back = size - 1;
    while(front < back) {
        t = l[front];
        l[front] = l[back];
        l[back] = t;
        front ++;
        back --;
    }
}
```

I used the same `prop_reverse_same` as we saw in Section 1.2 to verify the correctness of both the Haskell and C `reverse` functions. The four tests involved testing the C and Haskell code both with and without using the `fork` system call for each test.

For comparison of the slowdown the C based code was also tested with the Theft [Sco17] property testing framework. Each variation of the test was run 1000 times with the average of these in the appendix. For the testing using QuickCheck I used the default

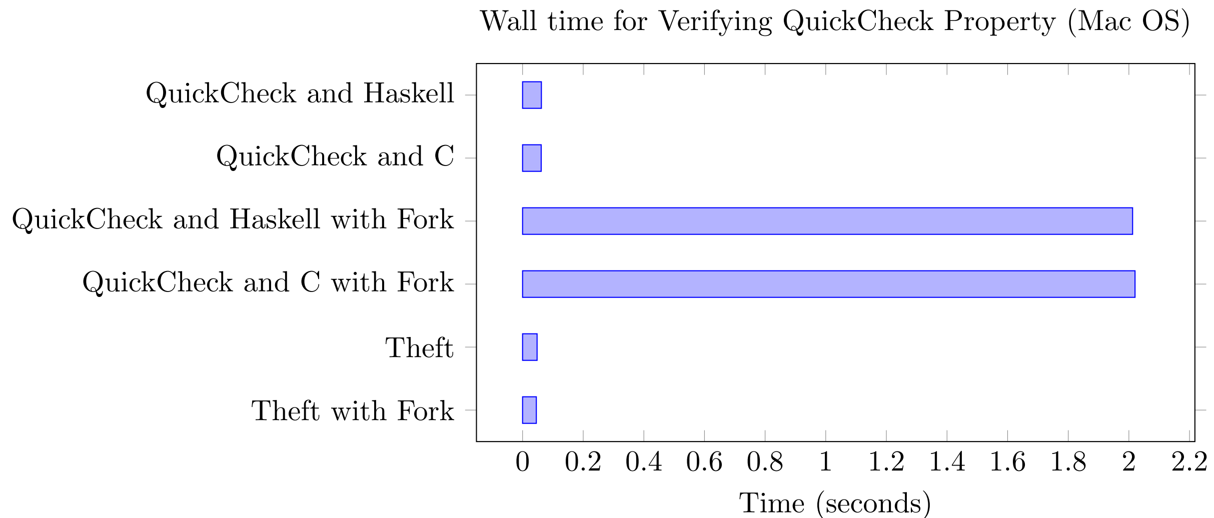
arguments except for printing which was disabled. The graph shows the wall time that the program spent running along with the testing framework and language the function to be tested was written in.

### Linux Desktop Results



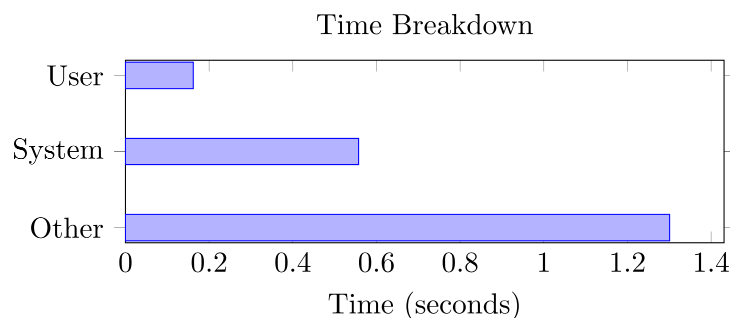

Both QuickCheck results without using the `fork` call performed roughly the same and adding the `fork` system call also made both about 2.5x slower. From this we can deduce that the use of the testing of the C based code over the foreign function interface had little impact relative to the large slowdown caused by calling the `fork` system call. We can also see that both the Theft and QuickCheck based solution experienced a similar magnitude of slowdown when executing the `fork` system call for each test. When I tested on the Mac OS based laptop we got some interesting results.

**Mac OS Results**

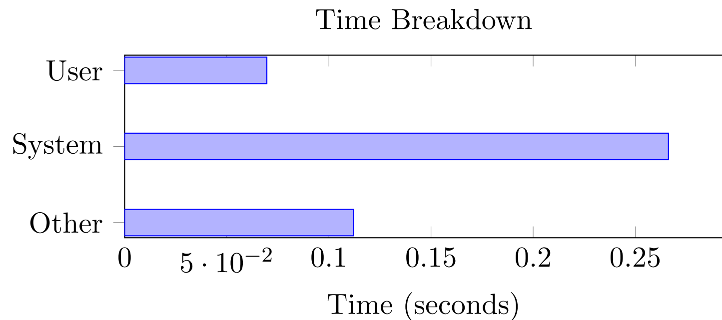



As the Theft implementation didn't have such a large slowdown when testing with the `fork` call enabled, it looked like the combination of Mac OS, GHC and the `fork` system call was causing a very large slowdown. As the feasibility investigation for using `fork` was completed quicker than expected I was able to do some investigation into what was causing this significant slowdown.

To start off I looked at the breakdown the `time` command gives. This breakdown includes the amount of time spent making system calls, the time spent running application code and the total time it took, including any waiting. Below is a breakdown of the time spent in the application code, making system calls and waiting for the QuickCheck testing of the C code with the `fork` system call enabled.



This indicates that the program spends a large proportion of time waiting. By calling `exitImmediately` just before the code handed back to GHC resulted in a significant reduction in the wall time. The `exitImmediately` call runs the C function `exit` by-passing any cleanup the GHC does when the process finishes. The new timings with the `exitImmediately` call are below.




The large reduction in the time taken, especially in the `other` section would indicate that a large portion of the time is spent cleaning up the child processes. Testing this on the Linux desktop didn't help with reducing the speed.

### 3.1.2 Result Analysis

From these results it is clear that the `fork` system call, especially on MacOS has a large overhead with our use case. As a developer might run these tests many times during their development cycle we want these properties to be verified as quickly as possible. So even though we have a working solution it is clear from these results that we need to find a way to reduce the number of `fork` calls `quickFork` makes.

## 3.2 New Design

To reduce the number of `fork` system calls that are made the child process should try and run as many tests as possible with the parent only re-spawning the thread if the child process were to crash. To accomplish this we need to make two changes to the

way `quickFork` runs its tests. Firstly, as the child process is now running more than one test over its lifetime the parent process needs to become aware of what test the child is running in case the child crashes. Secondly, as the child now has to run multiple property tests we need to change the way a developer runs the tests. Instead of using `quickFork` to transform the given property into one which transparently uses the `fork` system call we must now pass the property to the child process and the child will then invoke QuickCheck itself. From the fast list example we saw in Section 1.2 a developer would originally test the property in a safe manner by running:

```
quickCheck (quickFork prop_reverse_same)
```

Here the `quickFork` is transforming the property as described in Section 3.1. Using our new design which calls QuickCheck in the child, this would now simply become:

```
quickFork prop_reverse_same
```

As `quickFork` will now call QuickCheck internally, the developer can now pass their property directly to the `quickFork` function. Consequently the `quickFork` function now has the following type signature:

```
quickFork :: (Show a, Arbitrary a, Testable prop)
                 => (a -> prop) -> IO ()
```

With `a -> prop` being the property the developer wishes to test. This is a function which takes an `a` and will return a QuickCheck compatible property which it uses to determine whether that input has passed or failed. The new implementation will not return anything but rather print the results to standard out like the `quickCheck` function.

### 3.2.1 QuickCheck State

For each test iteration the child process may die, this causes all the information in the child process to be lost. To ensure the parent process knows which input the child is about to run we need to look into how QuickCheck generates its test data.

To generate its test data QuickCheck creates a `QCGen`. The `QCGen` is the initial seed from which QuickCheck generates the sequence of test data to be used as input into the given property. By using the same `QCGen` QuickCheck will generate the same sequence of test data. However this only allows us to determine the starting values, QuickCheck has no external method which allows us to identify where in the sequence QuickCheck is up to.

### 3.2.2 Successful Case

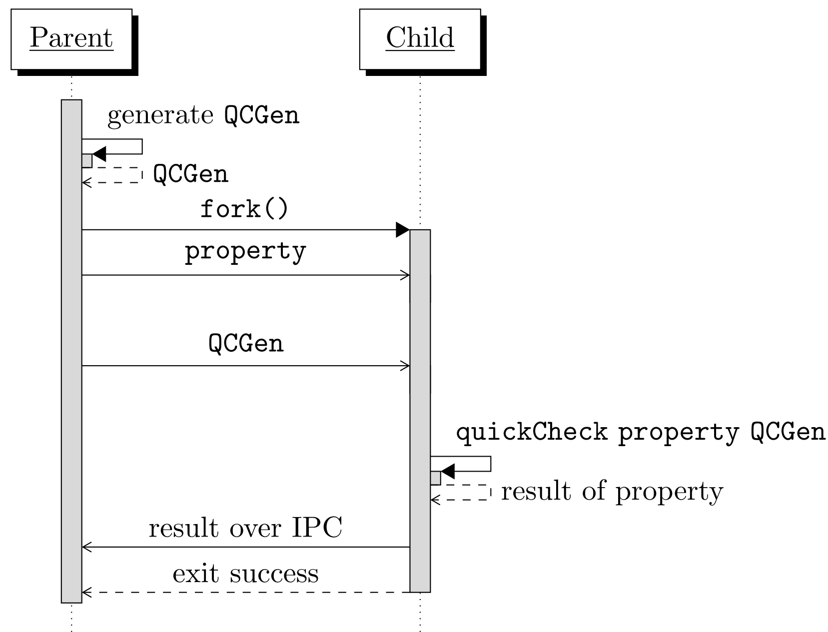



Figure 3.2: Successful property verification

Figure 3.2 shows the sequence of events that occurs when a property is able to be verified without the child process crashing. The first step is to create the seed, called the `QCGen` which as described in Section 3.2.1 is used by QuickCheck to generate the test data. The seed is generated in the parent process so it can be reused during the shrinking phase if the child process crashes.

Next the parent uses the `fork` to create the child process, the parent can then pass the property and the `QCGen` to the child process. The child process with then run

the `quickCheck` function against the given property with the `QCGen` as the starting seed. For this case where no crashes occur the child can send the result of whether the property held or not back to the parent process over IPC. As the `quickCheck` function will verify the property only one `fork` system call needs to be made when the child process doesn't crash. In the next section we will look at what happens when the child does crash.

### 3.2.3 Crash Case

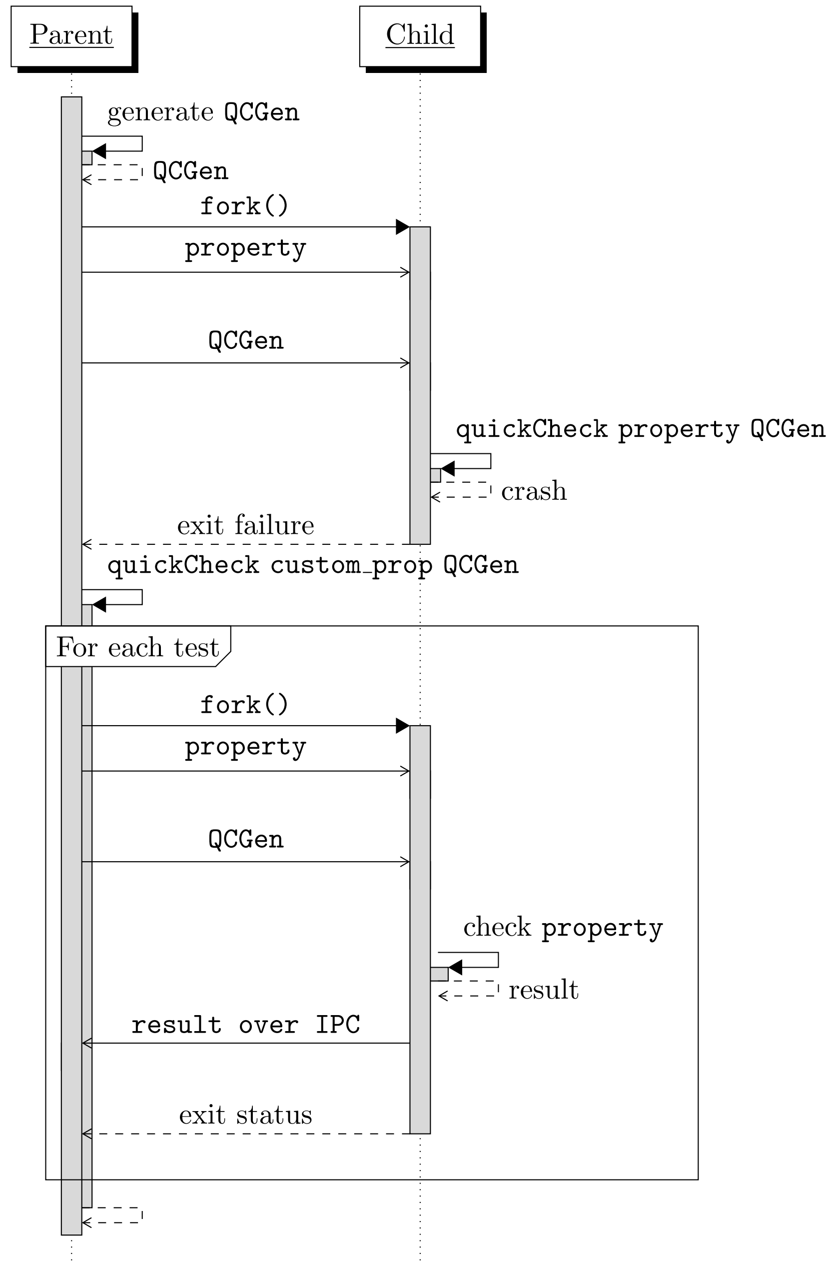



Figure 3.3: Crash property verification

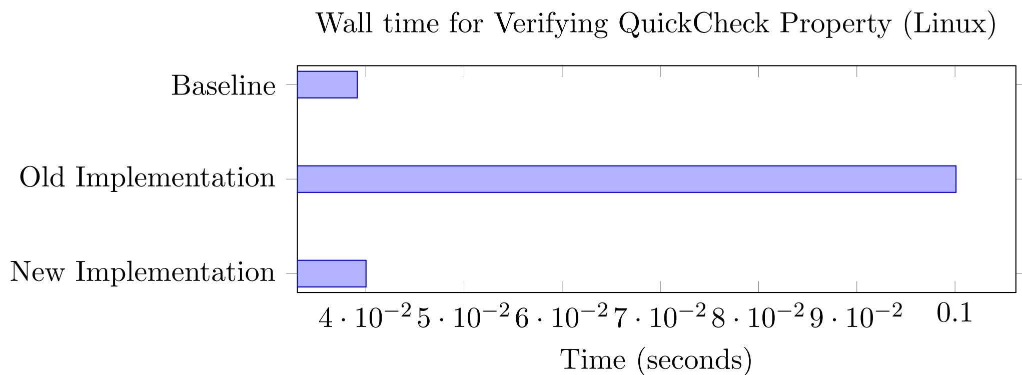
Figure 3.3 shows the flow of what happens when the child process crashes during its verification of the property. When the first child spawned crashes we now have a failing test case, however we want use QuickCheck's shrinking ability to find the smallest failing

input. To find the smallest input `quickFork` will switch over to using the same method discussed in Section 3.1. The key difference is that `quickCheck` is called with the `QCGen` which caused the first crash. This ensures that QuickCheck will generate the same test data which caused the crash.

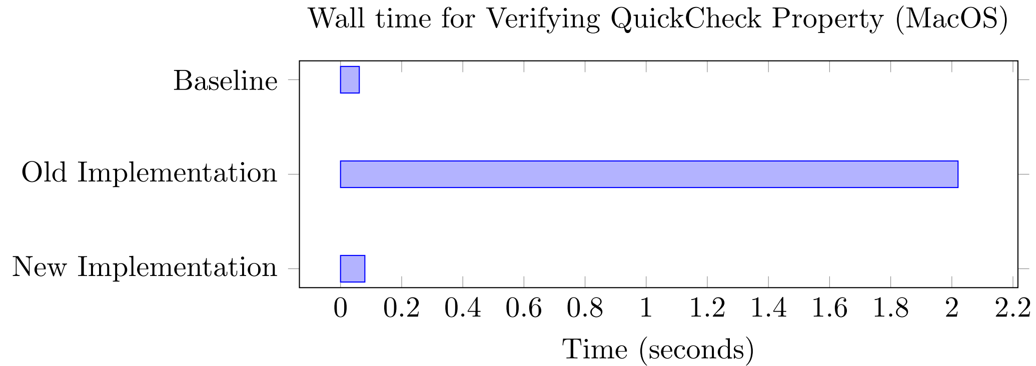
## 3.3 Results

To observe the performance of the new implementation I ran the same property test as described in Section 3.1.1 but now using the new implementation. For reference the old implementation and baseline times are taken from the C + Haskell test with and without using the `fork` system call respectively.

### 3.3.1 Linux Desktop Results



### 3.3.2 MacOS Results




### 3.3.3 Analysis

As this is the happy case, where no crashes occur we can see a significant speedup as we now only need to use the `fork` system call once. For the case where a crash does occur there would be no change in speed as `quickFork` will go back to using the original design. This is acceptable as we only go back to the slower method when we have found a `QCGen` which causes a crash and we want to start the shrinking process. During this shrinking process we are expecting most property checks to result in a crash as we are trying to locate the smallest value to cause the crash. We will look at how this could be improved in Section 3.5.1.

## 3.4 Limitations

### 3.4.1 Priority Queue Example

To test a more complicated example compared to the fast reverse function we saw in Section 1.2 I developed a model based test for a C based priority queue. The priority queue was developed to keep track of registered timers, with the timer which was to be fired next being the next element to be popped off the queue. The model based test works by generating a random set of operations and performing them on the concrete

model and an abstract model. The results of the operations are then compared and any differences are an error.

The main difficulty in implementing the testing code for the priority queue was linking the C to the Haskell code. Lots of boiler plate was needed to ensure that the C structs could be passed back and forth to the Haskell code. Two modifications were needed to be done to the C code. Firstly a function to allocate the priority queue had to be written as the implementation had assumed it was allocated elsewhere. Secondly the calls to the logging library were removed to speed up the integration with the Haskell code.

Another implementation issue was the number of arguments. The property took two arguments, the size of the queue and a list of actions to perform on the queue of that size. An issue occurred when `quickFork` went from its first phase to the shrinking phase. In the shrinking phase two pieces of test input were generated at different times compared to the initial run. Generating the inputs in a differing order produced different results when compared to the first phase. To remedy this the developer has to use the `uncurry` function to turn all arguments into a single tuple, forcing them to be generated at the same time. The command used to test the property `pq_prop` went from:

```
pq_prop :: QueueSize -> [Action] -> Property

quickFork :: (Show a, Arbitrary a, Testable prop)
             => (a -> prop) -> IO ()
quickFork pq_prop
```

to:

```
pq_prop :: QueueSize -> [Action] -> Property

quickFork :: (Show a, Arbitrary a, Testable prop)
             => (a -> prop) -> IO ()
quickFork (uncurry pq_prop)
```

### 3.4.2 Multithreading

As the `fork` call from Haskell only copies the single thread that `fork` is called from care must be taken when dealing with multi-threaded properties. Specifically using multiple threads from within a property is safe along with using threads in the C code which is tested. However if a thread was spawned before the `quickFork` function was called the thread won't be included when the `fork` call is made, so properties won't be able to communicate with the thread.

## 3.5 Future Work

### 3.5.1 Deep Integration with QuickCheck

By integrating the `quickFork` into the QuickCheck library it would allow more potential performance improvements. As `quickFork` would be part of QuickCheck it would allows greater control over the state. Having access to QuickCheck's internals would mean we could save the state before each test. With the state saved before each test we can then run as many tests in the child process, even during the shrinking process until one of the tests cause a crash.

# Chapter 4

# Conclusion

By using a wrapper around QuickCheck called `quickFork` we can safely use property based testing to test potentially crash prone C functions. By going through the Haskell FFI we can avoid the need to test entire programs like Quick Fuzz. By specifying the properties in Haskell we are able write easier to read and more succinct properties as compared to other solutions such as Theft. We have also seen through the Priority Queue Example that through small modifications, more complicated properties can be tested which deal with custom C data types.

# Bibliography


[CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[CL02] Yoonsik Cheon and Gary T Leavens. A simple and practical approach to unit testing: The jml and junit way. In *European Conference on Object-Oriented Programming*, pages 231–255. Springer, 2002.

[GCB16] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. Quickfuzz: An automatic random fuzzer for common file formats. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 13–20, New York, NY, USA, 2016. ACM.

[HB03] Eric Haugh and Matt Bishop. Testing c programs for buffer overflow vulnerabilities. In *NDSS*. Citeseer, 2003.

[Hug16] John Hughes. Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World*, pages 169–186. Springer, 2016.

[Jon01] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction*, 180:47, 2001.

[M+10] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[MPJS09] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[oG02] The University of Glasgow. System.posix.process.bytestring. `https://hackage.haskell.org/package/unix-2.7.2.2/docs/System-Posix-Process-ByteString.html#v:forkProcess`, 2002. Accessed: 2019-04-25.

[Sco17] Scott Vokes. theft: property-based testing for C. `https://github.com/silentbicycle/theft`, 2017. Accessed: 2019-04-22.