QuickFuzz: An Automatic Random Fuzzer for Common File Formats

Gustavo Grieco Martín Ceresa
CIFASIS-CONICET, Argentina
{ gg, ceresa }@cifasis-conicet.gov.ar

Pablo Buiras
Harvard University, United States
pbuiras@seas.harvard.edu

Abstract

Fuzzing is a technique that involves testing programs using invalid or erroneous inputs. Most fuzzers require a set of valid inputs as a starting point, in which mutations are then introduced. QuickFuzz is a fuzzer that leverages QuickCheck-style random test-case generation to automatically test programs that manipulate common file formats by fuzzing. We rely on existing Haskell implementations of file-format-handling libraries found on Hackage, the community-driven Haskell code repository. We have tried QuickFuzz in the wild and found that the approach is effective in discovering vulnerabilities in real-world implementations of browsers, image processing utilities and file compressors among others. In addition, we introduce a mechanism to automatically derive random generators for the types representing these formats. QuickFuzz handles most well-known image and media formats, and can be used to test programs and libraries written in any language.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools

Keywords Fuzzing, Haskell, QuickCheck, Hackage

1. Introduction

Modern software is able to manipulate complex file formats that encode richly-structured data such as images, audio, video, HTML documents, PDF documents or archive files. These entities are usually represented either as binary files or as text files with a specific structure that must be correctly interpreted by programs and libraries that work with such data. Dealing with the low-level nature of such formats involves complex, error-prone artifacts such as parsers and decoders that must check invariants and handle a significant number of corner cases. At the same time, bugs and vulnerabilities in programs that handle complex file formats often have serious consequences that pave the way for security exploits [3].

How can we test this software? As a complement to the usual testing process, and considering that the space of possible inputs is quite large, we might want to test how these programs handle *unexpected* input. *Fuzzing* [9, 6, 17] has emerged as a promising tool for finding bugs in software with complex inputs, and consists in random testing of programs using potentially invalid or erroneous inputs. There are two ways of producing invalid inputs: *mutational* fuzzing involves taking valid inputs and altering them through randomization, producing erroneous or invalid inputs that are fed into the program; *generational* fuzzing (sometimes also known as grammar-based fuzzing) involves generating invalid inputs from a specification or model of a file format. A program that performs fuzzing to test a target program is known as a *fuzzer*.

While fuzzers are powerful tools with impressive bug-finding ability [8, 13, 7], they are not without disadvantages. Mutational fuzzers usually rely on an external set of *input* files which they use as a starting point. The fuzzer then takes each file and introduces small bit-level mutations in them before using them as test cases for the program in question. The user has to collect/generate and maintain this set of input files manually for each file format they might want to test. By contrast, generational fuzzers avoid this problem, but the user must then develop and maintain models of the file format types they want to generate. As expected, creating such models requires a deep domain knowledge of the desired file format and can be very *expensive* to formulate.

In this paper, we introduce QuickFuzz, a tool that leverages Haskell's QuickCheck [4] (the well-known property-based random testing library) and Hackage (the community Haskell software repository) in conjunction with off-the-shelf bit-level mutational fuzzers to provide automatic fuzzing for several common file formats, without the need of an external set of input files and without having to develop models for the file types involved. QuickFuzz generates invalid inputs using a mix of generational and mutational fuzzing to try to discover unexpected behavior in a target application.

Hackage already contains Haskell libraries that handle well-known image, document, archive and media formats. These libraries have two important features: (a) they provide a *data type* T that serves as a lightweight specification and can be used to represent individual files of these formats, and (b) they provide a function to *serialize* elements of T to write into files. In general we call this function $encode$ and model it as having type $T \rightarrow ByteString$. Using ready-made Hackage libraries as models saves the programmers from having to write these by hand.

The key insight behind QuickFuzz is that we can make random values of T using QuickCheck's generators, then serialize them using $encode$ and pass them to an off-the-shelf fuzzer to randomize. Such mutation is likely to produce a corrupted version of the file. Then, the target application is executed with the corrupted file as input.

The missing piece of the puzzle is a mechanism to automatically derive the QuickCheck generators from the definitions of the data types in the libraries, which we do in Template Haskell and also provide in the form of a library, which we call MegaDeTH.

Haskell'16, September 22-23, 2016, Nara, Japan
ACM. 978-1-4503-4434-0/16/09...$15.00
http://dx.doi.org/10.1145/2976002.2976017

QuickCheck generates random tests cases by means of the Gen monad. A generator of type $Gen\ T$ for type T is given as an instance of the $Arbitrary$ type class. MegaDeTH inspects the structure of a data type and automatically constructs an appropriate instance of $Arbitrary$ for it. Unlike existing tools for deriving such instances, MegaDeTH can correctly handle mutually recursive data types as well as types with nested nonprimitive types.

Finally, if an abnormal termination is detected (for instance, a segmentation fault), the tool will try to minimize the size of the corrupted file in order to output the smallest test case. Such information can be very useful for the developers of the faulty program to fix the issue.

Thanks to Haskell implementations of file-format-handling libraries found on Hackage, QuickFuzz currently generates and mutates a large set of different file types out of the box. However, it is also possible for the user to add file types by providing a data type T and $encode$ function. Our framework can derive $Arbitrary$ instances fully automatically, to be used by QuickFuzz to discover bugs in new applications.

Although QuickFuzz is written in Haskell, we remark that it treats its target program as a black box, giving it randomly-generated, invalid files as arguments. Therefore, QuickFuzz can be used to test programs written in any language.

Our contributions can be summarized as follows:

- We present QuickFuzz, a tool for automatically generating inputs and fuzzing programs parsing several common types of files. QuickFuzz uses QuickCheck behind the scenes to generate test cases, and is integrated with fuzzers like *Radamsa*, *Honggfuzz* and other bug-finding tools such as *Valgrind* and *Address Sanitizer*.
- We released QuickFuzz as open-source and free of charge. As far as we know, QuickFuzz is the first fuzzer to offer the generation and mutation of more than a dozen complex file types without requiring the user to develop the models: just install, select a target program and wait for crashes. The tool is available at `http://quickfuzz.org/`.
- We introduce MegaDeTH, a library to derive Arbitrary instances for Haskell data types. MegaDeTH is fully automatic and capable of handling mutually recursive types and deriving instances from external modules. This library can be used to extend QuickFuzz with new data types.
- We evaluate the practical feasibility of QuickFuzz and show a list of security-related bugs discovered using QuickFuzz in complex real-world applications like browsers, image-processing utilities and file archivers among others.

The rest of the paper is organized as follows. Section 2 provides an overview of how QuickFuzz works using an example. Section 3 discusses how to generate $Arbitrary$ instances and the implementation of MegaDeTH. In Section 4 we highlight some of the key principles in the design and implementation of our tool using the QuickCheck framework. Later, in Section 5, we perform an evaluation of its applicability. Section 6 presents related work and Section 7 concludes.

2. A Quick Tour of QuickFuzz

In this section, we show QuickFuzz in action with a simple example. More specifically, we will see how to discover bugs in *giffix*, a small command line utility from giflib that attempts to fix broken GIF images. Our tool has built-in support for the generation of GIF files using the JuicyPixels library [18] and the $Arbitrary$ instances automatically derived by MegaDeTH . We treat MegaDeTH as a black box for now, and defer an in-depth look at MegaDeTH to Section 3.

To launch a fuzzing campaign on giffix, we simply execute:

```
$ QuickFuzz Gif 'giffix @' -a radamsa -s 10
```

With these command-line parameters, our tool generates GIF files using *Radamsa* to perform bit-level mutations.

After a few seconds, QuickFuzz stops since it found an execution that fails with a segmentation fault. At this point we can examine the output directory (*outdir* by default) to see the GIF file produced by our tool that caused giffix to fail.

Figure 1 shows the QuickFuzz pipeline and architecture. An execution of QuickFuzz consists of three phases: high-level fuzzing, low-level fuzzing and execution. The diagram also shows the interaction with MegaDeTH, which just provides $Arbitrary$ instances for the high-level fuzzing phase and can be run offline. Let us take a look at what happens in each phase in the giffix example.

2.1 High-Level Fuzzing

During this phase, QuickFuzz generates values of the data type T that represents the file format of the input to the target program. It relies on the Gen monad defined in QuickCheck, which provides convenient access to a random-number generator that can be used to construct randomized structured data compositionally. In our example this representation type T (borrowed from JuicyPixels) is called $GifFile$. A $GifFile$ contains a header (of type $GifHeader$) and the raw bitmap data specified as a list, among other things. Note that randomly generated elements of type $GifFile$ might not be valid GIF files, since the type system is unable to encode all invariants that should hold among the parts of the value. For example, the header might specify a width and height that doesn't match the bitmap data. For this reason, we consider that this step corresponds to generational fuzzing, where the data type definition serves as a lightweight approximate model of the GIF file format which generates potentially invalid instances of the file format. After running QuickFuzz, the output directory contains, for each test case, a text file that shows the value generated by this step. For instance, for the running example we get

$$
\begin{array}{l}
GifFile\ \{ \\
\quad gifHeader = GifHeader\ \{ \\
\quad\quad gifVersion = GIF87a, \\
\quad\quad gifScreenDescriptor = \\
\quad\quad\quad LogicalScreenDescriptor\ \{ \\
\quad\quad\quad screenWidth = 0, \\
\quad\quad\quad screenHeight = 0, \\
\quad\quad\quad backgroundIndex = 1, \\
\quad\quad\quad hasGlobalMap = False, \\
\quad\quad\quad colorResolution = 0, \\
\quad\quad\quad isColorTableSorted = False, \\
\quad\quad\quad colorTableSize = 1 \\
\quad\quad \}, \\
\quad\quad ...
\end{array}
$$

After generating a value of type $GifFile$ with QuickCheck's Gen monad, we use the Hackage library's $encode$ function for this file type (in this case $GifFile$ has an instance of $Binary$) to serialize the $GifFile$ into a $ByteString$, which is also written as-is to the output directory for further inspection by the user.

2.2 Low-Level Fuzzing

Usually the use of high-level fuzzing produced by the values generated by QuickCheck is not enough to trigger some interesting bugs. Therefore, this phase relies on an off-the-shelf mutation fuzzer to introduce errors/mutations at the bit level on the $ByteString$ pro-

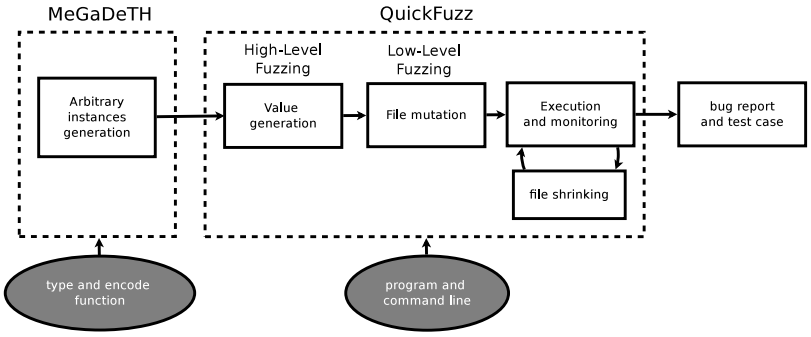

Figure 1: Summary of high-level and low-level fuzzing of QuickFuzz where grey nodes represent inputs provided by a user.

duced by the previous step. In particular, the current version supports the following fuzzers:

- Zzuf: a transparent application input fuzzer by Caca Labs [2].
- Radamsa: a general purpose fuzzer developed by the Oulu University Secure Programming Group [13].
- Honggfuzz: a general purpose fuzzer developed by Google [7].

One of the key principles of the design of QuickFuzz was to require no parameter tuning in the use of 3rd party fuzzers and bug-detection tools. Usually, the use of mutational fuzzers requires fine tuning of some critical parameters. Instead, we decided to incorporate sane default values to perform an effective fuzzing campaign even without fine-tuning values like mutation rates.

Additionally, QuickFuzz can use Valgrind [12] and Address Sanitizer [15] to detect more subtle bugs like a read out-of-bounds that would not cause a segmentation fault or the use of uninitialized memory.

2.3 Execution

The final phase involves running the target program with the mutated file as input. As we have seen, for each test case file producing runtime failure, we can also find in the output directory the intermediate values for each step of the process:

- A text file with the printed value generated by QuickCheck.
- The original encoded value, before the mutation by the mutational fuzzer.
- The actual mutated file which was passed as input to the target program and resulted in failure.

Developers can examine how the file was corrupted in order to understand why their program failed and how it can be fixed.

QuickFuzz found a test case to reproduce a heap-based overflow in giffix (CVE-2015-7555). This issue is caused by the lack of validation of the size of the logical screen and the size of the actual gif frames. In fact, if we run the tool during no more than 5 minutes in a single core, we will obtain dozens of test cases triggering failed executions (crashes and aborts). Crash de-duplication is currently outside the scope of our tool, so we manually checked the backtraces using a debugger and determined that giffix was failing in 3 distinctive ways.

The root cause of such crashes can be the same, for instance if the program is performing a read out-of-bounds. Nevertheless, QuickFuzz can still obtain valuable information finding different crashes associated with the same issue: they can be very useful to determine if the original issue is exploitable or not.

Test Case Minimization Immediately after a failed execution is found, QuickFuzz uses QuickCheck's *shrinking* feature to start the test case minimization. This procedure is very important for the developers looking to fix the issue, since the minimized test case should only trigger the code that is required to reproduce the unexpected behavior. It is also called input simplification [20]. QuickFuzz uses a simple strategy to reduce the number of bytes in the test cases causing the failed execution, defining the following function:

$$shrink :: ByteString \rightarrow [ByteString]$$
$$shrink\ bs = (tail\ \$\ tails\ bs) +\!\!+ (init\ \$\ inits\ bs)$$

The shrinking procedure will check every reduced input returned by our $shrink$ function in order to detect which are still causing an execution fail. Since the use of headers, blocks and trailer sections in files is common practice, we decided to simplify the $shrink$ function to prune either the beggining or the end of a file.

Given a $shrink$ function that returns a list of smaller files to check, the minimization procedure is performed automatically by QuickCheck. It is important to note that in our implementation there is no guarantee that a failed execution in the minimization process will trigger the same bug as the original one. Nevertheless, in the worst case the test case minimization process discovers a different issue.

3. Automatically Deriving Type Class Instances

As explained in Section 2, our first step is the generation of complex data. We obtain this by selecting a type that represents the data we want to generate, and giving a good $Arbitrary$ instance for that type. Then it is up to QuickCheck to generate test cases and test properties about them.

In order to define an instance of $Arbitrary$ for a particular type, we have to provide a definition of a monadic computation $arbitrary :: Gen\ a$, known as a *generator*, that generates arbitrary elements of type a.

In this section we present MegaDeTH, a tool to automatically derive suitable $Arbitrary$ instances for a given type.

3.1 MegaDeTH

MegaDeTH is a tool implemented in Template Haskell that gives the user the ability to provide instances to a type and all of its nested types. Given a type A MegaDeTH generates a list of all types that are needed in order to instantiate A and instantiates each one until A can be instantiated.

Let us consider the situation when a Haskell programmer wants to use a library and needs to print out on the screen certain results (of a library-specific data type) produced by it. If the library does

not provide a *Show* instance for it, writing such an instance can be tortuous task. The cause of this is that in general the top-level type refers to a number of other nonprimitive types, which we shall refer to as *nested types*. These types, in turn, might refer to further nested types that the user does not need to know about, and usually those nested types are not even exported. For example, consider the following data type definitions for a binary tree and a type for the contents of a node:

```
data Bin a = L a | B (Bin a) Node (Bin a)
data Node = Node {name :: String, l :: Int, d :: Int}
```

In order to define a *Show* instance for type *Bin*, our top-level type in this toy example, we need to provide a *Show* instance for type *Node*, a nested type for *Bin*.

Mega Derivation TH (MegaDeTH) offers a solution to this problem: it gives the user a way to *thoroughly* derive instances for all the intermediate nested types that are needed to make the top-level instance work.

MegaDeTH was implemented using Template Haskell [16], a metaprogramming mechanism built into GHC that is extremely useful to process the AST of Haskell programs and insert new declarations at compilation time. We use the power of Template Haskell to extract all the nested types for a given type and derive a class instance for each of them, finally instantiating the top-level type. Since Haskell gives the user the possibility of writing mutually recursive types, MegaDeTH implements a *topological sort* to find a suitable order in which to instantiate each type.

In Template Haskell, type names are reified into type *Name*, and declarations into type *Dec*. Considering that type class instances are a kind of declaration, we use a Template Haskell function of type $Name \to Q\ [Dec]$ to model a compile-time metafunction that derives an instance of a class for a given type, which we refer to as a *derivation function*. The function returns in the Q monad, which gives us access to Template Haskell's internal state and allows us to inspect the structure of types.

The main function exported by MegaDeTH is *megaderive*:

```
megaderive :: (Name → Q [Dec])
           → (Name → Q Bool)
           → Name → Q [Dec]
```

We can interpret this type as a way of lifting an existing derivation function (of type $Name \to Q\ [Dec]$) into a new derivation function that works on all nested types of the argument type. The first argument of *megaderive* is a function that takes care of providing an instance of the wanted class for a single type. As mentioned before, *megaderive*'s traversal is *deep*: it explores all the dependencies and tries to derive instances for all the nested types. Hence in order to exclude some types we can indicate those types by passing a filter function. Given that *megaderive* does not know the name of the class, we can use the filter function to stop MegaDeTH from instantiating types that were already defined. In order to give a clear user interface we provide a function *isinsName* (of type $Name \to Name \to Q\ Bool$) for that very purpose. This filter is extremely useful, it gives the user the ability to provide instances to a particular nested type without having to define all the other ones. However we keep the filter function as general as we can to give more control to the user.

For example, we can use the *makeShow* method given by the package Derive [11] to easily instantiate all the required nested types with MegaDeTH as follows:

```
devShow :: Name → Q [Dec]
devShow = megaderive (derive makeShow)
  (isinsName ''Show)
```

Now we can use MegaDeTH to derive the *Show* instance for *Bin*. In order to use Template Haskell we need to activate the extension *TemplateHaskell*, which provides us with a reification function to get the corresponding element of type *Name* for a given type ($''$).

```
{-# Language TemplateHaskell #-}
devShow ''Bin
```

Which will generate the following code and insert it in compilation time:

```
instance Show Node where
  showsPrec _ (Node x1 x2 x3)
    = ((showString "Node {name = ")
      ∘ ((showsPrec 0 x1)
      ∘ ((showString ", l = ")
      ∘ ((showsPrec 0 x2)
      ∘ ((showString ", d = ") ∘ ((showsPrec 0 x3)
        ∘ (showChar '}')))))))
instance Show a → Show (Bin a) where
  showsPrec p (L x1)
    = ((showParen (p > 10)) $((showString "L ")
      ∘ (showsPrec 11 x1)))
  showsPrec p (B x1 x2 x3)
    = ((showParen (p > 10))
      $((showString "B ")
        ∘ ((showsPrec 11 x1)
        ∘ ((showChar ' ')
        ∘ ((showsPrec 11 x2) ∘ ((showChar ' ')
          ∘ (showsPrec 11 x3)))))))
```

Given that the implementation of MegaDeTH was driven by the needs of QuickFuzz, we design a new instantiation method for the *Arbitrary* class. While Derive provides a derivation method for *Arbitrary* instances, in practice it proved to be not a good choice in the presence of mutually recursive types. Given a type A with constructor declarations *C1 a11 a21* and *C2 a12 a22*, Derive's instantiation for *Arbitrary A* will select with the same probability one of the constructors, *C1* or *C2*, and compute each of the arguments using *arbitrary* again, as the following example code:

```
instance Arbitrary A where
  arbitrary = do
    x ← choose (0 :: Int, 2)
    case x of
      0 → C1 ⟨$⟩ arbitrary ⟨⋆⟩ arbitrary
      1 → C2 ⟨$⟩ arbitrary ⟨⋆⟩ arbitrary
```

The main problem with this derivation method is that in the presence of mutually recursive types, say T and R, it is possible to always select the constructors of T that contain an R and vice versa, leading to non-termination. To avoid this problem, the *Gen* monad is equipped with a *size* parameter and a function $sized :: (Int \to Gen\ a) \to Gen\ a$ that is normally used to write generators that produce values of finite depth. So MegaDeTH implements all its *arbitrary* generators using *sized* to have more control over which constructor to choose. These generators simply decrease the *size* parameter each time they are called, and upon reaching the value 0 they limit the constructor selection to only nonrecursive constructors.

However if Derive's derivation function for *Arbitrary* instances were to be improved, we would gladly integrate it into our library as we did with *Show*. Again, thanks to the flexibility of Template Haskell we define a function that instantiates just one type, given its name, called *deriveArbitrary* and in composition with

megaderive we can generate a *mega* tactic *devArbitrary* to instantiate all the intermediate types:

```
deriveArbitrary :: Name → Q [Dec]
devArbitrary :: Name → Q [Dec]
devArbitrary = megaderive deriveArbitrary
  (isinsName ''Arbitrary)
```

Example. For the sake of the argument, we are going to simplify some types. Take for example the following type *GifFile* found in JuicyPixels's *Juicy.Pixels.Gif* module:

```
data GifFile = GifFile {
  gifHeader :: GifHeader,
  gifImages :: [(Maybe GraphicControlExtension,
                 GifImage)],
  gifLoopingBehaviour :: GifLooping
  }
```

We can simply derive all the required instances with $(*devArbitrary* ''*GifFile*), and MegaDeTH will generate the following instances (among others):

```
instance Arbitrary GifLooping where
  arbitrary = sized go
    where go n = oneof
      [return LoopingNever
      , return LoopingForever
      , LoopingRepeat <$> resize (n - 1) arbitrary]
instance Arbitrary GifFile where
  arbitrary = sized go
    where go n =
      GifFile <$> resize (n - 1) arbitrary
      <*>(listOf $(resize (n `div` 10) arbitrary))
      <*> resize (n - 1) arbitrary
```

Here we can observe that the *go* functions always decrease their argument n and call the QuickCheck function *resize* before calling *arbitrary*. Whenever we use a constructor, we reduce the size by one, and whenever we generate a list we reduce the size by an order of magnitude. Given that the instances are automatically generated, they seem very artificial, but it is crucial to reduce the size and insert the new size in the *Gen* monad in order to generate a good mix of elements with relatively low sizes.

In the implementation of *deriveArbitrary* we make a lot of assumptions, the most important one is how we reduce the size parameter inside the *Gen* monad, as we explained before. Implementing a good strategy to reduce the size parameter is crucial in order to finish execution in some reasonable time, but it is also important to decide how values are generated, and what those values are. Because if the size is reduced too abruptly some values are never going to be generated or the probability for some elements will be very high, while for others very low. However, given the expressive power of Haskell's data types, we delegate all the responsability to guide the generation (the *how* and *what*) to them and our sole goal is to terminate in reasonable time.

4. Design and Implementation

This section details how we defined suitable properties in QuickCheck to perform the different phases of the fuzzing process.

4.1 Detecting Unexpected Termination in Programs

One of the key concepts in fuzzing is the repeated execution of a target program. In Haskell, a program execution using certain arguments can be summarized using this type:

```
type Cmd = (FilePath, [String])
```

A program execution *fails* if we detect an abnormal termination. In the POSIX.1-1990 standard, a program can be abnormally terminated after receiving the following signals:

- A *SIGILL* when it tries to execute an illegal instruction.
- A *SIGABRT* when it called `abort`.
- A *SIGFPE* when it raised a floating point exception.
- A *SIGSEGV* when it accessed an invalid memory reference.
- A *SIGKILL* at any time (usually when the operating system detects it is consuming too many resources).

After a process finished, it is possible to detect signals associated with failed executions by examining its exit status code. Traditionally in GNU/Linux systems a process which exits with a zero exit status has succeeded, while a non-zero exit status indicates failure. When a process is terminated by a signal with number n, a shell sets the exit status to a value greater than 128. Most of the shells will use $128+N$. We capture such condition in the Haskell function *has_failed*, in order to catch when a program finished abnormally:

```
has_failed :: ExitCode → Bool
has_failed (ExitFailure n) =
  (n < 0 ∨ n > 128) ∧ n ≢ 143
has_failed ExitSuccess = False
```

We only excluded *SIGTERM* (with exit status of 143) since we want to be able to use a timeout in order to catch long executions without considering them *failed*.

4.2 High-Level Fuzzing Properties

In order to use QuickCheck to uncover failed executions in programs, we need to define a property to check. Given an executable program and some arguments, QuickFuzz tries to verify that there is no failed execution as we defined above for arbitrary inputs. We call this property *prop_NoFail*. It serializes inputs to files and executes a given program, so it should be defined using *monadicIO*. Its definition is very straightforward:

```
prop_NoFail :: Cmd → (a → ByteString)
            → FilePath → a → Property
prop_NoFail pcmd encode filename x = monadicIO $
  do
    run $ write filename (encode x)
    ret ← run $ execute pcmd
    assert (¬ (has_failed ret))
```

After that, we can *quickCheck* the property of no failed executions instantiating *prop_NoFail* with suitable values. For instance, let us assume we want to test the conversion from gif to png images using ImageMagick. The usual command to achieve this would be:

```
$ convert in.gif out.png
```

In terms of *prop_NoFail*, to test the command above we should *quickCheck* the following property:

```
prop_NoFail "/usr/bin/convert in.gif out.png"
  encodeGif
  "in.gif"
```

where *encodeGif* is a function to serialize *GifFiles*.

4.3 Low-Level Fuzzing Properties

In the next phase of the fuzzing process, we enhanced the value generation of QuickCheck with the systematic file corruption produced by off-the-shelf fuzzers. Intuitively, we augmented *prop_NoFail* with a low-level fuzzing procedure abstracted as a call to the *fuzz* function.

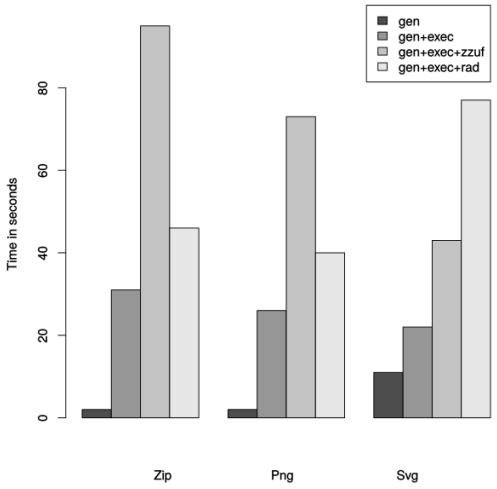

Figure 2: Overhead of QuickFuzz performing the fuzzing process.

$$fuzz :: Cmd \to FilePath \to IO\ ()$$

After calling $fuzz$, the content of a file will be changed somehow. Using this function, we defined a new property, $prop\_noFailFuzzed$ to perform a mutation of the serialized file before the execution takes place:

$$\begin{aligned}
&prop_noFailFuzzed :: Cmd \to Cmd \to (a \to ByteString)\\
&\qquad \to FilePath \to a \to Property\\
&prop_NoFailFuzzed\ pcmd\ fcmd\ encode\ filename\ x =\\
&\quad monadicIO\ \$\ \mathbf{do}\\
&\qquad run\ \$\ write\ filename\ (encode\ x)\\
&\qquad run\ \$\ fuzz\ fcmd\ filename\\
&\qquad ret \leftarrow run\ \$\ execute\ pcmd\\
&\qquad assert\ (\neg\ (has_failed\ ret))
\end{aligned}$$

5. Evaluation

5.1 Generation, Mutation and Execution Overhead

For the overhead evaluation of QuickFuzz in the different stages of the fuzzing process, we measured the time required for high-level fuzzing with and without execution (noted as `gen` and `gen+exec` respectively) as well as high and low-level fuzzing using zzuf and radamsa (noted as `gen+exec+zzuf` and `gen+exec+rad` respectively).

In order to strictly quantify the overhead in execution, we used `/bin/echo` which does not read any file. Therefore, it should always take the same amount of time to execute. Since this simple program will not crash, the shrinking process was not measured in this overhead evaluation. Each experiment was repeated 10 times in a dedicated core of an Intel i7 running at 3.40GHz. It is worth noting that every fuzzer performs one round of mutations and the parameters are defined in the tool itself, so these experiments should be easy to reproduce.

Figure 2 shows a comparison of the time that QuickFuzz took to perform each step of the fuzzing process for three different file types: Zip, Png and Svg. We selected these file types because Zip and Png are binary formats while Svg is a complex human-readable markup language, and we wanted to observe how overhead varied among those.

Our experiments suggest that the performance of the code generated by MegaDeTH for $Arbitrary$ instances is not limiting the other components of the tool. Additionally, as expected, there is a noticeable overhead in the execution. It is possible that most of the extra time executing is used for calling fork and exec primitives: this why is one the reasons some fuzzers implement a fork server.

Interestingly enough, the overhead introduced by the use of a fuzzer is not always consistent. On one hand, in the case of `zzuf`, which only XORs bits from the input files without reading them, it should be a constant overhead. But on the other hand, `Radamsa` is a fuzzer which looks at the structure of the data and performs some mutations according to it. In fact, it was specially designed to detect and fuzz markup languages: this can explain the higher overhead in the mutation of Svg files using it.

5.2 Real-World Vulnerabilities Detection

Thanks to Haskell implementations of file-format-handling libraries found on Hackage, QuickFuzz currently generates and mutates a large set of different file types out of the box. **Table 1a** shows a list of supported file types to generate and corrupt using our tool.

We tested QuickFuzz using complex real-world applications like browsers, image processing utilities and file archivers among others. All the security vulnerabilities presented in this work were previously unknown (also known as zero-days). The results are summarized in **Table 1b**. An exhaustive list is available at the official website of QuickFuzz, including frequent updates on the latest bugs discovered using the tool.

5.3 Limitations

QuickFuzz shares some of the limitations of QuickCheck. In particular, we observed that our $Arbitrary$ instances are not always effective in the generation of source code, since it requires to carefully define variable names and functions before trying to use them. Then the fuzzed generated source code will be very likely rejected in the first steps of the parsing of interpreters or compilers. This is a well-known issue that has been studied extensively by Pałka et al. [14] and Yang et al. [19] in the context of testing a compiler.

The use of third-party modules from Hackage is associated with some limitations. Some of the modules we used to serialize complex file types do not implement all the features. For instance, the bmp support in `Juicy.Pixels` cannot handle or serialize compressed files. Therefore this feature will not be effectively tested in the bmp parsers. Also, the $encode$ function used in the serialization includes its own bugs. Unsurprisingly some of them can be triggered by the generation of QuickCheck values. In this case, we have a simple workaround: if the $encode$ function throws an unhandled exception, we ignore it and continue the fuzzing process using the next generated value to serialize.

6. Related Work

Generational fuzzing The idea of a generational fuzzer is not new at all. One of the most mature and commercially supported generational fuzzers is Peach. This fuzzer was originally written in Python in 2007, and later re-written in C# for the latest release. It provides a wide set of features for generation and mutation of data, as well as monitoring remote processes. In order to start, it requires the specification of two main components to generate and mutate program inputs:

- Data Models: a formal description of how data is composed in order to be able to generate fuzzed data.
- Target: a formal description of how data can be mutated and how to detect unexpected behavior in monitored software.

As expected, the main issue with Peach is that the user has to write these configuration files, which requires very specific do-

Images	Code	Archives	Media
Bmp	Css	Tar	Ogg
Gif	Javascript	Zip	Wav
Jpeg	Python	Gzip	ID3
Png	Html	CPIO	MIDI
Pnm	Xml		
Svg	Dot		
Tga	GLSL		
Tiff	Json		
Ico	Regex		

(a) List of the file-types supported for fuzzing

Program	File-Type	Reference
Firefox	Gif	CVE-2016-1933
Firefox	Zip	CVE-2015-7194
VLC	Wav	CVE-2016-3941
GraphicsMagick	Svg	CVE-2016-2317
GraphicsMagick	Svg	CVE-2016-2318
GDK-pixbuf	Bmp	CVE-2015-7552
GDK-pixbuf	Gif	CVE-2015-7674
GDK-pixbuf	Tga	CVE-2015-7673
Jasper	Jpeg	CVE-2015-5203
libTIFF	Tiff	CVE-2015-7313
libXML	Xml	CVE-2016-3627
Jq	Json	CVE-2016-4074
Jasson	Json	CVE-2016-4425
cpio	CPIO	CVE-2016-2037

(b) Some of the security issues found by QuickFuzz

Table 1: Implementation and results

main knowledge. Another option is Sulley [1], a fuzzing engine and framework in Python. It is frequently presented as a simpler alternative to Peach since the model specification can be written using Python code. A more recent alternative open-sourced by Mozilla in 2015 is Dharma [10], a generation-based, context-free grammar fuzzer also in Python. It also requires the specification of the data to generate, but it uses a context-free grammar in a simple plain text format.

To make a fair comparison between fuzzers is always a challenge. First, it only makes sense to compare between fuzzers using similar techniques. Second, in the case of generative ones, the model to create data in all the compared fuzzers should be similar; otherwise, generating a complex input will most likely take varying amounts of time and could result in some fuzzers being unfairly flagged as *inefficient*.

Moreover, some fuzzers like Peach are not useful to start discovering bugs immediately after installing them since they include almost no models to start the input generation process. Usually, if you want to have a wide support of file-types or protocols to fuzz, you need to pay to access them, or hire someone to create them. In other cases like Sulley, fuzzers are developed to be more like a framework in which you can define models, mutate and monitor process. As a result, no file-type specifications are provided out of the box.

At first look, Dharma seems to be a good fuzzer to compare with QuickFuzz. Unfortunately, it only includes very specific grammars like Canvas2D used by the Mozilla Security team to stress a very specific API of Firefox. QuickFuzz currently does not support generation of these types of files.

Automatic algebraic data type test generation Claessen et al. [5] propose a technique for automatically deriving test data generators from a predicate expressed as a Boolean function. The derived generators are both efficient and guaranteed to produce a uniform distribution over values of a given size.

While MegaDeTH currently produces generators with ad-hoc distributions, it would be feasible to integrate this technique to the existing machinery to achieve more control over the test case generation process.

7. Conclusions and Future Work

We have presented QuickFuzz, a tool for automatically generating inputs and fuzzing programs that work on common file formats. Unlike other fuzzers, QuickFuzz does not require the user to provide a set of valid inputs to mutate, and it does not place the burden of writing specifications for file formats on the programmer. Our tool combines both generational and mutational fuzzing techniques by bringing together Haskell's QuickCheck library and off-the-shelf, robust mutational fuzzers. In addition, we introduce MegaDeTH, a library that can be used to generate instances of the $Arbitrary$ type classes. MegaDeTH works in tandem with QuickFuzz, allowing us to crowdsource the specifications for well-known file formats that are already present in Hackage libraries. We have tried QuickFuzz in the wild and found that the approach is effective in discovering interesting bugs in real-world implementations. Moreover, to the best of our knowledge QuickFuzz is the only fuzzing tool that provides out-of-the-box generation and mutation of dozens of complex, common file formats, without requiring users to write models or configuration files.

As future work, we intend to introduce mutations at different levels of the QuickFuzz pipeline, rather than just at the level of the serialized $ByteString$. In particular, we aim to explore code analysis of the serializations functions to detect and selectively break invariants, and to perform mutations on such functions to corrupt files. Finally, we would like to extend our approach to the generation and fuzzing of network protocols, as well as adding support for automatic derivation of formats with a monadic structure.

Acknowledgments

We would like to thank Alejandro Russo and Daniel Schoepe for interesting discussions, as well as the anonymous reviewers for their useful feedback and comments. This work was supported in part by The Sloan Foundation.

References

[1] Bitflip. Sulley: a pure-python fully automated and unattended fuzzing framework. `https://github.com/OpenRCE/sulley`, 2011.

[2] CACA Labs. zzuf - multi-purpose fuzzer. `http://caca.zoy.org/wiki/zzuf`, 2010.

[3] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12. IEEE Computer Society, 2012.

[4] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[5] K. Claessen, J. Duregård, and M. H. Pałka. *Generating Constrained RandomData withUniformDistribution*, pages 18–34. Springer International Publishing, Cham, 2014. ISBN 978-3-319-07151-0.

[6] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. *SIGPLAN Not.*, 2008.

[7] Google. honggfuzz: a general-purpose, easy-to-use fuzzer with interesting analysis options. `https://github.com/aoh/radamsa`, 2010.

[8] Michal Zalewski. American Fuzzy Lop: a security-oriented fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2010.

[9] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12): 32–44, Dec. 1990. ISSN 0001-0782.

[10] Mozilla. Dharma: a generation-based, context-free grammar fuzzer. `https://github.com/MozillaSecurity/dharma`, 2015.

[11] Neil Mitchell . Data.Derive is a library and a tool for deriving instances for Haskell programs. `http://hackage.haskell.org/package/derive`, 2006.

[12] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[13] Oulu University Secure Programming Group. A Crash Course to Radamsa. `https://github.com/aoh/radamsa`, 2010.

[14] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0592-1.

[15] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. USENIX ATC'12, pages 28–28, 2012.

[16] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002. ISSN 0362-1340. doi: 10.1145/636517.636528. URL `http://doi.acm.org/10.1145/636517.636528`.

[17] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[18] Vincent Berthoux. Juicy.Pixels: Haskell library to load & save pictures. `https://hackage.haskell.org/package/JuicyPixels`, 2012.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8.

[20] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.