QuickFuzz Testing for Fun and Profit

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

Agustín Mista
Universidad Nacional de Rosario, Argentina
amista@dcc.fceia.unr.edu.ar

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

Abstract

Fuzzing is a popular technique to find flaws in programs using invalid or erroneous inputs but not without its drawbacks. At one hand, mutational fuzzers require a set of valid inputs as a starting point, in which modifications are then introduced. On the other hand, generational fuzzing allows to synthesize somehow valid inputs according to a specification. Unfortunately, this requires to have a deep knowledge of the file formats under test to write specifications of them to guide the test case generation process.

In this paper we introduce an extended and improved version of QuickFuzz, a tool written in Haskell designed for testing unexpected inputs of common file formats on third-party software, taking advantage of off-the-self well known fuzzers.

Unlike other generational fuzzers, QuickFuzz does not require to write specifications for the files formats in question since it relies on existing file-format-handling libraries available on the Haskell code repository. It supports almost 40 different complex file-types including images, documents, source code and digital certificates.

In particular, we found QuickFuzz useful enough to discover many previously unknown vulnerabilities on real-world implementations of web browsers and image processing libraries among others.

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

Keywords Testing, Fuzzing, Haskell, QuickCheck

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 [7].

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* [25, 15, 35] 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; and *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 [24, 29, 16], 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 mutations in them before using them as test cases for the program in question. The user has to collect and maintain this set of input files manually for each file format she 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 she wants 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 [9], the well-known property-based random testing library and Hackage [18], the community Haskell software repository in conjunction with off-the-shelf 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. We selected libraries that 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 type T to write into files. In general we call this function $encode$ that takes a value of type T and returns a $ByteString$. 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 type T using QuickCheck's *generators*, the specialized machinery for type-driven random values generation. Then we serialize the test cases 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 call MegaDeTH.

Finally, if an abnormal termination is detected (for instance, a segmentation fault), the tool will report the input producing the crash.

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 the suitable serializing functions. Our framework can derive random generators 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 release 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 almost forty 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 random generators 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. Additionally, we describe the strategy adopted to improve the automated derivation of random generators by using not only the information found on a data type definition, but the one on its abstract interface as well. Moreover, we detail and exemplify the technique used to enforce some semantic properties in the generation of source code. This is implemented in our tool for widely used programming languages like JavaScript, Python and Lua among others.
- We evaluate the practical feasibility of QuickFuzz and show an extensive list of security-related bugs discovered using QuickFuzz in complex real-world applications like browsers, image-processing utilities and file archivers among others.

This paper is a revised and extended version of [17] which appeared in the Haskell Symposium 2016. This new version brings many theoretical and experimental contributions.

First, we extended our tool with the improved random generators using the information obtained from the abstract interface available for every library used.

Second, in the case of the source code generation, we presented a technique to enforce semantic properties immediately after the generation. We implemented this approach using meta-programming, in order to improve the random code generation of some widely used programming languages.

Third, we added three sets of experiments to explore how our tool generates and mutates files. The related work section and the experiments comparing to other fuzzers was also expanded to cover the latest developments in the field.

Finally, QuickFuzz now supports a greater number of file formats, including complex file formats found in public key infrastructure such as ASN.1, X509 and CRT certificates. Using all the proposed extensions, we have found more security related bugs, updating our results and conclusion sections accordingly.

The rest of the paper is organized as follows. Section 2 introduces fuzzing and the functional programming concepts useful to perform value generation. Section 3 provides an overview of how QuickFuzz works using an example. Section 4 discusses how to automatically derive random generators using MegaDeTH. In Section 5 we highlight some of the key principles in the design and implementation of our tool using the QuickCheck framework. Later, in Section 6, we perform an evaluation of its applicability. Section 7 presents related work and Section 8 concludes.

2. Background

2.1 Fuzzers

Fuzzers are very popular tools to test how a program handles *unexpected* input. There are two approaches for fuzzing [26]: *mutational* and *generational*.

Mutational *fuzzers* These tools produce inputs for testing programs taking valid inputs and altering them through randomization, producing erroneous or invalid inputs that are fed into the program. Typically they work producing a random mutation at the bit or byte level.

Nowadays, there are plenty of robust and fast mutational fuzzers. For instance, zzuf [6] is a fuzzer developed by Caca Labs that produced mutations in the program input automatically hooking the functions to read from files or network interfaces before a program is started. When the program reads an input, zzuf randomly flips a small percentage of bits, corrupting the data. Another popular mutational fuzzer is radamsa [29]. It was developed by the Oulu university secure programming group and works at the byte level randomly adding, removing or changing complete sequence of bytes of the program input. It features a large amount of useful mutations to detect bugs and vulnerabilities.

Both radamsa and zzuf are dumb mutation fuzzers since they do not use any feedback provided by the actual execution of the program to test. In the last few years, feedback-driven mutational fuzzers such as american fuzzy lop [24] and honggfuzz [16] were developed. These fuzzers use lightweight program instrumentation to collect information of every execution and use it to guide the fuzzing procedure.

While mutational fuzzers are one of the simpler and more popular type of fuzzers to test programs, they still require a good initial corpus to mutate in order to be effective.

Generational *fuzzers* These tools produce inputs for testing programs generating invalid or unexpected inputs from a specification or model of a file format.

This type of fuzzers are also popular in testing. For instance, one of the most mature and commercially supported generational fuzzers is Peach [11]. It 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, as well as monitoring remote processes. However, in order to start fuzzing, 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 domain knowledge. Another option is Sulley [31], 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 [27], 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.

In recent years, tools like AUTOGRAM [19] and GLADE [4] helped to learn and syntetize inputs grammars to test programs. These tools start from valid input files and using the analyzed program itself, they approximate the input grammar. AUTOGRAM uses dynamic taint analysis to syntetize the input grammar while GLADE executes the program as an oracle to answer membership queries (i.e., whether a given input is valid). Later such grammars can be used as model in generational fuzzers [4].

2.2 Haskell

Haskell is a general-purpose purely-functional programming language [22]. It provides a powerful type system with highly-expressive user-defined algebraic data types. With the power to precisely constrain the values allowed in a program, types in *Haskell* can serve as adequate lightweight specifications.

Data Types Data types in *Haskell* are defined using one or more *constructors*. A constructor is a tag that represents a way of creating a data structure and it can have zero or more arguments of any other type.

For instance, we can define the *List a* data type representing lists of values of type a by using two constructors: *Nil* represents the empty list, while *Cons* represents a non-empty list formed by combining a value of type a and a list (possibly empty) as a tail. Note that this is a recursive type definition.

```
data List a
  = Nil
  | Cons a (List a)
```

As an example, we define a few functions that we are going to use in the rest of this work.

```
length :: List a → Int
length Nil = 0
length (Cons x xs) = 1 + length xs
snoc :: a → List a → List a
snoc x Nil = Cons x Nil
snoc x (Cons y ys) = Cons y (snoc x ys)
reverse :: List a → List a
reverse Nil = Nil
reverse (Cons x xs) = snoc x (reverse xs)
```

The function *length* computes the length of a given list, *snoc* adds an element at the end of the list and finally *reverse* reverses the entire list. Their definitions are straightforward applications of pattern-matching and recursion. Free type variables in types, such as a above, are implicitly universally quantified.

Type Classes *Haskell* provides a powerful overloading system based on the notion of a *type class*. Broadly speaking, a type class is a set of types with a common abstract interface. The functions defined in the interface are said to be overloaded since they can be used on values of any member of the type class. In practice, membership in a type class is defined by means of an *instance*, i.e. a concrete definition of the functions in the interface specialized to the chosen type. For example, *Haskell* includes a built-in type class called *Eq* which defines the equality relation $(\equiv)$ for a given type. Assuming that a is in the *Eq* type class, we can define an instance of *Eq* for *List a*.

```
instance Eq a ⇒ Eq (List a) where
  Nil ≡ Nil = True
  (Cons x xs) ≡ (Cons y ys) = (x ≡ y) ∧ (xs ≡ ys)
  _ ≡ _ = False
```

Note that the $(\equiv)$ operator is used on two different types: in the expression $x \equiv y$ it uses the definition given in the instance for *Eq a* (equality on a), while in the expression $xs \equiv ys$ it is a recursive call to the $(\equiv)$ operator being defined (equality on *List a*). *Haskell* uses the type system to dispatch and resolve this overloading.

Applicative Functors In this work, we use a well-known abstraction for structuring side-effects in *Haskell*, namely *applicative functors* [23]. *Haskell* being a pure language means that all function results are fully and uniquely determined by the function's arguments, in principle leaving no room for effects such as random-number generation or exceptions, among others. However, such effects can be encoded in a pure language by enriching the output types of functions, e.g. pseudo-random numbers could be achieved by explicitly threading a seed over the whole program. Applicative functors is one of the ways in which we can hide this necessary boiler plate to implement effects.

Applicative functors in GHC are implemented as a type class. In order to define an applicative functor one has to provide definitions of two functions, *pure* and $(\langle\star\rangle)$, with the types given below.

```
class Applicative p where
  pure :: a → p a
  (⟨⋆⟩) :: p (a → b) → p a → p b
```

The function *pure* inserts pure values into the applicative structure (the boiler plate), and $(\langle\star\rangle)$ gives us a way to "apply" a function inside the structure to an argument. Due to overloading, computations written using this interface can be used with any applicative effect.

For example, assume that we have a function $(+) :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ that adds two numbers, and that we have a type *RNG* with an instance *Applicative RNG* that represents random-number generation, and moreover that there is a value $\mathit{gen} :: \mathit{RNG}\ \mathit{Int}$ that produces a random *Int*. We can express a computation that adds two random numbers using the applicative interface as follows: $\mathit{pure}\ (+)\ \langle\star\rangle\ \mathit{gen}\ \langle\star\rangle\ \mathit{gen}$. This expression has type *RNG Int* (which can be read as "an *Int* produced potentially from random data"), and it can be further used in other applicative computations as needed.

Hackage This work draws on packages found in *Hackage*. Hackage is the *Haskell* community's central package archive. As we will explain, we take from this archive the data types used to generate different file formats. For instance, the JuicyPixels library is available in Hackage [37], and it has support for reading and writing different image formats.

Hackage is a fundamental part of QuickFuzz, since it provides all the lightweight specifications for free and we carefully designed QuickFuzz to easily include new formats as they appear in this code repository.

2.3 QuickCheck

QuickCheck is a tool that aids the programmer in formulating and testing properties of programs, first introduced as a Haskell library by Koen Claessen and John Hughes [9]. *QuickCheck* presents mechanisms to generate random values of a given type, as well as a simple language to build new generators and specify properties in a modular fashion. Once the generators have been defined, the properties are tested by generating a large amount of random values.

Properties To use this tool, a programmer should define suitable properties that the code under test must satisfy. *QuickCheck* defines a property basically as a predicate, i.e. a function that returns a boolean value. For instance, we can check if the size of a list is preserved when we reverse it.

```
prop_reverseSize :: List a → Bool
prop_reverseSize xs = length xs ≡ length (reverse xs)
```

QuickCheck will try to falsify the property by generating random values of type *List a* until a counter example is found.

Generators *QuickCheck* requires the programmer to implement a generator for *List a* in order to test properties involving such data type, like *prop_reverseSize* above. The tool defines an applicative functor *Gen* and a new type class called *Arbitrary* for the data types whose values can be generated. Its abstract interface consists solely of a function that returns a generator for the data type a being instantiated. The applicative functor *Gen* provides the required mechanisms to generate random values. As seen in the previous subsection, effectful behavior requires an applicative structure.

```
class Arbitrary a where
  arbitrary :: Gen a
```

Then it is up to the programmer to define a proper instance of *Arbitrary* for *List a* using the tools provided by *QuickCheck*:

```
instance Arbitrary a ⇒ Arbitrary (List a) where
  arbitrary = genList
    where
      genList = oneof [genNil, genCons]
      genNil = pure Nil
      genCons = pure Cons
      ⟨⋆⟩(arbitrary :: Gen a)
      ⟨⋆⟩(arbitrary :: Gen (List a))
```

The function *oneof* chooses with the same probability between a *Nil* value generator or a *Cons* value generator. Note that *genCons* calls to *arbitrary* recursively in order to get a generated *List a* for its inner list parameter.

However, the previous implementation has a problem; it is possible for *oneof* to always choose a *genCons*, getting the computation in an endless loop. To solve this, *QuickCheck* provides tools to limit the maximum value generation size. An improved implementation uses the size dependent functions *sized* and *resize*, which take care of the maximum generation size, decreasing it after every recursive step. When the size reaches zero, the generation always returns *Nil*, ensuring that the value construction process never gets stuck in an infinite loop. The generation size is controlled externally and is represented in this case by the n parameter.

```
instance Arbitrary a ⇒ Arbitrary (List a) where
  arbitrary = sized genList
    where genList n = oneof [genNil, genCons n]
      genNil = pure Nil
      genCons 0 = genNil
      gencons n = pure Cons
      ⟨⋆⟩(resize (n − 1) arbitrary :: Gen Int)
      ⟨⋆⟩(resize (n − 1) arbitrary :: Gen IntList)
```

Using this instance, *QuickCheck* can properly generate *arbitrary* values of *List a* and test properties using them:

```
quickCheck prop_reverseSize
```

and if the test passed for all the randomly generated values, *QuickCheck* will answer:

```
++++ OK, passed 100 tests
```

3. A Quick Tour of QuickFuzz

In this section, we will show QuickFuzz in action with a simple example. More specifically, how to discover bugs in *giffix*, a small command line utility from *giflib* [14] that attempts to fix broken gif images. Our tool has built-in support for the generation of Gif files using the JuicyPixels library [37].

In order to find test cases to trigger bugs in a target program, our tool only requires from the user:

- A file format name to generate fuzzed inputs
- A command line to run the target program

It is worth to mention that no instrumentation is required in order to run the target program. For instance, to launch a fuzzing campaign on *giffix*, we simply execute:

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

Our tool replaces @@ by a random filename that it will represent the fuzzed gif file before executing the corresponding command line. The next parameter specifies the mutational fuzzer it uses (radamsa in this example) and the last one is the abstract maximum size in the gif value generation. Such limitation will effectively bound the memory and the CPU time used during the file generation.

After a few seconds, QuickFuzz stops since it finds 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 between the compile-time and the run-time of QuickFuzz. Let us take a look at what happens in each phase in the *giffix* example.

3.1 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 tools provided by QuickCheck. More specifically, the random number generation tools that can be used to construct randomized structured data in a compositionally manner. In our example this representation type T (borrowed from JuicyPixels) is called *GifFile*.

```
data Looping
  = LoopingNever
  | LoopingForever
  | LoopingRepeat Int
data GifFile = GifFile Header Images Looping
```

A *GifFile* contains a header (of type *Header*), the raw bitmap images (of type *Images*), and a looping behavior (of type *Looping*), specified by three *type constructors* denoting the possible behaviors. We left *Header* and *Images* data types unspecified for the sake of the example. 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 it.

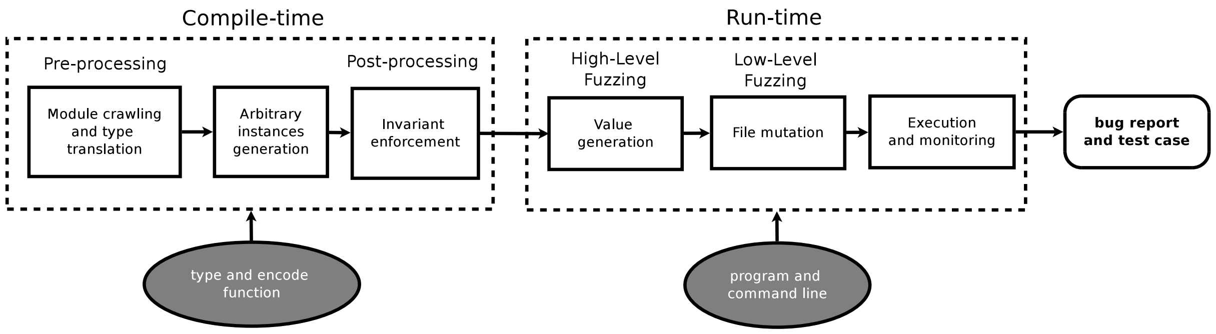


Figure 1: Summary of the random generators deriving using MegaDeTH at compile-time and the test case generation using QuickFuzz at run-time where gray nodes represent inputs provided by a user and bold nodes represent outputs.

After generating a value of type $GifFile$ with QuickCheck, we use the $encode$ function for this file type to serialize the $GifFile$ into a sequence of bytes, which is written into the output directory for further inspection by the user. Finally, the result of this phase is a gif image, most likely corrupted.

3.2 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 $ByteString$ produced by the previous step. In particular, the current version supports the following fuzzers:

- Zzuf: a transparent application input fuzzer by Caca Labs [6].
- Radamsa: a general purpose fuzzer developed by the Oulu University Secure Programming Group [29].
- Honggfuzz: a general purpose fuzzer developed by Google [16].

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

After this phase, the result will be a very corrupted gif file thanks to the combination of high-level and low-level fuzzing.

3.3 Execution

The final phase involves running the target program with the mutated file as input and check if it produces an abnormal termination. For each test case file producing a 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 test case file before the mutation by the mutational fuzzer.
- The actual mutated test case file which was passed as input to the target program and resulted in failure.

Using this information, developers can examine how the test case file was corrupted in order to understand why their program failed and how it can be fixed.

After corrupting a few gif files, QuickFuzz finds 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.

Additionally, QuickFuzz can use Valgrind [28] and Address Sanitizer [33] to detect more subtle bugs like a read out-of-bounds that would not cause a segmentation fault or the use of uninitialized memory.

4. Automatically Deriving Random Generators

In this section we explain the compilation-time stage of QuickFuzz, that can be separated into three methodologies depending on *how* the file format was implemented, and *which* file format is in order to enforce information not coded in the library:

- Automatically deriving $Arbitrary$ instances for target file formats data types. Explained in subsection 4.1.
- Crawling libraries interfaces related to the generation of the target file formats, and then, generating a higher level structure that represents manipulations of values using those interfaces. Explained in subsection 4.2.
- Post-processing the arbitrary generated values to enforce specific semantic properties. In particular, we use such technique to improve source code generation. Explained in subsection 4.3.

The last two stages are not required for every file format generation and fuzzing, however, they improve the variety of generated values as discussed on their respective subsections.

4.1 MegaDeTH

Mega Derivation TH (MegaDeTH) is a tool that gives the user the ability to provide class instances for a given type, taking care to provide suitable class instances automatically. As an example, we will analyze the $GifFile$ data type:

$$\begin{aligned}&\mathbf{data}\ Looping\\&\quad = LoopingNever\\&\quad \mid LoopingForever\\&\quad \mid LoopingRepeat\ Int\\&\mathbf{data}\ GifFile = GifFile\ Header\ Images\ Looping\end{aligned}$$

In order to define an $Arbitrary$ instance for $GifFile$, the programmer has to define such instances for $Header$, $Images$ and

Looping as well. We will refer to *GifFile* as our *target* data type, since it is the top-level data type we are looking to generate. Also, we will refer to *Header*, *Images* and *Looping* as the *nested* data types of *GifFile*. If any of these data types define further nested data types, this process has to be repeated until every data type involved in the construction of *GifFile* is a member of the *Arbitrary* type class.

Since Haskell benefits the practice of defining custom data type in an algebraic way, a data type definition can be seen as a hierarchical structure. Hence, deriving *Arbitrary* instances for every data type present at the hierarchy can be a repetitive task. MegaDeTH offers a solution to this problem: it gives the user a way to *thoroughly* derive instances for all the intermediate data types that are needed to make the desired data type instance work.

MegaDeTH was implemented using Template Haskell [34], a meta-programming mechanism built into GHC that is extremely useful to process the syntax tree of Haskell programs and to 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 one of them, finally instantiating the top-level data 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 data type satisfying their type dependencies.

We can simply derive all the required instances using MegaDeTH's function *devArbitrary* that automatically generates the following instances (among others), simplified for the sake of understanding:

```
instance Arbitrary Looping where
  arbitrary = sized gen
    where
      gen n = oneof
        [pure LoopingNever
        , pure LoopingForever
        , pure LoopingRepeat
            ⟨⋆⟩(resize (n − 1) arbitrary :: Gen Int)
            ]
instance Arbitrary GifFile where
  arbitrary = sized gen
  where gen n = pure GifFile
    ⟨⋆⟩(resize (n − 1) arbitrary :: Gen Header)
    ⟨⋆⟩(resize (n − 1) arbitrary :: Gen Images)
    ⟨⋆⟩(resize (n − 1) arbitrary :: Gen Looping)
```

As we can see, the derived code reduces the size whenever a type constructor is used and select which one is to be used with QuickCheck's *oneof* function. These automatic generated random generators follow directly the ideas presented in section 4, that is to choose between all the available constructors and generate the required arguments of it.

However, it is not always the case that we can choose between available constructors in order to generate rich structured values. We explore the limitations of this approach with further detail. The next example introduces a different manner to define a data type which exploits the limitations of MegaDeTH, and serves as introduction to the solution.

Designing a Html manipulating library. One of the main decisions involved when designing a domain-specific language [20] (DSL) manipulation library is the level of *embedding* this DSL will have. The most common approaches are *deep embedding* and *shallow embedding* [1]. Deep embedded DSLs usually define an internal intermediate representation of the terms this language can state, along with functions to transform this intermediate representation forth and/or back to the target representation. In this kind of embedding, the domain-specific invariants are mainly preserved by the internal representation. The previosly presented *GifFile* data type is an example of this technique. On the other hand, shallow embedded DSLs often use a simpler internal representation, leading the task of preserving the domain-specific invariants to the functions at the library abstract interface.

Since *Html* is a markup language, it is esentially conformed by plain text. Hence, instead of defining a complex data type using a different type constructor for each *Html* tag, the library designer could be tempted to use a shallow embedding representation, employing the same plain text representation for the library internal implementation:

```
module Html where
type Html = String
head :: Html → Html
body :: Html → Html
div :: Html → Html
hruler :: Html
(⟨+⟩) :: Html → Html → Html
toHtml :: String → Html
renderHtml :: Html → ByteString
```

In the definition above, the *Html* data type is a synonym to the *String* data type. Thus, the functions on its abstract interface are basically *String* manipulating functions with the implicit assumption that if they take a correct *html*, they will return a correct *html*, for instance:

```
head :: Html → Html
head hd = "<head>" ++ hd ++ "</head>"
hruler :: Html
hruler = "</hr>"
(⟨+⟩) :: Html → Html → Html
h1 ⟨+⟩ h2 = h1 ++ h2
```

Given that our guide in the derivation of random generators is the data type, MegaDeTH needs it to be structurally complex in order to generate complex data, remember that we based our generators on the assumption that we can choose with the same probability between different constructors in order to generate random values. If we derive a random generator for the given *Html* data type, its type definition does not provide enough structure to generate useful random values. Instead, the generated *Arbitrary* instance delegates this task to such instance of the *String* data type:

```
instance Arbitrary Html where
  arbitrary = (arbitrary :: Gen String)
```

The resulting *Html* values generated by this *Arbitrary* instance are just random strings, which rarely represents a valid *Html* value. Therefore, this kind of generators are useless for our purpose of discovering bugs on complex software parsing markup languages such as HTML.

This approach to define libraries is common to find in the wild, being *blaze-html* [21] or *language-css* [2] some examples of this. Instead of discarding them, next subsection introduces a different approach we took to derive powerful *Arbitrary* instances for this kind of libraries.

4.2 Encoding functions information into actions

Haskell's expressive power allows the library programmer to define a file format representation as a custom data type in several ways. As we have seen previously, MegaDeTH derive useful *Arbitrary* instances when the programmer had encoded invariants directly in the data type. On the other hand, as we have seen in the previous subsection, those invariants can be forced in the operations declared in the data type abstract interface. These operations manipulate the values of the data type, transforming well formed values into well formed results.

Since we need data types constructors to be able to use MegaDeTH, we use the concept of *Actions* [8]. Given a type T we can look up all the functions that return a T value and think of them as a way to create a new T value and call these functions actions. Henceforth, we can define a new data type where each function that creates a T value defines a constructor in this new type. In general, for a given data type we will refer to its actions-oriented data type by simply as *its actions data type*.

In order to illustrate this technique, we will reuse the Html manipulating library example defined in the previous subsection:

```
module Html where
type Html = String
```

To build a complex Html document, the programmer should use the functions defined in the abstract interface of this module. For example, a simple *Html* document could be represented as follows:

```
myPage :: Html
myPage =
  head (toHtml "my head")
  ⟨+⟩ body
    (div (toHtml "text")
    ⟨+⟩ hruler
    ⟨+⟩ div (toHtml "more text"))
```

The *Html* actions data type can be automatically generated, where each constructor represents a possible action over the original data type, whose type parameters corresponds to the ones at the original function this action intends to express. Note that, if an action has a parameter that comprises the original data type, it is replaced for its actions-oriented one, making this a recursively defined data type.

```
data HtmlAction
    = Action_head HtmlAction
   | Action_body HtmlAction
   | Action_div HtmlAction
   | Action_hruler
   | Action_toHtml String
   | Action_+ HtmlAction HtmlAction
```

Note that *renderHtml* will play the role of the encoding function in our representation, since it gives us a way to serialize *Html* values. Also, is worth to mention that it is not included as an action, since it does not return an *Html* value.

The previous value could be encoded using actions as follows:

```
myPageActions :: HtmlAction
myPageActions =
(Action_head (Action_toHtml "my head"))
  `Action_+`
  (Action_body
    ((Action_div (Action_toHtml "text")
    `Action_+`
    Action_hruler)
    `Action_+`
    Action_div (Action_toHtml "more text")))
```

Once an actions data type is derived for a given data type, a value of its type describes a particular composition of functions that returns a value of the original data type. Hence, we need a function *performHtml* that *performs* an action using the underlying implementation of the interface functions, returning corresponding values of the original type.

```
performHtml :: HtmlAction → Html
performHtml (Action_head v)
   = head (performHtml v)
performHtml (Action_body v)
   = body (performHtml v)
performHtml (Action_div v)
   = div (performHtml v)
performHtml Action_hruler
   = hruler
performHtml (Action_toHtml v)
   = toHtml v
performHtml (Action_+ v1 v2)
   = (performHtml v1) ⟨+⟩(performHtml v2)
```

Writting the action data type for common target data types is usually an straightforward task. A similar approach was taken in [3] in order to manually derive random generators for a particular data type of interest. However, this task also becomes repetitive, specially when the target data type contains several functions on its abstract interface. That is the reason why we automate this process by using Template Haskell. The function *devActions* is responsible for this, generating at compile time the actions data type and the performing function for a target data type. This process can be described as follows:

Step 1. Crawl the modules where the target data type is present, extracting all type constructors and functions declarations.

Step 2. Find any declarations that return a value of the target data type. Each one will become a type constructor at the actions data type.

Step 3. Generate the actions data type and the performing function for the target data type by using the previously obtained actions.

Once the actions data type and performing function have been generated for a given target data type, it is possible to use MegaDeTH to obtain an *Arbitrary* instance for the actions data type, and then, we can obtain such instance for the target data type by simply performing an arbitrary value of the first one:

```
instance Arbitrary Html where
  arbitrary = pure performHtml
     ⟨⋆⟩(arbitrary :: Gen HtmlAction)
```

We found this actions-oriented approach to be a convenient way to deal with Haskell libraries with no restrictive type definitions, wrapping their interfaces with a higher level structure and deriving suitable *Arbitrary* instances for them. Given that, it is possible to define useful *Arbitrary* instances for a variety of target data types based on the abstractions defined by the library writer, regardless of *how* the library was implemented.

There are limitations related to the generation of the actions data type. One of them involves definitions using complex types wrapping the target data type. For instance, suppose we extend the *Html* module adding a function for splitting Html values:

```
split :: Html → (Html, Html)
```

The result type for *split* does not match the target data type. However, we would like to translate it into an action as well, since the target data type (*Html*) is somehow wrapped by its result type ((*Html*, *Html*)). In order to translate *split* into an action, we need to know beforehand how to extract the target data type values from the wrapped value.

Another limitation is related to the special treatment required by polymorphic function definitions. Remember the definition of the polymorphic data type *List a* which represents a list of elements of type a, where a could be any data type:

$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$

We can define the following polymorphic functions for all a.

$\mathit{append} :: a \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a$
$\mathit{concat} :: \mathit{List}\ a \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a$

Our current approach can only handle non-polymorphic functions. We use a naive workaround to solve this consisting on instantiating every polymorphic function at the abstract interface of a module into non-polymorphic ones. This instantiation process is driven by the user, who decides which data types are interesting enough to be replaced. For instance, if the user decides to instantiate the previous list-handling functions with Int and String data types, our tool generates the following functions:

$\mathit{append\_1} :: \mathit{Int} \rightarrow \mathit{List\ Int} \rightarrow \mathit{List\ Int}$
$\mathit{append\_2} :: \mathit{String} \rightarrow \mathit{List\ String} \rightarrow \mathit{List\ String}$
$\mathit{concat\_1} :: \mathit{List\ Int} \rightarrow \mathit{List\ Int} \rightarrow \mathit{List\ Int}$
$\mathit{concat\_2} :: \mathit{List\ String} \rightarrow \mathit{List\ String} \rightarrow \mathit{List\ String}$

Then, these instantiated functions are treated like any other non-polymorphic ones at the stage of deciding which ones will be used as actions.

4.3 Enforce Variable Coherence

Using the previously explained machinery, our tool can randomly generate source code from various programming languages such as Python, JavaScript, Lua and Bash. The generation process relies on the type representing the abstract syntax tree (AST) of the code of each language.

Unfortunately, we found that automatically derived generators for such languages are not always effective at the generation of complex test cases, since they cannot account with all the invariants required for source code files to be semantically correct. In particular, one of the things that random code cannot account for is variable coherence, i.e., when we use a variable, it has to be defined (or declared).

We can see in the example below where QuickFuzz generates a complete program with variables and assignments but without any sense nor coherence between them. For example, the following program is rejected by any compiler within one of the first passes.

```
rpa = kk
meg = −18.3 == p
ize = le
```

In order to tackle this issue, we developed a generic technique to enforce properties in the resulting generated values (in this case, Python code). In particular, our goal is to correct generated source code as a first step to use QuickFuzz to test compilers and interpreters in deep stages of the parsing and executing process.

While there are some tools to test compilers, for instance CSmith [39] for stressing C compilers, they are specific tools developed for certain languages. Our approach is different, since we aim to develop a general technique that works in different complex languages provided some general guidance.

In this work, we decided to enforce variable coherence by making some corrections in the freshly generated test case. QuickFuzz goes through its AST collecting declared variables in a pool of variables identifications and changing unknown variables for previously declared variables arbitrarily taken from that pool. The special case when the pool is empty and a variable is required is sorted by generating an arbitrary constant expression.

As result we get programs where every variable used is already defined before it is used.

```
rpa = 4
meg = −18.3 == rpa
ize = meg
```

As we have seen in this section, it is possible to enforce user knowledge not encoded in either the type nor the library of a desired source code. It is also worth noting that this approach is as general as it can be. Therefore, we can implement complex invariants based on how we want to post-process the AST with all the information this structures provide.

5. Detecting Unexpected Termination of Programs using QuickCheck

This section details how we defined suitable properties in QuickCheck to perform the different phases of the fuzzing process and detect unexpected termination of programs.

Detecting Unexpected Termination in Programs In Haskell, a program execution using certain arguments can be summarized using this type:

$\textbf{type}\ \mathit{Cmd} = (\mathit{FilePath}, [\mathit{String}])$

First, we defined the notion of a *failed execution*. In our tool a program execution *fails* if we detect an abnormal termination. According to 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).

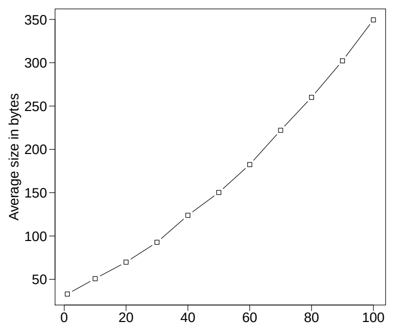
After a process finishes, it is possible to detect signals associated with failed executions by examining its exit status code. Traditionally in GNU/Linux systems a process which exits with a zero exit status has succeeded, while a non-zero exit status indicates failure. When a process terminates with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

$\mathit{has\_failed} :: \mathit{ExitCode} \rightarrow \mathit{Bool}$
$\mathit{has\_failed}\ (\mathit{ExitFailure}\ n) =$
$\quad (n < 0 \lor n > 128) \land n \not\equiv 143$
$\mathit{has\_failed}\ \mathit{ExitSuccess} = \mathit{False}$

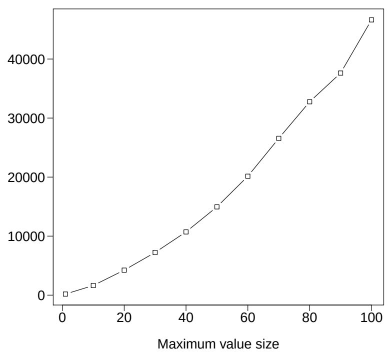
We only excluded *SIGTERM* (with exit status of 143) since we want to be able to use a timeout in order to catch long executions without considering them *failed*.

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 $\mathit{prop\_NoFail}$. It serializes inputs to files and executes a given program. Its definition is very straightforward:

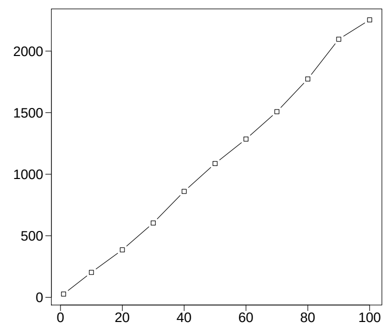
$\mathit{prop\_NoFail} :: \mathit{Cmd} \rightarrow (a \rightarrow \mathit{ByteString})$
$\qquad \rightarrow \mathit{FilePath} \rightarrow a \rightarrow \mathit{Property}$
$\mathit{prop\_NoFail}\ \mathit{pcmd}\ \mathit{encode}\ \mathit{filename}\ x =$
$\quad \textbf{do}$
$\qquad \mathit{run}\ (\mathit{write}\ \mathit{filename}\ (\mathit{encode}\ x))$
$\qquad \mathit{ret} \leftarrow \mathit{run}\ (\mathit{execute}\ \mathit{pcmd})$
$\qquad \mathit{assert}\ (\neg\ (\mathit{has\_failed}\ \mathit{ret}))$



(a) Png

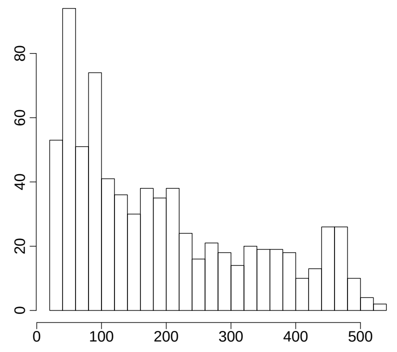


(b) Svg

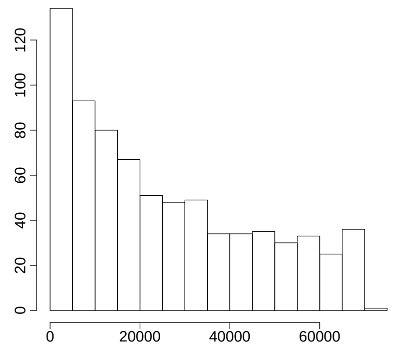


(c) zip

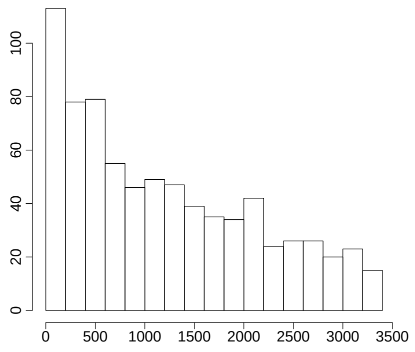
Figure 2: Average size in bytes of the generated files per file format



(a) Png

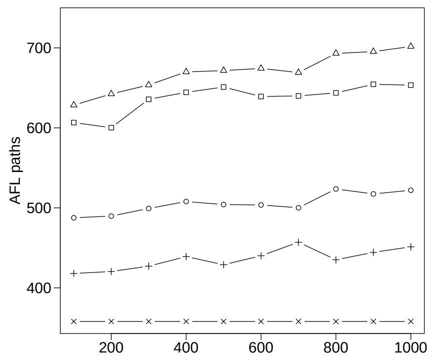


(b) Svg

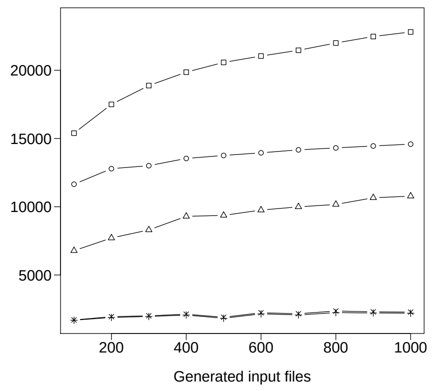


(c) zip

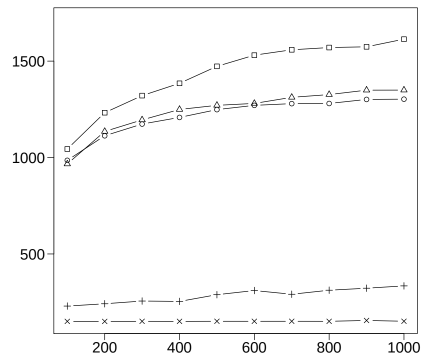
Figure 3: Frequency of generated file sizes in bytes



(a) Png files parsed by libpng 1.2.50



(b) Xml files parsed by libxml 2.9.1



(c) Jpeg files parsed by libjpeg-turbo 1.3.0

Figure 4: Average of *paths* discovered per file formats given the number of generated files. In this plots, circles (◯) represent execution of unaltered files, while triangles (△) are executions using files mutated by zzuf and squares (□) are executions using files mutated by radamsa. Pluses (+) and crosses (×) represent generation of random files with and without magic numbers respectively.

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 src.gif dest.png
```

In terms of $prop\_NoFail$, to test the command above we call the *QuickCheck* function using the following property:

```
let cmd = ("convert", ["src.gif", "dest.png"]) in
prop_NoFail cmd encodeGif "src.gif"
```

where $encodeGif$ is a function to serialize $GifFiles$. Finally, QuickCheck will take care of the GifFile generation, reporting any value that produces a failed assert in $prop\_NoFail$.

Low-Level Fuzzing Properties In the next phase of the fuzzing process, we enhance the value generation of QuickCheck with the systematic file corruption produced by off-the-shelf fuzzers. Intuitively, we augment $prop\_NoFail$ with a low-level fuzzing procedure abstracted as a call to the $fuzz$ function.

```
fuzz :: Cmd → FilePath → IO ()
```

After calling $fuzz$, the content of a file will be changed somehow. Using this new function, we define a new property called $prop\_NoFailFuzzed$ which mutates the serialized file before the execution takes place:

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

Finally, is up to QuickCheck to find a counter-example of $prop\_NoFailFuzzed$. This counter-example is a witness which causes the target program to fail execution.

As result of this process we can test any compiled program, written in any language, with a plethora of low-level fuzzers with $prop\_NoFailFuzzed$.

6. Evaluation

In this section we will describe different experiments to understand how QuickFuzz is generating and mutating input files. From the extensive list of file formats supported by QuickFuzz, shown in **Figure 6a**, we have selected five of them to perform our experiments: Zip, Png, Jpeg, Xml and Svg. We have selected these because they are binary and human-readable markup formats in different applications. We aim to observe how QuickFuzz behaves in the generation and fuzzing among those. Since the generation and fuzzing are intrinsically a random procedure, each experimental measure detailed in this section was repeated 10 times in a dedicated core of an Intel i7 running at 3.40GHz.

6.1 Generation Size

An important parameter for generational fuzzers is the maximum size of the resulting file. Such value should be carefully controlled, allowing the user to set it, according to the resources available for the fuzzing campaign. Otherwise, if the file generation results in a very large number of tiny input files or extremely large ones, it will not be effective to detect bugs. The resulting fuzzing campaign will be either useless to trigger bugs in the target program or will consume a huge amount of memory and abort.

To avoid this pitfall, our instances of $Arbitrary$ are carefully crafted to keep the size generated value under control using the $resize$ function provided by QuickCheck. **Figures 2a, 2b and 2c** show how the average size of bytes behaves when the maximum QuickCheck *size* is increased. The size of the resulting files grows linearly according to the maximum size allowed to generate by the QuickCheck framework.

It is also important to take a deeper look in the sizes of the generated files to understand how they are distributed, considering that a bias toward the generation of small files is useful in the context of the bug finding task. In fact, the benefit is twofold since (1) it keeps the amount of time spent in program executions low and (2) it prefers to generate small test cases. The resulting files triggering bugs or vulnerabilities tend to be quite small and therefore are easier to understand for developers looking to patch the faulty code.

In our experiments, we analyzed the size of the files of generated by QuickFuzz bucketing them in **Figures 3a, 3b and 3c**. In such figures, we can observe a bias for the generation of small input files.

6.2 Generation Effectiveness

Ideally, a fuzzer should generate or mutate inputs to produce a large number of distinctive executions to exercise different lines of code. Hopefully, this process should trigger conditions to discover unexpected behaviors in programs.

In order to explore the effectiveness of the generation of fuzzed files in QuickFuzz, we evaluate how many different executions we can obtain in the parsing and processing of the generated files. For the purposes of our experiments, we use the coverage measure know as *path* employed by American Fuzzy Lop [24], a well-known fuzzer, because:

- It was designed to be useful in the fuzzing campaigns: finding more *paths* is highly correlated with the discovery of more bugs [5].
- It was built using a modular approach: we can easily re-use the corresponding command line program to only extract *paths* and count them.
- It has a very fast instrumentation: it allows to extract *paths* at a nearly native speed.

Note that the AFL coverage metric might map different executions to the same *path*.

In our experiments, we use QuickFuzz to generate and fuzz Png, Jpeg and Xml files. Then, we run each fuzzed file as input to widely deployed open source libraries to parse and process them: we compiled instrumented libraries to parse Jpeg files using *libjpeg-turbo 1.3.0*, Png files using *libpng 1.2.50* and Xml files using *libxml 2.9.1*. **Figures 4a, 4c and 4b** show how many *paths* can be extracted from each instrumented implementation either using low-level mutators (zzuf and radamsa) or directly executing the generated file.

We also included two baseline measures to compare how the file structure created by our tool improves the *path* discovery. The first one generating files of random bytes and the second one using the corresponding magic numbers followed by a random bytes.

In the case of random generation, the image parsers *libjpeg-turbo* and *libpng* will try to find a valid image since they work with arbitrary binary data. The *libxml 2.9.1* rejects the random file very early in the parsing process even if it starts like a valid Xml file.

QuickFuzz discovers consistently more paths that these two baselines using random file generation.

Also, as expected, if the user generates more files using QuickFuzz, it is more likely to discover more *paths*. Additionally, the

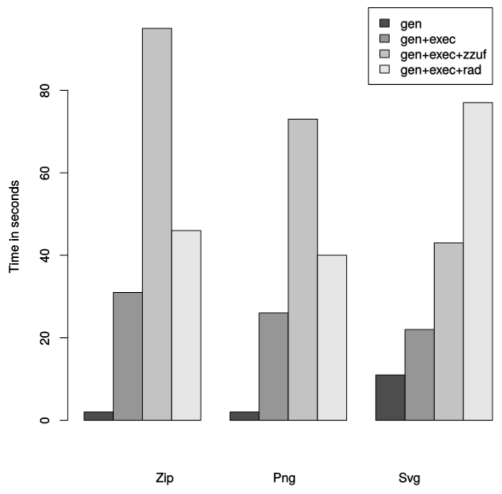


Figure 5: Overhead of QuickFuzz performing the fuzzing process.

number of discovered *paths* will grow very slowly after a few thousands files generated. This is understandable, since QuickFuzz works as *blind* fuzzer: it does not receive any feedback on the executions.

In some file formats the effect of low-level fuzzing becomes relevant. For instance, in the case of parsing fuzzed Xml files with libxml2, using radamsa as a low level fuzzers noticeable improves the number of discovered *paths*, compared to the executions of unaltered files.

Interestingly enough, mutating the files using zzuf produces quite the opposite effect: the number of *paths* is significantly reduced when this fuzzer is used. This behavior might be caused by the bit flipping of this fuzzer, causing the files to become too corrupted to be read, rejecting the files at the early stages of parsing.

6.3 Generation, Mutation and Execution Overhead

A good performance is critical in any fuzzer: we want to spend as little time as possible in the generation and mutation. 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+exec` and `gen` 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.

Figure 5 shows a comparison of the time that QuickFuzz took to perform each step of the fuzzing process for three different file types. Our experiments suggest that the performance of the code generated by MegaDeTH 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 [24].

We expected that the overhead introduced by the use of a fuzzer to be consistent regardless of the data to mutate. For instance, in the case of `zzuf`, a fuzzer which only XORs bits from the input files without reading them, it should be a constant overhead. However, the case of `Radamsa` is different. It 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.

6.4 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 6a** 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 6b**. An exhaustive list is available at the official website of QuickFuzz, including frequent updates on the latest bugs discovered using the tool.

Additionally, we reported some ordinary bugs. For instance, the use of variable coherence enforcement allowed us to find a bug that stalls the compilation in Python, and more than a twenty memory issues in GNU Bash and Busybox.

6.5 Comparison with Other Fuzzers

To make a fair comparison between fuzzers is a challenge. First, it only makes sense to compare between fuzzers using similar techniques. Second, in the case of generative ones, the model to produce files in all the compared fuzzers should be similar or somehow equivalent; otherwise, generating a complex input will most likely take varying amounts of time and could result in some fuzzers being unfairly flagged as *slow and 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 [12] or hire an specialist 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.

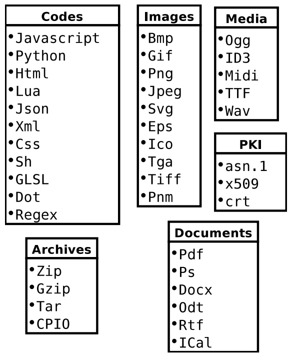
Recently, Mozilla released Dharma, a fuzzer to generate very specific files like Canvas2D and Node.js buffer scripts. It was designed by the Mozilla Security team to stress the API of Firefox. Nevertheless, this tool is good candidate to compare with QuickFuzz since it includes a grammar to generate Svg files and our tool currently supports to generate this kind of files through the types and functions of *svg-tree* package [36].

A comparison of the bugs and vulnerabilities discovered by both fuzzers is not possible: we could not find any public information regarding how many issues were reported thanks to Dharma. However, we suspect that the Mozilla Security already used it extensively to improve the quality of the Firefox parsers and the render engine.

Fortunately, it is certainly possible to compare the throughput of both fuzzers: QuickFuzz has approximately 1.9 times more throughput generating files Svg files than Dharma. While this measure is far from perfect, it gives a hint on how optimized is the generation of files using our tool.

6.6 Limitations

The use of third-party modules from Hackage carries 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. In this sense, types are used as incomplete specifications of file-formats.



(a) File-types supported for fuzzing

Program	File-Type	Reference	Program	File-Type	Reference
Firefox	Gif	CVE-2016-1933	Cairo	Svg	CVE-2016-9082
Firefox	Zip	CVE-2015-7194	libgd	Tga	CVE-2016-6132
Firefox	Svg	1297206	libgd	Tga	CVE-2016-6214
Firefox	Gif	1210745	GraphicsMagick	Svg	CVE-2016-2317
mujs	Js	CVE-2016-9109	GraphicsMagick	Svg	CVE-2016-2318
Webkit	Js	CVE-2016-9642	Mini-XML	Xml	CVE-2016-4570
Webkit	Regex	CVE-2016-9643	libical	Ical	CVE-2016-9584
gif2webp	Gif	CVE-2016-9085	Mini-Xml	Xml	CVE-2016-4571
VLC	Wav	CVE-2016-3941	GDK-pixbuf	Bmp	CVE-2015-7552
Jasper	Jpeg	CVE-2015-5203	GDK-pixbuf	Gif	CVE-2015-7674
libXML	Xml	CVE-2016-4483	GDK-pixbuf	Tga	CVE-2015-7673
libXML	Xml	CVE-2016-3627	GDK-pixbuf	Ico	CVE-2016-6352
Jq	Json	CVE-2016-4074	mplayer	Wav	CVE-2016-5115
Jasson	Json	CVE-2016-4425	mplayer	Gif	CVE-2016-4352
cpio	CPIO	CVE-2016-2037	libTIFF	Tiff	CVE-2015-7313

(b) Some of the security issues found by QuickFuzz

Figure 6: Implementation and results

We performed some experiments to compare how good is the input generation variety of QuickFuzz against a mature and complete test suite of png files. We used a test suite created by Willem van Schaik [38] that contains a variety of small PNG files. It covers different color types (gray-scale, rgb, palette, etc.), bit-depths, interlacing and transparency configurations allowed by the PNG standard. Also, in order to test robustness in the PNG parsers, this test suite includes valid images using odd sizes (for instance, very small and very large) and corrupted images. We counted the amount of distinctive *paths* after processing all the png files in the test suite using `pngtest` from libpng [32]. We performed the same experiment, but using QuickFuzz to generate and mutate png files 10,000 times.

The execution of test suite uncovers 6268 different *paths*, while the generation and fuzzing of 10,000 png files using QuickFuzz, only discovers 746 different *paths*. Therefore, our tool can only trigger $\sim 11\%$ of the *paths* we discover parsing a complex image format like PNG.

There are several explanations for such low coverage compared with a complete test suite like `pngtest`. On one hand, the generation of png files in QuickFuzz is limited by supported features in third party libraries like *Juicy.Pixels* [37]. For instance, this library lacks of the code to encode interleaved png images. On the other hand, good test suites like this one are very expensive to create since they require a very deep knowledge of the file format to test. The use of automatic tools for test suites synthesis still challenging.

Despite the automatic generation of a high quality corpus of a very complex file format like PNG is still unfeasible, it is a long term goal of our research.

Another limitation related with the $encode$ function is caused by the use of partial functions. Then the encoding could fail to execute correctly in large number of randomly generated inputs. For instance, if the $encode$ function requires some *hard* constrain to be present in the generated value such as some particular *magic* number to be guessed:

```
encodeHeader :: Int → ByteString
encodeHeader version =
  | version ≡ 87 = "GIF87"
  | version ≡ 89 = "GIF89"
  | otherwise = error "invalid version"
```

In this function, the encoding of gif format files only defines two version numbers 87 and 89: therefore, the approach to value generation defined in 4.1 is not going to be effective, since the probability of selecting a valid version number is 1 in $2,147,483,647$. Currently, this kind of issues are avoided manually selecting suitable libraries from Hackage to integrate in QuickFuzz.

Finally, the $encode$ function used in the serialization includes its own bugs. Unsurprisingly some of them can be triggered by the generation of QuickCheck values. We reported some of these issues as bugs [13] to the upstream developers of the libraries we use in QuickFuzz. In any case, we have a simple workaround when no fix is available: if the $encode$ function throws an unhandled exception, we ignore it and continue the fuzzing process using the next generated value to serialize.

7. Related Work

Automatic algebraic data type test generation Claessen et al. [10] 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.

Testing compilers generating random programs As we stated in 4.3, we observed that $Arbitrary$ 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. Therefore, 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. [30] in the context of testing a compiler.

The approach in that paper always generate valid lambda calculus terms, representing programs in Haskell. Then, they compiled the resulting terms using the Glasgow Haskell compiler in different optimization levels, to try to discover incorrectly compiled code.

In this sense, our tool also manages to generate source code and can be used to test compilers. Nevertheless they are designed with different goals in mind; on one hand, the authors of [30] generate a program of a strongly typed language. They define suitable rules for the generation, and how to backtrack in case of failing to use them.

On the other hand, QuickFuzz generates only source code from dynamically typed programs, without using any backtracking in order to keep the generation very fast, but not always correct.

8. 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, not to 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 crowd-source the specifications for well-known file formats that are already present in Hackage libraries. We 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 almost forty 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.

Another interesting feature to add to our tool is the input simplification procedure [40]. This procedure can be used just after a crash is detected and 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.

Our goal is to implement a general way to automatically derive specialized input simplification strategies for algebraic data types encoding different file formats. Moreover, by using the actions-based approach we would like to work in a higher level of abstraction, reducing a test case to the minimal sequence of actions needed to trigger an error on target programs.

Additionally, we observed that in general Haskell programmers implement their libraries in the more general way they can abusing of the expressive power of Haskell data-type ecosystem. Therefore the action-based approach is a good starting point to derive a Generalized Algebraic Data-types that can provides us with more information based in the functions found in the library, and we might capture effectful behaviors with this idea.

Finally, we would like to extend our approach to the generation and fuzzing of network protocols, since most of the vulnerabilities there can be remotely exploitable.

Acknowledgments

We would like to thank Franco Costantini, Mauro Jaskelioff, Guillermo Grinblat, Alejandro Russo and Daniel Schoepe for interesting discussions, as well as the anonymous reviewers for their useful feedback and comments.

References

[1]. Combining deep and shallow embedding of domain-specific languages. *Comput. Lang. Syst. Struct.*, 44(PB):143–165, Dec. 2015. ISSN 1477-8424.

[2] Anton Kholomiov. language-css: a library for building and pretty printing CSS 2.1 code. `https://hackage.haskell.org/package/language-css`, 2010.

[3] T. Arts, L. M. Castro, and J. Hughes. Testing erlang data types with quviq quickcheck. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 1–8, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. doi: 10.1145/1411273.1411275.

[4] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *Programming Language Design and Implementation (PLDI)*, 2017.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

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

[7] 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.

[8] K. Claessen and J. Hughes. Testing monadic code with quickcheck. *SIGPLAN Not.*, 37(12):47–59, Dec. 2002. ISSN 0362-1340. doi: 10.1145/636517.636527.

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

[10] K. Claessen, J. Duregård, and M. H. Pałka. Generating Constrained Random Data with Uniform Distribution. In M. Codish and E. Sumii, editors, *Functional and Logic Programming: 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 18–34, Cham, 2014. Springer International Publishing.

[11] Deja vu Security. Peach: a smartfuzzer capable of performing both generation and mutation based fuzzing. `http://peachfuzzer.com/`, 2007.

[12] Deja vu Security. Peach Pits and Pit Packs. `http://www.peachfuzzer.com/products/peach-pits/`, 2016.

[13] Franco Contanstini. language-python bug report: some stuff missing in Pretty instances. `https://github.com/bjpop/language-python/issues/30`, 2010.

[14] giflib. The GIFLIB project. `http://giflib.sourceforge.net/`, 1989.

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

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

[17] G. Grieco, M. Ceresa, and P. 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. ISBN 978-1-4503-4434-0.

[18] Hackage. The Haskell community's central package archive of open source software. `http://hackage.haskell.org/`, 2010.

[19] M. Höschele and A. Zeller. Mining input grammars with autogram. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 31–34, 2017.

[20] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8377-5.

[21] Jasper Van der Jeugt. blaze-html: a blazingly fast HTML combinator library for Haskell. `https://hackage.haskell.org/package/blaze-html`, 2010.

[22] S. Marlow. Haskell 2010 language report, 2010.

[23] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008. ISSN 0956-7968.

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

[25] 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.

[26] C. Miller and Z. N. Peterson. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 2007.

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

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

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

[30] 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.

[31] Pedram Amini and Aaron Portnoy. sulley: a pure-python fully automated and unattended fuzzing framework. `https://github.com/OpenRCE/sulley`, 2012.

[32] PNG Development Group. libpng: the official PNG reference library. `http://www.libpng.org/pub/png/libpng.html`, 2000.

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

[34] 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.

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

[36] Vincent Berthoux. svg-tree: SVG loader/serializer for Haskell. `https://github.com/Twinside/svg-tree`, 2007.

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

[38] Willem van Schaik. The öfficialẗest-suite for PNG. `http://www.schaik.com/pngsuite/`, 2011.

[39] X. Yang, Y. Chen, E. Eide, and J. Regehr. CSmith: a tool that can generate random C programs that statically and dynamically conform to the C99 standard. `https://embed.cs.utah.edu/csmith/`, 2011.

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