Thesis for The Degree of Doctor of Philosophy

Language-Based Techniques and Stochastic Models for Automated Testing

Agustín Mista

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg
Göteborg, Sweden, 2023

Language-Based Techniques and Stochastic Models for Automated Testing

Agustín Mista

ISBN 978-91-7905-851-7
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5317
ISSN 0346-718X

Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology | University of Gothenburg
SE-412 96 Göteborg
Sweden
Phone: +46(0)31 772 1000

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2023.

"Sometimes I'll start a sentence, and I don't even know where it's going. I just hope I find it along the way."
- Michael Scott

Language-Based Techniques and Stochastic Models for Automated Testing

Agustín Mista

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg

Abstract

As software systems become bigger and scarier, automating their testing is crucial to ensure that our confidence in them can keep up with their growth. In this setting, Generational Fuzzing and Random Property-Based Testing are two sides of the same testing technique that can help us find bugs effectively without having to spend countless hours writing unit tests by hand. They both rely on generating large amounts of random (possibly broken) test cases to be used as inputs to the system. Test cases that trigger issues such as crashes, memory leaks, or failed assertions are reported back to the developer for further investigation. Despite being fairly automatable, the Achilles heel of this technique lies in the quality of the randomly generated test cases, often requiring substantial manual work to tune the random generation process when the system under test expects inputs satisfying complex invariants.

This thesis tackles this problem from the Programming Languages perspective, taking advantage of the richness of functional, statically-typed languages like Haskell to develop automated techniques for generating good-quality random test cases, as well as for automatically tuning the testing process in our favor. To this purpose, we rely on well-established ideas such as coverage-guided fuzzing, meta-programming, type-level programming, as well as novel interpretations of centuries-old statistical tools designed to study the evolution of populations such as branching processes. All these ideas are empirically validated using an extensive array of case studies and supported by a substantial number of real-world bugs discovered along the way.

Keywords

Automated Software Testing, Property-Based Testing, Functional Programming Languages, Meta-Programming, Stochastic Models, Haskell

List of Publications

Appended publications

This thesis is based on the following publications, listed in chronological order:

[**Paper I**] *Gustavo Grieco, Martín Ceresa,* ***Agustín Mista****, and Pablo Buiras, QuickFuzz testing for fun and profit, Journal of Systems and Software (Volume 134), 340-354 (2017).*

[**Paper II**] ***Agustín Mista****, Alejandro Russo, and John Hughes, Branching Processes for QuickCheck Generators, ACM SIGPLAN Haskell Symposium (2018).*

[**Paper III**] ***Agustín Mista*** *and Alejandro Russo, Generating Random Structurally Rich Algebraic Data Type Values, 14th IEEE/ACM International Workshop on Automation of Software Test (2019).*

[**Paper IV**] ***Agustín Mista*** *and Alejandro Russo, Deriving Compositional Random Generators, 31st Symposium on Implementation and Application of Functional Languages (2019).*

[**Paper V**] ***Agustín Mista*** *and Alejandro Russo, BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs, 21st International Symposium on Trends in Functional Programming (2020).*

[**Paper VI**] *Matthías Páll Gissurarson and* ***Agustín Mista****, Short Paper: Weak Runtime-Irrelevant Typing for Security, ACM SIGSAC 15th Workshop on Programming Languages and Analysis for Security (2020).*

[**Paper VII**] ***Agustín Mista*** *and Alejandro Russo, MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Mutations, 16th IEEE International Conference on Software Testing, Verification and Validation (2023).*

Acknowledgments

To my advisor, Alejandro Russo, may someone finally beat you at Street Fighter.

To my friends and colleagues, my rants about Reviewer #2 are over.

To Reviewer #2, it's not you, it's me.

To my parents, for their constant support and love despite the distance.

To the love of my life, Carla, and to the love of her life, Archie.

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011) as well as the Swedish research agency Vetenskapsrådet.

Contents

Part I

Overview

Chapter 1

Introduction

Testing software is critical to ensure its correctness, robustness and reliability. So much so, that it is hard to imagine that any non-trivial system would be released without having first thoroughly validated it meets its functional and security requirements. Despite this, most of the software we use nowadays is frustratingly buggy in one way or another. For the most part, the bugs we experience daily are no more than little annoyances left there to remind us how much of a can of worms computers can be, e.g., Slack randomly crashing *only* when VirtualBox is running in the background.[1] However, every so often, some bugs are so influential that they make it all the way to the news. Most of these are yet another buffer overflow in an open-source library that was exploited to run arbitrary code and steal sensitive data from millions of users — news outlets have slow days, too. But not all of them are caused by using deprecated string functions that no one cared enough to update. The ones that stick are those that evoke empathy for the humans behind them. Let us take CVE-2014-1266 (a.k.a. Apple's *goto fail*) as an example:

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
... // Other checks
err = sslRawVerify(...);
...
fail:
    ... // Cleanup code
    return err;
```

[1] A real bug that some software gurús claim is most likely caused by having a screensaver enabled in the guest virtual machine. A different school of thought, however, believes the issue comes from using VirtualBox's bidirectional clipboard. None of these diagnoses have led to a permanent solution, and the bug is still at large at the time of writing. Intrigued readers can find the entrance of the rabbit hole at `https://www.virtualbox.org/ticket/20022`.

A code fragment similar to the one above was supposed to validate the authenticity of a secure SSL/TLS connection, but the repeated `goto fail;` causes it to skip a vital check `sslRawVerify` and return a non-error code. This, in turn, opened the door for potential eavesdropping by a malicious attacker, affecting anyone running iOS or macOS up until early 2014.

Interestingly, this critical security bug was not caused by a subtle race condition, a memory leak, or anything else making it hard to reason about. It was likely just a simple copy-and-double-paste mistake that put Apple's SSL/TLS system and thus the trust in the security of their products in shambles.

"Who is responsible for all this havoc?" One might ask.

One could start by claiming that C allowing **if** statements without curly braces in their **then** block is the main offender here. But C is more than 50 years old, and by now we should all be used to dealing with its quirks and features. Changing them would take more effort than it is worth spending, and still, we risk making matters worse, so we quickly establish that whatever the C standard accepts as a valid program is *the law*, and we must abide by it.

We want names here, so let us try to pin the crime on the development process instead. The first suspect is the team who wrote the buggy code. They seem to have done so at 16:45 on a Friday, though. We should give them some leeway to make an honest mistake. The second suspect in the police lineup is the team in charge of writing tests for that code, but it turns out their alibi was having a "case of the Mondays" at the time of the crime, so they deserve to get some slack too. The last suspect we gather is the project manager responsible for that code, under the presumption that they could have allocated more budget into testing it before it was deployed. Easier said than done, the defendant claimed. "Budget (either time or financial) is finite, and the project was already over it anyways." After conducting this thorough fictitious investigation, the case remains open.

Hopefully clear by now, we cannot expect us humans[2] to deliver perfectly reliable solutions at any step of the development process of a critical software system. One way to improve the quality of software is, perhaps unsurprisingly, to put as much effort as possible into testing and/or formally verifying it. Even more so when a system is both critical and widely used, as it becomes an attractive target for attackers looking for a challenge. However, finding programmers who enjoy and are good at testing (or verifying) software is hard. To tackle this problem in the supply chain, the software industry tends to complement any existing testing effort with raw computing power and automation. Instead of carefully crafting test code that covers all possible corners of a system, the system can be fed with *many* random, arbitrary inputs until something inevitably breaks — signaling the existence of a bug or even a security vulnerability. This simple idea is the essence of *software fuzzing* and it is remarkably powerful at finding bugs that uncover corner cases that humans cannot easily spot in advance.

Fuzzing is quite popular in part because it is very automatable: it can run 24/7 with minimal intervention, and changes in the codebase do not necessarily

[2]Assuming the reader is not a bot scraping this thesis to train an AI model.

require any change in the testing harness. In some occasions, however, fuzzing can be *too coarse* to be effective. This is because fuzzers normally handle full systems at once, which either break or crash with a given input or do not. The problem is that even if a system does not crash with a given input, there might still be plenty of opportunities for something to go wrong inside of it. With this in mind, a notable approach for testing systems is known as *Random Property-Based Testing* (RPBT), where we split the initial assertion "does the system crash?" into several *testing properties*, each one specialized on validating a particular aspect of the expected behavior of the system under test, e.g., the correctness of a concrete optimization pass in a compiler; the consistency of a database after a trigger is executed; the parser and pretty-printer of a programming language being "somewhat" inverse with each other, and so on. Running these specialized properties in tandem not only gives us a better understanding of the correctness of a system but also simplifies finding the origin of the bugs they find, as individual testing properties tend to encompass smaller portions of the codebase.

Although RPBT is very useful to validate the properties a system must satisfy, as these properties drift away from the relatively simple initial assertion "does the system crash?" they become more challenging to test effectively. This is in part because using test inputs resembling random noise to test complex properties can render the whole process ineffective, since the code they intend to test might rely on non-trivial internal assumptions, e.g., their inputs having already been validated against some specification. This is often referred to as having properties with "sparse" preconditions, and it is accepted that testing them effectively requires substantial manual intervention. In this scenario, programmers need to tailor the generation process that produces random testing inputs to satisfy these sparse preconditions on a reasonable basis so they can penetrate the surface layers of the system under test. This increases the chances of finding real bugs (because the inputs remain mostly within their specification) but, at the same time, increases costs and opens the door for human bias.

As mentioned earlier, being automatable is a key feature for these testing approaches to be used in real-world systems. Thus, the need for manual intervention when requirements become less trivial makes them much less appealing, and it seems as though we are also back to where we started from: having to trust humans to manually do the right thing, even on Mondays. Fortunately, there are many of us, some being lucky enough to have the opportunity to dive into this timely problem a little deeper, so the goals of this thesis are to:

1. Recognize the limitations of existing automated testing approaches, focusing on the aspects that deter effectiveness in the face of testing systems with sparse preconditions.

2. Develop both theoretical and empirical techniques to tackle these limitations. To this purpose, we rely on well-established ideas such as coverage-guided fuzzing, meta-programming, type-level programming, as well as novel interpretations of centuries-old statistical tools designed to study the evolution of populations such as branching processes.

3. Develop software tools implementing these ideas to enable the end-user to test their systems as automatically as possible. These tools are released as publicly available open-source software.

4. Demonstrate how these ideas are robust enough to improve the state-of-the-art of automated PBT, collecting and presenting enough empirical data for the reader to make informed decisions in their future testing endeavors.

Why Haskell? Strongly-typed programming languages like Haskell are prime tools for developing automated testing techniques. This is because programmers can statically encode much of the structure of their programs using the language's type system, e.g., by defining custom algebraic data types that precisely represent the program's inputs. This enables the compiler to collect useful information and pass it onto our automated testing framework, which can later use it to fine-tune the testing process.

Although we use Haskell as the lingua franca of this thesis, the ideas presented here should be reproducible in other programming languages with similar features up to a reasonable extent.

Thesis Structure This thesis includes seven peer-reviewed articles accepted to journals, conferences, symposia and workshops spread out across different academic communities, with Software Testing (Papers I, III, and VII) and Programming Languages (Papers II, IV, V and VI) being the main categories. Naturally, these boundaries are rather fuzzy so, to make some justice to this taxonomy, Figure 1 groups these articles by their areas of contribution.

The rest of this chapter briefly outlines the main ideas covered by this thesis, indicating when progress has been made in some of the peer-reviewed papers included in it.

1 Fuzzing

Fuzzing [1] is a technique used in software testing and security analysis (e.g., penetration testing [2]) which involves providing unexpected inputs to a system under test. A program that performs fuzzing campaigns to test a program is colloquially known as a *fuzzer*. The intuition behind a fuzzer is simple: it picks an input from some inputs repository, feeds it to the system under test, and monitors its behavior to signal different kinds of results, e.g., normal executions, crashes, memory leaks and failed code assertions. This process is repeated in a loop until something bad happens in the target system or, alternatively, until a stopping condition is reached (e.g., number of tests or total time). Then, any anomaly detected in the expected behavior of the system under test is reported along with the input producing it. Figure 2 shows a simplified representation of this approach.

Real-world fuzzers typically implement certain tweaks to boost the chances of finding different kinds of vulnerabilities with remarkable success [3]–[14].

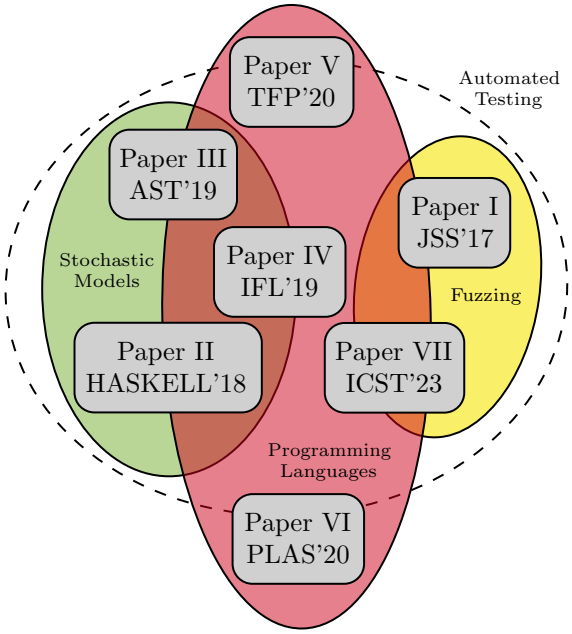

Figure 1: Areas of contribution of each paper included in this thesis.

One of the biggest differentiators between fuzzers is the nature of the inputs used to test the system under test, covering the full spectrum from completely random noise to completely semantically valid ones. Moreover, the origin of these inputs denotes an important distinction used to classify different kinds of fuzzing models [15], as described below.

- **Mutational Fuzzers:** they use an existing set of (usually valid) inputs that are combined in different ways through randomization. In practice, they often rely on an external set of input seeds provided by the user, known as a *corpus*. A mutational fuzzer takes one or more seeds from this corpus and produces a mutated version that it then uses as a test

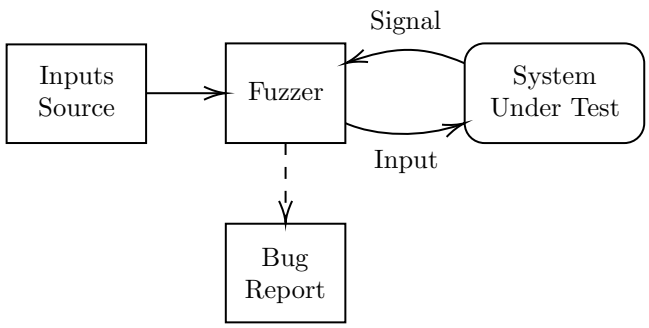

Figure 2: Simplified representation of a fuzzing environment.

input for the system under test. While this approach has shown to be quite powerful for finding bugs, its inherent disadvantage is that the user has to collect and maintain a carefully curated corpus manually for each kind of input that wants to test, e.g., for each input file format.

- **Generational Fuzzers:** they generate test inputs from scratch using a model that describes the format of the inputs expected by the system under test. These models do not necessarily need to fully match the specification of the system under test, and deliberately producing almost valid but yet broken inputs can still be useful to uncover certain kinds of bugs. In general, generational fuzzers avoid the problem of having to maintain an external corpus of inputs. However, users must then develop and maintain models of the input types they want to generate. As expected, creating such models requires deep domain knowledge, which can be tedious and expensive to achieve.

In this work, we focus particularly on the generational approach, although Paper VII demonstrates a hybrid technique that could partially fit both the generational and mutational categories — more on this later. We aim to develop automated techniques for the random generation of unexpected inputs based on statically-known information. This information can be extracted either directly from the system under test or from other external sources. In particular, Paper I is focused on automatically leveraging existing file-format manipulating libraries to derive random input generators used to test massively used programs. This led us to discover dozens of bugs and security vulnerabilities.

2 Property-Based Testing and QuickCheck

Instead of just feeding our software with random inputs and waiting for unexpected behavior, it is also possible to test our programs using randomly generated inputs in a more controlled way. The idea behind this is to verify our code against a more formal specification than just "does the system crash?" This specification can be defined, for instance, as a set of testing properties that our code must fulfill for every possible input. These properties do not necessarily involve only the input format of the system under test, but can also be specified in terms of intermediate or specialized data formats, e.g., a parsed abstract syntax tree, a serialized value, a set of command-line flags, etc. Then, these properties can be individually validated using a large number of randomly generated inputs. As mentioned earlier, this is the basis for the technique known as *Random Property-Based Testing* (RPBT).

In the Haskell realm, QuickCheck [16] is the de facto tool of this sort. Originally conceived by Koen Claessen and John Hughes, this tool counts with many success stories and inspired the ideas behind it to be replicated in other programming languages and systems with remarkable success [17]–[25].

Using QuickCheck requires the programmer to interact with two main components: *executable testing properties* and *random data generators*. Akin to a fuzzer, testing properties encapsulate the system under test (or a portion

of it) into an executable predicate that signals whether an input produces an unexpected result. Moreover, concrete input sources such as corpora are instead replaced with random data generators that produce random inputs on-the-fly. Figure 3 shows a simplified representation of this approach. Although this thesis focuses strictly on improving random data generators, automating the process of deriving testing specifications is also a non-trivial task that comprises a research field of its own [16], [26]. For completeness, the following subsections briefly introduce the reader to the usage of both components.

2.1 Testing Properties

One of the attractive aspects of QuickCheck is its simplicity. To illustrate this, suppose we write a Haskell function `reverse :: [Int] → [Int]` for reversing lists of integers. While specifying the expected behavior of this function, we might want to assert that our implementation is its own inverse, i.e., reversing a list twice always yields the original list. This property of our function can be written in QuickCheck simply as a Haskell predicate parameterized over its input, which we can think of as being universally quantified:

```
prop_reverse_ok :: [Int] → Bool
prop_reverse_ok xs =
  reverse (reverse xs) == xs
```

Then, verifying that our function holds this property becomes simply running QuickCheck over it:

```
ghci> quickCheck prop_reverse_ok
++++ OK, passed 100 tests
```

What happens under the hood is that QuickCheck will instantiate every input (`xs`) of our property using a large number of randomly generated lists of integers, asserting that `prop_reverse_ok` returns `True` for all of them.

Shall any of our properties not hold for some input, QuickCheck will try to find a minimal counterexample for us to further analyze. For instance, reversing a list of integers once will not always return the original list:

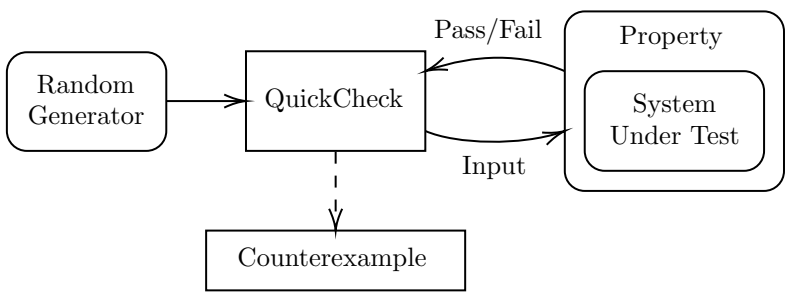

Figure 3: Random Property-Based Testing with QuickCheck.

```
prop_reverse_bad :: [Int] → Bool
prop_reverse_bad xs =
  reverse xs == xs
```

This property can be easily refuted using QuickCheck as before:

```
ghci> quickCheck prop_reverse_bad
*** Failed! Falsifiable (after 3 tests and 1 shrink):
[0,1]
```

And after a handful of random tests, we obtain a minimal counterexample (`[0,1]`) which falsifies `prop_reverse_bad` when used as an input.

This way, running a large number of random tests gives us statistical confidence about the correctness of our code against its specification.

2.2 Random Generators

One of the reasons behind the simplicity of the previous examples is that the random generation of test cases is transparently handled for us by QuickCheck. This is achieved by using Haskell's *type classes* [27]. In particular, QuickCheck defines the `Arbitrary` type class for the types that can be randomly generated:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a → [a]
```

The interface of this type class encodes two basic primitives. Firstly, `arbitrary` specifies a monadic random generator of values of type `a`. Such generators are defined in terms of the `Gen` monad which provides random generation primitives. Moreover, `shrink :: a → [a]` specifies how a given counterexample (of type `a`) can be minimized into different smaller ones. This function is used to report a minimal counterexample after a bug is found.

QuickCheck comes equipped with `Arbitrary` instances for most basic data types in the Haskell prelude. In particular, our previous testing examples simply use the default `Arbitrary` instances for integers and lists. This way, it is quite easy to test properties defined in terms of basic data types using QuickCheck. However, things get more complex when we start defining our own custom data types.

Algebraic Data Types Haskell has a powerful type system that can be extended with custom data types defined by the user. For instance, suppose we want to represent simple HTML pages as Haskell values. For this purpose, we can define the following custom algebraic data type:

```
data Html =
    Text String
  | Sing String
  | Tag String Html
  | Html :+: Html
```

```
instance Arbitrary Html where
  arbitrary = oneof
    [Text ⟨$⟩ arbitrary
    ,Sing ⟨$⟩ arbitrary
    ,Tag  ⟨$⟩ arbitrary ⟨*⟩ arbitrary
    ,(:+:) ⟨$⟩ arbitrary ⟨*⟩ arbitrary]
```

Figure 4: Naïve type-driven random generator of `Html` values.

This type allows building pages via four possible constructions: `Text` represents plain text values, `Sing` and `Tag` represent singular and paired HTML tags, respectively, and `(:+:)` concatenates two HTML pages one after another. These four constructions are known as data constructors (or constructors for short) and are used to distinguish which variant of the ADT we are constructing. Each data constructor is defined as a product of zero or more types known as fields. For instance, `Text` has a field of type `String`, whereas the infix constructor `(:+:)` has two recursive fields of type `Html`. When generating random values, we will say that a data constructor with no recursive fields is *terminal*, and *non-terminal* or *recursive* otherwise. Then, the example page:

```
<html>hello<hr>bye</html>
```

can be encoded using our freshly defined `Html` data type as:

```
Tag "html" (Text "hello" :+: Sing "hr" :+: Text "bye")
```

Later, suppose we implement two functions over `Html` values for simplifying and measuring the size of an HTML page:

```
simplify :: Html → Html
size :: Html → Int
```

The concrete implementation of these functions is not relevant here. What is important, though, is that with these functions in place, we might be interested in asserting that simplifying an HTML page never returns a bigger one. This can be encoded with the following QuickCheck property:

```
prop_simplify :: Html → Bool
prop_simplify html =
  size (simplify html) ⩽ size html
```

However, testing this property using random inputs is not possible yet. The reason behind this is simple: QuickCheck does not know how to generate random `Html`s to instantiate this property's input parameter. To solve this issue, we can provide a user-defined `Arbitrary` instance for `Html` as shown in Figure 4 (avoiding for simplicity the definition of `shrink`). To generate a random `Html` value, this generator picks a random `Html` data constructor

with uniform probability and proceeds to "fill" its fields recursively. This type-driven definition implements the simplest generation procedure for `Html` that is theoretically capable of generating any possible `Html` value.

After providing this concrete `Arbitrary` instance, QuickCheck can now proceed to test properties involving `Html` values.

3 Automated Derivation of Generators

Although simple, writing the random generator defined in Figure 4 can be quite tedious, so it is of no surprise that automated derivation mechanisms [28], [29] have emerged to relieve the programmer of the burden of this task—something especially valuable for large data types! Most of these tools use Template Haskell [30], the Haskell meta-programming framework, which allows one to examine the user code and synthesize new code based on it.

However, a suitable mechanism for deriving random generators cannot be as simple as just producing code like the one shown in Figure 4. This naïve generator is ridden with flaws, and QuickCheck users are often aware of them when implementing random generators — even an unfamiliar but attentive reader might have recognized them too. Concretely, to implement a suitable random generator we need to consider (at least) the following challenges:

Unbounded Recursion: Every time a recursive subterm is needed, the generator shown in Figure 4 calls itself recursively. This is a common mistake that can lead to infinite generation loops due to recursive calls producing (on average) one or more subsequent recursive calls. This problem can be more or less severe depending mostly on the shape of the data type our generator produces values of, being a practical limitation nonetheless. Fortunately, QuickCheck already provides a simple mechanism to overcome this issue—this is addressed by papers I-IV presented in this thesis.

Generation Parameters: The generator from Figure 4 picks the next random constructor on a uniform basis. This is the simplest approach we can mechanically follow but hardly the best choice in practice. In particular, generating values of any data type with more terminal than recursive data constructors using uniform choices will be biased towards generating very small values. QuickCheck provides mechanisms for adjusting the generation probability of each random choice it performs. However, doing so carries a second problem: it becomes quite tricky to assign these probabilities without knowing how they will affect the overall distribution of generated values — something we later discovered to be a science on its own. Both problems are addressed in detail in Paper II, where we use a stochastic model known as a *branching process* to model, predict and optimize the generation process on demand.

Abstraction Level: The generation process encoded in the generator shown in Figure 4 constructs values using the smallest possible level of granularity:

one data constructor at a time. In practice, this technique is often too weak to generate (with a non-negligible probability) values containing the complex patterns of constructors that could be required to test the corner cases of our code, leaving the door open for subtle bugs that might never get triggered within the testing budget.

On the other hand, the implementation of our code under test could rely on internal invariants that are necessary to make it work properly — consider for instance the case of the implementation of data structures like balanced trees, where its abstract interface must preserve the internal invariants used by their implementation. In this case, our testing properties will likely require providing somewhat well-formed inputs as a precondition. Thus, testing this kind of software becomes much more complicated using the approach described above, as constructing random values one data constructor at a time will very rarely produce values satisfying such preconditions. This issue is addressed in detail in Paper III, where we first show how extra static information present in the codebase can be used to generate better random data automatically by including abstract interfaces and functions' branching patterns in the mix. Later, in Paper IV we show how this enhancement can be implemented with modularity in mind in an elegant way using type-level programming. Using this idea, different static sources of structural information can be combined to generate data showing different lightweight invariants on a per-property basis.

4 Coverage Guided, Property-Based Testing

As described above, Papers I-IV contribute to the state-of-the-art of automatic derivation of random data generators based on static information. These generators are initially intended to be used with a black-box testing framework such as QuickCheck, where they show a noticeable improvement with respect to other existing automated techniques in certain real-world testing scenarios.

Being intentionally designed to follow a black-box approach, the only signal QuickCheck gets back from running a test is whether it passes, fails or gets discarded due to not passing the property's precondition. This design choice is in part what makes QuickCheck fast and easy to use. However, it also entails that if our random generators (automatically derived or otherwise) cannot generate data satisfying the sparse precondition of a testing property, then QuickCheck has no other choice but to give up early. This led us to believe that, regardless of the improvements we developed in the previous papers, automatically derived generators can still be remarkably ineffective when used to test properties with sparse preconditions if we limit ourselves to black-box RPBT. Concretely, the main limitation of the black-box approach is that neither the testing loop, the property nor the random generator can tune their behavior dynamically based on feedback taken from the execution of the system under test — a missed opportunity that the fuzzing community has taken advantage of many times in the past.

If we accept moving away from QuickCheck, we can enhance its testing loop with two key features to make it more flexible:

- **Target code instrumentation:** to capture execution information from each test case. For this, we instrument each branching point of the code in the system under test with a small wrapper that logs its execution in a global execution trace. This way, the testing loop can retrieve the path in the code of the system under test taken by each test case.

- **High-level, type-preserving mutations:** to produce syntactically valid test cases by altering existing ones at the data constructor level. Similar to automatically derived generators, these mutations can also be automatically computed by inspecting the data types used to represent input test cases using meta-programming.

Relying on code instrumentation in tandem with mutations is a well-established testing technique used outside of RPBT. Existing fuzzing tools use execution traces to recognize interesting test cases, e.g, those that exercise previously undiscovered parts of the target code [3], [4], [31]–[33]. Moreover, given a valid input test case, high-level, type-preserving mutations are a useful tool for producing new valid input test cases from existing ones.

These two features are tied together using a mutation pool that stores previously executed test cases. On one end, whenever a new test case discovers a new portion of the code in the system under test, its corresponding value is saved in the mutation pool. On the other end, values are drawn from this pool and mutated to create a new test case similar to the original one. If the mutation pool is empty, the testing loop simply generates a new random test case and repeats the process. Figure 5 outlines a simplified representation of this approach.

This technique was originally conceived by Lampropoulos *et al.* under the name of *Coverage-Guided, Property-Based Testing* (CGPT). In their original work, this technique leaves considerable room for improvement. Concretely, we observed a large reliance on randomness, which in conjunction with its simple, sub-par scheduling can prevent bugs from being found on a timely basis. To tackle these issues, Paper VII introduces MUTAGEN, a novel CGPT

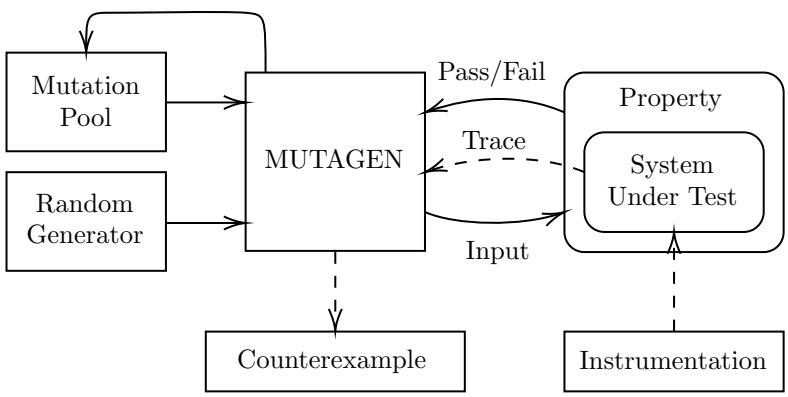

Figure 5: Coverage-Guided, Property-Based Testing with MUTAGEN.

tool that addresses the main limitations of the original CGPT approach using exhaustively computed mutations along with dynamic heuristics to improve its scalability.

5 Domain-Specific Programming Language Tools

In addition to automated software testing, this thesis covers two contributions to the state-of-the-art of domain-specific programming languages. This section discusses how the ideas behind these contributions could be used to solve some of the main problems this thesis aims to tackle.

5.1 Enhancing Embedded Domain-Specific Languages

Embedded Domain Specific Languages (EDSLs) are a useful approach to developing custom programming languages tailored to specific requirements without reinventing the wheel. Instead of having to manually implement lexers, parsers, type-checkers, optimizers or code-generators, to name a few, EDSLs can be designed to reuse some (or all) of these components from a host language. In this setting, Haskell excels at hosting EDSLs given its powerful type system and relatively extensible syntax, where its monadic **do** notation is extremely powerful to implement complex EDSLs.

Despite their evident appeal, EDSLs are not without limitations. Perhaps one of the most common ones is the lack of source metadata collected by earlier stages of the compilation pipeline. For instance, parsers often collect useful metadata that later compilation passes rely on to emit code or to generate error messages. Understandably, most compilers would not make this metadata available to the compiled program, GHC being no exception. This, in turn, limits how expressive our Haskell EDSLs can be, as they cannot access useful source-level details of the embedded programs such as source locations or variable names.

To alleviate this problem, Paper V describes BinderAnn, a flexible plugin for the GHC compiler that automatically inserts source code annotations into the user's monadic EDSL code. These annotations capture both the location (i.e., line number and file name) of every EDSL statement, as well as any name that the statement might be bound to (using the $\leftarrow$ operator). Enabling our EDSLs to use this (otherwise lost) information makes it possible to implement better code-generating tools, as well as improve the quality of domain-specific error messages.

We demonstrate using several examples how BinderAnn can be used to enhance existing EDSLs with source annotations, e.g, improving code-generation (where variables names in the generated code reflect those used in the EDSL code), as well as improving error messages (where EDSLs can guide the user to domain-specific errors using source-level locations). Moreover, we showed how EDSLs enhanced with source annotations can open the door for new programming patterns. In particular, we demonstrate how BinderAnn can be an instrumental tool to implement a simple interactive proof assistant.

Using the BinderAnn approach, it could be possible automatically insert source annotations into the system under test to guide the testing process towards new kinds of goals. In this setting, there exist several fuzzing techniques that take advantage of external information to direct their efforts towards exercising specific parts of the code, e.g., to maximize the time spent on fuzzing recent changes in the codebase or to target code that uses specific functions or that has potential vulnerabilities [35]–[37]. In our case, we could instruct a RPBT tool like MUTAGEN to use the extra source information added at compile time to optimize the effort put into testing certain EDSL code more effectively and at a higher level of abstraction — an interesting challenge to tackle in the future.

5.2 Weakening the Type System

Continuing with the topic of embedded languages, we will now focus on a specific kind: EDSLs that use Haskell's type-level capabilities to enforce domain-specific constraints. Perhaps the most remarkable example of this technique is the existence of security *libraries*, where programmers can specify Information-Flow Control policies in their programs by using type-level constraints to denote the sensitivity of the data handled by them. These constraints are checked along with the rest of the code, preventing the program to compile in the presence of forbidden information flows.

Despite providing strong domain-specific guarantees, EDSLs relying on type-level constraints often suffer from a lack of adoption. For instance, a reverse dependency search for the most (academically) popular security libraries (e.g., *MAC*, *LIO*, *HLIO*, etc.) in Hackage returns fewer than five results combined. We hypothesize that the extra friction added by using domain-specific type-level constraints takes a toll on the usability of these EDSLs, where users must fully abide by the language's domain-specific quirks before an otherwise functionally correct program can be even compiled. We argue that this all-or-nothing paradigm can be improved by letting the programmer start by developing functionally correct code, and then gradually tweak it until the domain-specific constraints are met. This can be particularly useful in the case of security libraries, where the programmer can progressively adapt their existing code to satisfy the desired security policy, but checking that the functional correctness of the program is also satisfied after each refactoring step — an approach akin to that popularized by gradually typed languages.

To tackle this issue, Paper VI describes WRIT, a type-checking plugin for the GHC compiler that *weakens* the type system to allow for certain ill-typed programs to compile in a controlled way. The kind of programs that WRIT allows GHC to compile are those that would otherwise compile just fine should their type-level constraints were removed. In other words, WRIT allows GHC to ignore type errors caused by unsolved *runtime irrelevant* constraints. We found that, oftentimes, domain-specific constraints are irrelevant at runtime, having no runtime representation, hence the compiler erases them completely during compilation. This is commonly seen when using type-level tools like phantom types and empty type classes. In such cases, WRIT allows the programmer to

transform runtime irrelevant type errors into warnings to be addressed later. Additionally, our plugin enables the EDSL designer to replace the generic error messages GHC produces when there are unsolved type-level constraints with specialized ones that refer to the concrete domain-specific constraint the client code violates.

We showed how this approach can be used to write programs matching a static IFC policy using the *MAC* library in a gradual manner and with more descriptive error messages when something goes wrong. In addition, we discussed how our approach can be extended to consider other scenarios when unsolved constraints are not runtime-irrelevant, but there exists a mechanical way to solve them by simulating dynamic typing as seen in other languages like Python or Erlang.

Weakening the type system in our favor could be used to further automate the testing process by dividing the complexity of generating test cases into different steps. For instance, if we want to test a security library, we could try to randomly generate programs using its abstract interface and verify that its security guarantees are preserved. This might not be an easy task, though. The type-level security constraints used by such a library might be too hard to satisfy using an automatically derived (or even a manually written) random generator [38], [39]. In turn, we could start by generating weaker programs that compile albeit with potential security flaws. These programs could be automatically lifted to use the library's interface via the WRIT plugin, generating security warnings in the process. Then, a subsequent step could use the information encoded into these security warnings to try to automatically "patch" their corresponding random programs into ones that still use the library's interface but compile without warnings. We believe this multi-level technique for generating random test cases could be useful to complement MUTAGEN's mutation approach to overcome the complexity of testing properties with complex security preconditions.

Chapter 2

Statement of contributions

This chapter lists the abstracts of the individual paper chapters and outlines the personal contributions for each.

Paper I - QuickFuzz testing for fun and profit

Gustavo Grieco, Martín Ceresa, **Agustín Mista** and Pablo Buiras

Abstract

Fuzzing is a popular technique to find flaws in programs using invalid or erroneous inputs but not without its drawbacks. On 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 synthesizing 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 in real-world implementations of web browsers and image processing libraries among others.

Contributions

This project was a collaboration between people from CIFASIS-Conicet and Chalmers University of Technology. Agustín contributed to this project by i) developing an extension to the existing generators' derivation mechanism, which contemplates the common case of existing libraries written using shallow embeddings of the target file format; and ii) carrying out a complete rewrite of the testing harness from scratch, maximizing the use of meta-programming to ease the task of adding support for new file-format targets.

Moreover, Agustín collaborated with the writing of the journal paper resulting from this project.

Paper II - Branching Processes for QuickCheck Generators

Agustín Mista, Alejandro Russo and John Hughes

Abstract

In *QuickCheck* (or, more generally, random testing), it is challenging to control random data generators' distributions—especially when it comes to *user-defined algebraic data types* (ADT). In this paper, we adapt results from an area of mathematics known as *branching processes*, and show how they help to analytically predict (at compile-time) the expected number of generated constructors, even in the presence of mutually recursive or composite ADTs. Using our probabilistic formulas, we design heuristics capable of automatically adjusting probabilities in order to synthesize generators whose distributions are aligned with users' demands. We provide a Haskell implementation of our mechanism in a tool called *DRAGEN* and perform case studies with real-world applications. When generating random values, our synthesized *QuickCheck* generators show improvements in code coverage when compared with those automatically derived by state-of-the-art tools.

Contributions

This project was a collaboration with Alejandro Russo. Agustín was responsible for i) developing a generic meta-programming mechanism for deriving random generators using the stochastic model based on branching processes (the first version of DRAGEN), and ii) designing and carrying out the evaluation of these ideas, comparing the results of different generator derivation techniques in terms of the code coverage observed when feeding real-world applications with randomly generated inputs.

The technical writing of this paper was initially done in equal parts between Alejandro and Agustín. John Hughes joined at a later stage with invaluable feedback.

Paper III - Generating Random Structurally Rich Algebraic Data Type Values

Agustín Mista and Alejandro Russo

Abstract

Automatic generation of random values described by algebraic data types (ADTs) is often a hard task. State-of-the-art random testing tools can automatically synthesize random data generators based on ADTs definitions. In that manner, generated values comply with the structure described by ADTs, something that proves useful when testing software that expects complex inputs. However, it sometimes becomes necessary to generate structurally richer ADTs values in order to test deeper software layers. In this work, we propose to leverage static information found in the codebase as a manner to improve the generation process. Namely, our generators are capable of considering how programs branch on input data as well as how ADTs values are built via interfaces. We implement a tool, responsible for synthesizing generators for ADTs values while providing compile-time guarantees about their distributions. Using compile-time predictions, we provide a heuristic that tries to adjust the distribution of generators to what developers might want. We report on preliminary experiments where our approach shows encouraging results.

Contributions

This project was a collaboration with Alejandro Russo. Agustín contributed to this project by i) extending the previous derivation tool and its underlying stochastic model with support for extracting and generating function patterns and API calls automatically (this extension is called DRAGEN2); and ii) designing and carrying out the evaluation of these ideas, comparing the effects of including more static information when deriving random data generators versus using a simple type-directed derivation approach.

The technical writing of this paper was done in equal parts between Alejandro and Agustín.

Paper IV - Deriving Compositional Random Generators

Agustín Mista and Alejandro Russo

Abstract

Generating good random values described by algebraic data types is often quite intricate. State-of-the-art tools for synthesizing random generators serve the valuable purpose of helping with this task, while providing different levels of

invariants imposed over the generated values. However, they are often not built for composability nor extensibility, a useful feature when the shape of our random data needs to be adapted while testing different properties or subsystems.

In this work, we develop an extensible framework for deriving compositional generators, which can be easily combined in different ways in order to fit developers' demands using a simple type-level description language. Our framework relies on familiar ideas from the à la Carte technique for writing composable interpreters in Haskell. In particular, we adapt this technique with the machinery required in the scope of random generation, showing how concepts like generation frequency or terminal constructions can also be expressed in the same type-level fashion. We provide an implementation of our ideas, and evaluate its performance using real-world examples.

Contributions

This project was a collaboration with Alejandro Russo. Agustín was responsible for i) carrying out the technical development, using meta-programming and type-level features available in Haskell to derive composable random data generators; and ii) designing and evaluating these ideas, which focus on the runtime overhead induced by the usage of composable random data generators.

The majority of the technical writing was done by Agustín. Alejandro provided invaluable feedback throughout the process.

Paper V - BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs

Agustín Mista and Alejandro Russo

Abstract

Embedded Domain-Specific Languages (EDSLs) are an alternative to quickly implement specialized languages without the need to write compilers or interpreters from scratch. In this territory, Haskell is a prime choice as the host language. EDSLs in Haskell, however, are often incapable of reifying useful static information from the source code, namely variable binding names and source locations. Not having access to variable names directly affects EDSLs designed to generate low-level code, where the variables names in the generated code do not match those found in the source code—thus broadening the semantic gap between source and target code. Similarly, many existing EDSLs produce poor error messages due to the lack of knowledge of source locations where errors are generated.

In this work, we propose a simple technique for enhancing monadic EDSLs expressed using **do** notation. This technique employs *source-to-source plugins*, a relatively new feature of GHC, to annotate every **do** statement of our EDSLs with relevant information extracted from the source code at compile time. We show how these annotations can be incorporated into EDSL designs either

directly inside values or as monadic effects. We provide *BinderAnn*, a GHC source plugin implementing our ideas, and evaluate it by enhancing existing real-world EDSLs with relatively minor modification efforts to contemplate the source-level static information related to variables names and source locations.

Contributions

This project was a collaboration with Alejandro Russo. Agustín contributed to this project by i) designing and implementing BinderAnn with support for multiple annotation styles based on valuable input from Alejandro, Koen Claessen and John Hughes; and ii) evaluating these ideas, showing how BinderAnn could solve existing real-world problems, as well as allowing for new programming patterns to emerge.

The majority of the technical writing was done by Agustín. Alejandro Russo provided invaluable feedback throughout the process.

Paper VI - Short Paper: Weak Runtime-Irrelevant Typing for Security

Matthías Páll Gissurarson and **Agustín Mista**

Abstract

Types indexed with extra type-level information are a powerful tool for statically enforcing domain-specific security properties. In many cases, this extra information is runtime-irrelevant, and so it can be completely erased at compile-time without degrading the performance of the compiled code. In practice, however, the added bureaucracy often disrupts the development process, as programmers must completely adhere to new complex constraints in order to even compile their code.

In this work we present WRIT, a plugin for the GHC Haskell compiler that relaxes the type-checking process in the presence of runtime-irrelevant constraints. In particular, WRIT can automatically coerce between runtime equivalent types, allowing users to run programs even in the presence of some classes of type errors. This allows us to gradually secure our code while still being able to compile at each step, separating security concerns from functional correctness.

Moreover, we present a novel way to specify which types should be considered equivalent for the purpose of allowing the program to run, how ambiguity at the type level should be resolved and which constraints can be safely ignored and turned into warnings.

Contributions

This project was a collaboration with Matthías Páll Gissurarson, and the result of a joint project for the course Language-Based Security led by Andrei

Sabelfeld. Agustín contributed to this project by helping with the design of the WRIT GHC plugin and its case studies.

The technical writing of this paper was done in equal parts between Matthías and Agustín.

Paper VII - MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Mutations

Agustín Mista and Alejandro Russo

Abstract

Automatically-synthesized random data generators are an appealing option when using property-based testing. There exists a variety of techniques that extract static information from the codebase to produce random test cases. Unfortunately, such techniques cannot enforce the complex invariants often needed to test properties with sparse preconditions.

Coverage-guided, property-based testing (CGPT) tackles this limitation by enhancing synthesized generators with structure-preserving mutations guided by execution traces. Albeit effective, CGPT relies largely on randomness and exhibits poor scheduling, which can prevent bugs from being found.

We present MUTAGEN, a CGPT framework that tackles such limitations by generating mutants *exhaustively*. Our tool incorporates heuristics that help to minimize scalability issues as well as cover the search space in a principled manner. Our evaluation shows that MUTAGEN not only outperforms existing CGPT tools but also finds previously unknown bugs in real-world software.

Contributions

This project was a collaboration with Alejandro Russo. Agustín contributed to this project by i) carrying out most of the technical development, with several rounds of helpful feedback from Alejandro, Koen Claessen and John Hughes; and ii) designing and performing the evaluation of these ideas, adapting existing case studies with the assistance of Leonidas Lampropoulos.

The technical writing of this paper was done in equal parts between Alejandro and Agustín, with invaluable feedback from John, Koen and Robert Feldt.

Bibliography

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

[2] B. Arkin, S. Stender and G. McGraw, 'Software penetration testing', *IEEE Security Privacy*, 2005.

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

[4] R. Swiecki, *Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options*, `https://honggfuzz.dev`, 2017.

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

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

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

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

[9] J. Wang, B. Chen, L. Wei and Y. Liu, 'Superion: Grammar-aware greybox fuzzing', in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[10] 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*, 2016.

[11] 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*, ACM, 2016.

[12] P. Godefroid, A. Kiezun and M. Y. Levin, 'Grammar-based whitebox fuzzing', in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 2008.

[13] S. K. Cha, M. Woo and D. Brumley, 'Program-Adaptive Mutational Fuzzing', in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[14] P. Godefroid, M. Y. Levin and D. Molnar, 'Sage: Whitebox fuzzing for security testing', *Communications of the ACM*, vol. 55, 2012.

[15] C. Miller and Z. N. Peterson, 'Analysis of mutation and generation-based fuzzing', *Independent Security Evaluators, Techincal Report*, 2007.

[16] K. Claessen and J. Hughes, 'QuickCheck: A lightweight tool for random testing of Haskell programs', in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[17] J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson and H. R. Nielson, 'Effect-driven QuickChecking of compilers', *In Proceedings of the ACM on Programming Languages, Volume 1*, no. ICFP, 2017.

[18] 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*, 2008.

[19] T. Arts, J. Hughes, U. Norell and H. Svensson, 'Testing AUTOSAR software with QuickCheck', in *IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015.

[20] J. Hughes, U. Norell, N. Smallbone and T. Arts, 'Find more bugs with QuickCheck!', in *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2016.

[21] J. Hughes, 'QuickCheck testing for fun and profit', in *International Symposium on Practical Aspects of Declarative Languages*, Springer, 2007.

[22] L. Bulwahn, 'The new QuickCheck for isabelle', in *International Conference on Certified Programs and Proofs*, Springer, 2012.

[23] T. Arts, J. Hughes, J. Johansson and U. Wiger, 'Testing telecoms software with Quviq QuickCheck', in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, 2006.

[24] P. Holser, *Junit-quickcheck: Property-based testing, JUnit-style*, 2019.

[25] L. Pike, 'Smartcheck: Automatic and efficient counterexample reduction and generalization', in *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 2014.

[26] R. Braquehais and C. Runciman, 'Fitspec: Refining property sets for functional testing', in *Proceedings of the 9th International Symposium on Haskell, 2016*, 2016.

[27] P. Wadler and S. Blott, 'How to make ad-hoc polymorphism less ad hoc', in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1989.

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

[29] J. Duregård, P. Jansson and M. Wang, 'Feat: Functional enumeration of algebraic types', in *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2012.

[30] T. Sheard and S. L. P. Jones, 'Template meta-programming for Haskell', *SIGPLAN Notices*, vol. 37, 2002.

[31] *LibFuzzer: A library for coverage-guided fuzz testing.* `http://llvm.org/docs/LibFuzzer.html`, 2019.

[32] S. Dolan and M. Preston, 'Testing with crowbar', in *OCaml Workshop*, 2017.

[33] R. Kersten, K. Luckow and C. S. Păsăreanu, 'Poster: Afl-based fuzzing for java with Kelinci', in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[34] L. Lampropoulos, M. Hicks and B. C. Pierce, 'Coverage guided, property based testing', *Proceedings of the ACM on Programming Languages, (OOPSLA)*, 2019.

[35] M. Böhme, V.-T. Pham, M.-D. Nguyen and A. Roychoudhury, 'Directed greybox fuzzing', in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[36] X. Zhu and M. Böhme, 'Regression greybox fuzzing', in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[37] S. Österlund, K. Razavi, H. Bos and C. Giuffrida, 'Parmesan: Sanitizer-guided greybox fuzzing', in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020.

[38] C. Hritcu, J. Hughes, B. C. Pierce *et al.*, 'Testing noninterference, quickly', *ACM SIGPLAN Notices*, vol. 48, 2013.

[39] C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky *et al.*, 'Testing noninterference, quickly', *Journal of Functional Programming*, vol. 26, 2016.

[40] 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*, 2012.

[41] B. P. Miller, L. Fredriksen and B. So, 'An Empirical Study of the Reliability of UNIX Utilities', *ACM Communications*, vol. 33, 1990.

[42] K. Claessen and J. Hughes, 'QuickCheck: a lightweight tool for random testing of Haskell programs', *ACM SIGPLAN Notices*, vol. 46, 2011.

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

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

[45] M. Höschele and A. Zeller, 'Mining input grammars with AUTOGRAM', in *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.

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

[47] S. Marlow, *Haskell 2010 language report*, 2010.

[48] C. McBride and R. Paterson, 'Applicative programming with effects', *Journal of Functional Programming*, vol. 18, 2008.

[49] V. Berthoux, *Juicy.Pixels: Haskell library to load and save pictures*, `https://hackage.haskell.org/package/JuicyPixels`, 2012.

[50] Eric S. Raymond, *GIFLIB: A library and utilities for processing GIFs*, `http://giflib.sourceforge.net`, 1989.

[51] N. Nethercote and J. Seward, 'Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation', *SIGPLAN Notices*, vol. 42, 2007.

[52] K. Serebryany, D. Bruening, A. Potapenko and D. Vyukov, 'AddressSanitizer: A fast address sanity checker', in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[53] P. Hudak, 'Modular domain specific languages and tools', in *Proceedings of the 5th International Conference on Software Reuse (ICSR)*, 1998.

[54] J. Svenningsson and E. Axelsson, 'Combining deep and shallow embedding of domain-specific languages', *Computer Languages, Systems & Structures*, vol. 44, 2015.

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

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

[57] K. Claessen and J. Hughes, 'Testing monadic code with QuickCheck', *SIGPLAN Notices*, vol. 37, 2002.

[58] 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.

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

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

[61] Willem van Schaik, *The official test-suite for PNG*, `http://www.schaik.com/pngsuite/`, 2011.

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

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

[64] K. Claessen, J. Duregård and M. H. Palka, 'Generating constrained random data with uniform distribution', in *Proceedings of the Functional and Logic Programming FLOPS*, 2014.

[65] M. Pałka, K. Claessen, A. Russo and J. Hughes, 'Testing an optimising compiler by generating random lambda terms', in *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2011.

[66] A. Zeller and R. Hildebrandt, 'Simplifying and isolating failure-inducing input', *IEEE Transactions on Software Engineering*, vol. 28, 2002.

[67] J. Hughes, B. C. Pierce, T. Arts and U. Norell, 'Mysteries of Dropbox: Property-based testing of a distributed synchronization service', in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016.

[68] C. Runciman, M. Naylor and F. Lindblad, 'Smallcheck and Lazy Smallcheck: Automatic exhaustive testing for small values', in *Proceedings of the ACM SIGPLAN Symposium on Haskell*, 2008.

[69] N. Mitchell, 'Deriving generic functions by example', in *Proceedings of the 1st York Doctoral Syposium*, 2007.

[70] G. Grieco, M. Ceresa and P. Buiras, 'QuickFuzz: An automatic random fuzzer for common file formats', in *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2016.

[71] G. Grieco, M. Ceresa, A. Mista and P. Buiras, 'QuickFuzz testing for fun and profit', *Journal of Systems and Software*, vol. 134, 2017.

[72] L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce and L. Xia, 'Beginner's luck: A language for property-based generators', in *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, 2017.

[73] H. W. Watson and F. Galton, 'On the probability of the extinction of families', *The Journal of the Anthropological Institute of Great Britain and Ireland*, 1875.

[74] P. Haccou, P. Jagers and V. Vatutin, *Branching processes. Variation, growth, and extinction of populations*. Cambridge University Press, 2005, ISBN: 978-0-521-83220-5.

[75] L. Lampropoulos, Z. Paraskevopoulou and B. C. Pierce, 'Generating good generators for inductive relations', *In Proceedings ACM on Programming Languages (POPL)*, 2017.

[76] P. Duchon, P. Flajolet, G. Louchard and G. Schaeffer, 'Boltzmann samplers for the random generation of combinatorial structures', *Combinatorics, Probability and Computing*, vol. 13, 2004.

[77] M. Bendkowski, K. Grygiel and P. Tarau, 'Boltzmann samplers for closed simply-typed lambda terms', in *In Proceedings of the ACM International Symposium on Practical Aspects of Declarative Languages*, 2017.

[78] S. M. Poulding and R. Feldt, 'Automated random testing in multiple dispatch languages', *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[79] R. Feldt and S. Poulding, 'Finding test data with specific properties via metaheuristic search', in *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[80] F. Chan, T. Chen, I. Mak and Y. Yu, 'Proportional sampling strategy: Guidelines for software testing practitioners', *Information and Software Technology*, vol. 38, 1996.

[81] T. Y. Chen, H. Leung and I. K. Mak, 'Adaptive random testing', in *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, 2005.

[82] A. Arcuri and L. Briand, 'Adaptive random testing: An illusion of effectiveness?', in *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[83] I. Ciupa, A. Leitner, M. Oriol and B. Meyer, 'ARTOO: Adaptive random testing for object-oriented software', in *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, 2008.

[84] N. Balakrishnan, V. Voinov and M. Nikulin, *Chi-Squared Goodness of Fit Tests with Applications. 1st Edition*. Academic Press, 2013, ISBN: 9780123971944.

[85] A. Mista, A. Russo and J. Hughes, 'Branching processes for QuickCheck generators', in *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2018.

[86] C. Klein and R. B. Findler, 'Randomized testing in PLT Redex', in *ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2009.

[87] M. Bendkowski, O. Bodini and S. Dovgal, 'Polynomial tuning of multiparametric combinatorial samplers', in *Proceedings of the 15th Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, 2018.

[88] P. Godefroid, N. Klarlund and K. Sen, 'DART: Directed automated random testing', in *ACM Sigplan Notices*, vol. 40, 2005.

[89] A. Mista and A. Russo, 'Generating random structurally rich algebraic data type values', in *Proceedings of the 14th International Workshop on Automation of Software Test*, 2019.

[90] W. Swierstra, 'Data types à la carte', *Journal of Functional Programming*, vol. 18, 2008.

[91] T. Schrijvers, M. Sulzmann, S. P. Jones and M. Chakravarty, 'Towards open type functions for haskell', in *Proceedings of the 19th International Symposium on Implemantation and Application of Functional Languages*, 2007.

[92] R. A. Eisenberg, S. Weirich and H. G. Ahmed, 'Visible type application', in *European Symposium on Programming*, 2016.

[93] H. Kiriyama, H. Aotani and H. Masuhara, 'A lightweight optimization technique for data types a la carte', in *Companion Proceedings of the 15th International Conference on Modularity*, 2016.

[94] C. Okasaki, 'Red-black trees in a functional setting', *Journal of Functional Programming*, vol. 9, 1999.

[95] B. O'Sullivan, *Criterion: A haskell microbenchmarking library*, 2014. [Online]. Available: `http://www.serpentine.com/criterion/`.

[96] P. Wadler, *The expression problem*, 1998. [Online]. Available: `https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

[97] L. E. Day and G. Hutton, 'Compilation à la carte', in *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages (IFL)*, 2013.

[98] A. Persson, E. Axelsson and J. Svenningsson, 'Generic monadic constructs for embedded languages', in *International Symposium on Implementation and Application of Functional Languages (IFL)*, 2011.

[99] N. Wu, T. Schrijvers and R. Hinze, 'Effect handlers in scope', 2014.

[100] B. Delaware, B. C. d S Oliveira and T. Schrijvers, 'Meta-theory à la carte', in *ACM SIGPLAN Notices*, vol. 48, 2013.

[101] P. Hudak *et al.*, 'Building domain-specific embedded languages', *ACM Computing Surveys*, vol. 28, 1996.

[102] P. Wadler, 'Monads for functional programming', in *International School on Advanced Functional Programming*, 1995.

[103] J. Launchbury, 'Lazy imperative programming', in *ACM Workshop on State in Programming Languages*, 1993.

[104] E. Axelsson, K. Claessen, G. Dévai *et al.*, 'Feldspar: A domain specific language for digital signal processing algorithms', in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, 2010.

[105] T. Elliott, L. Pike, S. Winwood *et al.*, 'Guilt free ivory', in *ACM SIGPLAN Notices*, vol. 50, 2015.

[106] L. Pike, A. Goodloe, R. Morisset and S. Niller, 'Copilot: A hard real-time runtime monitor', in *International Conference on Runtime Verification*, 2010.

[107] M. Pickering, N. Wu and B. Németh, 'Working with source plugins', in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, 2019.

[108] A. Gill, *Dotgen: A simple interface for building .dot graph files.* 2008. [Online]. Available: `https://hackage.haskell.org/package/dotgen`.

[109] L. Erkok, *Sbv: SMT based verification: Symbolic Haskell theorem prover using SMT solving.* 2010. [Online]. Available: `https://hackage.haskell.org/package/sbv`.

[110] E. Axelsson, 'Compilation as a typed EDSL-to-EDSL transformation', *arXiv preprint arXiv:1603.08865*, 2016.

[111] A. Ekblad, *Shellmate: Simple interface for shell scripting in Haskell.* 2014. [Online]. Available: `https://hackage.haskell.org/package/shellmate`.

[112] S. Marlow, S. P. Jones *et al.*, *The Glasgow Haskell compiler*, 2004.

[113] S. L. P. Jones and A. M. Santos, 'A transformation-based optimiser for Haskell', *Science of Computer Programming*, vol. 32, 1998.

[114] C. V. Hall, K. Hammond, S. L. Peyton Jones and P. L. Wadler, 'Type classes in Haskell', *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, 1996.

[115] S. P. Jones, M. Jones and E. Meijer, 'Type classes: An exploration of the design space', in *Haskell Workshop*, 1997.

[116] M. Algehed, P. Jansson, S. H. Einarsdóttir and A. Gerdes, 'Saint: An API-generic type-safe interpreter', in *Trends in Functional Programming (TFP)*, 2019.

[117] B. Barras, S. Boutin, C. Cornes *et al.*, 'The Coq proof assistant reference manual: Version 6.1', 1997.

[118] G. Dévai, D. Leskó and M. Tejfel, 'The EDSL's struggle for their sources', in *Central European Functional Programming School*, 2013.

[119] M. Snoyman, *Developing web applications with Haskell and Yesod.* O'Reilly Media, Inc., 2012, ISBN: 978-1449316976.

[120] G. Giorgidze and H. Nilsson, 'Embedding a functional hybrid modelling language in Haskell', in *Symposium on Implementation and Application of Functional Languages (IFL)*, 2008.

[121] G. Mainland and G. Morrisett, 'Nikola: Embedding compiled GPU functions in Haskell', in *ACM SIGPLAN Notices*, vol. 45, 2010.

[122] G. Giorgidze, T. Grust, T. Schreiber and J. Weijers, 'Haskell boards the ferry', in *Symposium on Implementation and Application of Functional Languages (IFL)*, 2010.

[123] G. Mainland, 'Why it's nice to be quoted: Quasiquoting for Haskell', in *Proceedings of the ACM SIGPLAN workshop on Haskell*, 2007.

[124] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones and S. Weirich, 'Closed type families with overlapping equations', *ACM SIGPLAN Notices*, vol. 49, 2014.

[125] A. Russo, 'Functional pearl: Two can keep a secret, if one of them uses haskell', in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015.

[126] D. Leijen and E. Meijer, 'Domain specific embedded compilers', in *Proceedings of the 2nd Conference on Domain-Specific Languages*, 2000.

[127] J. Cheney and R. Hinze, 'Phantom types', Cornell University, Tech. Rep., 2003.

[128] R. Pucella and J. A. Tov, 'Haskell session types with (almost) no class', in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, 2008.

[129] D. Stefan, D. Mazières, J. C. Mitchell and A. Russo, 'Flexible dynamic information flow control in the presence of exceptions', *Journal of Functional Programming*, vol. 27, 2016.

[130] J. Bracker and A. Gill, 'Sunroof: A monadic dsl for generating javascript', in *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages*, 2014.

[131] M. P. Gissurarson, 'Suggesting valid hole fits for typed-holes (experience report)', in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, 2018.

[132] D. Vytiniotis, S. Peyton jones, T. Schrijvers and M. Sulzmann, 'Outsidein(x) modular type inference with local assumptions', *Journal of Functional Programming*, vol. 21, 2011.

[133] G. Team. 'The ghc-8.10.1 library Constraint module'. (2020), [Online]. Available: `https://hackage.haskell.org/package/ghc-8.10.1/docs/src/Constraint.html`.

[134] I. S. Diatchki, 'Improving haskell types with smt', in *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 2015.

[135] D. Otwani and R. A. Eisenberg, 'The thoralf plugin: For your fancy type needs', in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, 2018.

[136] T. Schrijvers, S. Peyton Jones, M. Chakravarty and M. Sulzmann, 'Type checking with open type functions', in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

[137] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis and J. P. Magalhães, 'Giving haskell a promotion', in *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2012.

[138] J. Breitner, R. A. Eisenberg, S. Peyton Jones and S. Weirich, 'Safe zero-cost coercions for haskell', in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2014.

[139] A. Serrano and J. Hage, 'Type error customization in ghc: Controlling expression-level type errors by type-level programming', in *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages (IFL)*, 2017.

[140] J. Peterson, 'Dynamic typing in haskell', Technical Report YALEU/DCS/RR-1022, Yale University, Department of Computer Science, Tech. Rep., 1993.

[141] M. Toro, R. Garcia and E. Tanter, 'Type-driven gradual security with references', *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 40, 2018.

[142] J. Hughes, 'Erlang/quickcheck', in *Ninth International Erlang/OTP User Conference, Älvsjö, Sweden. November 2003*, 2003.

[143] M. Papadakis and K. Sagonas, 'A proper integration of types and function specifications with property-based testing', in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, 2011.

[144] M. Dénès, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou and B. C. Pierce, 'Quickchick: Property-based testing for coq', in *The Coq Workshop*, 2014.

[145] J. Chen, J. Patra, M. Pradel *et al.*, 'A survey of compiler testing', *ACM Computing Surveys*, 2020.

[146] Á. Perényi and J. Midtgaard, 'Stack-driven program generation of webassembly', in *Asian Symposium on Programming Languages and Systems*, 2020.

[147] 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)*, 2011.

[148] C. Holler, K. Herzig and A. Zeller, 'Fuzzing with code fragments', in *21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012.

[149] W. G. Hatch, P. Darragh and E. Eide, *Xsmith software repository.* `https://www.flux.utah.edu/project/xsmith`, 2020.

[150] C. Boyapati, S. Khurshid and D. Marinov, 'Korat: Automated testing based on java predicates', *ACM SIGSOFT Software Engineering Notes*, vol. 27, 2002.

[151] L. Meertens, 'First steps towards the theory of rose trees', *CWI, Amsterdam*, 1988.

[152] J. A. Goguen and J. Meseguer, 'Security policies and security models', in *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 1982.

[153] A. Sabelfeld and A. Myers, 'Language-based information-flow security', *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003.

[154] A. Azevedo de Amorim, N. Collins, A. DeHon *et al.*, 'A verified information-flow architecture', *SIGPLAN Notices*, vol. 49, 2014.

[155] I. Rezvov, *wasm: WebAssembly Language Toolkit and Interpreter*, `https://hackage.haskell.org/package/wasm`, 2018.

[156] A. Haas, A. Rossberg, D. L. Schuff *et al.*, 'Bringing the web up to speed with webassembly', in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[157] W. M. McKeeman, 'Differential testing for software', *Digital Technical Journal*, vol. 10, 1998.

[158] A. Chou, J. Yang, B. Chelf, S. Hallem and D. Engler, 'An empirical study of operating systems errors', in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[159] A. Mista and A. Russo, *MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Structure-Preserving Mutations (Replication Package)*, version 0.3, 2022. DOI: 10.5281/zenodo.7197927. [Online]. Available: https://doi.org/10.5281/zenodo.7197927.

[160] M. Böhme, V.-T. Pham and A. Roychoudhury, 'Coverage-based greybox fuzzing as markov chain', *IEEE Transactions on Software Engineering*, vol. 45, 2017.

[161] S. Gan, C. Zhang, X. Qin *et al.*, 'Collafl: Path sensitive fuzzing', in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[162] N. Havrikov and A. Zeller, 'Systematically covering input structure', in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[163] H. L. Nguyen and L. Grunske, 'BeDivFuzz: Integrating behavioral diversity into generator-based fuzzing', in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[164] R. Padhye, C. Lemieux, K. Sen, M. Papadakis and Y. Le Traon, 'Semantic fuzzing with zest', in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.