Design and Implementation of a Coverage-Guided Ruby Fuzzer

Matt Schwager
matt.schwager@trailofbits.com
Trail of Bits
New York, NY, USA

Dominik Klemba
dominik.klemba@trailofbits.com
Trail of Bits
New York, NY, USA

Josiah Dykstra
josiah.dykstra@trailofbits.com
Trail of Bits
New York, NY, USA

ABSTRACT

The dynamic nature of cybersecurity threats necessitates the development of advanced tools capable of identifying vulnerabilities in software applications. Fuzz testing is a core method for cybersecurity research and practice because it exemplifies the scientific pursuit for proactive and secure computing environments through its emphasis on systematic experimentation and analysis. Ruby is a dynamic language used to build some of the leading websites and software applications. This paper presents Ruzzy, a coverage-guided fuzzer for Ruby, inspired by Google's Atheris and developed to address the lack of a modern fuzzing tool within the Ruby community. By integrating with the libFuzzer ecosystem and providing support for both pure Ruby code and Ruby C extensions, Ruzzy represents a significant advancement in automated security testing for Ruby applications. We describe the motivations behind Ruzzy's creation, its architecture, and its potential impact on improving the security posture of Ruby-based software.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Ruby, fuzzing, security engineering

ACM Reference Format:
Matt Schwager, Dominik Klemba, and Josiah Dykstra. 2024. Design and Implementation of a Coverage-Guided Ruby Fuzzer. In *Workshop on Cyber Security Experimentation and Test (CSET 2024), August 13, 2024, Philadelphia, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3675741.3675749

1 INTRODUCTION

Ruby's popularity for web development and the critical nature of web applications have underscored the need for robust security testing tools. Fuzzing, a dynamic testing method involving the injection of malformed or unpredictable data into systems, has proven effective in other programming environments but has been underutilized within the Ruby community due to a lack of suitable tools. Ruzzy aims to fill this gap by providing a coverage-guided fuzzing tool tailored for Ruby, drawing inspiration from successful fuzzers in other languages while addressing unique challenges in Ruby applications.

Ruby is a dynamic language that is particularly known for Ruby on Rails, used to build some of the leading websites and software applications including the Metasploit Framework. Ruby's elegant syntax makes it a popular among web developers, including at Hulu, Airbnb, and GitHub [7]. Further, because it is object-oriented, Ruby is ideal for both new and experienced software developers.

Fuzzing is an important and common testing methodology for evaluating application security by uncovering coding errors that can lead to security issues [16]. Fuzzing represents a dynamic testing method that inputs malformed or unpredictable data to a system to detect security issues, bugs, or system failures. It is a proactive approach to discover bugs during software development and maintenance.

As a method within the cybersecurity research domain, fuzzing exemplifies the scientific pursuit for more secure computing environments through its emphasis on systematic experimentation and analysis. It directly engages with principles of reliability, by generating repeatable testing scenarios that identify vulnerabilities; validity, through its capacity to accurately reflect the system's behavior under malformed inputs; and reproducibility, enabling researchers to verify findings across different contexts and systems. Moreover, fuzzing's adaptability allows for the transferability of methodologies and insights across various software and hardware platforms, enhancing the overall knowledge base of cybersecurity defenses. Ethical considerations are paramount, as fuzzing must be conducted with respect for privacy and without causing harm, ensuring that vulnerabilities are disclosed responsibly. Lastly, scalability challenges are addressed through the development of automated and distributed fuzzing systems that can handle the complexity and size of modern software projects. This intersection of scientific rigor and ethical responsibility positions fuzzing as a critical component in the evolution of cybersecurity research and practice, pushing the boundaries of what is possible in securing digital infrastructure against emerging threats.

For practitioners, fuzzing has become an indispensable technique in identifying security vulnerabilities, with numerous examples of its effectiveness across different programming languages. OSS-Fuzz, Google's open source fuzz testing service, claims that it has helped to find and fix over 10,000 security vulnerabilities and 36,000 bugs with fuzzing [6]. The Ruby community has lagged behind in adopting this methodology due to the absence of an accessible, well-maintained fuzzing tool.

Prior attempts at creating Ruby fuzzers have been hindered by maintenance, usability, and feature-completeness issues. Such examples include kisaten [8], afl-ruby [1], and FuzzBert [4]. Only

	Fuzzer	Pure Ruby Coverage	C Extension Coverage	Coverage-Guided
kisaten	AFL	Yes	No	Yes
afl-ruby	AFL	Yes	No	Yes
FuzzBert	custom	Yes	Yes	No
Ruzzy	libFuzzer	Yes	Yes	Yes

Table 1: Comparison of Ruby fuzzer features

FuzzBert appears to have support for C extensions. Ruzzy was developed with the intention of overcoming these limitations by leveraging the extensive ecosystem of libFuzzer. Table 1 shows a comparison of features across Ruby fuzzers. Because fuzzing helps find bugs in software that processes untrusted input, in pure Ruby this may result in unexpected exceptions leading to denial-of-service. The ability to fuzz C extensions allows greater use cases than Ruby alone, particularly because C code could contain high-impact memory corruption bugs relevant to potential exploitation [18].

In this preliminary work we describe our contribution to improve the scientific approach to cybersecurity through Ruby fuzz testing. In Section 2 we describe the motivation and implementation details behind Ruzzy. Section 3 presents use cases and evaluation, and we present future work in Section 4.

2 RUZZY DESIGN OVERVIEW

Ruzzy was inspired by Atheris, a Python fuzzer developed by Google [5]. Having previously used Atheris and found it useful for fuzzing pure Python and C extensions, we took a similar approach. Like Atheris, Ruzzy also aims to have a streamlined user experience that does not require alterations to the underlying project being fuzzed. Ruzzy, like Atheris, also uses libFuzzer for its coverage instrumentation and fuzzing engine. We considered using AFL++ but found less community support and adoption. Ruzzy is available as a free and open-source project.[1]

Ruzzy was built on three practice-oriented goals, which are explained further in this section:

(1) Fuzz both pure Ruby code and Ruby C extensions.
(2) Make fuzzing easy for users by providing a RubyGems installation process and straightforward interface.
(3) Integrate with the extensive libFuzzer ecosystem.

Ruzzy was built with mature software development practices including automated unit tests that help to ensure specific functionality works as development continues. This increases confidence that current and future developers do not break the software. Automated tests also keep integrity and development standards high for external contributors. Today, Ruzzy has automated testing for LLVM versions 15-18 and Ruby versions 3.0-3.3.

2.1 Support for Pure Ruby and Ruby C Extensions

The first goal was support for pure Ruby code and Ruby C extensions. Pure Ruby is Ruby code without external dependencies. This refers to code written purely in Ruby, without relying on any libraries or frameworks written in other languages such as C or C++. This type of code can be more difficult to write and maintain for complex applications, but it can also be more portable and easier to understand.

Ruby C extensions are a way to integrate the power and speed of C code into Ruby programs. C is generally much faster than Ruby and there may exist a powerful C library that the developer wants to utilize in a Ruby application. So, a Ruby program can call a C function directly from Ruby code.

2.2 Installation and Use

Ruzzy has a straightforward installation using the RubyGems package manager: `gem install ruzzy`.

Fuzzing pure Ruby code requires two Ruby scripts: a tracer script and a fuzzing harness. The tracer script is required due to an implementation detail of the Ruby interpreter. Every tracer script will look nearly identical. Example scripts can be seen in Appendix A.

However, a problem arises when we consider that libFuzzer is written in C/C++. When using libFuzzer as a library, we need to pass a C function pointer to `LLVMFuzzerRunDriver` to initiate the fuzzing process. We discovered that the `rb_proc_call` C extension function could be used to pass arbitrary Ruby code to a C/C++ library, thereby bridging the gap between Ruby code and the libFuzzer C/C++ implementation.

An example of fuzzing Ruby C extensions can be seen in Appendix B.

2.3 Integration with libFuzzer

libFuzzer is a coverage-guided fuzzing engine that is part of the LLVM project [10]. It is linked with the library being tested, and feeds fuzzed inputs to the library through a specific fuzzing entry point. The fuzzer then tracks which areas of the code are reached and generates mutations on the corpus of input data in order to maximize the code coverage. We have adopted this fitness function to generate inputs learned during execution that maximize the chances of discovering bugs or security issues in the software being tested.

2.4 Coverage-Guided Fuzzing

Coverage refers to knowing which parts of a Ruby program's code have been executed during the fuzzing process. This analysis helps guide the generation of new test inputs in order to explore different code paths that have not yet been covered. By tracking which parts of the code have been executed (i.e., covered) during fuzzing, developers can gain insights into the effectiveness of their testing and identify areas of the code that may need further scrutiny. There is a very strong correlation between the code coverage achieved by a fuzzer and the number of bugs it detects, making coverage an effective proxy measure for fuzzer effectiveness [3].

Ruzzy achieves code coverage differently depending on whether the input is pure Ruby or uses C extensions. For pure Ruby code we rely on the Ruby `Coverage` module. Ruzzy passes Ruby coverage information to libFuzzer in real time as the fuzzing engine executes. When fuzzing Ruby C extensions, however, Clang provides coverage instrumentation for C code. Ruzzy supports `AddressSanitizer` [14] and `UndefinedBehaviorSanitizer` [11] when fuzzing C extensions. `AddressSanitizer` is a well-supported LLVM sanitizer that

[1]https://github.com/trailofbits/ruzzy

helps developers find memory errors and corruption bugs, making it appropriate for fuzzing. It is commonly used in other fuzzers including Atheris and in software development testing when running unit and integration tests. `UndefinedBehaviorSanitizer` is another LLVM sanitizer used to identify undefined behavior in C/C++. Undefined behavior can lead to unexpected software crashes, application vulnerabilities, and a general reduction of application robustness. Detecting undefined behavior and its resulting deficiencies is a typical subject of static analysis and fuzzing.

Fuzzers often rely on a control-flow graph (CFG) to reason about coverage gathering. libFuzzer offers three instrumentation points based on a CFG: edge, basic block, and function entry [15]. Ruby's `Coverage` module was not built with these in mind as it does not emit branch information in a way that is consistent with libFuzzer's instrumentation points. As a result, Ruzzy interfaces between Ruby `Coverage` events and libFuzzer's coverage instrumentation expectations. Ruzzy does this by hooking branching events and emitting them as if they were basic blocks. Ideally, Ruby would emit coverage events such that a CFG could be constructed but this is not the case today. Such events would include both basic blocks and edges, the building blocks of a CFG. Instrumenting function entry points will likely yield the least memory usage but also the worst fuzzing performance. However, if this data was presented by the Ruby standard library then Ruzzy could test various instrumentation points against one other to maximize fuzzing performance. Fuzzing performance is not simply maximizing coverage; it is better to maximize coverage per unit time. As one AFL developer noted, "It's genuinely hard to compete with brute force when your 'smart' approach is resource-intensive. If your instrumentation makes it 10x more likely to find a bug, but runs 100x slower, your users are getting a bad deal" [17].

Another consideration is how to store coverage information. Ruzzy uses an in-memory buffer to track coverage counters. Counters are indexed inside the buffer by a two-tuple of file path and line number. When a branch is executed at a specific file path and line number, the corresponding counter is incremented. This is how Ruzzy leverages libFuzzer to track coverage information. However, this is not ideal because it does not uniquely identify branches in Ruby. For example, there can be multiple 'if' statements on a single line which produces the same counter tuple and thus increments the same counter for multiple branches. Branches should instead by uniquely identifiable.

The design for Ruzzy further required consideration for how to store potentially very large numbers of branch counters in a finite amount of memory. This is a traditional trade-off between space and time. In other words, allocating a larger in-memory buffer allows you to track more unique branch events, but comes at the cost of additional memory usage and increased time to search the buffer. Furthermore, there is a finite amount of memory, so considerations must be made regarding how large the buffer should be. Ruzzy initially chose an arbitrary number of 8,192 unique counter values. When the number of unique branches exceeds this value it performs a modulo operation and rolls back to the beginning of the buffer. This trades off fuzzing performance for memory size and allows the fuzzer to encounter an infinite number of branches and continue fuzzing. Improvements to this design should be considered.

3 EVALUATION AND USE CASES

Building the tool is just the beginning; the real value comes when it starts to find bugs. In this section we describe how Ruzzy performed during development and then we present additional use cases.

3.1 Evaluation and Preliminary Results

Fuzzing tools can be formally evaluated in several dimensions. The number of distinct bugs a tool identifies is a common measure of effectiveness, and one approach to comparing fuzzing tools [9].

During development, we tested Ruzzy on GitHub repositories found to use Ruby C extensions. Our search in GitHub for active Ruby projects that includes an extension ("lang:ruby path:extconf.rb NOT is:archived") revealed 7,600 files as of May, 2024. From this list, we fuzzed 15 target libraries based on a convenience sample of parsers, decoders, and deserializers (see Appendix C). These types of libraries commonly contain complex logic and receive attacker-controlled input which makes them potentially vulnerable. Each of our target files was fuzzed once with a 20 minute timeout. This produced four distinct bugs during our testing which were disclosed to their respective developers.[2]

3.2 Potential Use Cases

One use case that Ruzzy already supports is round-trip fuzzing. Round-trip fuzzing is a technique that tests the robustness of software by generating input data, having the program encode it, capturing the output, having the program decode it, and then ensuring the decoded output is equivalent to the original input. This approach can be useful for detecting logic and consistency bugs as opposed to software crashes.

Another Ruzzy use case is "continuous fuzzing." This is similar to the concept of continuous integration (CI) in software engineering. The goal of continuous fuzzing is to continuously fuzz appropriate parts of a library or application. This is typically done as a CI job, similar to a project's unit or integration tests. But this raises some questions for practitioners: How long should one fuzz a specific commit or merge request before deeming it acceptable to enter the codebase? What parts of a codebase should be fuzzed and which should not? These questions often intersect with some more academic in nature, such as: What is the optimal amount of time to fuzz a target before you get diminishing returns in finding bugs? When should one re-evaluate an evolutionary fuzzing algorithm or coverage gathering system versus simply spending more time fuzzing?

Continuous fuzzing via tools such as `CIFuzz` and `ClusterFuzzLite` and projects such as Google's OSS-Fuzz, have been proven a success. Finding and helping fix tens of thousands of bugs is no small feat. This concept of continuous fuzzing and projects such as OSS-Fuzz consider fuzzing on a more macroscopic scale. In other words, Ruzzy combined with OSS-Fuzz could have ecosystem-wide impact within the Ruby community. Bringing a simple, developer-friendly fuzzing tool and applying it at scale using concepts like continuous fuzzing and OSS-Fuzz open the door for improving the security and assurance properties of the Ruby ecosystem as a whole. A similar outcome was achieved for Python with Atheris, OSS-Fuzz, and `CIFuzz` [13].

[2]https://github.com/trailofbits/ruzzy#trophy-case

4 CONCLUSION AND FUTURE WORK

In this preliminary work paper, we have described the motivation, design, and early results from a new Ruby fuzzer. Ruzzy supports coverage-guided fuzzing for both pure Ruby code and Ruby C extensions. Motivated and grounded by both reserach and practice, Ruzzy is designed for ease of use and continued development and iteration.

The limitations in Ruzzy's code coverage was intentional as a minimum-viable product and constrained by the language and tools. Nevertheless, the work was grounded in our experience as practitioners and we achieved an initial product that was functional for community use and expansion. In the future, we plan to formally benchmark and optimize code coverage for comparison with existing and future tools [2]. We have already begun working with the Ruby community to enable Ruby C extensions to more easily process branch coverage events in real-time. Today, Ruby has rudimentary coverage instrumentation and Ruzzy relies on internal, unofficial APIs. If adopted, however, Ruzzy would improve ability to gather coverage information.

Ruzzy lays the foundation for further research in several areas. First, its accessibility can be improved by integration with established frameworks such as OSS-Fuzz. This would streamline the workflow for developers by allowing them to leverage existing test suites and seamlessly incorporate fuzzing into their development process. Additionally, such integration would enable the use of existing test oracles to assess the correctness of execution paths discovered during fuzzing.

Second, Ruzzy's capabilities can be extended through advanced input generation strategies. These might include grammar-based fuzzing to target specific functionalities, protocol-aware fuzzing for applications interacting with external protocols, and taint tracking to prioritize fuzzing inputs that handle sensitive data. Such advancements would empower Ruzzy to identify a wider range of deeper bugs and vulnerabilities within Ruby code.

Finally, usability testing is necessary to evaluate how effectively software developers and vulnerability researchers can integrate Ruzzy into their workflows [12]. This would provide valuable insight to guide further improvements to the tool's design and documentation. By addressing these areas of future work, Ruzzy has the potential to become an indispensable tool for safeguarding Ruby applications.

In conclusion, Ruzzy exemplifies a pioneering step in advancing Ruby application security, showcasing the pivotal role of coverage-guided fuzzing in uncovering and addressing critical vulnerabilities. This tool not only elevates the security posture of Ruby-based systems but also sets a framework for future innovations in secure software development practices. As we navigate an increasingly complex cybersecurity landscape, the adoption and continual enhancement of tools like Ruzzy are essential, ensuring that the cybersecurity community remains at the forefront of technological resilience and defense.

REFERENCES

[1] afl-ruby. 2022. *A minimal ruby gem to allow fuzzing native ruby code with afl.* Retrieved March 15, 2024 from https://github.com/richo/afl-ruby
[2] Jim Alves-Foss, Aditi Pokharel, Ronisha Shigdel, and Jia Song. 2023. Calibrating Cybersecurity Experiments: Evaluating Coverage Analysis for Fuzzing Benchmarks. In *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA).* IEEE, 251–257.
[3] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering.* 1621–1633.
[4] Martin Boßlet. 2013. *FuzzBert.* Retrieved March 15, 2024 from https://github.com/krypt/FuzzBert
[5] Google. 2022. *A Coverage-Guided, Native Python Fuzzer.* Retrieved March 15, 2024 from https://github.com/google/atheris
[6] Google. 2023. *OSS-Fuzz.* Retrieved April 26, 2024 from https://google.github.io/oss-fuzz/#trophies
[7] Michael Hartl. 2022. *Ruby on Rails Tutorial: Learn Web Development with Rails, 7th Edition.* Addison-Wesley Professional.
[8] kisaten. 2018. *Ruby MRI extension for fuzzing Ruby code with afl-fuzz.* Retrieved March 15, 2024 from https://github.com/twistlock/kisaten
[9] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security.* 2123–2138.
[10] libFuzzer. 2024. *libFuzzer - a library for coverage-guided fuzz testing.* Retrieved March 15, 2024 from https://llvm.org/docs/LibFuzzer.html
[11] LLVM developers. 2024. *UndefinedBehaviorSanitizer.* Retrieved April 26, 2024 from https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[12] Stephan Plöger, Mischa Meier, and Matthew Smith. 2023. A Usability Evaluation of AFL and libFuzzer with CS Students. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* 1–18.
[13] Matt Schwager. 2024. Continuously fuzzing Python C extensions. Retrieved May 16, 2024 from https://blog.trailofbits.com/2024/02/23/continuously-fuzzing-python-c-extensions/
[14] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12).* 309–318.
[15] The Clang Team. 2024. SanitizerCoverage - Clang 10.0.0.git Documentation. Retrieved May 16, 2024 from https://clang.llvm.org/docs/SanitizerCoverage.html#instrumentation-points
[16] Trail of Bits. 2024. *Testing Handbook.* Retrieved March 15, 2024 from https://appsec.guide/
[17] Michał Zalewski. [n. d.]. Historical notes. Retrieved May 16, 2024 from https://lcamtuf.coredump.cx/afl/historical_notes.txt
[18] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.

A EXAMPLE WITH PURE RUBY CODE

A.1 Example tracer script, test_tracer.rb

```
require 'ruzzy'
Ruzzy.trace('test_harness.rb')
```

A.2 Example fuzzing harness, test_harness.rb

```
require 'ruzzy'

def fuzzing_target(input)
  if input.length == 4
    if input[0] == 'F'
      if input[1] == 'U'
        if input[2] == 'Z'
          if input[3] == 'Z'
            raise
          end
        end
      end
    end
  end
end

test_one_input = lambda do |data|
  fuzzing_target(data) # Your fuzzing target would go here
  return 0
end

Ruzzy.fuzz(test_one_input)
```

A.3 Running Ruzzy to produce a crash from the test harness

```
$ LD_PRELOAD=$(ruby -e 'require "ruzzy"; print Ruzzy::ASAN_PATH') ruby test_tracer.rb

INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2311041000
...
/app/ruzzy/bin/test_harness.rb:12:in `block in ': unhandled exception
    from /var/lib/gems/3.1.0/gems/ruzzy-0.7.0/lib/ruzzy.rb:15:in `c_fuzz'
    from /var/lib/gems/3.1.0/gems/ruzzy-0.7.0/lib/ruzzy.rb:15:in `fuzz'
    from /app/ruzzy/bin/test_harness.rb:35:in `'
    from bin/test_tracer.rb:7:in `require_relative'
    from bin/test_tracer.rb:7:in `
'
...
SUMMARY: libFuzzer: fuzz target exited
MS: 1 CopyPart-; base unit: 24b4b428cf94c21616893d6f94b30398a49d27cc
0x46,0x55,0x5a,0x5a,
FUZZ
artifact_prefix='./'; Test unit written to ./crash-aea2e3923af219a8956f626558ef32f30a914ebc
Base64: RlVaWg==
```

B EXAMPLE WITH RUBY C EXTENSIONS

A test file (dummy.c) is available at https://github.com/trailofbits/ruzzy/blob/main/ext/dummy/dummy.c

B.1 Running Ruzzy to produce a crash from dummy.c

```
$ export ASAN_OPTIONS="allocator_may_return_null=1:detect_leaks=0:use_sigaltstack=0"

$ LD_PRELOAD=$(ruby -e 'require "ruzzy"; print Ruzzy::ASAN_PATH') \
    ruby -e 'require "ruzzy"; Ruzzy.dummy'

INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2527961537
...
==45==ERROR: AddressSanitizer: heap-use-after-free on address 0x50c0009bab80 at pc 0xffff99ea1b44 bp
0xffffce8a67d0 sp 0xffffce8a67c8
...
SUMMARY: AddressSanitizer: heap-use-after-free /var/lib/gems/3.1.0/gems/ruzzy-0.7.0/ext/dummy/dummy.c:18:24
in _c_dummy_test_one_input
...
==45==ABORTING
MS: 4 EraseBytes-CopyPart-CopyPart-ChangeBit-; base unit: 410e5346bca8ee150ffd507311dd85789f2e171e
0x48,0x49,
HI
artifact_prefix='./'; Test unit written to ./crash-253420c1158bc6382093d409ce2e9cff5806e980
Base64: SEk=
```

C FUZZ TARGETS

```
https://github.com/cabo/cbor-ruby
https://github.com/emancu/toml-rb
https://github.com/flori/json
https://github.com/jgarber/redcloth
https://github.com/k0kubun/hamlit
https://github.com/kgiszczak/tomlib
https://github.com/mongodb/bson-ruby
https://github.com/msgpack/msgpack-ruby
https://github.com/ohler55/ox
https://github.com/puma/puma/tree/master/ext/puma_http11
https://github.com/ruby/erb
https://github.com/ruby/openssl
https://github.com/ruby/rdoc
https://github.com/ruby/zlib
https://github.com/slim-template/slim
```