RUG: Turbo LLM for Rust Unit Test Generation

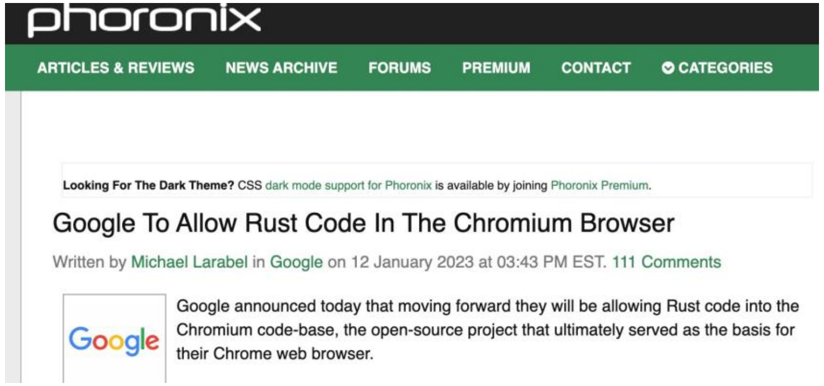
Xiang Cheng, Fan Sang, Yizhuo Zhai, Xiaokuan Zhang*, and Taesoo Kim
Georgia Institute of Technology, *George Mason University

Content

- Motivating
- Challenge & Insights
- Solution
  - LLM
  - Fuzzing
- Evaluation

Rust's Adoption is very Fast

- From DARPA/Microsoft/Linux/Google: use Rust to ensure memory safety


phoronix

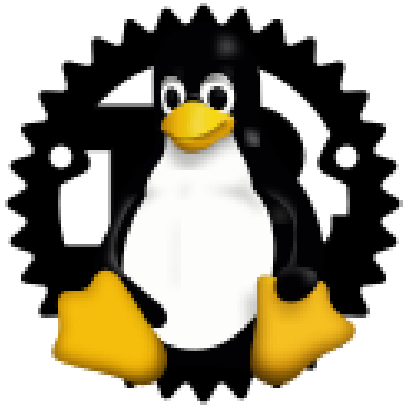
ARTICLES & REVIEWS NEWS ARCHIVE FORUMS PREMIUM CONTACT CATEGORIES

Looking For The Dark Theme? CSS dark mode support for Phoronix is available by joining Phoronix Premium.

Google To Allow Rust Code In The Chromium Browser

Written by Michael Larabel in Google on 12 January 2023 at 03:43 PM EST. 111 Comments

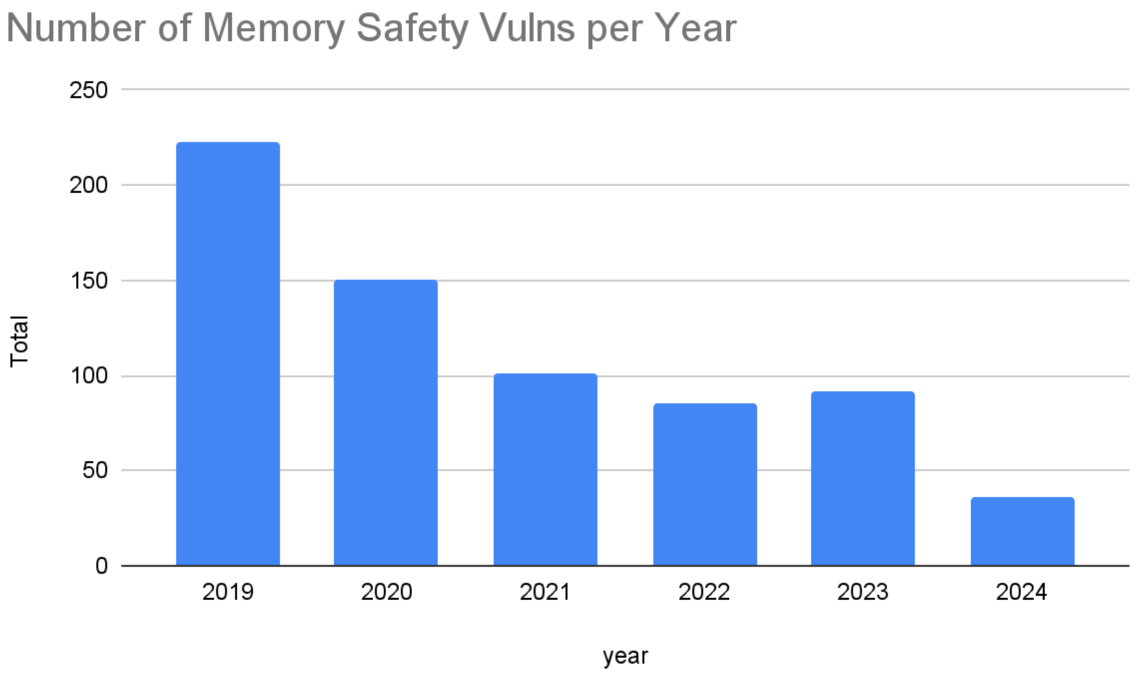
Google announced today that moving forward they will be allowing Rust code into the Chromium code-base, the open-source project that ultimately served as the basis for their Chrome web browser.

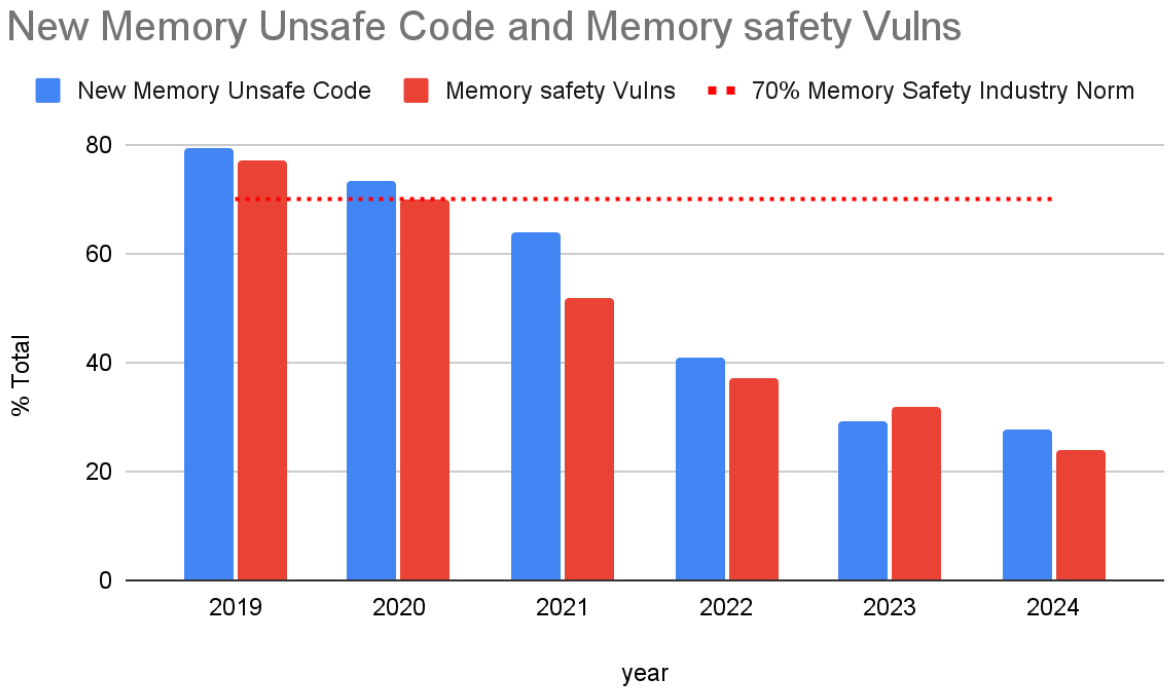


Picture: from chrome/windows/android/Rust-for-linux homepage

From Android's Practice

- AOSP starts to use Rust in 2019
- The total number of memory safety errors starts to drop even with unsafe Rust





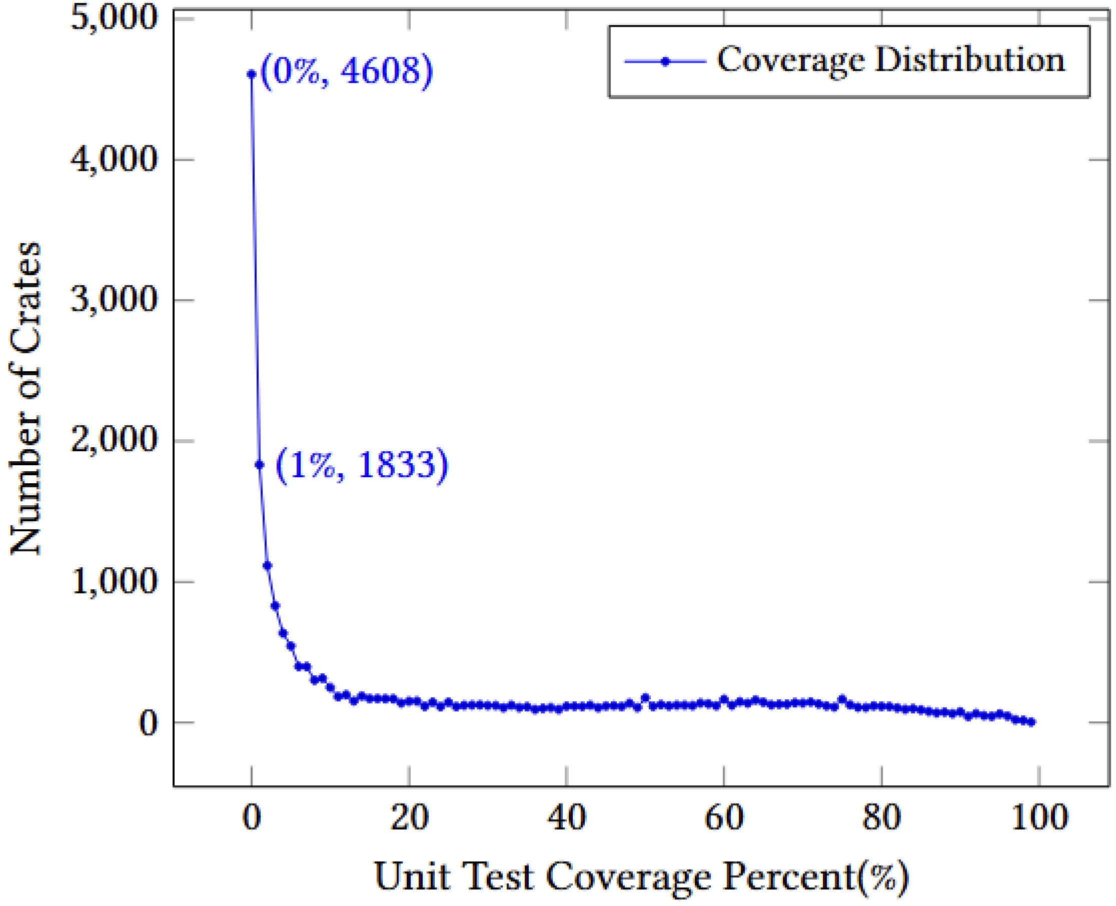
Picture: [Google Online Security Blog: Eliminating Memory Safety Vulnerabilities at the Source]

Rust Unit Testing

- Unit test is a type of software test that focuses on testing individual units or components of a program in isolation.
- Rust has a good support to unit test:
  - Annotations #[test] to mark a region as test
  - Temporally break encapsulation when testing in the same file
  - Provide driver build for all the tests:
    - cargo test – run all tests in the target repo

Unit testing status of Rust

- Crates.io is like npm/maven/pip, serves as the dependency manager in Rust
  - As of Dec 2024, serves 164k crates, and more than 96B downloads
- We evaluate the top 30K popular crates
  - 60% crates have less than 10% test coverage
  - 16% crates doesn't have any tests



Content

- Motivating
- Challenge & Insights
- Solution
  - LLM
  - Fuzzing
- Evaluation

Why unit test is difficult

- Unit test requires to build a minimal context to trigger the target function, it's not easy to implement in some cases
  - Rust is applied for low-level systems: embedding device, OS driver…
- Developers are tired of writing unit tests:
  - E.g. 3 conditions + 2 loops to test

```rust
pub fn unfill(text: &str) -> (String, Options<'_>) {
    let prefix_chars: &[_] = &[' ', '-', '+', '*', '>', '#', '/'];

    let mut options = Options::new(0);
    for (idx, line) in text.lines().enumerate() {
        options.width = std::cmp::max(options.width, display_width(line));
        let without_prefix = line.trim_start_matches(prefix_chars);
        let prefix = &line[..line.len() - without_prefix.len()];

        if idx == 0 {
            options.initial_indent = prefix;
        } else if idx == 1 {
            options.subsequent_indent = prefix;
        } else if idx > 1 {
            for ((idx, x), y) in prefix.char_indices().zip(options.subsequent_indent.chars()) {
                if x != y {
                    options.subsequent_indent = &prefix[..idx];
                    break;
                }
            }
            if prefix.len() < options.subsequent_indent.len() {
                options.subsequent_indent = prefix;
            }
        }
    }
```

Insight: LLM + Fuzzing

- Chain-of-thought is proven to be effective for LLM reasoning
  - => RUG uses type dependencies to automatically apply chain of thought
- The result of each subproblem needs to be verified, otherwise the error will accumulate
  - => RUG verifies the result for each subproblem, ensuring the final correctness
- The LLM isn’t good at exploring program paths
  - => RUG leverages fuzzing as post-processor to extend the testing coverage

Georgia Tech

Content

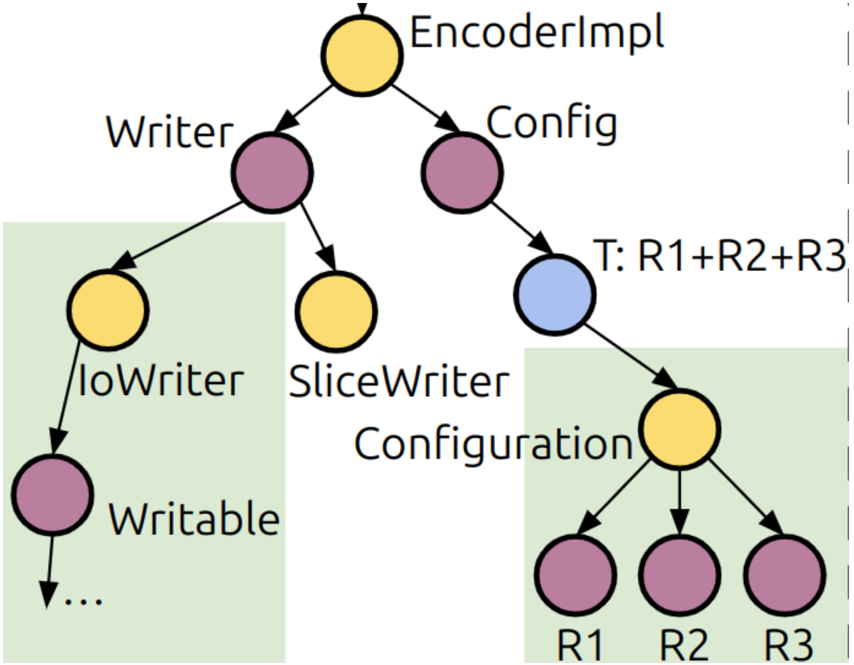
- Motivating
- Challenge & Insights
- Solution
  - LLM
  - Fuzzing
- Evaluation

Semantic Aware Decomposition

- Challenge: to build the testing context, LLM is expected to correctly built all its necessary dependents in one shot, leading to errors
- Semantic aware decomposition: Type has dependencies, RUG automatically decomposes the context building into sub-problems

```
fn encode<E: Encoder> (&self :char, encoder: E)
    -> Result<EncodeError> // target testing function
// def for EncoderImpl
pub struct EncoderImpl<W, C: Config>
// impl for Encoder trait
impl<W: Writer, C: Config> Encoder for EncoderImpl
// impls for Writer Trait
impl Writer for SliceWriter
impl enc::write::Writer for IoWriter
// impls for Config Trait
pub trait Config: R1 + R2 + R3 {}
impl<T> Config for T where T: R1 + R2 + R3
// def for Configuration, impls R1, R2, R3
pub struct Configuration<A = R1, B = R2, C = R3>
```

```rust
fn encode<E: Encoder> (&self :char, encoder: E)
    -> Result<EncodeError> // target testing function
// def for EncoderImpl
pub struct EncoderImpl<W, C: Config>
// impl for Encoder trait
impl<W: Writer, C: Config> Encoder for EncoderImpl
// impls for Writer Trait
impl Writer for SliceWriter
impl enc::write::Writer for IoWriter
// impls for Config Trait
pub trait Config: R1 + R2 + R3 {}
impl<T> Config for T where T: R1 + R2 + R3
// def for Configuration, impls R1, R2, R3
pub struct Configuration<A = R1, B = R2, C = R3>
```



- Build type dependency graph as G = {E, V}, V denotes type entities and E denotes dependency relations, G is directed
- Divide the context generation into sub-problems and resolve them individually
- Rely on static analysis to guide and verify the LLM output

Pros and Corner Cases

- Semantic Aware Decomposition
  - We minimize the context for each iteration (only the direct dependent instances in the green square)
  - We verify the result of each iteration output to ensure the correctness
  - We memorize the result based on type and saves tokens in project scope
- Corner Cases:
  - Cycles: we will randomly decide the order and use natural language description of the dependencies as context
  - LLM failed in the middle: we will mark the node as unfinished and continue, using natural language description of dependencies
  - Candidate selection: our evaluation shows its influence is limited for different strategies

Fuzzing as post-processing

- Fuzzers are efficient to explore different paths in few seconds
- We build a harness transformation program based on Rustc to convert the generated tests into fuzzing harnesses
- To control the number of unit tests, we propose a greedy selection and ranking algorithm to select the fuzzing inputs

Evaluation

- How efficient is our approach compared with traditonal Rust/unit testing tools?
- How efficient is our problem division compared with other LLM based tools?
- How efficient is fuzzing?
- Can RUG be applied in real-world software development?

https://github.com/cxworks/rug

RUG vs Synthesizing Tools

- RustyUnit: SBST in Rust
- SyRust: SAT based Rust synthesizer
- RUG: using GPT-3.5

Crate	**Func**	**Region**	**Func**	**Region**
	RustyUnit		RUG	
gamie	55.54%	30.79%	68.67%	72.24%
humantime	45.55%	26.67%	50.33%	64.92%
lsd	32.58%	40.23%	37.66%	43.98%
quick-xml	17.38%	24.61%	54.5%	62.76%
tight	24.70%	30.27%	32.24%	36.90%
time	75.26%	70.78%	68.13%	56.94%
mean	**37.23%**	**34.70%**	**49.96%**	**54.84%**
	SyRust		RUG	
data-structure	26.11%	31.19%	52.10%	56.03%
encoding	30.69%	28.51%	55.47%	48.54%
mean	**28.40%**	**30.65%**	**53.79%**	**52.28%**

RUG vs Synthesizing Tools

- RustyUnit: SBST in Rust
- SyRust: SAT based Rust synthesizer
- RUG: using GPT-3.5
- RUG out-performs both tools for 20.14% and 21.63%

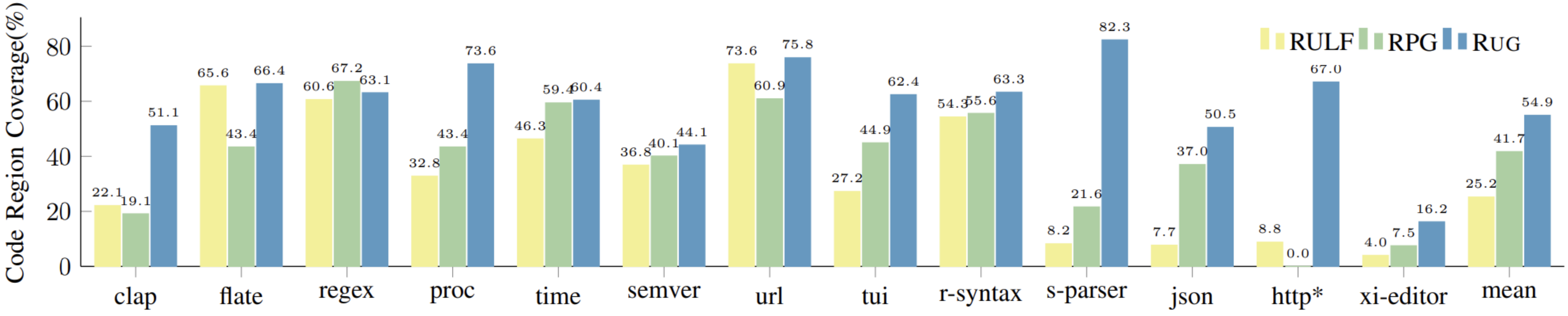
Crate	Func	Region	Func	Region
	RustyUnit		RUG	
gamie	55.54%	30.79%	68.67%	72.24%
humantime	45.	[illegible]	[illegible]	%
lsd	32.	[illegible]	[illegible]	%
quick-xml	17.	[illegible]	[illegible]	%
tight	24.	[illegible]	[illegible]	%
time	75.2	[illegible]	[illegible]	4%
mean	**37.23%**	**34.70%**	**49.96%**	**54.84%**
	SyRust		RUG	
data-structure	26.1	[illegible]	[illegible]	3%
encoding	30.	[illegible]	[illegible]	%
mean	**28.**	[illegible]	[illegible]	**%**

RUG improves RustyUnit coverage by 20.14%

RUG improves SyRust coverage by 21.63%

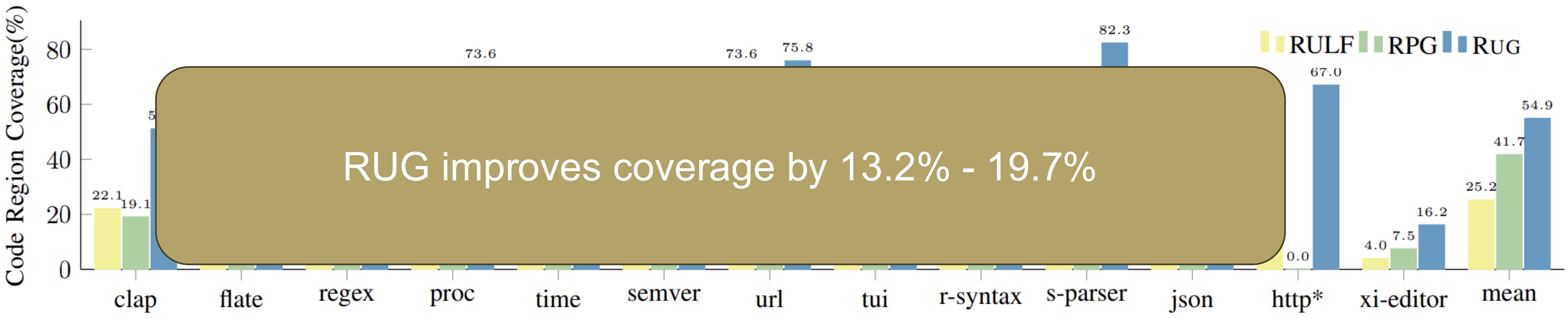
RUG vs Fuzzing based Testing

- RULF: type dependency based fuzzing harness generator for Rust
- RPG: improved RULF with harness selection algorithm
- RUG achieves 54.9% code coverage, while RULF as 25.2% and RPG has 41.7%



RUG vs Fuzzing based Testing

- RULF: type dependency based fuzzing harness generator for Rust
- RPG: improved RULF with harness selection algorithm
- RUG: using GPT-3.5
- RUG achieves 54.9% code coverage, while RULF as 25.2% and RPG has 41.7%



RUG vs LLM Approach

- Base: RUG improves baseline for 21.81% in GPT-3.5 and 10.20% in GPT-4
- Sensitivity test of RUG for: GPT model/fuzzing/problem decomposition

Crate Name (Downloads)	Tests ac/rej	GPT-3.5				GPT-4					Human Test Coverage
		Base		RUG		Base		RUG		Newly API Cov Rate	
		w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing		
bincode(49M)	4/0	1.57%	1.57%	22.92%	23.91%	16.63%	18.79%	44.67%	47.91%	74.11%	**64.58%**
chrono(128M)	22/13	37.88%	44.07%	47.2%	58.29%	54.04%	59.24%	56.90%	62.67%	73.05%	**76.66%**
hashes(266M)	**P**(7)	43.84%	43.84%	68.28%	68.28%	57.71%	57.71%	68.96%	**85.16%**	61.41%	**85.17%**
humantime(98M)	**P**(5)	63.09%	64.40%	67.02%	75.39%	74.08%	75.92%	74.61%	**80.37%**	40.00%	79.32%
itoa(221M)	1/0	26.00%	26.00%	82.00%	96.00%	96.00%	98.00%	100.00%	**100.00%**	83.33%	86.00%
json(203M)	-	28.10%	35.69%	44.60%	52.07%	62.26%	67.00%	70.25%	70.49%	47.33%	**72.36%**
mio(145M)	-	20.47%	20.47%	25.20%	25.20%	26.77%	26.77%	33.86%	**33.86%**	38.89%	24.19%
nom(114M)	6/1/**P**(14)	25.81%	25.81%	39.93%	40.04%	51.13%	51.17%	53.84%	53.87%	28.64%	**76.20%**
num-traits(185M)	-	36.02%	36.47%	43.20%	43.95%	46.94%	46.94%	47.23%	47.98%	90.36%	**50.58%**
demangle(93M)	**P**(14)	21.32%	21.62%	21.83%	65.99%	20.00%	74.82%	26.60%	**76.55%**	18.75%	72.25%
crc32fast(104M)	-	62.35%	64.71%	70.59%	71.76%	87.06%	88.24%	87.06%	**88.24%**	92.86%	68.24%
ryu(185M)	0/3	52.51%	95.28%	61.65%	97.64%	76.40%	99.42%	81.72%	**99.42%**	100.00%	87.85%
semver(168M)	18/0	61.40%	62.96%	62.54%	73.36%	72.36%	74.64%	74.22%	76.50%	95.24%	**84.33%**
textwrap(134M)	1/0	88.84%	92.56%	90.15%	94.31%	92.78%	94.75%	92.56%	**94.97%**	83.34%	87.53%
time(200M)	**P**(3)	33.08%	35.06%	48.98%	51.72%	55.34%	55.34%	79.89%	79.89%	66.06%	**96.48%**
toml(125M)	-	32.43%	37.06%	47.28%	49.02%	59.58%	64.40%	38.90%	38.90%	25.14%	**70.81%**
uuid(108M)	1/0	58.66%	64.44%	69.30%	77.20%	73.86%	75.08%	75.68%	**76.60%**	88.89%	61.40%
mean	-	40.79%	45.41%	53.69%	62.60%	60.17%	66.37%	65.11%	71.37%	65.14%	**73.18%**
Identifier	P	A	B	C	D	E	F	G	H	I	J

RUG vs LLM Approach

- Base: RUG improves baseline for 21.81% in GPT-3.5 and 10.20% in GPT-4
- Sensitivity test of RUG for: GPT model/fuzzing/problem decomposition

Crate Name (Downloads)	Tests ac/rej	GPT-3.5				GPT-4					Human Test Coverage
		Base		RUG		Base		RUG		Newly API Cov Rate	
		w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing		
bincode(49M)	4/0	1.57%	1.57%	22.92%	23.91%	16.63%	18.79%	44.67%	47.91%	74.11%	**64.58%**
chrono(128M)	22/13	37.88%	44.07%	47.2%	58.29%	54.04%	59.24%	56.90%	62.67%	73.05%	**76.66%**
hashes(266M)	**P**(7)									61.41%	**85.17%**
humantime(98M)	**P**(5)									40.00%	79.32%
itoa(221M)	1/0									83.33%	86.00%
json(203M)	-									47.33%	**72.36%**
mio(145M)	-									38.89%	24.19%
nom(114M)	6/1/**P**(14)									28.64%	**76.20%**
num-traits(185M)	-									90.36%	**50.58%**
demangle(93M)	**P**(14)									18.75%	72.25%
crc32fast(104M)	-	62.35%	64.71%	70.59%	71.76%	87.06%	88.24%	87.06%	**88.24%**	92.86%	68.24%
ryu(185M)	0/3	52.51%	95.28%	61.65%	97.64%	76.40%	99.42%	81.72%	**99.42%**	100.00%	87.85%
semver(168M)	18/0	61.40%	62.96%	62.54%	73.36%	72.36%	74.64%	74.22%	76.50%	95.24%	**84.33%**
textwrap(134M)	1/0	88.84%	92.56%	90.15%	94.31%	92.78%	94.75%	92.56%	**94.97%**	83.34%	87.53%
time(200M)	**P**(3)	33.08%	35.06%	48.98%	51.72%	55.34%	55.34%	79.89%	79.89%	66.06%	**96.48%**
toml(125M)	-	32.43%	37.06%	47.28%	49.02%	59.58%	64.40%	38.90%	38.90%	25.14%	**70.81%**
uuid(108M)	1/0	58.66%	64.44%	69.30%	77.20%	73.86%	75.08%	75.68%	**76.60%**	88.89%	61.40%
mean	-	40.79%	45.41%	53.69%	62.60%	60.17%	66.37%	65.11%	71.37%	65.14%	**73.18%**
Identifier	P	A	B	C	D	E	F	G	H	I	J

RUG improves baseline for 10.2%-21.8%

RUG vs LLM Approach(cont'd)

- Base: RUG improves baseline for 21.81% in GPT-3.5 and 10.20% in GPT-4
- Sensitivity test of RUG for: GPT model/fuzzing/problem decomposition

Crate Name (Downloads)	Tests ac/rej	GPT-3.5				GPT-4					Human Test Coverage
		Base		RUG		Base		RUG		Newly API Cov Rate	
		w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing	w/o	w. fuzzing		
bincode(49M)	4/0	1.57%	1.57%	22.92%	23.91%	16.63%	18.79%	44.67%	47.91%	74.11%	**64.58%**
chrono(128M)	22/13	37.88%	44.07%	47.2%	58.29%	54.04%	59.24%	56.90%	62.67%	73.05%	**76.66%**
hashes(266M)	**P**(7)	43.84%	43.84%	68.28%	68.28%	57.71%	57.71%	68.96%	**85.16%**	61.41%	**85.17%**
humantime(98M)	**P**(5)	63.09%	64.40%	67.02%	75.39%	74.08%	75.92%	74.61%	**80.37%**	40.00%	79.32%
itoa(221M)	1/0	26.00%	26.00%	82.00%	96.00%	96.00%	98.00%	100.00%	**100.00%**	83.33%	86.00%
json(203M)	-	28.10%	35.69%	44.60%	52.07%	62.26%	67.00%	70.25%	70.49%	47.33%	**72.36%**
mio(145M)	-	20.47%	20.47%	25.20%	25.20%	26.77%	26.77%	33.86%	**33.86%**	38.89%	24.19%
nom(114M)	6/1/**P**(14)	25.81%	25.81%	39.93%	40.04%	51.13%	51.17%	[illegible]	[illegible]	[illegible]	[illegible]
num-traits(185M)	-	36.02%	36.47%	43.20%	43.95%	46.94%	46.94%	[illegible]	[illegible]	[illegible]	[illegible]
demangle(93M)	**P**(14)	21.32%	21.62%	21.83%	65.99%	20.00%	74.82%	[illegible]	[illegible]	[illegible]	[illegible]
crc32fast(104M)	-	62.35%	64.71%	70.59%	71.76%	87.06%	88.24%	[illegible]	[illegible]	[illegible]	[illegible]
ryu(185M)	0/3	52.51%	95.28%	61.65%	97.64%	76.40%	99.42%	[illegible]	[illegible]	[illegible]	[illegible]
semver(168M)	18/0	61.40%	62.96%	62.54%	73.36%	72.36%	74.64%	[illegible]	[illegible]	[illegible]	[illegible]
textwrap(134M)	1/0	88.84%	92.56%	90.15%	94.31%	92.78%	94.75%	[illegible]	[illegible]	[illegible]	[illegible]
time(200M)	**P**(3)	33.08%	35.06%	48.98%	51.72%	55.34%	55.34%	[illegible]	[illegible]	[illegible]	[illegible]
toml(125M)	-	32.43%	37.06%	47.28%	49.02%	59.58%	64.40%	38.90%	38.90%	25.14%	**70.81%**
uuid(108M)	1/0	58.66%	64.44%	69.30%	77.20%	73.86%	75.08%	75.68%	**76.60%**	88.89%	61.40%
mean	-	40.79%	45.41%	53.69%	62.60%	60.17%	66.37%	65.11%	71.37%	65.14%	**73.18%**
Identifier	P	A	B	C	D	E	F	G	H	I	J

RUG with GPT-4 generates comparable tests to human developers

Evaluation on real world development

- We run the project's original test and compare with RUG, find the coverage differences
- For the untested regions, we submit RUG's tests as Pull Request to the project
  - 113 tests submitted: 53 merged/17 rejected/43 pending for response

```rust
#[test]
fn test_unfill_consecutive_different_prefix() {
let (text, options) = unfill("foo\n*\n/");
assert_eq!(text, "foo * /");
...
}
```

```rust
pub fn unfill(text: &str) -> (String, Options<'_>) {
    let prefix_chars: &[_] = &[' ', '-', '+', '*', '>', '#', '/'];

    let mut options = Options::new(0);
    for (idx, line) in text.lines().enumerate() {
        options.width = std::cmp::max(options.width, display_width(line));
        let without_prefix = line.trim_start_matches(prefix_chars);
        let prefix = &line[..line.len() - without_prefix.len()];

        if idx == 0 {
            options.initial_indent = prefix;
        } else if idx == 1 {
            options.subsequent_indent = prefix;
        } else if idx > 1 {
            for ((idx, x), y) in prefix.char_indices().zip(options.subsequent_indent.chars()) {
                if x != y {
                    options.subsequent_indent = &prefix[..idx];
                    break;
                }
            }
            if prefix.len() < options.subsequent_indent.len() {
                options.subsequent_indent = prefix;
            }
```

Evaluation on real world development

- We run the project’s original test and compare with RUG, find the coverage differences
- For the untested regions, we submit RUG’s tests as Pull Request to the project
  - 113 tests submitted: 53 merged/17 rejected/43 pending for response
- RUG received positive feedback

“The PR looks good, great job at finding a test case to improve the coverage like this.”

RUG can help developer in submit missed unit tests with 53/70 accepted cases

Conclusion

- RUG leverages program analysis to guide LLM for Rust unit test generation and address the concerns of compiling errors
- RUG can help developers to build uncovered tests and achieves a coverage comparable with experienced human efforts

Thanks for listening

Paper

Code, AE