RemGen: Remanufacturing A Random Program Generator for Compiler Testing

The 33rd IEEE International Symposium on Software Reliability Engineering (ISSRE 2022)

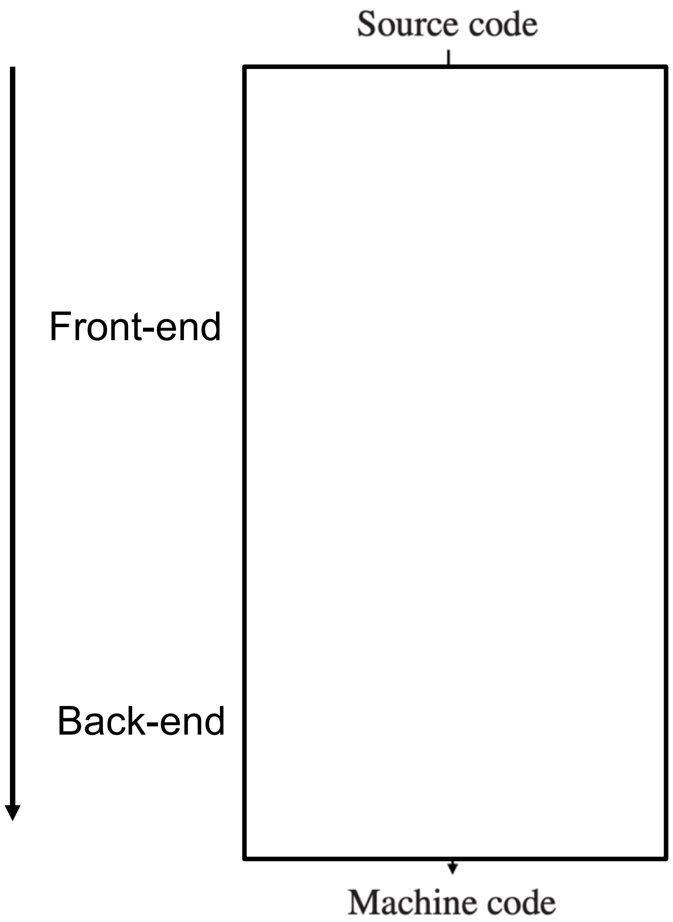
Haoxin Tu, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou
(Dalian University of Technology)
Lingxiao Jiang (Singapore Management University)

Outlines

- ❑ Background
- ❑ Motivation
- ❑ Approach
  - RemGen
- ❑ Evaluation
- ❑ Conclusion

Part 1: Background

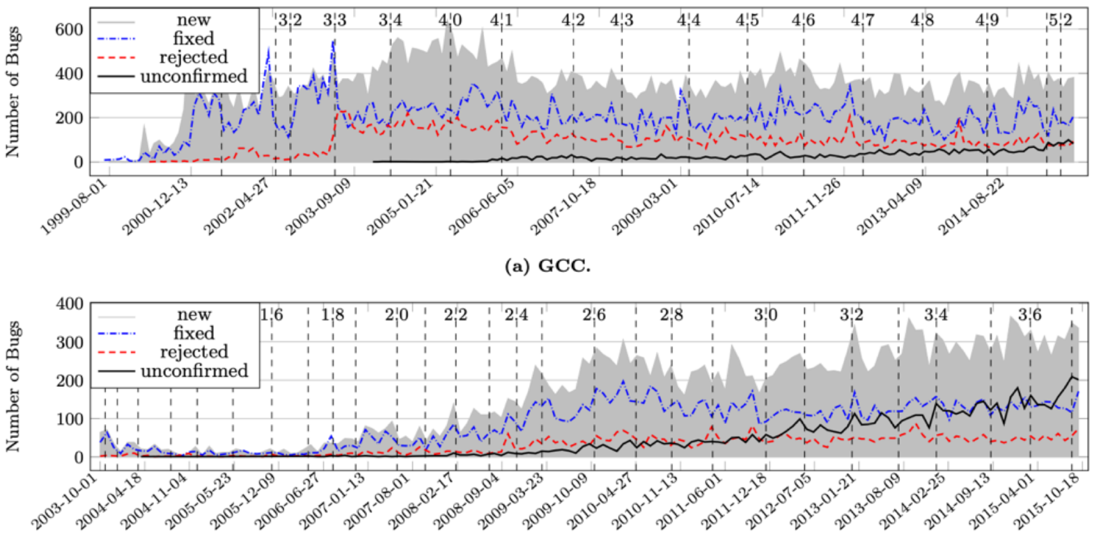
What is a compiler ?



- Two mainstream compilers
  - GCC (created by 1987)
    - ~ 5,000,000 lines of code
  - LLVM (created by 2003)
    - ~ 1,600,000 lines of code

- Implementation of compilers is complex
  - they can be unreliable and buggy

Compilers are important but unreliable



(a) GCC.

(b) LLVM

- Cited from [1]

XcodeGhost Bug: affect 3418 apps

CVE-2009-1897: Kernel crash to Dos attack

Improving the reliability of compilers is still a hot topic.

Constructing test programs for compiler testing

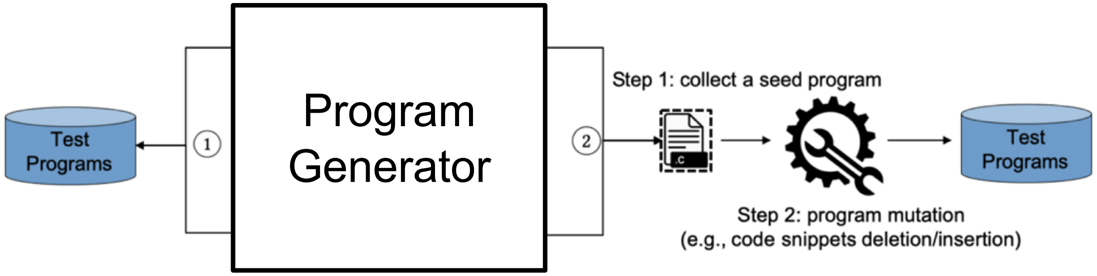
Program Generator
(e.g., Csmith)

- **Two primary approaches**
  - **1. Generation-based**
    - CCG [3], Csmith [4], and Yarpgen [5]
  - **2. Mutation-based**
    - Orion [6], Athena [7], and Hermes [8]

- **Observation: existing construction approaches all start from a random program generator!**

Part 2: Motivation

Motivation



- **Complaints from compiler testing studies or compiler expertise**
  - Csmith has found bugs before, but current production compilers are already **resilient** to it (from [5,6])
  - *Compilers have now caught up with CCG (since it's been pretty **hard to spot crashes** last time I tried.* (from CCG [1])
  - *I hadn't run Csmith for a while and it turns out LLVM is now amazingly **resistant** to it, ran a million tests overnight without finding a crash or miscompilation.* (from John Regehr [9])
  - ***Same** with YARPGen.* (from Dmitry Babokin [10])

Research question: Is it possible to make those generators effective again?

Remanufacturing

- **Definition [2]**
  - A process of bringing **a used product** to **a “like new” product**, which is being regarded as a sustainable mode of manufacturing
- Applications
  - Automobile, heavy-duty equipment, aerospace, machinery, medical devices, photocopiers, IT products [2]

❶ Preparation for remanufacturing

↓

❷ Remanufacturing

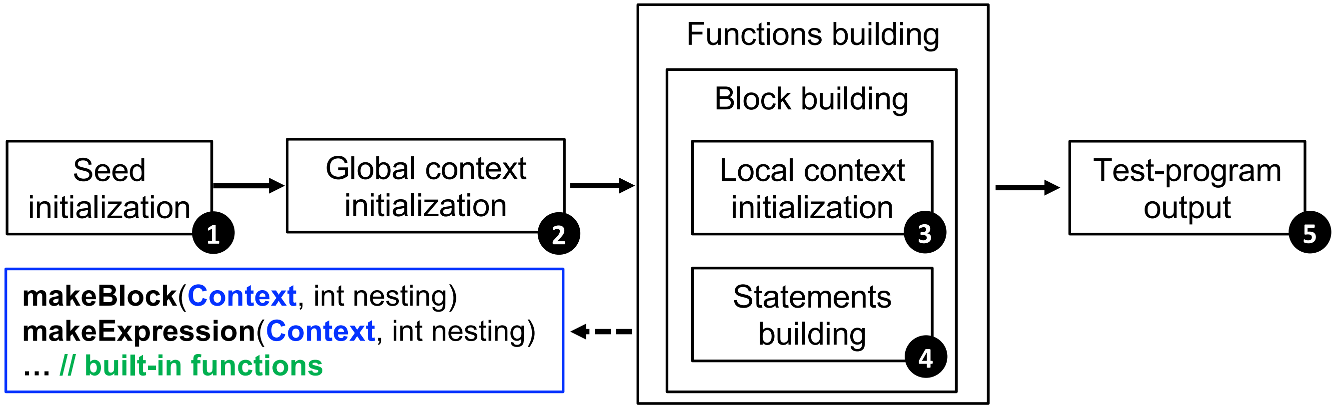
↓

❸ Testing the remanufactured product

Any chance to conduct remanufacturing on a program generator?

Leverage capabilities in program generators

- **General workflow of a generator**



- **Key capabilities**
  - (1) they support various built-in functions to generate different new valuable code snippets
  - (2) the context (i.e., one of the parameters used in the built-in functions) used in generating code snippets can be reserved and then reused in a lightweight manner

Motivation

- **An example**

```
int a, b, c, d;
void e() {
    ... // code snippets generated by CCG
    a = 7;
    for (; a <= 78; a++) {
        d = 3;
        for (; d <= 73; d++) {
            // code produced by makeBlock() highlighted in gray
            int f = 0;
            b += c;
            if (b) {
                int g = 0;
                for (f = 5; f; g);
            }
        }
    }
} /* Grammar Coverage : G={0,0,0,2,0,0,0,0,0,0,0} */
```

- **Limitation of existing approaches**
  - Generation-based approaches: randomness
  - Mutation-based approaches: (1) limited synthesize template to produce code snippets and (2) costly
- **Our approach**
  - Leverage the unexplored capabilities in generators to synthesize new code snippets

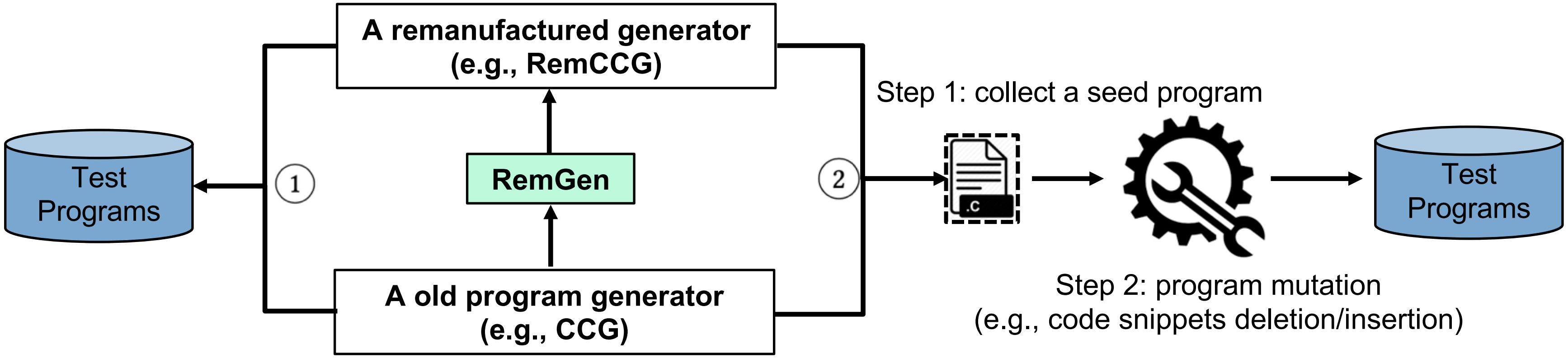
Challenges

- **1. The synthesis of diverse code snippets with low effort**
  - We do not know what the trigger for a compiler bug looks like [4].
  - Efforts in synthesizing code snippets should be lightweight
- **2. The selection of the bug-revealing code snippet for constructing test programs**
  - Not all **code snippets** are equal and only few can trigger bugs [12]
  - limited computing or human resource

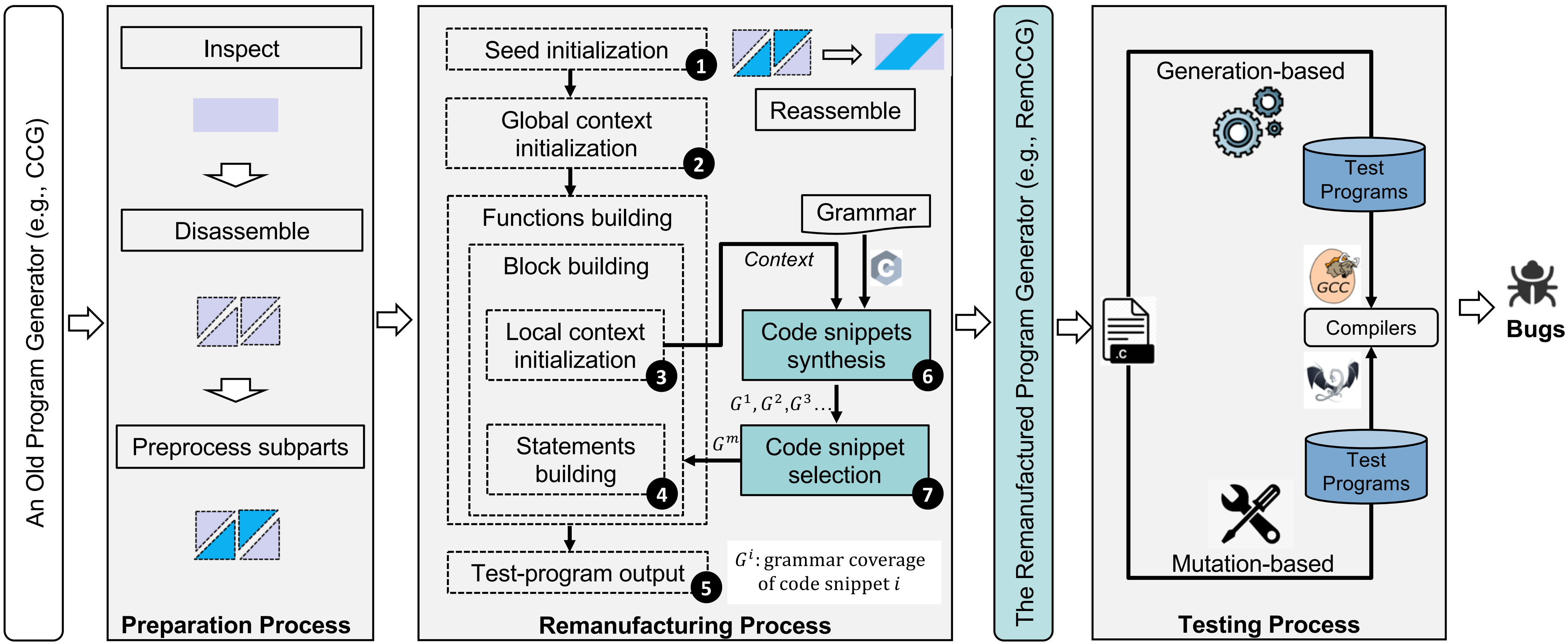
Part 3: Approach

Our approach: RemGen

- **Highlight**

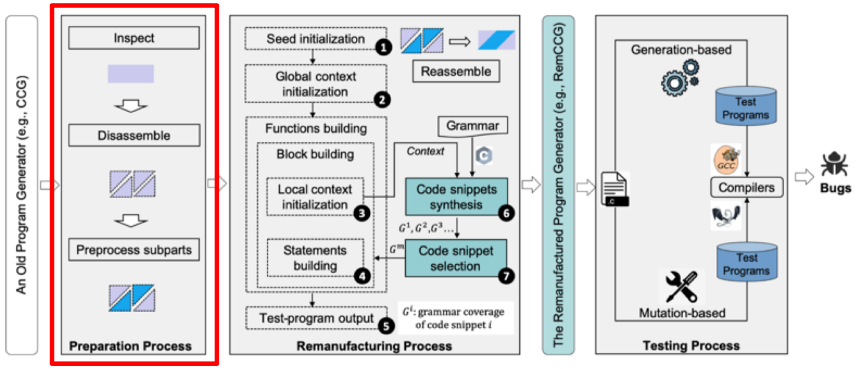


RemGen: Overview



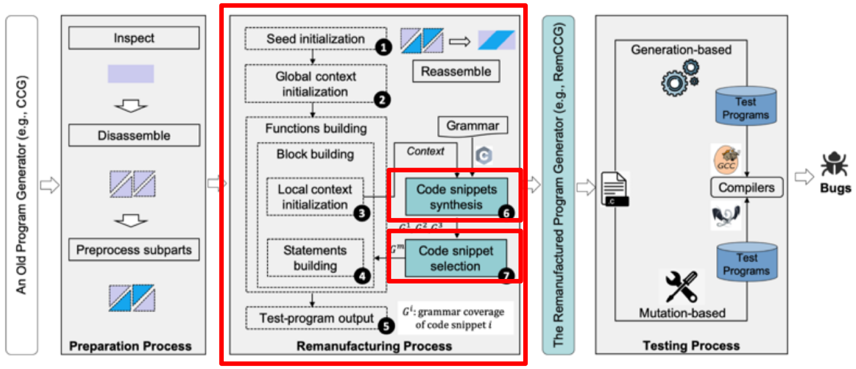
Preparation for remanufacturing ⇨ Remanufacturing ⇨ Testing the remanufactured product

RemGen: Preparation process



- **Inspect**
  - Checking the functionality from the input generator's "appearances"
- **Disassemble**
  - Decomposing the test program generation components to be modularized
- **Preprocess subparts**
  - Reconstructing required components (e.g., built-in functions) to be easily integrated with other components

RemGen: Remanufacturing process

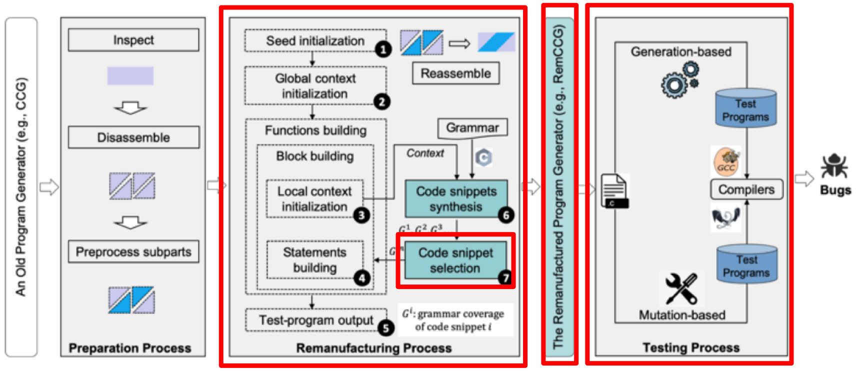


- **Remanufacturing: two new components**
  - Code snippets synthesis
  - Code snippet selection

❻ **Diverse code snippets synthesis (grammar-aided)**

- Collect the required context (i.e., global and local)
  - Low effort
- Invoke the built-in functions to generate new code snippets
- utilize our new “diversity”: *grammar coverage*
  - the number of grammar rules (e.g., *if* or *for* statements) invoked during the synthesis

RemGen: Remanufacturing process



❼ **Bug-revealing code snippet selection**

- Leverage grammar coverage in the prior component
- Order produced code snippets
  - Calculate the sum of the square of each grammar coverage
- Integrate the selected code snippet to construct bug-revealing test program

- **Reassemble**

- **Testing process**
  - More details in evaluation part

Part 3: Evaluation

Experimental Setup

- **We remanufactured an old program generator CCG into RemCCG under RemGen**

- **Research questions**
  - RQ1: Can RemCCG boost both generation-based and mutation-based approaches for compiler testing?
  - RQ2: Can RemCCG find new compiler bugs in practice?

- **Test settings**
  - For ***RQ1***, we run over the same compiler versions used in [15] (GCC-4.4.3, LLVM-2.6)
    - Running 90 hours 10 times, count the average number of bugs detected
  - For ***RQ2***, we run over current development versions of two compilers
    - Run RemCCG over the latest version of compilers

Evaluation (1/2)

- **RQ1:** Can RemGen boost generation-based approaches for compiler testing?
  - Compare with generation-based approach: CCG [1] (baseline)
  - Compare with mutation-based approach: Hermes [8]
    - Use CCG/RemCCG to generate seed programs

TABLE I: Results of Boosting in Generation-based Approach

Subject	Tools	Average Statistics			
		Cra.	*Perf.*	*Sum.*	*Imp.*
GCC	CCG [3]	2.9	0.3	3.2	16%
	REMCCG	3.1	0.6	3.7	-
LLVM	CCG [3]	9.2	2.7	11.9	11%
	REMCCG	9.7	3.5	13.2	-

TABLE II: Results of Boosting in Mutation-based Approach

Subject	Tools	Average Statistics			
		Cra.	*Perf.*	*Sum.*	*Imp.*
GCC	Hermes(CCG)	3.0	0.5	3.5	14%
	Hermes(REMCCG)	3.2	0.8	4.0	-
LLVM	Hermes(CCG)	9.8	3.6	13.4	11%
	Hermes(REMCCG)	10.6	4.3	14.9	-

Evaluation (2/2)

- **RQ2:** Can RemCCG find new compiler bugs in practice?

TABLE III: Results of All the Reported Bugs

Bug Status	GCC	LLVM	Total
Fixed	8	29	37
WorksForMe	0	2	2
Duplicate	2	3	5
Pending	0	12	15
Total	10	46	56

TABLE IV: Results of Bug Types of Fixed Bugs

But Types	GCC	LLVM	Total
Crash	6	16	22
Performance	2	13	15
Total	8	29	37

TABLE V: Details of Fixed Bugs

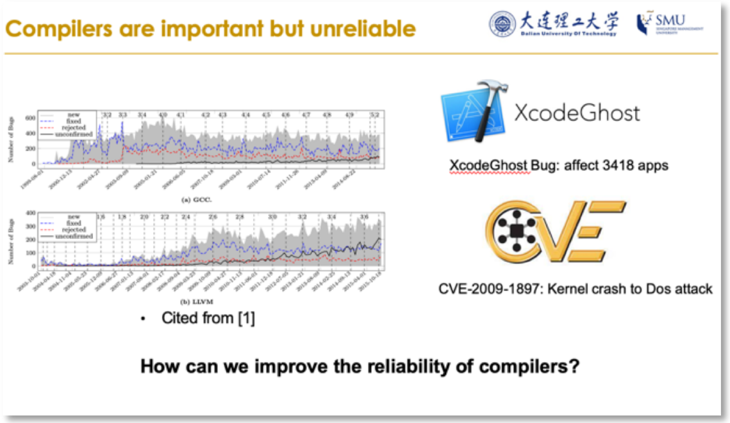
	Compiler-ID	Priority	Type	Affected. Opt.	Affected Versions
1	GCC-99694	P2	*Perf.*	-O1,2,3	**9.3-11.0 (trunk)**
2	GCC-99880	P2	*Crash*	-O3	10.2-11.0 (trunk)
3	GCC-99947	P1	*Crash*	-O3	11.0 (trunk)
4	GCC-100349	P2	*Crash*	-O2,3,s	11.0-12.0 (trunk)
5	GCC-100512	P3	*Crash*	-O2,3,s	12.0 (trunk)
6	GCC-100626	P2	*Crash*	-O1,2,3,s	11.0-12.0 (trunk)
7	GCC-102057	P3	*Crash*	-O1,2,3,s	12.0 (trunk)
8	GCC-102356	P3	*Perf.*	-O3	11.0-12.0 (trunk)
9	LLVM-49171	P3	*Perf.*	-O3	13.0 (trunk)
10	LLVM-49205	P3	*Perf.*	-O1,2,3,s	11.0-13.0 (trunk)
11	LLVM-49218	P3	*Crash*	-O1	12.0-13.0 (trunk)
12	LLVM-49396	P3	*Crash*	-O2,3,s	12.0-13.0 (trunk)
13	LLVM-49451	P3	*Crash*	-Os	13.0 (trunk)
14	LLVM-49466	P3	*Crash*	-O2	13.0 (trunk)
15	LLVM-49475	P3	*Perf.*	-O1	12.0-13.0 (trunk)
16	LLVM-49541	P3	*Perf.*	-O2,s	**7.0-13.0 (trunk)**
17	LLVM-49697	P3	*Crash*	-O3	**7.0-13.0 (trunk)**
18	LLVM-49785	P3	*Perf.*	-O3	13.0 (trunk)
19	LLVM-49786	P3	*Perf.*	-O2	13.0 (trunk)
20	LLVM-49993	P3	*Crash*	-O3	13.0 (trunk)
21	LLVM-50009	P3	*Crash*	-Os	12.0-13.0 (trunk)
22	LLVM-50050	P3	*Crash*	-O2,3,s	13.0 (trunk)
23	LLVM-50191	P3	*Crash*	-O2	13.0 (trunk)
24	LLVM-50238	P3	*Crash*	-O1,2,3,s	13.0 (trunk)
25	LLVM-50254	P3	*Perf.*	-O2,3	13.0 (trunk)
26	LLVM-50279	P3	*Perf.*	-O3	13.0 (trunk)
27	LLVM-50302	P3	*Perf.*	-O3	13.0 (trunk)
28	LLVM-50307	P3	*Crash*	-Os	13.0 (trunk)
29	LLVM-50308	P3	*Perf.*	-O1,2,3,s	12.0-13.0 (trunk)
30	LLVM-51553	P3	*Crash*	-O3	14.0 (trunk)
31	LLVM-51584	P3	*Perf.*	-O1,2,3,s	14.0 (trunk)
32	LLVM-51612	P3	*Crash*	-O2,3	14.0 (trunk)
33	LLVM-51656	P3	*Crash*	-O2,3	14.0 (trunk)
34	LLVM-51657	P3	*Perf.*	-O2,3,s	12.0-14.0 (trunk)
35	LLVM-51762	P3	*Perf.*	-O1	14.0 (trunk)
36	LLVM-52018	P3	*Crash*	-O3	14.0 (trunk)
37	LLVM-52024	P3	*Crash*	-O2	14.0 (trunk)

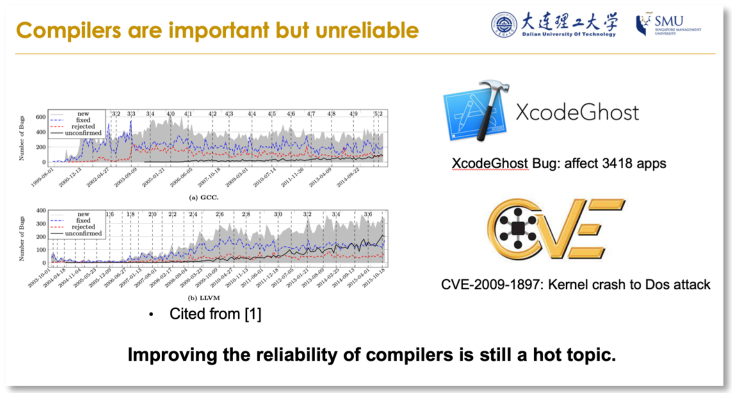
Discussion

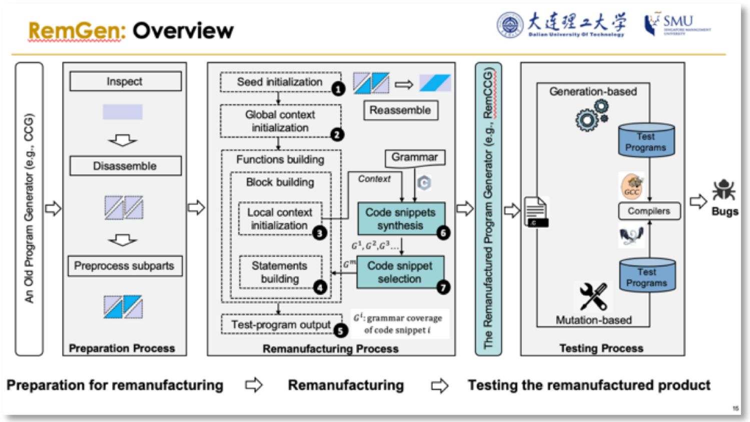
- Effectiveness of the two proposed components
  - we compare RemCCG with its variants
- Comparison with Csmith [4] and YARPGen [5]
  - Find 164%/363% and 120%/595% more bugs than Csmith and YARPGen, in GCC/LLVM, respectively
  - This is reasonable due to the different design goal between those tools
- Limitation of RemCCG
  - Inherits the limitation from CCG: can only find two kinds of (i.e., crash and performance) bugs

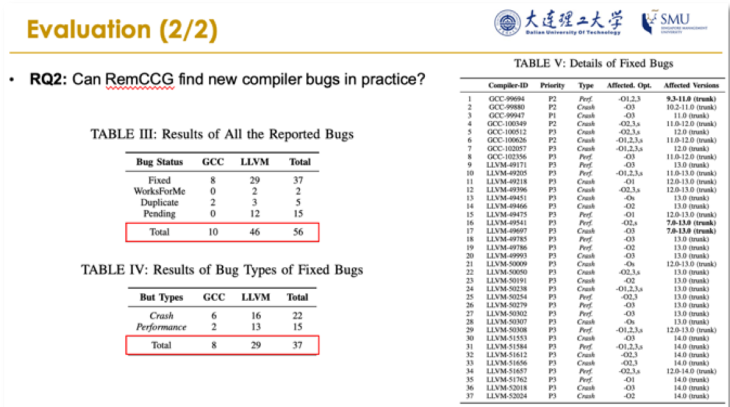
Part 5: Conclusion

Conclusion









Code: https://github.com/haoxintu/RemCCG
Email: haoxintu.2020@phdcs.smu.edu.sg
(Please feel free to pull requests or raise any questions if you have!)

References

[1] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. **Toward understanding compiler bugs in GCC and LLVM**. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016). Association for Computing Machinery, New York, NY, USA, 294–305.

[2] Mitsutaka Matsumoto, Shanshan Yang, Kristian Martinsen, and Yasutaka Kainuma. 2016. **Trends and research challenges in remanufacturing**. International Journal of Precision Engineering and Manufacturing-Green Technology 3 (01 2016), 129–142.

[3] https://github.com/Mrktn/ccg.git

[4] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011**. Finding and understanding bugs in C compilers**. In PLDI 2011. Association for Computing Machinery, New York, NY, USA, 283–294.

[5] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. **Random testing for C and C++ compilers with YARPGen**. Proc. ACM Program. Lang. 4, OOPSLA, Article 196 (November 2020), 25 pages.

[6] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. **Compiler validation via equivalence modulo inputs**. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).

[7] Vu Le, Chengnian Sun, and Zhendong Su. 2015. **Finding deep compiler bugs via guided stochastic program mutation**. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).

[8] Chengnian Sun, Vu Le, and Zhendong Su. 2016. **Finding compiler bugs via live code mutation**. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016).

[9] https://twitter.com/johnregehr/status/1134866965028196352

[10] https://twitter.com/DmitryBabokin/status/1134907976085516290

[11] https://github.com/Mrktn/ccg/blob/master/README

[12] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. **Taming Compiler Fuzzers**. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13). Association for Computing Machinery, New York, USA, 197–208.

[13] https://github.com/antlr/grammars-v4/blob/master/c/C.g

[14] https://github.com/protocolbuffers/protobuf

[15] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. **An empirical comparison of compiler testing techniques**. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16). Association for Computing Machinery, New York, NY, USA, 180–190.

Thank you && Questions?

RemGen: Remanufacturing A Random Program Generator for Compiler Testing

The 33rd IEEE International Symposium on Software Reliability Engineering (ISSRE 2022)

Haoxin Tu, He Jiang, Xiaochen Li, Zhilei Ren, Zhide Zhou
(Dalian University of Technology)
Lingxiao Jiang (Singapore Management University)