RABIEF: Range Analysis Based Integer Error Fixing

Xi Cheng
School of Software, Tsinghua University
Beijing, China
chengxi13@mails.tsinghua.edu.cn

ABSTRACT

We propose RABIEF, a novel and fully automatic approach to fix C integer errors based on range analysis. RABIEF is inspired by the following insights: (1) fixes for various integer errors have typical patterns including sanitization, explicit cast and declared type alteration; (2) range analysis provides sound basis for error detection and guides fix generation. We implemented RABIEF into a tool ARGYI. Its effectiveness and efficiency have been substantiated by the facts that: (1) ARGYI succeeds in fixing 93.9% of 5414 integer bugs from Juliet test suite, scaling to 600 KLOC within 5500 seconds; (2) ARGYI is confirmed to correctly fix 20 errors from 4 real-world programs within only 240 seconds.

CCS Concepts

•**Software and its engineering** → **Automated static analysis;**

Keywords

integer error, range analysis, fixing pattern

1. PROBLEM AND MOTIVATION

Integer errors in C include overflow, sign misinterpretation, lossy truncation and other undefined behaviors such as divided-by-zero. They can lead to serious system crash (e.g. the failure of Ariane 5 [19]), or become an important source of vulnerabilities [1]. Integer use in C is quite error-prone, as C integers actually have semantics of fixed-size bit-vectors and implicit conversion is allowed. Therefore, addressing integer errors is an important and long-standing problem. Existing automated program repair techniques either require external specifications for practical effectiveness [18, 8, 11], or aggressively transform bugs into program crash [4, 20]. To overcome these limitations, we propose RABIEF, a novel approach to fix integer errors based on range analysis. The essential ideas behind RABIEF are: (1) many integer bugs can be prevented by improving precision instead of aggressive sanitization ; (2) existing fixes have typical patterns; (3) the criterion for integer errors are general and they can be captured by range analysis [7].

We found that fixes for most real-world integer errors can be summarized as three typical patterns, presented in Figure 1. Sanitization (SA) checks whether an expression have safe value for certain critical operation. Explicit type cast (TC) enforces the type of certain expression to prevent underlying implicit conversion. Declared type alteration (TA) changes the type of variable to its underlying type which is different from the declared one.

Table 1: Three fixing patterns.

	Before Fixing	After Fixing
SA	long size; void *mem; mem = malloc(size);	long size; void *mem; **if(size < 0 || size > SIZE_MAX)** **abort();** mem = malloc((size_t)size);

TC	int a = -1;
	unsigned b = 0;
	if(a > b)
	error();
		int a = -1;
		unsigned b = 0;
		if(**(int)**a > **(int)**b)
		error();

TA	int a = rand(0, INT_MAX);
	int b = rand(0, 10);
	a = a + b;
		long a = rand(0, INT_MAX);
		int b = rand(0, 10);
		a = a + b;

The condition of safe integer use is general. In most cases, developer's expectation of integer semantics can be summarized as the following assumptions:

- **A1** Semantics of a bit manipulation and reaching definitions of its operands are expected as over bit-vector.
- **A2** Semantics of an integer used in arithmetic or relational operation is expected as over $\mathbb{Z}$;

Thus, an integer error is supposed to occur when one of the assumptions is violated by mixing different semantics in integer use. It is noteworthy that although these assumptions are not absolute, they are sufficient for most of integer bugs in practice.

2. BACKGROUND AND RELATED WORK

Tools for automated generic bug fixing based on generate-and-validate [18, 17, 16, 12, 8, 9] or program synthesis [11, 10] can cover integer errors. However, they face several difficulties: (1) the effectiveness heavily relies on specifications which are often insufficient or incomplete; (2) generated patches rarely ensure correctness [13]. Moreover, even the state-of-the-art approach [9] generally requires hours of searching for a practical fix.

There are some other approaches specific to integer errors. Z. Coker et al. apply code transformations [3] based on several design decisions, however overflows are transformed into unrecoverable exceptions by using safe integer library. Dynamic approaches [4, 5, 14, 20] essentially sanitize critical integer operations, but they would miss bugs on non-critical sites leading to the erroneous runtime behavior.

3. APPROACH AND UNIQUENESS

The workflow of RABIEF is illustrated in Figure 1. At first, range analysis is performed on preprocessed source code to collect

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983961

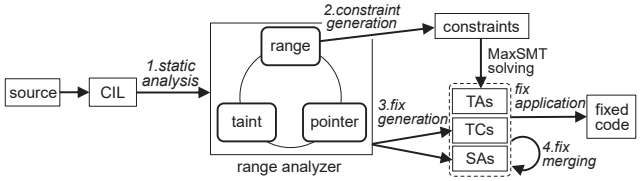

Figure 1: Overview of RABIEF approach.

range information, then we generate (1) MaxSMT [6] constraints of underlying types for variables, and (2) fixes of TC and SA. Fixes of TA are generated by solving the type constraints. Finally, fixes are applied to the original source.

Static analysis. Mainly three analyses are employed. Range analysis infers and propagates ranges of expressions. Pointer analysis captures pointer-to and alias relation. Taint analysis [15] tracks values involved in bit manipulation. This analysis is used to preserve intentional wraparound with respect to **A1**. To improve precision, we do not propagate value which prohibits the continued execution. For example, when $a \in [2, 5]$ and $b \in [-2, 2]$, the range of a/b is inferred as $[-5, 5]$ while the divided-by-zero case is discarded.

Constraint generation. Constraints are logical formulae specifying underlying types for variables. They are built with the following two basic predicates:

- $\mathsf{cover}(t_1, t_2)$: $R_{t_2} \subseteq R_{t_1}$ (R_t denotes the range of t);
- $\mathsf{equiv}(t_1, t_2)$: t_1 and t_2 are equivalent types.

Given a statement s and range state $\mathcal{R}$ that maps expression e to range $I \subseteq \mathbb{Z}$, its MaxSMT constraint C is determined by rules shown in Table 2. Constraint for external declaration is *hard* indicating that its type should not be altered. Other constraints are *soft* and assigned with a weight value (such as α and β), indicating the penalty of violation. A MaxSMT solver can give a solution minimizing total penalty. In particular, a zero-penalty solution assigns variables with types fully compatible with correctness assumptions on integer behavior. Practically, we have $\beta > \alpha$ since type alteration has higher priority than minimizing modification on code.

Table 2: Constraint generation rules.

Stmt. s	**Constraint** C
`extern T x`	$\mathsf{equiv}(t_{\mathtt{x}}, \mathtt{T})$
`T x`	$\alpha : \mathsf{equiv}(t_{\mathtt{x}}, \mathtt{T}) \wedge \beta : \mathsf{cover}(t_{\mathtt{x}}, \mathtt{T})$
`x = e`	$\beta : \mathsf{cover}(t_{\mathtt{x}}, t')$, where $\mathcal{R}(e) = I, t' \prec I^{\dagger}$

† $t' \prec I$ denotes t' *minimally covers* I, i.e. no $t'' \neq t'$ exists such that $R_{t'} \supseteq I$, $R_{t''} \supseteq I$ and $\mathsf{cover}(t', t'')$.

Fix generation. Formally, we use $F_{\mathsf{SA}}(e)$ to denote SA on expression e, $F_{\mathsf{TA}}(v, t)$ to denote TA on variable v to type t and $F_{\mathsf{TC}}(e, t)$ to denote TC on e to t. $F_{\mathsf{SA}}(e)$ is generated when e is on critical site where type alteration is not permitted, such as function parameter and return statement. We have $F_{\mathsf{TA}}(v, t)$ if v is assigned with t in MaxSMT solution. Finally, for binary expression $e_1 \bowtie e_2$, $F_{\mathsf{TC}}(e_1, t)$ and $F_{\mathsf{TC}}(e_2, t)$ is generated when there exists t such that $R_t \prec (\mathcal{R}(e_1) \cup \mathcal{R}(e_2) \cup \mathcal{R}(e_1 \bowtie e_2))$.

Fix merging. Multiple fixes can be assigned to one syntactic element. Let $\mathcal{C}$ be the function that merges multiple fixes into one. We have $\mathcal{C}(F_{\mathsf{TC}}(e, t_1), \ldots, F_{\mathsf{TC}}(e, t_n)) = F_{\mathsf{TC}}(e, t)$ if t exists such that $t \prec R_{t_1} \cup \cdots \cup R_{t_n}$, otherwise t is the type in $t_1, \ldots, t_n$ with the widest range. Also, $\mathcal{C}(F_{\mathsf{TC}}(e, t), F_{\mathsf{SA}}(e)) = F_{\mathsf{SA}}(F_{\mathsf{TC}}(e, t))$. Other combinations are impossible. For example, there could not be multiple $F_{\mathsf{TA}}(v, t)$ since t is determined by MaxSMT solution.

The novelty of RABIEF is summarized as follows. First, RABIEF is fully automatic without relying on additional specifications or existing correct patches. Second, RABIEF can generate fixes which keep as much acceptability of continued execution as possible. Third, RABIEF is fully static and applicable for code segments.

4. RESULT AND CONTRIBUTION

We implemented RABIEF into an tool ARGYI, which is evaluated on NIST's Juliet test suite (a collection of test bench programs in C/C++ developed for assessing the capability of program analysis tools) and 4 real-world open-source projects for effectiveness and efficiency. Weights α and β are set to 1 and 80 respectively, which is an optimal setting according to pilot experiments.

Table 3: Result on Juliet test suite.

CWE	#File	KLOC	BAD (Time: s)				GOOD
			#Fixed	T_{FIX}	T_{RUN_ORIG}	T_{RUN_FIXED}	#Corrupt
190	1938	230.060	**1817**	1876	1.30	1.22	**0**
191	1178	141.385	**971**	1112	0.65	0.65	**0**
194	760	79.768	**760**	956	CRASH	1.02	**0**
195	760	78.400	**760**	830	CRASH	0.55	**0**
196	18	1.547	**18**	17	0.03	0.03	**0**
197	570	49.245	**570**	525	0.41	0.27	**0**
680	190	19.600	**190**	179	CRASH	0.26	**0**
Σ	5414	600.005	**5086**	5495	N/A	4.00	**0**

From Juliet test suite, we choose programs from 7 CWE categories involving integer error. CWE 190, 191 and 680 are overflow errors. CWE 194-197 are related to unexpected conversions. Each test program contains a bad function (containing exactly one defect) and at least one good function.

Table 3 shows the experimental results. ARGYI succeeds in fixing 5086/5414=93.9% bugs in 5495 seconds. False negatives are caused by limited precision of native C integers for overflow issues of 64-bit integers. By profiling the running of ARGYI, we found that the main bottleneck on efficiency is static analysis phase, which occupies 96.8% of total time. Column 6-7 show that applied fixes bring negligible overhead. Moreover, the results show that none of good functions are corrupted. This substantiates the robustness of RABIEF for flawless code.

Table 4: Result on real-world projects.

Program	Version	#BugFixed	T_{FIX} (Time: s)
PostgreSQL	9.0.15	4	24.249
GIMP	2.6.7	11	204.341
gzip	1.3.9	1	2.124
FFmpeg	2.0.1	3	9.334

ARGYI is also evaluated on 4 open-source projects based on recent reported defects in CVE database. We directly apply ARGYI to the function with target defect and check if new program does not crash or return unexpected result with certain test case. The experimental results are shown in Table 4. ARGYI is confirmed to fix totally 20 bugs in 240 seconds. In particular, the fixed expression in gzip contains multiple integer errors mixed together. We also have two important observations: (1) MaxSMT solving becomes expensive when the code contains many intricate numerical operations, which is the case for several bugs in GIMP; (2) A large portion of real-world integer errors involve using overflown value in buffer manipulation (e.g. IO2BO bugs [2]).

Our contributions are highlighted as follows. (1) We propose a novel and automatic approach RABIEF, which generates fixes of three typical patterns by leveraging range analysis. (2) We implemented RABIEF into a tool ARGYI and evaluate it with Juliet test suite and several real-world projects. ARGYI correctly fixes most of integer errors efficiently while does not corrupt any flawless programs. The effectiveness and efficiency of RABIEF are of practical benefits.

5. REFERENCES

[1] CVE-2001-0144. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144.

[2] CWE-680: Integer Overflow to Buffer Overflow. https://cwe.mitre.org/data/definitions/680.html.

[3] Z. Coker and M. Hafiz. Program transformations to fix C integers. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 792–801, 2013.

[4] R. B. Dannenberg, W. Dormann, D. Keaton, R. C. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum. As-if infinitely ranged integer model. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 91–100, 2010.

[5] W. Dietz, P. Li, J. Regehr, and V. S. Adve. Understanding integer overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.*, 25(1):2, 2015.

[6] Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, pages 252–265, 2006.

[7] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. Software Eng.*, 3(3):243–250, 1977.

[8] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178, 2015.

[9] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 298–312, 2016.

[10] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 448–458, 2015.

[11] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 772–781, 2013.

[12] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 254–265, 2014.

[13] Z. Qi, F. Long, S. Achour, and M. C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 24–36, 2015.

[14] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 33:1–33:11, 2013.

[15] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331, 2010.

[16] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 43–54, 2015.

[17] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 356–366, 2013.

[18] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374, 2009.

[19] Wikipedia. Ariane 5 Flight 501. https://en.wikipedia.org/wiki/Ariane_5_Flight_501. Accessed: 2016-06-25.

[20] C. Zhang, W. Zou, T. Wang, Y. Chen, and T. Wei. Using type analysis in compiler to mitigate integer-overflow-to-buffer-overflow threat. *Journal of Computer Security*, 19(6):1083–1107, 2011.