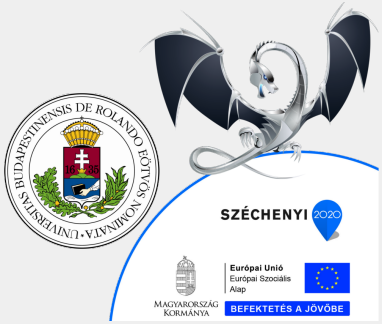
The Role of Implicit Conversions in Erroneous Function Argument Swapping in C++

Richárd Szalay, Ábel Sinkovics,
Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages & Compilers

IEEE SCAM 2020,
2020. 09. 27–28.
The work presented is supported by the European Union,
co-financed by the European Social Fund in project
EFOP-3.6.3-VEKOP-16-2017-00002.



1 Implicit conversions

2 Mixable parameter ranges

3 Empirical setup

4 Results

5 Summary

Argument selection defects

Coined by Rice *et al.*[1]

Happens when wrong argument (from available variables, expressions) passed to function.

```
void f(String hostName, int port, String message);

String author = "Richard.Szalay", greeting = "Hello, World!";
f(author, 8080, greeting);
```

[1] Rice et al., “Detecting Argument Selection Defects”.

Argument swaps ($\subset$ argument selection defects)

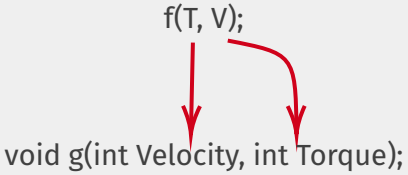
Special case when arguments are as intended, but out of order.

Previous literature findings:

- adjacency increases chance of mistake
- too many parameters increases chance of mistake

```
void f2(String message, String hostName, int port);
```

Reactive $\longrightarrow$ Proactive




```
void g₂(velocity_t V,
        torque_t  T);
```

- Detect call sites
- Significant enough mismatch **in name** $\rightarrow$ report

- Use the type system to guard us!
- Swapped expressions $\rightarrow$ *↯ compile error*

Is this a swapped call? Is this a compile error?

```
struct Complex { double Re, Im; };
void h(int Scalar, Complex Comp);

void test() {
    int     S = 8;
    Complex C = Complex{.5f, -.25f}; // = (1/2 - 1/4 i)

    h(C, S); // ← ?
}
```

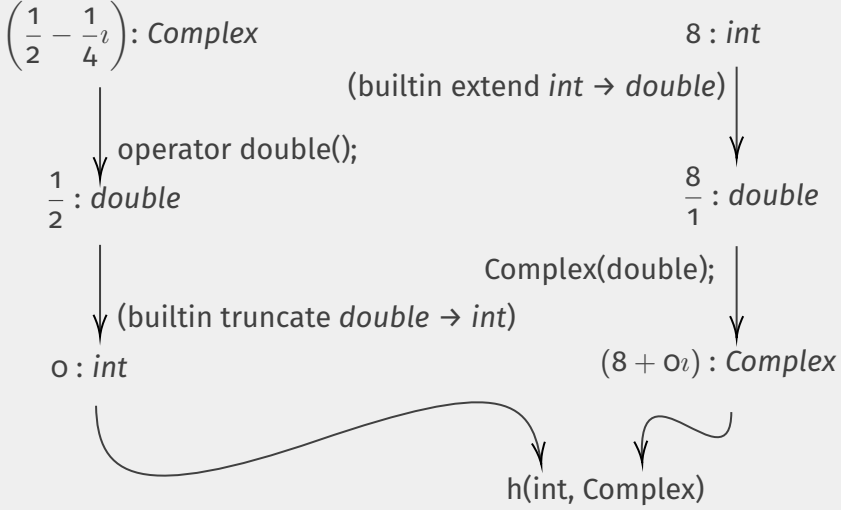
It depends ...

```
struct Complex {
    double R, I;
    Complex(double real) : R(real), I(0.0) {}
    operator double() const { return R; }
    // ⋮
};
void h(int Scalar, Complex Comp);

void test() {
    int     S = 8;
    Complex C = Complex{.5f, -.25f}; // = (1/2 − 1/4 i)

    h(C, S); // X
}
```

Implicit conversion



Subtle mistake, expanded

```
struct Complex {
    double R, I;
    Complex(double real) : R(real), I(0.0) {}
    operator double() const { return R; }
    // ⋮
};
void h(int Scalar, Complex Comp);

void test() {
    h(Complex{.5f, -.25f}, 8); // ≡ h(C, S); from before...
    h(0, Complex{8, 0});
}
```

Implicit conversion

An ***implicit conversion sequence*** $\mathcal{T}_1 \to \mathcal{T}_2$ exists and defined as:[2]

1. at most one standard conversion sequence (max. 4 steps)
2. at most one user-defined conversion (one function call!)
3. at most one standard conversion sequence (max. 4 steps)

…**if** the path taken is *unique*.

$\implies$ Fortunately, it means it's *bounded* at least…

[2] ISO/IEC JTC 1/SC 22, *ISO/IEC 14882:2017 Information technology — Programming languages — C++, version 17 (C++17)*.

Mixable adjacent parameters

```
void
p (int i, int j, double d, Complex c, std::string s);
```

Mixable adjacent parameters

```
void
p (int i, int j, double d, Complex c, std::string s);
```

- Is **`int`** mixable with **`int`**? ✓ *Same type, trivially*
- Is **`int`** mixable with **`double`**? ↗ *standard conversion*
- Is **`int`** mixable with `Complex`? ↗ *standard + user*
- Is **`double`** mixable with `Complex`? ↗ *user conversion*
- Is `Complex` mixable with `std::string`?

Mixable adjacent parameters

```
void
p (int i, int j, double d, Complex c, std::string s);
   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^
```

- Is **`int`** mixable with **`int`**? ✓ *Same type, trivially*
- Is **`int`** mixable with **`double`**? ↷ *standard conversion*
- Is **`int`** mixable with `Complex`? ↷ *standard + user*
- Is **`double`** mixable with `Complex`? ↷ *user conversion*
- Is `Complex` mixable with `std::string`? ✗ *No*

Mixable adjacent parameters

```
void
p (int i, int j, double d, Complex c, std::string s);
   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^
```

- Is **`int`** mixable with **`int`**? ✓ *Same type, trivially*
- Is **`int`** mixable with **`double`**? ↗ *standard conversion*
- Is **`int`** mixable with `Complex`? ↗ *standard + user*
- Is **`double`** mixable with `Complex`? ↗ *user conversion*
- Is `Complex` mixable with `std::string`? ✗ *No*
- … (at most $\mathcal{O}\left(\frac{n(n-1)}{2}\right)$ checks)

Analysis

Implemented as *Clang-Tidy* [3] analysis rule.
Developed relaxations and filtering to make more important findings stand out.

We sampled *GitHub*'s most active projects for an analysis and selected 7 **C** and 7 **C++**.

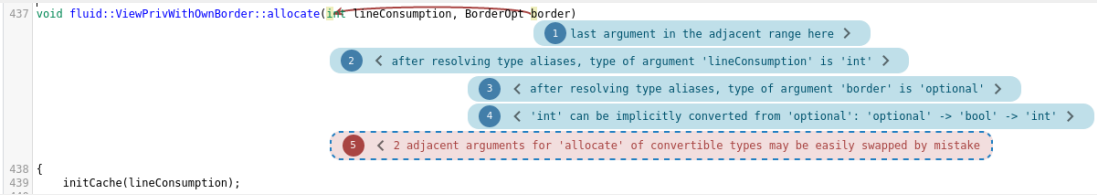


Figure: One finding (from OpenCV[4]) visualised using *CodeChecker* [5].

[3] clang.llvm.org/extra/clang-tidy
[4] Xperience AI, *OpenCV*.
[5] github.com/Ericsson/codechecker

Avg. number of mixable ranges by length

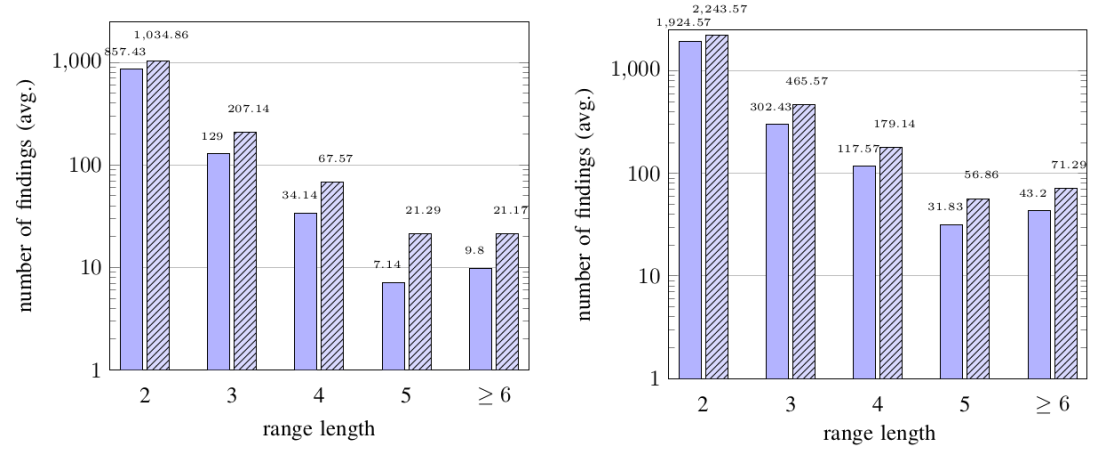


Figure: Findings for C (left) and C++ (right). Striped column indicates relaxed (conversions considered mixable, not just strict equality) mode.

Avg. number of mixable ranges by length

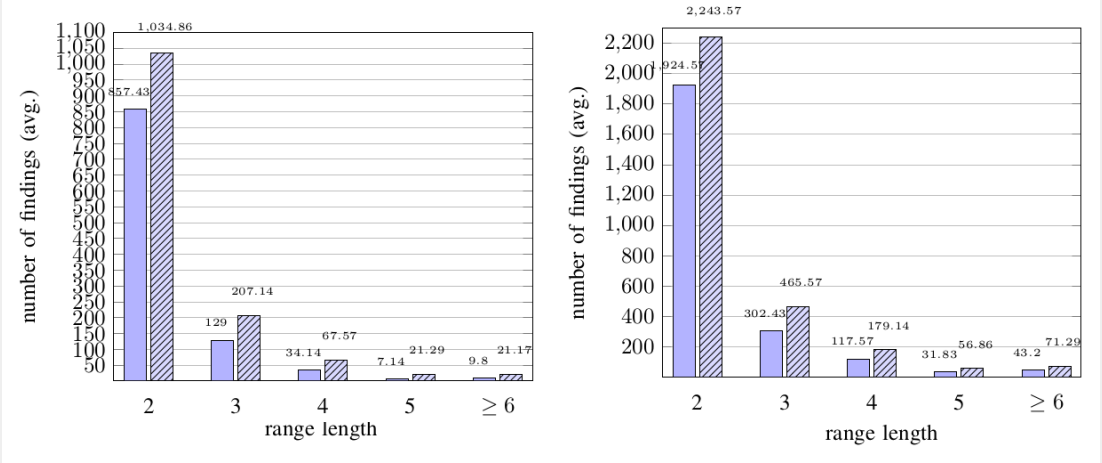


Figure: Findings for C (left) and C++ (right). Striped column indicates relaxed (conversions considered mixable, not just strict equality) mode.

How many functions are with potentially hidden problems?

TABLE I
DETAILED BREAKDOWN FOR THE NUMBER OF FUNCTIONS MATCHED, ACROSS VARIOUS CONFIGURATIONS.

Lang.	Project	Functions considered	N (Normal)		CV (Section III-B)			Imp (Section IV)			CV ∪ Imp			
			T (total)	*R* (without related)	*T*	+ vs. *N*	*R*	*T*	+ vs. *N*	*R*	*T*	+ vs. *CV*	+ vs. *Imp*	*R*
C	curl [38]	875	134	73	153	19	84	210	76	138	229	76	19	149
	git [39]	5 721	1 428	626	1 477	49	654	1 610	182	771	1 660	183	50	798
	netdata [40]	780	236	110	119	18	119	304	68	173	320	66	16	181
	PHP [41]	6 310	1 272	644	1 306	34	659	1 515	243	831	1 548	242	33	846
	Postgres [42]	9 506	2 705	1 314	2 817	112	1 365	3 721	1 016	2 116	3 837	1 020	116	2 167
	Redis [43]	2 834	589	242	628	39	257	700	111	332	744	116	44	351
	TMux [44]	1 043	250	108	261	11	113	300	50	158	308	47	8	163
C++	Bitcoin [45]	1 969	422	146	440	18	156	723	301	313	745	305	22	326
	guetzli [46]	165	81	35	84	3	37	83	2	39	91	7	8	46
	LLVM/Clang [37]	36 804	7 635	2 638	7 714	79	2 677	9 376	1 734	3 754	9 592	1 869	214	3 865
	OpenCV [47]	11 760	5 162	2 175	5 456	294	2 300	6 064	895	2 903	6 352	889	288	3 032
	ProtoBuf [48]	2 038	339	128	343	4	129	424	85	198	433	90	9	204
	Tesseract [49]	1 841	754	331	758	4	332	850	96	428	857	99	7	431
	Xerces [50]	1 655	492	196	508	16	200	555	69	241	671	163	116	299

How many functions are with potentially hidden problems?

TABLE I
DETAILED BREAKDOWN FOR THE NUMBER OF FUNCTIONS MATCHED, ACROSS VARIOUS CONFIGURATIONS.

Lang.	Project	Functions considered	N (Normal)		CV (Section III-B)			Imp (Section IV)			CV ∪ Imp			
			T (total)	*R* (without related)	*T*	+ vs. *N*	*R*	*T*	+ vs. *N*	*R*	*T*	+ vs. *CV*	+ vs. *Imp*	*R*
C	curl [38]	875	134	73	153	19	84	210	76	138	229	76	19	149
	git [39]	5 721	1 428	626	1 477	49	654	1 610	182	771	1 660	183	50	798
	netdata [40]	780	236	110	119	18	119	304	68	173	320	66	16	181
	PHP [41]	6 310	1 272	644	1 306	34	659	1 515	243	831	1 548	242	33	846
	Postgres [42]	9 506	2 705	1 314	2 817	112	1 365	3 721	1 016	2 116	3 837	1 020	116	2 167
	Redis [43]	2 834	589	242	628	39	257	700	111	332	744	116	44	351
	TMux [44]	1 043	250	108	261	11	113	300	50	158	308	47	8	163
C++	Bitcoin [45]	1 969	422	146	440	18	156	723	301	313	745	305	22	326
	guetzli [46]	165	81	35	84	3	37	83	2	39	91	7	8	46
	LLVM/Clang [37]	36 804	7 635	2 638	7 714	79	2 677	9 376	1 734	3 754	9 592	1 869	214	3 865
	OpenCV [47]	11 760	5 162	2 175	5 456	294	2 300	6 064	895	2 903	6 352	889	288	3 032
	ProtoBuf [48]	2 038	339	128	343	4	129	424	85	198	433	90	9	204
	Tesseract [49]	1 841	754	331	758	4	332	850	96	428	857	99	7	431
	Xerces [50]	1 655	492	196	508	16	200	555	69	241	671	163	116	299

“Relatedness” filtering: `int min(int a, int b)` should not be reported.

Distribution of culprit types

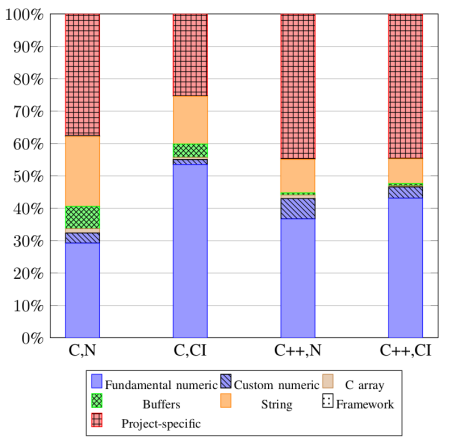


Figure: Distribution of types (hand-made classification) that contribute to adjacent mix possibility across project. (*N*: strict type equality, *CI*: conversions measured)

References

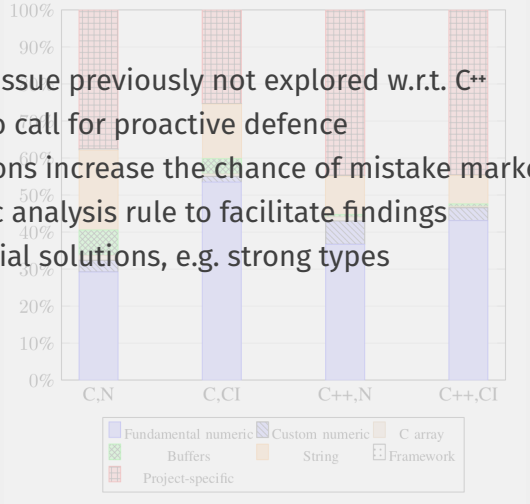
[1] ISO/IEC JTC 1/SC 22. *ISO/IEC 14882:2017 Information technology — Programming languages — C++, version 17 (C++17)*. Geneva, Switzerland: International Organization for Standardization, Dec. 2017, p. 1605.

[2] Andrew Rice et al. “Detecting Argument Selection Defects”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), 104:1–104:22. ISSN: 2475-1421. DOI: 10.1145/3133928.

[3] Xperience AI. *OpenCV*. version 4.2.0 (bda89a6), accessed 2019-12-30. 2019-. URL: http://opencv.org.

Conclusion

- Investigated the issue previously not explored w.r.t. C++
- Changed scope to call for proactive defence
- Implicit conversions increase the chance of mistake markedly
- Tool-driven static analysis rule to facilitate findings
- Discussed potential solutions, e.g. strong types



Questions...

- Have you ever made this mistake yourself?
- Are there conventions you know people follow when designing interfaces?
- Which domains do you think should be the first target of refactoring?
- Should we aim for domain-specific solutions, or try for general ones?
- How should we go through with a potential refactoring?