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

Richárd Szalay, Ábel Sinkovics, Zoltán Porkoláb
Department of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University
Budapest, Hungary
szalayrichard@inf.elte.hu, {abel,gsd}@elte.hu

***Abstract*—Argument selection defects, in which the programmer has chosen the wrong argument to a function call is a widely investigated problem. The compiler can detect such misuse of arguments based on the argument and parameter type in case of statically typed programming languages. When adjacent parameters have the same type, or they can be converted between one another, the potential error will not be diagnosed. Related research is usually confined to exact type equivalence, often ignoring potential implicit or explicit conversions. However, in current mainstream languages, like C++, built-in conversions between numerics and user-defined conversions may significantly increase the number of mistakes to go unnoticed. We investigated the situation for C and C++ languages where functions are defined with multiple adjacent parameters that allow arguments to pass in the wrong order. When implicit conversions are taken into account, the number of mistake-prone function declarations significantly increases compared to strict type equivalence. We analysed the outcome and categorised the offending parameter types. The empirical results should further encourage the language and library development community to emphasise the importance of strong typing and the restriction of implicit conversion.**

***Index Terms*—static analysis, function parameters, argument selection defect, type safety, strong typing, error-prone constructs, C++ programming language**

I. Introduction

In statically typed programming languages, each parameter of a function is given a type, and the compiler is responsible for ensuring that only expressions of the expected type are given as argument.[1] Unfortunately, the detection mechanisms in compilers are defeated if multiple parameters are declared adjacent to each other with the same type. A swap of adjacent arguments at a call site slips through semantic checks as the types of the swapped arguments still match the interface specified. Given a function `fn(int x, int y)`, both `fn(1, 2)` and `fn(2, 1)` are valid calls. In addition, due to *implicit conversions* that are possible in various mainstream programming languages, such as C++, `fn(1.5, 3)` is also a valid call, even though the function is not directly taking floating-point values. Developers often use the identifier name of the parameter to convey semantic information about the values expected in place of a parameter. While research has been done on understanding natural language for multiple aspects of the software, including identifiers names [1]–[3], the semantic information conveyed by the identifiers themselves are not considered by virtually any compilers of mainstream languages.

Various issues might arise in case the developers inadequately pass arguments to functions and do not get a diagnostic about it from the compiler. Run-time issues might cause unexpected results and incorrect execution which could lay hidden unless extensive functional and integration testing is performed, or worse, a trouble report is raised by users or customers affected by the issue. Inadequately typed function parameters hinder the program's maintainability, as any development or understanding effort is set back by questioning why a particular expression was passed to a particular parameter even though the types match. These issues are hard to identify within traditional development pipelines, unless, on top of testing the program's behaviour, tools specific to these cases are employed. Several tools considering the *names* of arguments and parameters are discussed in Section II. However, by employing language features, the programming interfaces of software projects can be made more resilient against argument selection defects.

We sampled the Internet for results of argument swaps reported to cause issues. Such a case was an argument swap issue in the GCC compiler's implementation to a particular architecture had two integer arguments of the same type reversed in a subtraction [4]. While not C++ code, a mixed string replace issue was found hidden in LLVM's build system [5]. Regular expression libraries commonly take the `needle` and `haystack` parameters as just strings, and as such, allow for such issues to go undetected by the compiler. In addition, in the Mozilla Firefox browser, a function taking a number and a boolean was called with the arguments reversed [6]. However, in this case, the two parameters did not have the same type. C and C++ allow for implicit conversions between types, the risks of such argument swaps are not investigated in earlier literature.

In this paper, we present an automatic static program analysis rule that diagnoses function definitions that contain mul-

This work presented in this paper was supported by the European Union, co-financed by the European Social Fund in project *EFOP-3.6.3-VEKOP-16-2017-00002*.

[1]In line with the literature of the field, we will refer to formal parameters appearing in functions' declarations and definitions as *parameters*, while the expressions from which actual parameters are calculated will be referred to as *arguments*.

tiple adjacent parameters which have the same type. Targeting definitions instead of call sites has the benefit that it warns the developers for an error-prone interface already during the function's development. The main additional contribution is that we also consider potential implicit conversions from one parameter's type to the other – a problem which was not evaluated in previous literature. The rule can be applied with minimal effort, as it only uses the source code, and no domain-specific information is required from the user. Our particular implementation was developed on top of the LLVM/Clang Compiler Infrastructure project. Given the reliance on a well-known compiler, the analysis can be integrated directly into active development, assuming the project can be compiled with Clang.

We measured open-source C and C++ projects of various scale and domain to obtain results on how function interfaces might be misused. We found that considering *implicit conversions* between the parameters' types, the number of potential mistakes increases markedly. Implicit conversions are a feature of the language that is generally never warned about by the compiler, further allowing potential misuses to go unnoticed. Although our analysis is applicable for investigating existing projects, the main goal of the rule is to prevent the creation of possibly error-prone code constructs early on in a project's design phase.

This paper is organised as follows. We discuss prior literature related to the topic of function parameters and argument selection defects in Section II. We define and detail the main target of our analysis rule, *type-equivalent parameter ranges* in Section III. Implicit conversions as a language feature and their theoretical effects on the results are presented in Section IV. In Section V, we discuss our empirical findings on various open-source projects. Potential solutions to the problem are given in Section VI. Restrictions of this research are mentioned in Section VII. Conclusions are drawn in Section VIII.

II. Related Work

The fact that compilers do not give any semantic worth to human-written identifiers had been identified as an issue of code comprehension and refactoring efforts. Several works discuss how poorly chosen identifier names, including formal function parameters, hinder code comprehension [7]. Multiple kinds and contexts of identifier names have been studied in [8]. A paper by Caprile and Tonella [9] discusses how function names are constructed, and that semantic information – lost to compilers and purely syntactic tools – is encoded in the name. Their subsequent work in [10] proposes a method for automatically standardising identifier names. Selecting good arguments to function calls has been studied by Zhang et al. [11]. They showed an automated technique, *Precise*, which suggests arguments at a call site based on a database of calls to the same library from other existing code.

The problem of a potential mismatch between formal parameters and passed arguments slipping through the semantic checks of the compiler had also been studied before. Pradel and Gross [12] emphasise the power of type checking systems with regards to finding *anomalies* in arguments passed to function calls. The anomaly manifests as multiple arguments of compatible types passed out of order. They developed an automated tool that requires no additional knowledge apart from the source code of the project itself. It works by gathering information of all call sites for each function. If argument names at a particular call site are unlike than all other call sites, a warning is issued. They have analysed their approach on a sizeable real-life corpus of Java applications and found good precision for the detection of anomalies. A subsequent paper [13] expands upon the previous work by applying their tool for finding anomalies on more Java and C programs, showing that the problem is not restricted to just Java. The authors improved the accuracy of their method. They also included an additional feature in their tool that searches insufficiently named parameters. Parameters' names are deemed insufficient if most calls to a function agree on a particular nomenclature, but it differs from the names of the parameters themselves.

Liu et al. [14] investigated the connection between formal parameter names and the names of arguments passed, and concluded in their empirical study that the similarity in most cases takes the two extremities: either very high (almost or precisely the same), or very low (dissimilar). Their study was run on 60 real-world Java programs. They also studied their approach for two use cases: for suggesting renames of misnomers inferred from the call sites, and for selecting a different argument at a call site, with high precision. This work compares arguments at one call site with the parameters of the called function, and most parts work across all arguments of the function with no regards to type. The suggestion of better matching arguments does consider type conformity.

Extending these works, Rice et al. [15] have integrated an automated check for argument mismatches to their development pipeline at Google Inc. and evaluated their implementation on substantially sized corpora of Java projects spanning 200 million lines of code, including proprietary and open-source. They measured the relative power of string distance functions by hand labelling a test set of ~ 4000 pairs of argument–parameter names to fine-tune the distance functions' thresholds. The paper discusses 84 true positive findings on real projects, one of which was a severe security vulnerability that had laid dormant for more than two years. They also found that the probability of an argument selection anomaly increases quickly once a method has more than 5 parameters. Similar approaches using string distance functions were implemented in [16], [17] to warn about potential swapped arguments for function calls with the help of compiler tools for C and C++ projects.

Our approach is similar to the works of Pradel, Liu, and Rice in using accurate semantic information obtained from compilers. However, their works contain an explicit precondition that arguments and parameters must be named, or calculated from the surrounding context in some fashion. This is a severe restriction, as it excludes all function calls where literals are passed, such as `fn(1, 2)`. Butler et al. [18] described means to extract meaningful identifier names from Java source code. These works fall into the same domain as our paper, but they all attempt at warning developers for mistakes already made, whereas our paper suggests taking a defensive design and leverage the type system.

The most notable case in the C and C++ world where name-based analysis is insufficient is from the *"memset bug family"*. The function's signature is `void* memset(void* buf, int value, size_t num)`. It sets `num` bytes in the provided `buf` buffer to the given `value`. While there are plenty of issues related to *memset*, one particular to our paper is the potential to swap the two numeric arguments, that is, to call the function as `memset(&t, sizeof(T), 0)`, which will result in no changes. No argument names appear in this swapped call, and even if a heuristic "names" the second argument `size`, the *name* of the second parameter is distinct from "size". A `memset`-related bug fix pops up recurringly, and it is such a problematic interface with a high chance of misuse that dedicated analysis rules [19], [20] exist to match `memset`'s case specifically. While it is unlikely that a standard library function dating back several decades will ever be changed, a highlight of type-based analysis is that it can warn immediately for a potentially offending function, prompting a potential clarification of the design. Our approach would warn about this in the code that defines `memset`.

There are several works in the literature that discuss the automated, tool-driven synthesis of type constraints. Guo et al. [21] detail how run-time interaction between variables can be used to infer similarities in the concept represented by some variables, and thus unite these variables to have a common, shared type. Hangal and Lam [22] propose a tool that automatically corrects errors in Java programs related to *dimensionality* – e.g. using an integer variable representing a square number (such as *area*) for a scalar (such as *length*) parameter. This tool does interprocedural context-sensitive analysis and infers possible dimensions or units for variables from their usage points. *RefiNym* [23] is an automated unsupervised learning tool that models the flow of values and expressions from one variable to the other and suggests more fine-grained types based on the information gathered.

These works may, in the future, serve as further steps to take for making software more type-safe. Combining our analysis and previous works discussed, one can obtain a set of "pain points" on which these inferring tools can target.

`Chrono`, the C++11 standard library for representing time and duration, is related to our work in terms of leveraging the type system to express and enforce dimensions and similar to [22]. Chrono can be viewed as a solution that aimed to solve some issues discussed in this paper for a particular domain. Other solutions that enforce dimensionality through the type system also exist for physical units [24].

The problem applies to other mainstream programming languages with varying degree. For example, in Scala the set of possible implicit conversions is broader than in C++ as Scala allows implicitly converting the instance before member access is performed [25], [26]. On the other hand, Rust supports no notion of implicit conversions [27], [28].

Several well-known guidelines, restrictions, and domain-specific spin-offs for C or C++ such as MISRA C [29] or the SEI-CERT secure coding guidelines [30] contain rules that guard against implicit conversions of numbers in any context, not specific to function parameters.

III. Type-equivalent parameter ranges

We extend the scope of the problem discussed in previous works regarding swapped or badly ordered arguments to signatures of the called functions. There are a few critical differences in the language's workings and the compilation process when Java (for which most of the related work has been done) and C++ is compared. One difference is the existence of *separate compilation* [31] in C++: the compiler is restricted to the information contained in the *translation unit* – the source file and all headers and modules included – of the current source file being compiled; unlike Java, which compiler is allowed to read other source files' contents. The names of the parameters do not form part of the symbol table, and as such, a header file, or for C++20 a module interface unit, may contain only the types of the parameters. Given a function signature `int fun(int, int);` it is evident that any call to this function contains a *possibility* for the arguments to be swapped, as when called with the right type, the compiler will deem the call correct. While the above signature is correct from the language's perspective, such constructs are extremely rare as developers tend to write the variable names in the header files to be able to give the extra semantic information that is conveyable through identifier names [32].

The issue from passing arguments that are type-equivalent in a potentially harmful order is not in itself a novel finding. When faced with possibilities of misuse and anti-patterns, teams, project or a broader community of developers tend to create rules of thumb or guidelines. One such guideline for C++ is the *C++ Core Guidelines*, drafted initially and curated by the creator of C++, Bjarne Stroustrup. This guideline contains a rule named *"Avoid adjacent unrelated parameters of the same type"* [33]. To our knowledge, there were no free and open-source automated tools that check for possible violations of this rule before. The meaning of two parameters being *related* to one another is not discussed in the guidelines. For example, if the above `fun` function were `int max(int a, int b);` then the parameters would be related as swapping arguments in a call is still valid. Such a case should not be warned about, as there is no sensible way of resolving the "ambiguity" with changing parameter types. We offer some heuristics to select related arguments which we discuss in Section V.

A possibility of mixing arguments at the call site might not be apparent at first glance. Due to language features such as references, lifetime extension, and type aliases, a deeper understanding of the context in which the signature appears is necessary. While developing the static analysis rules for our tool, we investigated the language rules to match the non-trivial cases, such as as the one depicted in Listing 1.

```
typedef Number int;
int fn(int i, const int& iref, Number i2);
```

Listing 1. All three parameters of function `fn` are mixable with each other at any call site, due to all three parameters binding to an argument of type **`int`**.

Definition 1. *Two parameters are* ***mixable*** *if there exists a type T for which an expression of that type might bind to both parameters' types at a call site.*

Definition 2. *Given all adjacent ranges in which parameters have mixable types with each other, the* ***type-equivalent parameter ranges*** *will be the longest such.*

There are several language constructs in C and C++ which require special handling when deciding whether a parameter of a type can be mixed with another:

A. `typedef`s and `using`s

`typedef`s in C and C++, and the `using` keyword in C++ version $\geq$ 11 denote type aliases. Such alias names are interchangeable with the type referred by them. These constructs are often employed by projects to emphasise the different role of the same type – which is futile, as in this case, the alias name has the same semantic power of the variable's identifier, and is not checked by the compiler – or to hide platform-specific variations to a central point of a library.

B. `const`, `volatile`, `restrict` qualified types

Type qualifiers can be used to create variables of any type for which the behaviour of access is changed. Qualifiers do not extend the possible operations on a type, nor they change the type's representation. The *C++ Core Guidelines* rule in [33] show an example where a memory copy function should take the `source` parameter as **`const`**, indicating that the source buffer should not be written to by this particular function. However, it is possible to call such a function with two `T*`s, as **`const`** only changes the behaviour inside the function. Potential removal of qualifiers at a given call site is diagnosed by the compiler, as a warning for C and a hard error for C++, and as such, the bogus ordering will be made apparent to the developer.

C. References (`&`, `&&`)

References in C++ allow creating variable symbols with different names which all bind to the same instance in memory. From the user's point of view, reference variables at a point of usage behave precisely like normal variables. We considered the "binding power" of an expression of type T to a reference parameter to ensure *mixability* is calculated correctly. A `T` and a **`const`** `T &` parameter mix perfectly due to *lifetime extension* [31], [34]. This is not true for other kinds of references, such as non-**`const`** *lvalue* or an *rvalue* (`&&`) [35]. A temporary value of an expression, e.g. a function call's result, can not be bound to the former. A local variable which is within its *lifespan* cannot be bound to the latter. Thus we consider `T &` and `T &&` as distinct types which do not mix.

```
struct BoundHost { BoundHost(int hostID); };
packet transmit(BoundHost host, int amount);

// Suppose two local scalars...
int H = 2130706433, s = 4096;
```
In C++, both orderings of arguments are valid as written.
```
transmit(H, s); // → 4096 bytes sent
transmit(s, H); // → 2130 million bytes sent
```
In Java, the conversion to `BoundHost` must be explicit.
```
// ϟ error: "incompatible types:
// int cannot be converted to BoundHost"
transmit(H, s);

transmit(H, new BoundHost(s)); // ϟ error!
transmit(new BoundHost(H), s);
```

Listing 2. Implicit user-defined conversions for C++, such as a *converting constructor* can hide a possibility to select wrong order of arguments.

IV. Implicit conversions

Previous works mentioned in Section II mainly focused on Java. While some of the works implemented type-conformity checks, compared to Java, in C++ there is the possibility to create user-defined *implicit conversions*. The issue with implicit conversions is depicted in Listing 2: the user may pass two **`int`**s at a call site which gives different values depending on the order, but the type conversion is not apparent, and no warning is emitted.

Implicit conversions are considered by the compiler if an expression of type T_1 is to be assigned a variable of type T_2, such as in the case of `T2 var = makeT1();`. This can be done if and only if there is exactly one, unambiguous *implicit conversion sequence* [31], which in C++17 and C++20 consists of the following three steps:

- At most one *standard conversion sequence*
- At most one *user-defined conversion* – executing **either** a suitable converting constructor or a conversion operator
- At most one *standard conversion sequence*

To allow modelling implicit conversions when checking for potentially swappable parameters, the definitions in Section III are extended as follows:

Definition 3. *Two parameters of not necessarily the same type T_1 and T_2 are* ***mixable*** **through implicit conversions** *if the implicit conversion from T_1 to T_2 or from T_2 to T_1 is possible.*

Implicit mixability is not a symmetric property. As an expression of type T is always assignable to a variable of type T, taking $T_1 = T_2 = T$ gives us Definition 1. Thus, implicit mixability is a broader set than type-equivalent mixability.

Definition 4. *Given all adjacent ranges in which parameters are* mixable through implicit conversions *with each other, the* ***implicitly mixable parameter ranges*** *will be the longest such.*

User-defined types in C can not have member methods, and as such, no constructors or conversions may exist. In case the

```
struct FromInt { FromInt(int); };
struct ToInt { operator int(); };
enum En { x, y, z };
void f(int i, const int& ir, double d,
       ToInt ti, En e, FromInt fi);
```

Listing 3. Example where *implicit conversions* are possible between most of the parameters for the entire function. For the two `struct`s, only one-way implicit conversion is possible.

analysis is done for C source files, *implicit conversions* will refer to the possibility of converting any numeric value to another. We ignore the potential numeric conversion between pointers of any `T *` to an unrelated `U *`. This case is diagnosed by a compiler as a warning in C and had been made an illegal operation in C++. An example of Definition 4 is depicted in Listing 3.

A. Standard conversions

Standard implicit conversions may have up to 4 steps, in order: *lvalue transformation*, *numeric conversion*, *function pointer conversion* and *qualification adjustment*. At most one of each step might be present in a *standard conversion sequence*. *Lvalue transformation* (e.g. array-to-pointer conversions) and *function pointer conversion* (assigning a function pointer denoted as "may throw" to a `noexcept`, not throwing function) steps are not relevant for our paper – in case these steps were needed to take place at a parameter passing, they will take place also if the order at a call site is swapped. *Qualification adjustments* are handled as discussed in Section III-B.

Numeric transformations are further broken down to two categories, *promotions* and *conversions*. The distinction between the two is that promotions preserve value, while conversions may truncate the value of the expression at hand. Other aspects of this distinction – such as how overload resolution prefers to select the function to which the argument maps with a *promotion* – are not relevant for our study. Numeric transformations refer to the rules that any integer or floating-point number may be converted to any other integer or floating-point number, at any point deemed necessary, giving a two-way passage between any "number". For C++ `enum`s, the enumeration constant is always convertible to an integral or floating-point number. This is not true for *scoped enumerations* (`enum struct` or `enum class`), which are not implicitly convertible in any direction. For C `enum`s, the conversion from a number to the enumeration constant is also possible. Upcasting a derived type's pointer to the base class is also defined as a *numeric conversion*.

B. User-defined conversions

User-defined conversions take the form of *converting constructors* (depicted in Listing 2) and *conversion operators* (depicted in Listing 4). At most one of such method can be executed as part of an implicit conversion attempt. By applying the `explicit` keyword on a conversion method, the library developer can specify that the method must not be part of an implicit conversion sequence. This applies to all cases where the now-disabled conversion method would fit.

```
struct Complex {
  Complex(float Re, float Im);
};
struct Int {
  int value;
  Int(Complex c) { value = c.Re; }
  operator Complex() const {
    return Complex(value, 0);
  }
};
```

Listing 4. Conversion operators allow for a type to define how it converts **to** another type. It should be noted that passing the two arguments to `Complex`'s constructor in itself contains an implicit conversion from `int` to `float`.

V. Evaluation

We created a practical implementation [36] for the analysis built on top of the open-source LLVM/Clang compiler infrastructure [37], which allowed us to find and report occurrences of the problem automatically. The implementation works by checking function definitions in the project and calculating whether two parameters could be mixed at a call site based on their type. These diagnostics reports are written in a user-friendly way, and thus, the *checker* can be easily integrated into continuous integration (CI) systems. The analysis rule is in the process of being reviewed and accepted into the upstream Clang code at the time of writing this paper. A sample of open-source projects was analysed from medium to large scale, encompassing various domains from system tools to machine-learning image processing libraries. The system requirements of the analysis is consistent with other compiler-based tools, taking 10 to 60 seconds for each project – excluding LLVM itself, which took 33 minutes – on a 24-core system.[2]

A. How many functions are affected?

A detailed breakdown of the number of functions that have mixable adjacent parameters based on the parameter types is shown in TABLE I. We compared different configurations corresponding to different levels of relaxation in the rules. Users of the analysis can toggle between these relaxations to fine-tune the strictness.

We only *considered* and evaluated findings from functions which are *defined* in files of the project analysed – functions from headers included as "system headers", usually those from third-party libraries, are ignored. Similarly, we ignored all findings that are of functions taking pairs of *iterators*, which is a common cause of two adjacent parameters being swappable with each other.[3] Typed variadic functions are treated with their variadic parameter counted as a single parameter.

[2]Most of the time spent is for the compiler's semantic analysis, which is irrespective of our specific rule, which needs parsed representation first.

[3]At the time of writing, a proposal labelled *Ranges* is on track for inclusion into the next release of C++, currently set to be *C*++23, which will allow replacing pairs of same-type *iterators* with a single parameter.

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

- In **Normal** mode, only exact type-equal ranges are matched, with `typedef`s and references (Sections III-A and III-C) always diagnosed.
- **CV** mode allows mixing types that only differ in their qualifiers (Section III-B), e.g. `int, const int`
- **Imp** mode enables calculating and considering *implicit conversions* (Section IV), e.g. `double, int`
- In **CV ∪ Imp** mode, both relaxations are enabled, e.g. the following are mixable: `double, const int`

Unfortunately, there are functions which cannot be reasonably changed to guard against swaps. In accordance with the C++ Core Guidelines rule [33] we implemented some heuristics to filter out functions where only *related* parameters are mixable. These are our heuristics – the Guideline at the time of writing this paper does not detail what predicates should be used. We defined the following criteria for relatedness:

- Parameters which appear in the same expression – such as an assignment, a comparison, a function call – together, such as `f(a, b)`, which takes care of the most common case of direct relatedness, such as the `max` function.
- Parameters which are passed to another function's same parameter, but on different code paths, such as `f(a)` and `f(b)`, which filters out forwarding or dispatching.
- Parameters which are returned inside the function on different code paths. This also helps filter out dispatchers, and selector functions which return one or the other of their parameters based on program state.

Users may toggle relatedness-checking for their project as they see fit. When two, otherwise mixable, parameters are deemed to be related, the equivalency set (see Definition 2) is split. The **R** columns in TABLE I indicate the number of functions that are still mixable, despite related parameters. On average, a 40% reduction of report count was achieved, which is beneficial in an industrial setting where analyses necessarily let false negatives slip through for the benefit of culling false positives [51]–[53] and allowing developers to spend their time better.

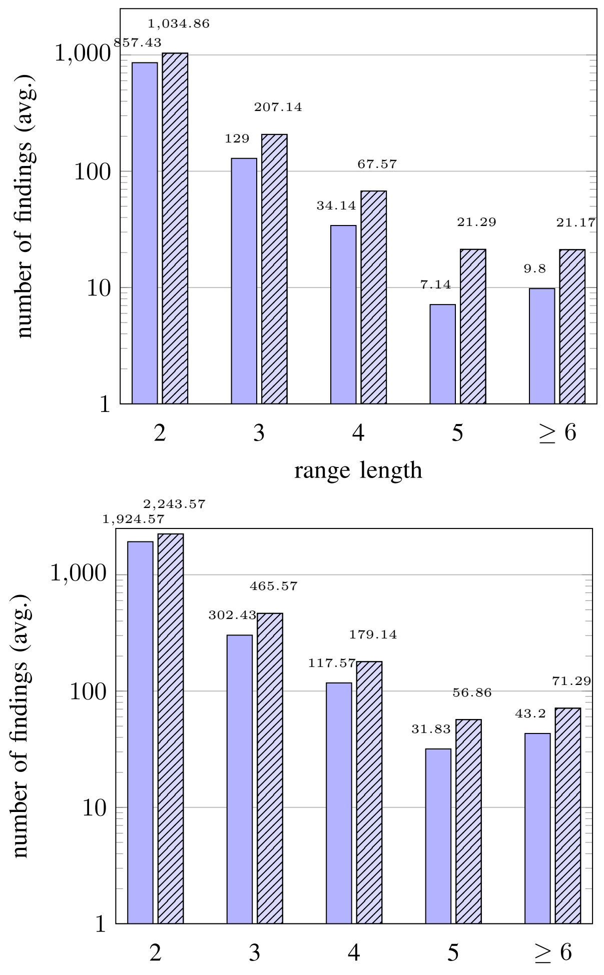

Fig. 1. Count of findings by length of mixable adjacent parameter range, averaged across the analysed projects. Filled columns depict normal mode, striped columns depict *CV* and *implicit* modelling turned on. The above picture is for **C**; the below is for **C++** projects.

TABLE II
NUMBER OF REPORTED RANGES HAVING A PARTICULAR LENGTH FOR C AND C++ PROJECTS. THE *max.* COLUMN INDICATES THE LONGEST FINDING, IF ≥ 6. `N` - NORMAL MODE, `CI` - *CV* AND *implicit conversions* CONSIDERED (SEE SECTIONS III-B, IV)

Project	**2**		**3**		**4**		**5**		**≥ 6**		**(max.)**	
	N	*CI*	*N*	*CI*	*N*	*CI*	*N*	*CI*	*N*	*CI*	*N*	*CI*
curl [38]	125	212	12	19	5	8	1	2				
git [39]	1 337	1 521	145	214	50	65	2	6	3	6	*6*	*7*
netdata [40]	206	264	37	56	10	16	5	9	9	15	*12*	*12*
PHP [41]	1 085	1 263	221	294	33	56	4	12		6		*8*
Postgres [42]	2 510	3 112	371	714	108	269	26	93	31	88	*9*	*20*
Redis [43]	535	626	75	103	14	32	5	21	1	5	*6*	*7*
TMux [44]	204	246	42	50	19	27	7	6	5	7	*7*	*7*
Bitcoin [45]	345	623	82	107	14	39	3	8	1	5	*6*	*7*
guetzli [46]	75	71	17	18	8	9	1	4		2		*7*
LLVM/Clang [37]	7 191	8 571	764	1 258	194	375	58	127	29	77	*13*	*13*
OpenCV [47]	4 461	4 789	1 056	1 604	513	687	118	226	157	371	*20*	*21*
ProtoBuf [48]	315	369	23	54	5	15	1	3	7	7	*10*	*11*
Tesseract [49]	652	678	123	156	58	85	10	26	22	36	*11*	*11*
Xerces [50]	433	604	52	62	31	44		4		1		*6*

B. How long are the mixable ranges?

Reports of ranges of length 2 are the most prevalent across all projects and configurations, making up more than half of the total findings. These results are consistent with findings in related works (see Section II) [13], [15], [17] employing name-based analysis to find ordering issues, where single adjacent arguments' swaps were the majority of noteworthy findings. Exact counts of findings for each project for normal – most restrictive – and *CV & Implicit* – least restrictive – configurations are shown in TABLE II. The average number of findings of a particular length is depicted in Fig. 1. We plotted the values of C and C++ separately due to the broader set of what is considered *implicit conversions* in C++. While it is natural from the rules of the languages that relaxing the "equal type" predicate and searching for longest subranges result in longer ranges being matched or adjacent ranges being joined together, the order of increment between most and least restrictive configurations shows a powerful creep towards more lengthy function signatures.

C. How different types contribute to the issue?

Another interesting result of our measurements is the distribution of types that are involved in the findings. We have investigated the parameter ranges reported and hand-categorised the types into the following categories:

1) *Fundamental numeric* types are all fundamental, built-in, "keyword" numeric types, including integers, floating-point numbers and enumerations.
2) *Custom numeric* refers to other types of a single numeric nature, such as custom precision integers (e.g. `int512`).
3) *C arrays* refer to C-style array expressions, such as `int T[]`.
4) *Buffers* refer to all types of type-erased (`void*`) or template (`arrayRef<T>`) wrappers over buffers, including files, and sockets, and arrays of `std::byte`.

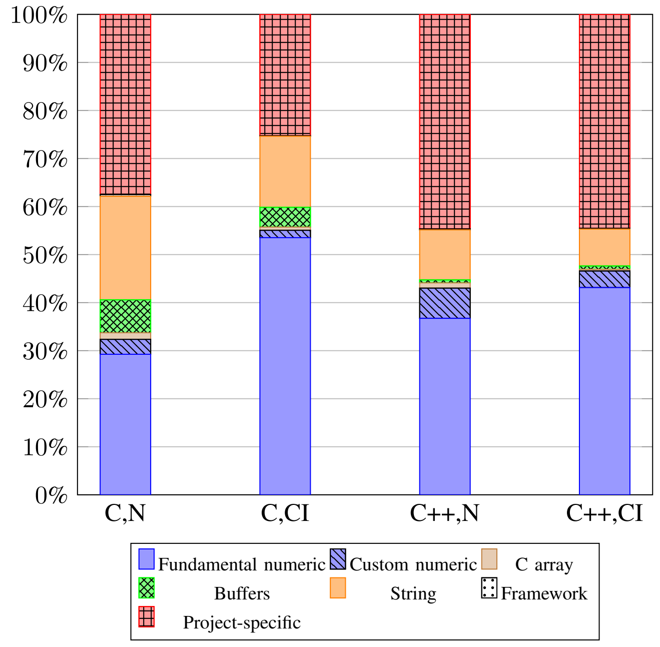

Fig. 2. Relative cardinality of type categories. Pointers/references of `T` counted as `T`. *Uncategorised* refers to user types that are specific to the project. (`N` - normal mode, `CI` - *CV* and *implicit conversions* considered)

5) *Strings* refer to all parameters that take `char*`, `std::string` or types related to string operations (like `std::string_view`, etc.). We admit here that it is not trivial to distinguish directly from the signature whether a `char*` is used for a string or for a buffer of byte-sized elements.
6) *Framework types* are all standard (C standard or C++ *STL*) types that do not fit into the previous categories, and every type that comes from a well-known framework

the project depends on – such in Bitcoin's [45] case, *Qt*.

7) The last category labelled *Project-specific*, is the "catch-all" bucket where every other type – mostly user types – are put.

Pointers or references of a type in category $\mathcal{S}$ are calculated as category $\mathcal{S}$. It should be noted that a non-pointer and a pointer is never reported as mixable for clarity of results, even for C where numbers and pointers are mixable with each other, as compiler warnings cover this case. The relative number of types involved in the findings for a particular configuration is depicted in Fig. 2.

Fundamental numeric and *uncategorised* categories being the two largest across the evaluation follows natural expectations. The authors were surprised that the size of the former increases markedly for C and considerably for C++ when implicit conversions are reported, showing that there is a corpus of functions similar in nature to `f(int a, double b)`.

Our findings confirm that the level of detail in projects' types seem insufficient to prevent misuse through poorly chosen arguments. This marks the need for tools to help prevent the mistakes happening by considering implicit conversions in addition to type equivalence. One such tool could be name-based analysis (see related works in Section II). However, there are cases where name-based analysis might be inapplicable.

D. Details on noteworthy findings

A typical cause of lengthy findings is numerous `bool` parameters, such in the case of LLVM/Clang's [37] function `AnalysisDeclContextManager` taking **12** toggles. Several other functions like `WriteSecHdrEntry`, `resolveRelocation` take ~ 10 numeric parameters with no restriction or semantic information to be inferred from the type.

OpenCV [47] uses the types `InputArray` and `OutputArray` as wrappers to indicate whether their functions take input or output parameters. These types can be constructed, according to the documentation, deliberately from seemingly all major data structures used in the project in an implicit fashion and should *"never be used directly"*.[4]Due to the subtype relation `OutputArray` <: `InputArray` holding between the classes and the copy constructor not being explicitly deleted, there is an implicit conversion possible from `Output` to `Input`. There are several functions with large sets of mixable arguments resulting from this "type erasure": `cv::rectify3Collinear` takes 8 `InputArrays`, then a numeric type, then 4 `InputArrays` and 7 `OutputArrays`. All matched "type-erased" arguments have names akin to generic arguments, such as `Rmat12`, `Qmat`. The relatedness heuristics in Section V-A break these long ranges up to adjacent ranges of 4 or less parameters.

In PostgreSQL [42], the longest result is a function named `TypeCreate` that has 20 numeric parameters adjacently. Other functions – such as `rrdset_create_custom` – do not distinguish between the various string-like arguments received, accepting any `const char*`s. There are similar matches in Tesseract OCR [49] of functions with ≥ 9 adjacent numeric arguments. Relatedness heuristics did not filter these extremely long cases.

VI. Potential solutions

In the following, we will overview a few solutions that could prove useful to disallow badly ordering similarly typed arguments. Some of the solutions are useful in industrial-scale projects if the developers consistently implement it, while some are theoretical for the general situation, with implementations existing for specific use cases.

A. Declaring forbidden overloads

The issue of implicit conversions can be side-stepped in C++ by explicitly creating overloads that are marked with the `= delete;` specifier. For example, given functions `void f(int) = delete;` and `void f(long) {}`, calling `f(42);` will resolve to the *deleted* overload as opposed to performing an implicit conversion, and a compile error will be emitted.

While theoretically such a system would disable the issue with implicit conversion, generating all possible overloads for all possibly affected functions is a daunting task. In addition, it would result in code bloat by having $\mathcal{O}(n^2) - 1$ disallowing declarations for each pair of overloads present.

B. Explicit type aliases

One possible solution to badly ordered arguments is to make the adjacent types incompatible with each other. An example of wrapping two `int`s can be seen in Listing 5. This technique is commonly called an *explicit type alias* or a *semantic typedef* and works by creating a wrapper type over the wrapped type and providing wrap and unwrap methods. There is no runtime performance drawback of the technique, as all major compilers optimise the relevant calls away. Once the types of parameters are succinctly distinct, any mixed arguments will be immediately reported by the compiler as an error. This makes the conversion explicit in the code, similar to what is required in Java (see Listing 2). Given the additional function calls being optimised out and due to *value semantics*, the size and behaviour of the semantic typedef instance are the same as the single variable contained within, with no additional steps to take at destruction. This is not the case for Java, where the heap allocation is done, and the boxing types cause a performance hit [54].

Semantic typedefs offer an easy and straightforward solution but cause an explosion in the number of types visible in scope, which may hurt compilation time and lessen development productivity [55]. Built-in support for such language elements is part of neither C nor C++. Other languages, such as Haskell support a similar notion via the `newtype` directive. There had been proposals [56], [57] to include *opaque typedefs* for C++ but these have not make it into the language yet.

[4]Quote from the documentation of `cv::_InputArray` in [47]: *"The class is designed solely for passing parameters. That is, normally you should not declare class members, local and global variables of this type."*

```
void drawBad(int width, int height);

struct Width {
  int value;

  // For C++:
  explicit Width(int v) : value(v) {}
  explicit operator int() const {
    return value;
  }
  int operator()() const {
    return value;
  }
};
struct Height { /* Analogous... */ };

void draw(Width w, Height h) {
  int wi1, he1 = w.value, h.value;

  // Obtaining values in C++:
  int wi2, he2 = w(), h();
  int wi3 = (int)w, he3 = (int)h;
}

// ↯ error: no implicit conversion
//   from 'int' to 'Width'
draw(1, 2, RED);
// ↯ compile error, type mismatch
draw(Height{2}, Width{1}, RED);
draw(Width{1}, Height{2}, RED); // ✓ Works!
```

Listing 5. Transformation from the same type to a *semantic typedef* or wrapping type disables mixing, potential implicit conversion and misuse at a call site.

Similarly, Baráth and Porkoláb [58] discusses a wrapper class over numeric conversions. The LLVM project, in which several functions take multiple boolean parameters adjacent to each other (see Section V-D) have been using comments to indicate which parameter is assigned a literal value, and community members have suggested implementing wrappers around such instances [59].

Function signatures might commonly repeat identifier-like phrases, such as `f(ShouldFlip flip, ShouldStretch stretch)`. What is more, looking at the function declaration might not offer enough clarity – except for a potential heuristic that lets developers assume `bool` parameters from a `ShouldXXX` – for more complex cases, resulting in excess navigation to the wrapper type's definition.

C. Strong typing

A particular issue with wrapping types is that their usage solves only the problem of adjacent argument mix-ups. Apart from argument-forwarding functions, the developers would always wrap and then unwrap the value, and within the business logic of the program, the wrapped type would be used. Strong

```
#include <chrono>
using namespace std::chrono;
using namespace std::literals;

// Bad: prone to bad order of arguments.
bool submit_at_1(
  int year, int month, int day,
  int hour, int minute, int second);
// Bad: "Seconds" is not descriptive.
bool submit_at_2(double seconds);

bool submit_at_good(
    time_point<system_clock, seconds> T) {
  auto DLDay = 2020y / Aug / 14;
  auto DLSecond = 24h - 1s; // = 86 399 sec
  auto AOEDeadline = zoned_time(
    "Etc/GMT+12", DLDay + DLSecond);

  return T ≤ AOEDeadline.get_sys_time();
}

int main() {
  // Order of arguments mixed up.
  submit_at_1(11,59,59,2020,8,14);
  // Semantically incorrect, yet compiles.
  submit_at_2(get_milliseconds());

  // ↯ compile error: no conversion.
  submit_at_good(2020);

  submit_at_good(system_clock::now());
}
```

Listing 6. Comparing traditional, not safe versions with using stronger types and type-safe "strong" literals for representing time and deadlines with the `chrono` library. Program execution shows as exit status whether the deadline has not been hit yet. ('+' sign in timezone name is inverted according to ISO standards, "`Etc/GMT+12`" indicates *UTC-12*.)

typing [60], in which the expressive capabilities of the type system and types used in the program is increased, has been investigated for their effect on language design [61] and as method to increase type coherence for persistent systems [62] and to prevent security vulnerabilities in web applications [63].

A more actionable solution to the issue is to increase the type safety of the project by introducing user types and relying on the compiler to find type non-conformance violations. It is very likely that there are hidden invariants [55], [64], [65] behind most of the `int` or `char*` parameters, that are checked somewhere during execution. Such cases could be transformed into types that ensure invariants. One such invariant could be that a numeric value must be within a specific range, narrower than the fundamental type would allow. Expressing this is possible in Ada with the `Range Lower..Upper of Integer` syntax [66]. Another case could be if there exist specific patterns a string-like parameter must adhere to, e.g. it is a time code or a name.

Using stronger – from the compiler's perspective user-defined – types will immediately make adjacent parameters of different invariants non-mixable.

While strong typing is a powerful solution in theory, user and library developer-friendly generic language elements are not widely researched. We plan to investigate the solutions in detail as part of future work.

D. Strong literals, strong dimensions

While a generic, "one size fits all" strong typing solution is not yet created in practice, some libraries offer elaborate solutions with regards to units and dimensions. The most notable example is *Chrono* [67], which was introduced in C++11. Chrono applies strong types and safe conversions with regards to date and time operation by employing C++ template metaprogramming. In C, and pre-C++11, the only way to represent time was to use the `time_t` type, which precision and exact definition was left to the implementation to specify.[5]

Chrono introduced the representation of various clocks and a versatile way of dealing with time precision. Most importantly, instead of a single – potentially floating-point – argument representing "the" time, the concepts of hour, minute, etc. was added. User-defined literals[6] allow expressing these concepts in an easily readable way, such as `2020y`. Listing 6 shows an example of a "deadline checking" program. The deadline itself is immediately readable due to the use of *user-defined literals*. Employing various other features of C++, the expressive capabilities of the code is further increased. Building upon the foundations and success of Chrono, various other libraries, such as one for physics dimension calculations [24] exist.

VII. Threats to Validity

Due to restrictions in the Clang static analysis framework, language constructs related to templates were not wholly modelled in our study. We opted to emit the warnings at the point of definition, as the location where any "fix" might be applied is the definition's source file. This presented a challenge for templates as they are *defined* with generic code often in header files, while concrete *instantiations* are done by the compiler [68]. We diagnose only *primary templates* and *explicit template specialisations* and provide no warning for cases similar in nature to `template<T, U> f(T, U)` instantiated by a call `f(1, 2);`. For this instantiation, `T` and `U` are both `int`, but for the *primary template*, `T` and `U` are distinct placeholder types.

Another case not modelled accurately and thus ignored from the analysis is when the adjacent parameters' types' equivalence of convertibility may only be proved through dependent names. The function depicted in Listing 7 contains

[5]It was not a requirement pre-C11 for this type to be a floating-point number. While most implementations settled for representing time since the *UNIX epoch* – either in seconds or milliseconds integer, or seconds floating-point –, this was not mandatory either.

[6]In this context, *user-defined* refers to the literal not being defined by the core language itself (such as `0.5f`), but rather loaded from the code being compiled – even if the code for such literals come from the *Standard Template Libarary*.

```
template <typename T> struct vector {
  typedef       T   value_type;
  typedef const T & const_reference;
};
template <typename T>
void g(typename vector<T>::const_reference,
 const typename vector<T>::value_type &);
```

Listing 7. A case of type-equivalent adjacent parameters through dependent types for function `g` not modelled by the analysis. In many cases, the two parameters have the same type (`const T &`). However, this depends on how `vector<T>` is defined, there could be *explicit specialisations*.

the possibility of mixing up the two parameters, but this is not diagnosed.

These issues are only causing false negatives, and do not pose a threat to the already found functions discussed in Section V. We plan to work with the open-source community to refactor the framework in a way that diagnosing these cases will be possible and accurate in the future.

Furthermore, changes in the library version in the package manager and the development environment might change which functions are compiled in a project, and thus which functions are analysed.

VIII. Conclusion

Similarly typed parameters of functions allow for potential misuse at call sites. These cases might go unnoticed as the match of the types of arguments to their parameters is the only requirement written in language specifications. Unless extensive testing or analysis tools are employed, a real issue affecting the program's behaviour remains hidden. The proliferation of coarse-grained types requires the usage of descriptive identifier names, which may only be understood by humans and experimental tools, not the mainstream development pipeline elements.

In this paper, we presented an analysis method that detects type-equivalent and type-similar adjacent parameter ranges. We showed that the usage of various language features, most importantly *implicit conversions*, increases the potential of misuse markedly. The rule can immediately warn when a function definition is found to be a carrier for potential misuse.

The analysis rule is developed on top of the LLVM/Clang Compiler Infrastructure project's static analysis framework, and as such, could easily be integrated into a development pipeline. Various integrated developer environments (*IDE*s), such as Eclipse [69] or CLion [70] already integrate, or through the Language Server Protocol [71] allow integrating analysis tools into the same views where code is written.

While our discussion focused primarily on C and C++ programming languages, the idea can be applied to other languages where implicit conversions might be prevalent, such as in Scala. We hope that the empirical results further encourage the language and library development community to emphasise the importance of finer-grained, stronger types, and the restriction of implicit conversions.

References

[1] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 815–825. [Online]. Available: http://dl.acm.org/doi/10.5555/2337223.2337319

[2] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, May 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6311409

[3] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. USA: IEEE Computer Society, 2009, p. 307–318. [Online]. Available: http://doi.org/10.1109/ASE.2009.94

[4] andreser. (2017) `_subborrow_u64 argument order inconsistent`. Free Software Foundation - GNU GCC. Accessed 2019-12-16. [Online]. Available: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=81294

[5] M. Storsjö. Fix accidentally swapped input/output parameters of `string(REPLACE)`. The LLVM Foundation. Accesssed 2019-12-16. [Online]. Available: http://reviews.llvm.org/rGe16434a0497bdb2da587390171a496b56f1c41b6

[6] M. Capella. (2016) Suspicious code with probably reversed parms in call to `IsSingleLineTextControl(bool, uint32_t)`. Mozilla. Accessed 2019-11-23. [Online]. Available: http://bugzilla.mozilla.org/show_bug.cgi?id=1253534

[7] A. Peruma, "Towards a model to appraise and suggest identifier names," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 639–643. [Online]. Available: http://ieeexplore.ieee.org/document/8918988

[8] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *2010 14th European Conference on Software Maintenance and Reengineering*, March 2010, pp. 156–165. [Online]. Available: http://ieeexplore.ieee.org/document/5714430

[9] B. Caprile and P. Tonella, "Nomen est omen: analyzing the language of function identifiers," in *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, Oct 1999, pp. 112–122. [Online]. Available: http://ieeexplore.ieee.org/document/806952

[10] ——, "Restructuring program identifier names," in *Proceedings 2000 International Conference on Software Maintenance*, Oct 2000, pp. 97–107. [Online]. Available: http://ieeexplore.ieee.org/document/883022

[11] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical api usage," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 826–836. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337321

[12] M. Pradel and T. R. Gross, "Detecting anomalies in the order of equally-typed method arguments," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 232–242. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001448

[13] M. Pradel and T. R. Gross, "Name-based analysis of equally typed method arguments," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1127–1143, Aug 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6419711

[14] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 1063–1073. [Online]. Available: http://ieeexplore.ieee.org/document/7886980

[15] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 104:1–104:22, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133928

[16] J. Varjú. (2016) Suspicious call argument checker. source code. The LLVM Foundation. Accessed 2019-12-27. [Online]. Available: http://reviews.llvm.org/D20689

[17] ——, "Felcserélt függvényhívási paraméterek detektálása statikus elemzés segítségével (Detecting swapped arguments in function calls with static analysis)," Master's thesis, Eötvös Loránd University, Faculty of Informatics, 2017.

[18] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *ECOOP 2011 – Object-Oriented Programming*, M. Mezini, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 130–154. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-22655-7_7

[19] Google, Inc. (2009) CppLint rule: "runtime/memset". Accessed 2019-12-16. [Online]. Available: http://github.com/google/styleguide/blob/gh-pages/cpplint/cpplint.py

[20] R. N. Kovács. (2017) Clang-Tidy rule: "`bugprone-suspicious-memset-usage`". The LLVM Foundation. Accessed 2019-12-16. [Online]. Available: http://clang.llvm.org/extra/clang-tidy/checks/bugprone-suspicious-memset-usage.html

[21] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic inference of abstract types," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 255–265. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146268

[22] S. Hangal and M. S. Lam, "Automatic dimension inference and checking for object-oriented programs," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 155–165. [Online]. Available: http://ieeexplore.ieee.org/document/5070517

[23] S. K. Dash, M. Allamanis, and E. T. Barr, "Refinym: Using names to refine types," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 107–117. [Online]. Available: http://doi.org/10.1145/3236024.3236042

[24] M. Pusz. (2019, May) Implementing physical units library for C++. presentation. C++Now (formerly BoostCon). Accessed 2019-12-27. [Online]. Available: http://youtube.com/watch?v=wKchCktZPHU

[25] Tour of Scala: Implicit Conversions. EFPL and Lightbend, Inc. Accessed 2020-01-07. [Online]. Available: http://docs.scala-lang.org/tour/implicit-conversions.html

[26] G. A. Nagy and Z. Porkoláb, "Performance issues with implicit resolution in Scala," in *Proceeedings of the 10th International Conference on Applied Informatics*, ser. ICAI '17. Eger, Hungary: Eszterházy Károly University, jan 2017, p. 211–223. [Online]. Available: http://icai.uni-eszterhazy.hu/icai2017/uploads/papers/2017/final/ICAI.10.2017.211.pdf

[27] Rust by Example `§5.1` "Casting". Rust Programming Language. Accessed 2020-01-07. [Online]. Available: http://doc.rust-lang.org/rust-by-example/types/cast.html

[28] Rust by Example `§6.1` "`From` and `Into`". Rust Programming Language. Accessed 2020-01-07. [Online]. Available: http://doc.rust-lang.org/rust-by-example/conversion/from_into.html

[29] Motor Industry Software Reliability Association, *MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems*. HORIBA MIRA, 2019. [Online]. Available: http://books.google.hu/books?id=PnoMxQEACAAJ

[30] Carnegie Mellon University Software Engineering Institute. (2016) `INT02-C`: Understand integer conversion rules. Accessed 2020-07-13. [Online]. Available: http://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules

[31] 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. [Online]. Available: http://iso.org/standard/68564.html

[32] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, June 2006, pp. 3–12. [Online]. Available: http://ieeexplore.ieee.org/document/1631100

[33] (2017) `§i.24` "Avoid adjacent unrelated parameters of the same type" in the C++ Core Guidelines. online article. Standard C++ Foundation. Version `0.8`, accessed 2019-12-28. [Online]. Available: http://github.com/isocpp/CppCoreGuidelines/blob/v0.8/CppCoreGuidelines.md#Ri-unrelated

[34] R. Kovács, G. Horváth, and Z. Porkoláb, "Detecting C++ lifetime errors with symbolic execution," in *Proceedings of the 9th Balkan Conference on Informatics*, ser. BCI'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: http://doi.org/10.1145/3351556.3351585

[35] S. Meyers, *Effective Modern C++: 42 specific ways to improve your use of C++11 and C++14*. Sebastopol, California, USA: O'Reilly Media, 2015. [Online]. Available: http://oreilly.com/library/view/effective-modern-c/9781491908419/

[36] Whisperity. (2019, 10) Add `cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type` check. source code. The LLVM Foundation. Accessed 2020-01-07. [Online]. Available: http://reviews.llvm.org/D69560

[37] The LLVM Foundation. (2001-) Clang: a C language family frontend for the LLVM compiler infrastructure. Version `9.0` (`0399d5a`), accessed 2019-12-30. [Online]. Available: http://clang.llvm.org

[38] D. Stenberg *et al.* (1996-) `curl`. Version `7.67.0` (`2e9b725`), accessed 2019-12-30. [Online]. Available: http://curl.haxx.se

[39] L. Torvalds *et al.* (2005-) `git`. Version `2.24.1` (`53a06cf`), accessed 2019-12-30. [Online]. Available: http://git-scm.org

[40] Netdata Corporation. (2013-) Netdata. Version `1.19.0` (`5000257`), accessed 2019-12-30. [Online]. Available: http://my-netdata.io

[41] The PHP Group. (1999-) PHP: Hypertext preprocessor. `php-src` version `7.4.1` (`b1a8ab0`), accessed 2019-12-30. [Online]. Available: http://php.net

[42] The PostgreSQL Global Development Group. (1996-) PostgreSQL. Version `12.1` (`578a551`), accessed 2019-12-30. [Online]. Available: http://postgresql.org

[43] S. Sanfilippo *et al.* (2006-) Redis. Version `5.0.7` (`4891612`), accessed 2019-12-30. [Online]. Available: http://redis.io

[44] N. Marriott *et al.* (2007-) `tmux`. Version `3.0` (`bbcb199`), accessed 2019-12-30. [Online]. Available: http://github.com/tmux/tmux

[45] S. Nakamoto, The Bitcoin Core Developers *et al.* (2009-) Bitcoin. Version `0.19.0.1` (`1bc9988`), accessed 2019-12-30. [Online]. Available: http://bitcoincore.org

[46] Google, Inc. (2016) `guetzli`. Version `1.0.1` (`a0f47a2`), accessed 2019-12-30. [Online]. Available: http://github.com/google/guetzli

[47] Xperience AI. (2019-) OpenCV. Version `4.2.0` (`bda89a6`), accessed 2019-12-30. [Online]. Available: http://opencv.org

[48] Google, Inc. (2008-) Protocol buffers. Version `3.11.2` (`fe1790c`), accessed 2019-12-30. [Online]. Available: http://developers.google.com/protocol-buffers/

[49] R. Smith, Google, Inc. *et al.* (2006-) Tesseract OCR engine. Version `4.1.0` (`5280bbc`), accessed 2019-12-30. [Online]. Available: http://github.com/tesseract-ocr/tesseract

[50] The Apache Software Foundation. (1999) Xerces C++. Version `3.2.2` (`71cc0e8`), accessed 2019-12-30. [Online]. Available: http://xerces.apache.org

[51] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5 – 21, 2008, proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008). [Online]. Available: http://sciencedirect.com/science/article/pii/S1571066108003824

[52] P. Godefroid, "The soundness of bugs is what matters (position statement)," in *BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, 2005. [Online]. Available: http://cs.umd.edu/~pugh/BugWorkshop05/papers/11-godefroid.pdf

[53] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, pp. 363–387, 04 2011. [Online]. Available: http://sciencedirect.com/science/article/abs/pii/S0950584910002235

[54] Y. Chiba, "Redundant boxing elimination by a dynamic compiler for java," in *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, ser. PPPJ '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 215–220. [Online]. Available: http://doi.org/10.1145/1294325.1294355

[55] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, July 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4497212

[56] W. E. Brown, "Toward opaque typedefs for `C++1y`," ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee (ISOCPP) proposals, Tech. Rep., 01 2013. [Online]. Available: http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3515.pdf

[57] ——, "Function aliases + extended inheritance = opaque typedefs," ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee (ISOCPP) proposals, Tech. Rep., 09 2015. [Online]. Available: http://open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0109r0.pdf

[58] A. Baráth and Z. Porkoláb, "Life without implicit casts: Safe type system in C++," in *Proceedings of the 7th Balkan Conference on Informatics Conference*, ser. BCI '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: http://doi.org/10.1145/2801081.2801114

[59] D. Greene. (2019) Provide a semantic `typedef` class and operators. The LLVM Foundation. Accessed 2019-11-15. [Online]. Available: http://reviews.llvm.org/D66148

[60] B. Meyer, "Ensuring strong typing in an object-oriented language (abstract)," in *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 89–90. [Online]. Available: http://doi.org/10.1145/141936.290558

[61] O. L. Madsen, B. Magnusson, and B. Mølier-Pedersen, "Strong typing of object-oriented languages revisited," in *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA/ECOOP '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 140–150. [Online]. Available: http://doi.org/10.1145/97945.97964

[62] A. Kemper and G. Moerkotte, "A framework for strong typing and type inference in (persistent) object models," in *Database and Expert Systems Applications*, D. Karagiannis, Ed. Vienna: Springer Vienna, 1991, pp. 257–263. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-7091-7555-2_43

[63] W. Robertson and G. Vigna, "Static enforcement of web application integrity through strong typing," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, p. 283–298. [Online]. Available: http://dl.acm.org/doi/10.5555/1855768.1855786

[64] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte, "Verification of object-oriented programs with invariants," *Journal of Object Technology*, vol. 3, no. 6, pp. 27–56, 2004. [Online]. Available: http://jot.fm/issues/issue_2004_06/article2.pdf

[65] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 255–265. [Online]. Available: http://dl.acm.org/doi/10.5555/2337223.2337254

[66] International Organization for Standardization and International Electrotechnical Commission, *ISO/IEC DIS 8652: information technology — programming languages — their environments and system software interfaces, programming language Ada, language and standard libraries, draft, version 5.0, 1 June 1994, IR-MA-1363-4*, ser. Draft international standard. Cambridge, MA, USA: Intermetrics, Inc., 1994. [Online]. Available: http://iso.ch/cate/d22983.html

[67] B. Schäling, *The Boost C++ libraries*. XML Press, 2014, accessed 2020-03-07. [Online]. Available: http://theboostcpplibraries.com

[68] D. Vandevoorde, N. M. Josuttis, and D. Gregor, *C++ Templates: The Complete Guide (2nd Edition)*, 2nd ed. Addison-Wesley Professional, 2017.

[69] (2001) Eclipse IDE - C/C++ development tooling (CDT). The Eclipse Foundation. Accessed 2020-01-06. [Online]. Available: http://projects.eclipse.org/projects/tools.cdt

[70] CLion - a cross-platform IDE for C and C++. JetBrains, Inc. Accessed 2020-01-06. [Online]. Available: http://jetbrains.com/clion

[71] Microsoft Corporation. (2015) LSP: Language server protocol. Microsoft Corporation. Accessed 2020-01-06. [Online]. Available: http://microsoft.github.io/language-server-protocol