The Journal of Systems & Software 181 (2021) 111048

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Practical heuristics to improve precision for erroneous function argument swapping detection in C and C++☆,☆☆

Richárd Szalay *, Ábel Sinkovics, Zoltán Porkoláb

Department of Programming Languages and Compilers, Institute of Computer Science, Faculty of Informatics, ELTE Eötvös Loránd University, Budapest, Hungary

ARTICLE INFO

Article history:
Received 5 March 2021
Received in revised form 26 May 2021
Accepted 13 July 2021
Available online 22 July 2021

Keywords:
Static analysis
Function parameters
Argument selection defect
Type safety
Strong typing
Error-prone constructs

ABSTRACT

Argument selection defects, in which the programmer chooses the wrong argument to pass to a parameter from a potential set of arguments in a function call, is a widely investigated problem. The compiler can detect such misuse of arguments only through the argument and parameter type for statically typed programming languages. When adjacent parameters have the same type or can be converted between one another, a swapped or out of order call will not be diagnosed by compilers. 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 developers can define functions with multiple adjacent parameters that allow arguments to pass in the wrong order. When implicit conversions – such as parameter pairs of types (`int`, `bool`) – are taken into account, the number of mistake-prone functions markedly increases compared to only strict type equivalence. We analysed a sample of projects 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 to restrict the proliferation of implicit conversions. However, the analysis produces a hard to consume amount of diagnostics for existing projects, and there are always cases that match the analysis rule but cannot be "fixed". As such, further heuristics are needed to allow developers to refactor effectively based on the analysis results. We devised such heuristics, measured their expressive power, and found that several simple heuristics greatly help highlight the more problematic cases.

1. 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] This helps guard against a bad call to the function if the user specified the arguments out of order, resulting in compile-time errors. Unfortunately, this detection mechanism in compilers is 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 ordering of the arguments, `fn(1, 2)` and `fn(2, 1)`, give valid calls. In addition, due to *implicit conversions* 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 parameter's name 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 (Zhong et al., 2009; Pandita et al., 2012; Robillard et al., 2013), the semantic information conveyed through names are not considered by virtually any compilers of mainstream languages.

Various issues might arise if 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 that 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.

☆ Prepared with the professional support of the Doctoral Student Scholarship Program of the Co-operative Doctoral Program of the Ministry of Innovation and Technology financed from the National Research, Development and Innovation Fund, Hungary.

☆☆ Editor: W.K. Chan.

* Corresponding author.
E-mail addresses: szalayrichard@inf.elte.hu (R. Szalay), abel@elte.hu (Á. Sinkovics), gsd@elte.hu (Z. Porkoláb).

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

https://doi.org/10.1016/j.jss.2021.111048

Inadequately typed function parameters hinder the program's maintainability. Any development or comprehension 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, developers employ tools specific to catching these issues. Several existing tools are discussed in Section 2 that aim to find argument selection defects using the *names* of arguments and parameters and finding mismatches between them. However, these tools are all "external" and only integrate into the development process: to catch the argument selection defects, vigilance and enforced automation must be done by the projects, such as in form of a continuous integration system. In contrast, the semantic checks performed by the type system is an integral part of the language's definition, and the compilation process. By encouraging users to use *language features* instead of external tools, the software project's quality, understandability, and safety can be proactively enhanced. A proactive solution also helps guard against future mistakes that are undetected before the fact by tools that compare names. In this paper, we show the use case for *stronger typing* and how to highlight program elements where such strengthening might be necessary. In addition, the design, thought process, and refactoring tools involved in moving to stronger types also allow uncovering additional issues in the project.

Name-based analyses were able to find some severe vulnerabilities that existed in production codebases for long times. A study at Google Inc. (Rice et al., 2017) has found a case where, in a function call involving authentication, the key (of type `string`) was passed out of order to another `string` parameter. They deemed this bug the highest priority in the study after laying dormant for more than two years. In that case, the *exact* match between the parameter name and the argument name made the discovery easy. However, it is realised easily that an authentication key should not be a plain `String` variable, and as such, stronger typing helps this case immensely. Several other defects were discussed in the paper with various lifetime, with an average of 66 days.

Unfortunately, name-based analyses fall short when there are no practical ways to give names to argument expressions. While searching the bug trackers and histories for well-known projects, we found a mixed string replace issue in LLVM's build system (Storsjö, 2019). Regular expression libraries commonly take the `needle` and `haystack` parameters as just strings. As such, they allow for swapping issues to go undetected if the passed values are created from string literals, not named or nameable variables. We found a similar case in the *GNU GCC implementation*, where an implementation-specific call for subtraction swapped the two arguments (andreser, 2017).

Although existing tools usually focused on type equivalence or languages with no implicit conversions, the argument ordering mismatch becomes an even greater problem when investigated in the context of C and C++. We detail *implicit conversions* in Section 4. A bug in production stemming from implicit conversions was found in *Mozilla Firefox*'s code (Capella, 2016). This finding is interesting in contrast with previous results, as the function had the parameter types `bool`, `int`, and two swapped calls to it in the form of `f(element, false)`, making implicit conversions the culprit.

An even more notable case for both C and C++ is the *"memset swap"*. The standard *memset* function is defined as `void* memset(void* buf, int value, size_t num)`. A call to it sets `num` bytes in the provided `buf` buffer to the given numeric `value`. This function is most commonly used to zero-fill a buffer holding a `T` object or array by calling `memset(&t, 0, sizeof(T))`. Unfortunately, developers often place the to-fill buffer and the `sizeof` relating to the buffer's type next to one another, in a syntactically valid swapped call, `memset(&t, sizeof(T), 0)`, which results in 0 bytes being set to the value, i.e. no changes. Usually no argument names appear in the call, and thus even if a heuristic "names" the second argument `size`, the second parameter's *name* is sufficiently distinct from "size" to not report the swap. Such `memset`-related bugs pop up often due to the function having a problematic interface, as evident by various static analysis tools having rules made explicitly for *memset* (Google, Inc., 2009; Kovács, 2017). 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 the code that defines `memset` if it was defined in the project being analysed and not coming from external sources.

As long-living libraries and legacy projects are painful, expensive, and in many cases near impossible to fix, the importance of proactive defence increases. In this paper, we present an automatic static program analysis that diagnoses function definitions that contain multiple adjacent parameters which have compatible types. Targeting definitions instead of call sites benefits developers by warning about an error-prone interface early 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 that 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's tools, the analysis can be integrated directly into active development, assuming the project can be compiled with Clang.

To measure the applicability of the analysis rule on existing projects, we gathered and measured open-source C and C++ software of various scale and domain. We concluded our study with multiple configurations of the analysis, as detailed in Section 5. When applied to existing projects, we found that the analysis rule produces a massive volume of reports, which are hard to fix by themselves, and the process requires refactoring of the project to eliminate the issue. However, we also found that considering implicit conversions, not just strict type equality, markedly increases the number of potentially bug-prone functions. Although our analysis is applicable for existing projects too, the rule's primary goal is to prevent the spread of mistakes by defencive design early on in the life cycle of projects. We found that several of the heuristics presented with regards to name-based analyses also apply in our case to allow the hiding of less relevant reports. Additional heuristics for squelching noisy results were derived from the usage pattern of variables.

This paper is an improved and extended version of our previous publication, *"The Role of Implicit Conversions in Erroneous Function Argument Swapping in C++"* (Szalay et al., 2020) published at the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM). The improved and extended version was created with significant changes as follows. We extended the Introduction and Related Work sections with a more detailed contextualisation of the problem. The sections that detail the model of our analysis were rewritten to be more precise. During the theoretical development of the model that was published in our previous paper, we introduced a shortcoming by accepting only *unidirectional* implicit conversions between two parameters as sufficient for a warning. This was an over-approximation, as in many cases, when the implicit conversion is only unidirectional, the language prohibits the swapped call, and the compiler reports it as an error. For this paper, this modelling was refined. This

required all of the analyses to re-run and re-evaluated. During the re-evaluation, we manually removed results from generated code. A new Section 6 was added, which details the heuristics for making the more important diagnostics stand out more. We added additional projects to the result set and improved the discussion of the results.

This paper is organised as follows. We discuss prior literature related to the topic of function parameters and argument selection defects in Section 2. We define and detail the main target of our analysis rule, *type-equivalent parameter ranges* in Section 3. Implicit conversions as a language feature and their theoretical effects on the results are presented in Section 4. In Section 5, we discuss our empirical findings on various open-source projects. Ways to highlight the more pressing issues and suppress those intentional or hard to fix are shown in Section 6. Potential solutions to the problem are overviewed in Section 7. Restrictions of this research are mentioned in Section 8. Conclusions are drawn in Section 9.

2. Related work

Argument selection defects have been investigated for various programming languages in the past. The major difference between our work and the previous state of the art is that the related work is always *reactive* and uses *a posteriori* information. The existing literature mainly uses name-based analysis to identify potentially mistaken calls reactively. Some do consider type equivalence and prevent false positive suggestions that do not work because the changed code would contain semantic errors due to types.

Pradel and Gross (2011) emphasise the power of enhanced type checking and static analysis regarding finding *anomalies* in arguments passed to function calls. They developed an automated tool that requires no additional knowledge apart from the source code of the project itself. The anomaly manifests as some arguments of the same type passed out of order. The analysis tool works by gathering information of all call sites for each function. If argument names at a particular call site are sufficiently unlike 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 detecting anomalies. A subsequent paper by the same authors (Pradel and Gross, 2013) 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. (2016) 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 involved 60 real-world Java programs. They also studied their approach for two practical use cases: for suggesting renames of misnomer parameters as inferred from the call sites and for selecting a different argument at a call site from a set of other potential candidates with high precision. The potential candidates for the suggestions are generated from other expressions at the call site and other calls to the same function. Thus the suggestions are broader than only changing the order of arguments passed. This work compares arguments at one call site with the parameters of the called function, and the similarity analysis is done across all arguments of the function, with no regards to the types. The suggestion of better matching arguments, however, does consider type conformity and ignores suggestions that might make the rewritten not pass semantic analysis.

Extending the works mentioned above, Rice et al. (2017) have integrated an automated check for argument mismatches to their development pipeline at Google Inc. They evaluated their implementation on substantially sized corpora of Java projects spanning 200 million lines-of-code, including company proprietary and open-source. They measured the relative power of string distance functions by hand-labelling a test set of approximately 4000 pairs of argument–parameter names to fine-tune the distance functions' thresholds. The authors devised some additional logic that involves generating a "name" for an argument expression where a name is not trivially obvious, such as when complex expressions' results are passed as arguments. 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 to those discussed, using string distance metrics were implemented in (Varjú, 2016, 2017) for C and C++ using compiler-based tools to warn about a potentially swapped argument at a particular call site.

Scott et al. (2020) have furthered the investigation of argument and parameter name mismatches by using morphemes in *SwapD*. The tool discussed in their work can find more developer mistakes by comparing the morphemes in the names instead of the whole string, allowing the discovery of crucial recurring "tags" in human-written names. For example, finding that a parameter `arrayLength` and an argument expression `X.getLength()` are not as dissimilar as a full string distance would deduce, due to both containing the *length* morpheme. *SwapD* also employs statistical approaches similar to previous work. The authors found 154 hand-evaluated true positive swaps in a 417 million lines-of-code C and C++ input corpus. They compared their results against the results of Rice et al. (2017) and concluded that the distribution of the probability of swaps when plotted against the number of parameters a function takes differ greatly in C and C++ versus Java, possibly due to the comparatively "weakly typedness" of C and C++. Implicit conversions, a language element discussed and one of the main focus of this paper is a contributing factor to this comparative weakness.

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 (Peruma, 2019). Multiple kinds and contexts of identifier names have been studied by Butler et al. (2010). Caprile and Tonella (1999) discuss how function names are constructed and that semantic information – lost to compilers and purely syntactic tools – is encoded in the name. Their subsequent work (Caprile and Tonella, 2000) proposes a method for automatically standardising identifier names. Selecting good arguments to function calls has been studied by Zhang et al. (2012). 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.

Our approach is similar to the aforementioned works 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. (2011) 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 proactive, defencive design and leverage the type system with *a priori* information.

Several works in the literature discuss the automated, tool-driven synthesis of type constraints. Guo et al. (2006) 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 (2009) 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* (Dash et al., 2018) is an automated unsupervised learning tool that models the flow of values and expressions from one variable to another and suggests more fine-grained types based on the information gathered.

In the future, these works may 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 what is shown by Hangal and Lam (2009). Chrono can be viewed as a solution that aimed to solve some issues discussed in this paper for a particular domain. We detail methods that are also used in Chrono in Section 7.4. Other solutions that enforce dimensionality through the type system also exist for physical units (Pusz, 2019).

The problem applies to other mainstream statically typed programming languages with varying degree. In Fortran, the built-in numeric data types `integer`, `real`, and `complex` convert between each other implicitly during assignments (ISO/IEC JTC 1/SC 22, 1978). The user may place an explicit type signature and a disabling of implicit conversions for the variable, such as by saying `implicit none;`, followed by `integer :: x`. No implicit conversions are performed in Fortran for procedure calls.

Similarly, in Java, the implicit conversions possible in the language are numerical conversions between built-in scalar types, e.g. `int` and `float`. In addition, there are conversions between stack-allocated scalars – e.g. `int` – and their heap-allocated, `Object`-inheriting counterparts, dubbed "boxing types" – e.g. `Integer` – (Arnold et al., 2000). Any other conversions between user-defined types must be explicitly executed by either calling an appropriate constructor of the target type or some other converting function.

Rust provides no implicit type conversions between the built-in primitive types of the language (Rust Programming Language, 2004a). Conversions between user-defined types are done by implementing the `From` trait (Rust Programming Language, 2004b) for a particular type. Given an input type `TIn` and a result type `T`, the `From<TIn>` trait has one member function, `from(val: TIn)` $\rightarrow$ `T`, which implements the conversion. This implementation method can be viewed from a C++ way of thinking as if converting constructors were implemented as explicit template specialisations, outside the concerned class's body. The inverse operation of `From` can be implemented analogously using the `Into` trait. In Rust, every source file implicitly defines its own *namespace*. Access of private members – often needed by conversion functions – is only possible inside the file where the target type is defined. Due to this, we can conclude that while explicit, user-defined conversions are possible in Rust, there is at most one implementation for such conversions for every (T_1, T_2) pairs of conversion functions.

Scala, however, makes the set of implicit conversions ever broader than in C++ and gives more options to the users to define implicit conversion functions (EFPL, Lightbend Inc., 2006). Scala performs implicit conversions not only during assignment or parameter passing but also during member access. The first case is similar to that of C++. In the second case, if the user tries to access a member of an expression by saying `e.M`, in case such member is not defined in the type of `e`, Scala will try to search and perform an implicit conversion that converts from the type of `e` to a type that has an `M` member. In addition, implicit conversions in Scala are *scoped* and *context-sensitive*: in contrast with C++ and Rust where the converting functions (implicit or explicit) are defined for the types involved, but otherwise "globally" for the project, implicit conversions in Scala depend upon the visibility of the conversion function. All functions that have the `implicit` keyword on their definition and are some `S` $\Rightarrow$ `T` functions are implicit conversion functions. This increase in complexity for implicit resolution was found to be a significant issue for compiler performance in Scala projects (Nagy and Porkoláb, 2017). Scala 3, the next upcoming major release of the language at the time of writing our paper, is expected to change the syntactic requirements around implicit definitions – such as implicit conversion functions – by requiring implicit conversions to be derivatives of the `Conversion` type, and all synthesised code be expressed with the new `given` keyword (EFPL, 2021).

Several well-known guidelines, restrictions, and domain-specific spin-offs for C or C++, such as MISRA C (Motor Industry Software Reliability Association, 2019) or the SEI-CERT secure coding guidelines (Carnegie Mellon University Software Engineering Institute, 2016), contain rules that guard against implicit conversions of numbers in any context, not specific to function parameters.

3. Type-equivalent parameter ranges

To facilitate moving projects to a preventive design, we shift the problem from detecting bogus call sites to working on the functions' interfaces instead. There are a few critical differences in C++'s workings compared to Java, which most of the previous related works were targeting. The first such is the extent and way *separate compilation* (ISO/IEC JTC 1/SC 22, 2017) is done. In C and C++, the compiler is restricted to use the information available only in the *translation unit* – the currently compiled source file and all headers and module data included in it –, unlike Java, where the compiler is allowed and regularly reads other files where the implementation of functions are available. Certain C++ tools do support so-called cross-translation unit analysis, but it is experimental and limited (Horváth et al., 2018). The name and the type qualifiers of the parameters are not required to be available in a code that is only using a function. Given a function signature such as `int f(int, int);`, it is evident that the call to the function contains the *possibility* for arguments to be swapped, and the compiler will still 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 give the extra semantic information that is conveyable through identifier names (Lawrie et al., 2006).

The issue of 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 parameters of the same type"* (Stroustrup and Sutter, 2017). To our knowledge, there were no free and open-source automated tools that check for possible violations of this rule before.

```
#define CPU_WORD_TYPE int
typedef int Number;
using CNumRef = const Number&;
void fn(int i, const int& ir, Number i2, CNumRef ir2,
      CPU_WORD_TYPE w);
```

Listing 1: All 5 parameters of function `fn` are mixable with each other at a call site, as all can be passed a value of type `int`. However, this is not deducible at first glance by simply reading what is written in the function's signature – especially if the `typedef`s are in a different location – unless language rules are understood and modelled.

Definition 1. Given a function signature $f(T_1, T_2)$ and two expressions v_x, v_y of types T_1, T_2, respectively, the parameter pair is **mixable** if a semantically valid, theoretical function call $f(v_x, v_y)$ is also valid in the swapped order $f(v_y, v_x)$.

Definition 2. Given a function signature $f(T_1, \ldots, T_N)$, the **type-equivalent parameter ranges** are ordered subintervals of the function's parameters, and in each such interval, every pair of parameters are *mixable* with each other.

However, solving the trivial "equation" $T_1 = T_2$ for the two types is not enough. A possibility of mixing or swapping arguments at a call site might not be apparent at first glance. There are several language features of C and C++ that need to be modelled to deduce the fact that a pair of parameters is mixable. Listing 1 shows an example of a function where even though all parameters have lexically different types defined, they all may be mixed with one another due to how the language works.

3.1. Type aliases

Type aliases – introduced with the `typedef` or `using`[2] keyword – introduce synonyms between two types in the program. The alias name can be used in any context the language expects a type, and in effect, the program will behave as if the aliased type is referred to. This feature is convenient for several use cases, such as when the actual type depends on configuration parameters, and in C++ template metaprogramming (Vandevoorde et al., 2017) which also employs it. In addition, developers use type aliases to emphasise the various roles a particular type may play in the program — which is futile, as the alias name conveys no additional semantic information on the language level and can only be understood by the developers themselves. However, the created type alias is a *weak type alias* (compared to the *strong type alias*, see Section 7.2), because it is completely interchangeable with the aliased type. Due to type aliases always referring the same type "in practice", type aliases must be resolved and their underlying types considered for the purposes of Definition 1.

3.2. Reference types (C++)

Using a reference type for a variable allows creating variables with different names that all bind to the same instance. Unlike pointers, references cannot be set to point to another object. From the user's point of view, a reference variable at a usage point behaves precisely like any other variable. During a function call, reference variables allow modifying the state of a variable living outside the function's scope or accessing an object without copying. In these situations, the "binding power" of an expression must be considered to model the parameters' mixing possibility. There are two kinds of references, *lvalue* (`&`) and *rvalue*[2] (`&&`) references, the former binding to named variables, while the latter binding to results of temporary expressions. This means that usually, parameters of non-reference type and different kinds of references to the same type do not allow mixing in the general case.

Unfortunately, there is a special case: an *lvalue reference* to a `const` can bind both temporaries and named variables. The variable cannot be modified through the reference – as the referred type is `const` –, but in the case of temporaries, *lifetime extension* is performed. During *lifetime extension*, a temporary object's lifetime is extended to cover the lifetime of the variable the temporary is bound to. In an expression such as `const Matrix& N = M * v;`, the temporary `Matrix` result of the `*` operator ("multiplication") would end its lifetime at the program reaching the end of the full expression, at the `;`, running the destructor, and thus releasing owned memory and other resources. Because this temporary is bound to the `N` variable, the destruction is only performed when `N`'s lifetime ends. Thus, for our model, we define that type `T` and `const T&` are *mixable*, allowing call sites to pass arguments out of order in every situation. This is not true for other combinations of reference and non-reference types, and thus, we consider those combinations non-mixable.

3.3. Qualified types

Types in C and C++ can be given qualifiers, which does not change the representation and semantics of the value behind a variable but change the behaviour of the access itself. There are three qualifiers defined in the language specifications: `const`, which makes a variable read-only; `volatile`, which makes accesses to the value forbidden from being optimised — used usually in the case of hardware interactions; and `restrict`,[3] which enables further optimisations by declaring pointer variables to be disallowed from aliasing the same object. Every type may be qualified with `const` or `volatile`, or both. Type qualifiers form a partially ordered set: an expression of type `T` may always be assigned to variables that are *more qualified* than `T`, i.e. a local variable of type `volatile int` may be passed to a parameter of `const volatile int`. Making a type more qualified is only a forward operation. Given two expressions of unqualified type `T` and a function `f(T*, const T*)`, both orderings of the arguments in a call are possible, as either `T` can increase in qualifiedness. The reversed operation of losing qualifiers is diagnosed already by all major compilers, thus if precisely one of the expressions were of type `const T*` instead, the mixed call is caught, eliminating the need for deducing such a pair of parameters as a possibly error-prone situation. This diagnosis manifests as *warnings* for C software, and a *compiler error* for C++ projects.

Due to projects and coding conventions about how elaborate should qualifiers be outside of function interfaces, we decided that the decision on whether types differing only in their qualifiers are deemed mixable should be a parameter of our model. Some rules of thumb, such as the *C++ Core Guidelines* (Stroustrup and Sutter, 2017) consider different qualified types to be non-mixable.

4. Implicit conversions

In addition to the aforementioned language features, both C and C++ feature *implicit conversions* between types, which increase the possibility of arguments being passed in the wrong order,

[2] Available in C++11 and newer standards.

[3] `restrict` is available only in C, starting with the C99 standard. It is not part of any C++ standards.

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

// Suppose two local scalars. . .
int H = 2130706433, s = 4096;
```

In C++, both orderings of arguments are valid.

```
transmit(H, s); // → 4096 bytes sent
transmit(s, H); // → 2130 million bytes sent
```

In Java, the conversion to `Host` must be explicit.

```
// ↯ error: "incompatible types:
// int cannot be converted to Host"
transmit(H, s);

transmit(H, new Host(s)); // ↯ error!
transmit(new Host(H), s); // ✓ works.
```

Listing 2: Implicit conversion in C++, such as the use of a *converting constructor* may allow for passing arguments out of order.

while also decreasing the visibility of such mistakes when a small view of the code is investigated without tools. The ability for user-defined types and user-defined conversion methods in C++ further extends the problem space. The example in Listing 2 shows how implicit conversions silently allow arguments passed out of order.

If an expression of type T_1 is used in a context where a dissimilar type T_2 is expected – e.g. during assignments: `T2 v = T1("xy")` – the language specifies the consideration of implicit conversions. One of the cases where implicit conversions might be performed is during function calls, where arguments are bound to parameters.

The C++ language standard (ISO/IEC JTC 1/SC 22, 2017) defines the *implicit conversion sequence* as the following sequence of 3 operations.

- At most one *standard conversion sequence*, which further divides to a sequence of 4 operations.
  - At most one *lvalue-to-rvalue transformation* **or** *pointer decay*.
  - At most one *numeric promotion* **or** *numeric conversion*.
  - At most one *function pointer conversion*.[4]
  - At most one *qualification conversion*.
- At most one *user-defined conversion*, i.e. executing **either** a *converting constructor* **or** a *conversion operator*.
- At most one *standard conversion sequence*, with the same 4 possible sub-operations as detailed above.

The implicit conversion from the T_1-typed expression to the T_2-typed context is performed if and only if there exists precisely one, unambiguous, implicit conversion sequence from T_1 to T_2.

In the language standard, the implicit conversion process is defined to be necessarily unidirectional. In the context of argument swapping, given a function $f(T_1, T_2)$, both $T_1 \Rightarrow T_2$ and $T_2 \Rightarrow T_1$ need to be deduced. If either of these implicit conversions is impossible, then a hypothetical swapped function call will already be diagnosed by the compiler.

User-defined types in C cannot have member methods, and as such, no constructors or conversion operators exist. In place of

[4] Available in C++17 and newer standards.

```
struct IntBox {
   int boxed_value;
   IntBox(int);       // Converting constructor.
   operator int() const; // Conversion operator.
};

void f(int i, const int& ir, double d, long l,
     IntBox box1, volatile IntBox box2);
```

Listing 3: Example where *implicit conversions* allow **all** 6 arguments of `f` to be mixed with one another at a call site. There are various conversions between the built-in scalar types, and the user has declared `IntBox` to be convertible both to and from an `int`. A `double D = 0.5` variable can be given as all arguments of the function call.

the C++ standard conversion sequence, C defines a similar set of conversions which deal with converting numeric values between one another. In C, every numeric or pointer expression – of type `T*` – may be converted to other scalars or pointers to a different – `U*` – type, e.g. valid and type-conforming arguments in a call to an `f(int, long*)` can be swapped. However, most compilers diagnose such cases already, so we skipped from considering such parameter pairs *mixable* (as per Definition 1).

We emphasise that the model is meant to find whether expressions that are of the parameters' types may be swapped or mixed with one another, and not whether there exists an unrelated type U which may be implicitly converted to both T_1 and T_2. The latter analysis would be more intensive computationally and produce an explosion in the results, with a vast majority of those results being noise and false positives. Consider a function that takes a `Colour` and a `Direction`, both of which may be implicitly converted from `int`. There would be approximately 20 reports with implicit conversion modelling – all numeric types, their unsigned counterparts, every converted `enum`, etc. – to just this one function alone.

4.1. Standard conversions

Standard conversions are defined for the built-in types or with regards to language elements, irrespective of the type used.

Lvalue-to-Rvalue conversion deals with allowing an expression referring a left-value – without going into too much detail about C++'s handling of value categories, *lvalues* are "basically" variables – to be converted to a right-value – a "value", without assigned identity. These conversions are generated and performed regularly when values of expressions are "read". For our analysis, modelling this conversion is not required, as it is always performed at the call site if needed. Variables, and thus, parameters cannot be declared as *rvalues* (only *rvalue references*, which we discussed in Section 3.2), and the conversion is part of the semantics for the function call.

Pointer decays specify that an expression that names either an array or a function may decay implicitly into a pointer that points to either the first element of the array or a function pointer. Parameters of an array type are syntactic sugar, and the parameter becomes an element pointer in every context. Thus, given two parameters where at least one is an array and the other is an element pointer – e.g. `f(int* p, int A[2])` –, a bidirectional possibility of mixing is deduced. The case of function decay, however, can also be ignored. In C and C++, functions are not *objects*, and thus, variables cannot be declared to have a *function type*. Any expression that names a function immediately decays to a suitable pointer, after which Definition 1 applies normally.

Numeric promotions and conversions allow converting scalar values, such as implicitly considering an `int` expression of value 0 to be used in place of a `double` with value 0.0, or that the 0 value is considered as the *null pointer* or the boolean value `false`. Thus, there is a two-way passage between numerical types. Enumeration types (`enum`), but not scoped enumerations[2] (`enum class` or `enum struct`), are convertible to their underlying scalar type in C++. The inverse operation, creating a value of an enumeration type from a scalar value, is only possible in C. It is noteworthy that the implicit upcast of a pointer from a derived object to the base instance is also defined as a *numerical conversion*.

Function pointer conversions[4] is a separate case from the pointer conversions defined as part of *numerical conversions*. This conversion allows a pointer to a non-throwing (`noexcept`) function to be used as a potentially-throwing (non-`noexcept`) function, i.e. `noexcept`-ness can be discarded. The issue with this conversion at call sites is analogous to the issue of qualified types.

Qualification conversions allow passing a "less qualified" expression – e.g. `int` – to a "more qualified" – `const int` – usage context. Similarly to how `noexcept`-ness can always be lost, `const`-ness and `volatile`-ness can always be gained. We discussed this in detail in Section 3.3.

4.2. User-defined conversions (C++)

In addition to the standard conversions, when either type involved in the implicit conversion is user-defined record, the languages allows the execution of at most one *user-defined conversion*. User-defined conversions take the form of *converting constructors* and *conversion operators*, depicted in Listing 3. Converting constructors take a different type as their parameter, while conversion operators produce a return value of a different type. By applying the `explicit` keyword on a conversion method, the code author can specify that it must not be part of any implicit conversion sequence.

Two properties of C++ allow the resolution of implicit conversions to be computationally easy, both for the compiler and in our analysis, when compared to languages with more elaborate implicits, such as Scala (see Section 2, Nagy and Porkoláb (2017)). First, the user-defined conversion methods must be declared in the involved types' definitions as member methods. User-defined types may be incomplete – *forward declared* –, i.e. their names are introduced, but no actual definition is given. In this case, the code which sees only the incomplete types may only perform a limited set of operations on it: most notably, pointers and references can be created to instances of incomplete types. As there is no information about the type's members, no fields or methods might be accessed or called, and instances cannot be constructed. It follows that in case the compiler sees the complete definition ("body") of the type – which necessarily holds if a parameter of said type is taken by value –, then it also sees all available methods. Second, the length of implicit conversions is greatly limited due to the ability to only execute one method in total.

For example, one conversion applied in Listing 3 is

`double` —numeric conversion→ `int` —converting constructor→ `IntBox` —qualification adjustment→ `volatile IntBox`.

5. Evaluation

The LLVM Compiler Infrastructure project and its C/C++ compiler frontend, Clang (LLVM Foundation, 2001), offer an elaborate, object-oriented design for accessing the syntax tree of C++ programs. Clang is first and foremost a C++ compiler, but the diverse set of libraries integrated with Clang make it possible to write analysis routines, code transformations, and other tools. We created an implementation (Whisperity, 2019) of the analysis discussed earlier in *Clang-Tidy*, a sister project of LLVM/Clang (named *"Clang Extra Tools"*), which is a framework of syntactic static analysis for C and C++. Clang-Tidy allows the users to run various *checks* – analysis rules – on their source code and emits the resulting diagnostics in the same fashion as running Clang, the compiler would emit compiler errors. The list of checks to run can be configured by the user, and checks may take additional configuration options too. Clang-Tidy already integrates into various development environments (IDEs) and continuous integration systems.

The analysis is done by requesting the *Abstract Syntax Tree* (`AST`, the Clang-specific data structure representing the compiled/analysed source code at hand) for each function *definition* in the analysed translation unit and applying the rules discussed in Sections 3 and 4 using Clang's API. At the time of writing this paper, the analysis rule is under discussion for introduction to the official LLVM/Clang codebase. The modelling routines of the implemented analysis themselves were tested with synthetic unit tests during development, which were combinatorially derived from the language rules.

5.1. Running the analysis

Until the analysis is accepted and merged into the official LLVM/Clang code, the analysis can be run manually by obtaining a new enough version of LLVM's source code. At the time of writing this paper, the official repository of LLVM resides on GitHub.[5] Version 12 of LLVM is the newest release on top of which the code can be applied. Our implementation's code (Whisperity, 2019) and the subsequent, dependent patches must be applied over the official source code first to gain access to the implementation. Note that the code – as appearing in LLVM's code review system – might change as the code review progresses. Once the patching has been done successfully, the local copy of LLVM is ready to be compiled. LLVM is written in C++ and uses CMake for compilation. It is essential to define the `-DLLVM_ENABLE_PROJECTS` variable during the build configuring `cmake` call to be `"clang;clang-tools-extra"`. The latter element, *clang-tools-extra* enables building *Clang-Tidy*; otherwise, it would not appear amongst the build targets. Additional details regarding the configuration variables and the build process, such as the underlying build system – using `ninja` is preferred – is discussed in the official documentation.[6]

The configuration options for the implementation can be given directly to the Clang-Tidy invocation or via a `.clang-tidy` file in the project's tree. All relaxations (see Section 5.3) and filtering heuristics (see Section 6) can be enabled through these configuration options. To build the documentation for the checker, install the Sphinx documentation generator and specify `-DLLVM_BUILD_DOCS=ON -DLLVM_ENABLE_SPHINX=ON`.

Once `cmake` has finished generating the build, calling `ninja clang-tidy` will build Clang-Tidy, its dependencies and the related tools. The resulting executable is available, relative to the *build directory* – where `cmake` was executed – as `bin/clang-tidy`. For the documentation, executing `ninja docs-clang-tools-html` will generate them, and make them available under `tools/clang/tools/extra/docs/html`.

To run the analysis on a source file or a project, first, the analysis's subject project must be downloaded and its *compilation database*[7] file created. This file contains the exact compiler

5 http://github.com/llvm/llvm-project.

6 http://llvm.org/docs/GettingStarted.html#getting-the-source-code-and-building-llvm (accessed 2021-05-08).

7 http://clang.llvm.org/docs/JSONCompilationDatabase.html (accessed 2021-05-08).

invocation flags the project is being compiled with. If the project uses CMake, it can automatically emit the compilation database by specifying `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` during configuration. In case the project is using conventional *Makefile*s, a build logging tool, such as *CodeChecker* (Ericsson, 2014) can splice itself into the build system and emit such a file.

Once the test project is built, and the compilation database is ready, the analysis can be run by calling `clang-tidy --checks='-*,bugprone-easily-swappable-parameters' -p build-directory source-file.cpp`. The argument passed to `--checks` sets only our analysis routine to execute and disables every other rule in Clang-Tidy. The results of the analysis is printed to the standard output of the executed binary in the form of compiler diagnostics.

For whole-project analysis, CodeChecker's automation via the `check` command can be used instead. It automatically executes Clang-Tidy for every source file of the project, as listed in the compilation database, and saves the diagnostics. These results can later be viewed in a Web application by running a `CodeChecker server` locally and uploading via the `CodeChecker store` command (Márton and Krupp, 2020). The web browser interface allows the user to interactively search the reports based on the file reported in, or filter, or sort by the output message. In addition, for teamwork with other developers, a shared web server allows other developers to assign textual comments to the reports or manually mark them as true or false positives. We used these capabilities built into CodeChecker during our evaluation, especially for the case study discussed in Section 5.7.

5.2. Analysed projects

We gathered a sample of open-source projects, from small to large scale, encompassing various domains, from system tools to machine-learning image processing libraries. Some of the projects gathered were part of a "test set" for contemporary articles targeting C++ static analysis (Horváth et al., 2018; Horváth et al., 2020), while others were selected by trying to find projects similar in scope and size to those analysed in previous work, which mostly focused on Java. The list of the projects, with the versions we used for analysis, is shown in Table 1.

The system requirements of the analysis are consistent with other compiler-based tools, taking between mere seconds and 5 min for each project per configuration on a contemporary middle-class 8-core system. Most of the time during the analysis is spent in the semantic analyser of the compiler, which operation is irrespective of our specific rule or its implementation. As our analysis runs on the syntax tree, the parsing must be performed. The only outlier in terms of execution time was LLVM itself, for which the analysis took 50 min.

We executed our measurements on a computer with Intel® Core™ i5-8350U processor with 4 physical cores – with HyperThreading™ allowing 8 logical cores – at 2.8 GHz speed. It had 16 GiB DDR4-2400 MHz memory, and a Samsung® PM981 256 GiB NVMe™ storage device. C++ compilation and static analysis are single-threaded for a translation unit, using between 1 and 1.5 GiB of memory for each process.

5.3. How many functions are affected?

A detailed breakdown of the number of functions we found to have at least one pair of adjacent mixable parameters is shown in Table 2. We compared the individual relaxations or extensions of the type equivalence (strict mode) to type convertibility to see how many functions produced reports. Users of the analysis can toggle between these relaxations with configuration options to fit their project's needs.

Table 1
The details of the projects that formed the test set for our analysis. *LoC* is also shown to indicate the volume of the project. The LoC metric includes only C and C++ source files, without build-generated content.

Language	Project	Lines of code	Release	Commit
C	curl (Stenberg et al., 1996)	138 067	7.67.0	2e9b725
	git (Torvalds et al., 2005)	223 924	2.24.1	53a06cf
	netdata (Netdata Corporation, 2013)	75 546	1.19.0	5000257
	PHP (PHP Group, 1999)	708 424	7.4.1	b1a8ab0
	Postgres (PostgreSQL Developers, 1996)	841 577	12.1	578a551
	Redis (Sanfilippo et al., 2006)	122 561	5.0.7	4891612
	TMux (Marriott et al., 2007)	45 153	3.0	bbcb199
C++	Bitcoin (Nakamoto, 2009)	145 949	0.19.0.1	1bc9988
	CodeCompass (Ericsson, 2016)	23 810	–	1796dcb
	guetzli (Google, Inc., 2016)	7 328	1.0.1	a0f47a2
	LLVM/Clang (LLVM Foundation, 2001)	3 100 139	9.0	0399d5a
	OpenCV (Xperience AI, 2019)	952 484	4.2.0	bd89a6
	ProtoBuf (Google, Inc., 2008)	214 895	3.11.2	fe1790c
	Tesseract (Smith, 2006)	148 968	4.1.0	5280bbc
	Xerces (Apache Software Foundation, 1999)	173 540	3.2.2	71cc0e8
	Z3 (Microsoft Research, 2012)	462 454	4.8.9	79734f2

The **A** column shows the number of functions that the analysis routine picked up on and started calculating the convertibility between types for. This is not all the functions that are found in the project. First, we only consider functions that are *definitions* – as opposed to *declarations* – in the project's source code. We ignored functions that come from "system headers", which are commonly used by developers in the project configuration to tell compilers and tools that the code in those files are from third-party libraries outside of the developers' jurisdiction. Producing diagnostics only for the function *definitions* in the analysed project's code allows us to only report issues that the developers have a chance to fix. Two additional kinds of functions were omitted from the analysis. We ignored overloaded operators, which may only have at most 2 parameters in C++, due to their high false positive rate. Using overloaded operator symbols is usually an indication of implementation of custom mathematical operations. Also, we did not match and model for *instantiated function templates*. The reasoning behind the latter is explained in detail in Section 8. None of the functions that match the aforementioned ignore criteria are part of the count of functions in the **A** column. Compared to our previous paper (Szalay et al., 2020), these factors contributed to the change in the number of analysed functions. In the case of *OpenCV*, this drop was massive, reducing the analysed functions from 11 760 to 6 746. However, such a drop is reasonable as *OpenCV* uses user-defined overloaded operators immensely in their codebase in the underlying mathematical structures' implementation.

We also implemented a way for users to specify certain parameter names and parameter types that shall be ignored. By default, and during our evaluation, the parameter type names representing various *iterator* types were filtered, as iterators, passed always as two parameters highlighting a left-closed, right-open range, would be a common cause for superfluous warnings. The *Ranges* library introduced in C++20 allows swapping pairs of *iterators* with a single, more strongly typed parameter. We decided to exclude all parameters that are *unnamed* in the definition – the rationale behind this being that unnamed parameters of the definition cannot be used by the function body in any context, and such cases usually indicate a necessary interface that cannot be changed, usually because the function's signature has to match a polymorphic function inherited from a base class or a function type required by an external library. Such functions were part of the analysed function set, and thus, counted in the value in the **A** column, as the implementation investigated the function, and only when doing so, deemed the parameter not to be analysed and reported in a potential mix.

Table 2
Detailed count of functions producing *at least one* mixable adjacent parameter diagnostic across the analysis rule's relaxation/extension configurations. Note that the same function might be matched by either of the relaxations individually, and the results' cardinality from enabling both relaxations is not merely the sum of the individual modes'.

Lang.	Project	Functions analysed (**A**)	Strict (**S**, Section 3)		CV (Section 3.3)			Imp (Section 4)			CV ∪ Imp			
			T (total)	*% of A*	*T*	*% of A*	*+ vs. S*	*T*	*% of A*	*+ vs. S*	*T*	*% of A*	*+ vs. CV*	*+ vs. Imp*
C	curl	865	134	15.49%	153	17.69%	19	209	24.16%	75	228	26.36%	75	19
	git	5 641	1 418	25.14%	1 466	25.99%	48	1 595	28.28%	177	1 644	29.14%	178	49
	netdata	719	227	31.57%	243	33.80%	16	294	40.89%	67	308	42.84%	65	14
	PHP	5 984	1 272	21.26%	1 304	21.79%	32	1 509	25.22%	237	1 539	25.72%	235	30
	Postgres	9 436	2 696	28.57%	2 804	29.72%	108	3 708	39.30%	1 012	3 820	40.48%	1 016	112
	Redis	1 745	393	22.52%	418	23.95%	25	454	26.02%	61	484	27.74%	66	36
	TMux	1 032	248	24.03%	259	25.10%	11	298	28.88%	50	306	29.65%	47	8
C++	Bitcoin	1 773	394	22.22%	412	23.24%	18	499	28.14%	105	512	28.88%	109	22
	CodeCompass	191	27	14.14%	27	14.14%	0	28	14.66%	1	28	14.66%	1	0
	guetzli	153	72	47.06%	76	49.67%	4	75	49.02%	3	81	52.94%	5	6
	LLVM-CTE	32 339	6 109	18.89%	6 181	19.11%	72	6 817	21.08%	704	6 898	21.33%	713	81
	OpenCV	6 746	3 286	48.71%	3 414	50.61%	128	3 590	53.22%	304	3 714	55.05%	300	124
	ProtoBuf	1 997	313	15.67%	317	15.87%	4	386	19.33%	73	392	19.63%	75	6
	Tesseract	1 962	793	40.42%	797	40.62%	4	874	44.55%	81	880	44.85%	83	6
	Xerces	1 594	446	27.98%	462	28.98%	16	465	29.17%	19	532	33.38%	70	67
	Z3	9 673	2 600	26.88%	2 608	26.96%	8	2 793	28.87%	193	2 801	28.96%	193	8

We manually removed all diagnostics from the result set that pointed to what was clearly and trivially identifiable generated code. Only a few of the projects – most notably CodeCompass and LLVM – had such examples. We considered files that were not part of the project's source tree before the build or were explicitly created in directories such as `build/` as *generated code*. Generated code is not written directly by the users, and the insertion point of error is not in the generated code text.

The individual analysis modes refer to configurations as follows.

- In **Strict** mode, only exact type-equal ranges are matched, with `typedef`s and references (see Sections 3.1 and 3.2) always diagnosed.
- **CV** mode allows mixing types that only differ in their qualifiers (see Section 3.3), e.g. allowing (`int`, `const int`) to match.
- **Imp** mode enables calculating and considering *implicit conversions* (see Section 4), e.g. (`double`, `int`).
- In **CV** ∪ **Imp** mode, both relaxations are enabled, e.g. the following are also mixable: (`double`, `const int`).

On average, 23% of the functions in the projects matched in the strict type equivalence check, while this figure rises to 33% with implicit conversions modelled.

5.4. How long are the mixable ranges?

Reports of length 2 ranges are the most prevalent across all projects and configurations, making up three-fourths of the total findings. These results are consistent with findings in existing literature (see Section 2, Pradel and Gross (2013), Rice et al. (2017), Varjú (2017)) 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 strict – most restrictive – and *CV & Implicit* – least restrictive – configurations are shown in Table 3. The average number of findings of a particular length is depicted in Fig. 1. We plotted the results for C and C++ separately due to the broader set of what is considered *implicit conversions* in C++. While it is natural from the languages' rules 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 the longer ranges.

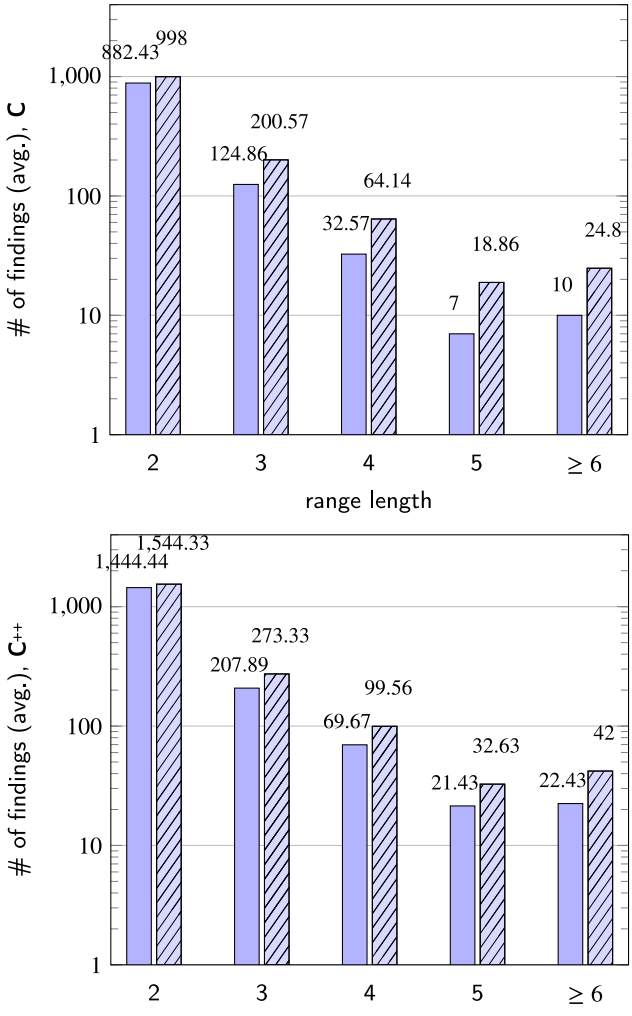

Fig. 1. Number of findings of a particular length averaged for C and C++ test projects. Filled columns depict strict mode, striped columns depict *qualifier mixing* and *implicit conversions* (Sections 3.3 and 4) turned on.

Table 3
Count of individual mixable ranges of a particular length reported for the tested projects. The "*max len.*" column shows the length of the longest finding if more than 5. Column *S* details strict type equality (Section 3), *CV-I* details qualifier mixing and implicit conversions (Sections 3.3 and 4) enabled.

Lang.	Project	2		3		4		5		≥ 6		(max len.)	
		S	*CV-I*	*S*	*CV-I*	*S*	*CV-I*	*S*	*CV-I*	*S*	*CV-I*	*S*	*CV-I*
	curl	125	211	12	19	5	8	1	2				
	git	1 337	1 505	143	211	51	64	2	7	3	6	6	7
	netdata	198	253	36	55	10	16	5	9	9	15	12	12
C	PHP	1 076	1 251	232	302	31	51	4	12	1	7	8	8
	Postgres	2 501	3 103	371	708	105	265	26	91	32	89	9	20
	Redis	358	417	40	61	7	18	4	5				
	TMux	204	246	40	48	19	27	7	6	5	7	7	7
	Bitcoin	318	415	80	91	13	33	3	9	1	4	6	7
	CodeCompass	20	17	6	10	2	2						
	guetzli	59	59	15	18	6	6	3	5	1	2	7	7
	LLVM-CTE	5 708	6 171	666	878	175	284	58	86	26	48	13	13
C++	OpenCV	3 051	3 112	674	914	283	373	65	124	84	178	20	21
	ProtoBuf	289	336	23	45	5	16	1	1	7	7	10	10
	Tesseract	683	700	127	160	63	87	10	22	22	35	11	11
	Xerces	406	484	32	47	29	31		1				
	Z3	2 466	2 605	248	297	51	64	10	13	16	20	11	11

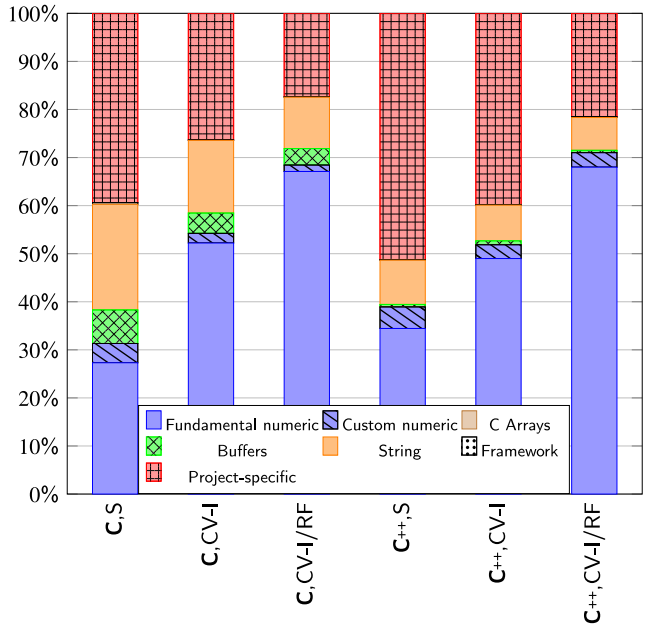

Fig. 2. Relative share of the hand-made type categories that are involved in mixable parameter ranges. *S* columns depict strict type equality (Section 3) analysis, *CV-I* show *qualifier mixing* and *implicit conversions* (Sections 3.3 and 4), *CV-I/RF* show *CV-I* with the filters (Section 6) turned on.

5.5. How different types contribute to the issue?

Not all types are used equally in projects. To tackle the issue and understand how refactoring might take place, it is interesting to look at the distribution of involved types. We have investigated the reports in the test projects and hand-categorised the types of the parameters found in reported ranges into the following 7 categories.

1. *Fundamental numeric*: The fundamental, built-in, "keyword" numeric types, including integers and floating-point numbers, and trivial type aliases of these.
2. *Custom numeric*: Other types of scalar nature, such as custom precision integers (e.g. `int512`).
3. *C arrays*: Classic C-style array types, such as `int T[]`, whether known or unknown size.
4. *Buffers*: Type-erased (`void*`) or template (`arrayRef<T>`) wrappers over buffers – including sockets – and arrays of or pointers to `std::byte`.
5. *Strings*: Parameters that take `char*`, `std::string` or types related to string operations, like `std::string_view`, `llvm::SmallString`. Due to `std::byte` only introduced in C++17, several projects that deal with "buffers" do it through `char*`, and it is not easy to distinguish the two cases.
6. *Framework types*: All standard (POSIX, C Standard Library, 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 case, *Qt*.
7. The last category, *Project-specific*, is the fallback bucket where every other type not fitting the previous categories are put. These types user-defined for the project at hand.

There is no possibility of passing a non-pointer in place of a pointer without the compiler catching it. The difference between a reference parameter and a value parameter is only in terms of whether the changes to the object is visible outside the function's scope. Due to this, pointers or references of a type *T* is counted in the same bucket as if *T* was taken by value, except for `char*`. The relative number of types involved in the findings for a particular configuration is depicted in Fig. 2.

Fundamental numeric and *project-specific* 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, and their share only grows even when filtering heuristics (see Section 6) are applied, showing that there is a corpus of functions similar in nature to `f(int a, double b)`.

Our findings confirm that the low level of detail in projects' types allows for 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.

5.6. Details on exceptional findings

Several functions in *LLVM*, such as `WriteSecHdrEntry`, `resolveRelocation` take 10 numeric parameters with no restriction or semantic information to be inferred from the type. *OpenCV* 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

Table 4
The classification of individual bug reports of strict (Section 3) and most relaxed (Sections 3.3 and 4) mode analysis across the hand-evaluated projects. Reports were put into either of the three categories *true positive*, *false positive*, or *heuristically discard*. The **HD** ✓ rows show the number of reports that were successfully discarded from the *HD*-classified reports.

Project		Bitcoin		CodeCompass		Xerces	
Strict	Total	456		33		512	
	TP	185	40.57%	26	78.79%	260	50.78%
	FP	185	40.57%	2	6.06%	149	29.10%
	HD	86	18.86%	5	15.15%	103	20.11%
	HD ✓	84	97.67%	5	100%	102	99.03%
CV-Imp	Total	809		35		725	
	TP	532	65.76%	27	77.14%	429	59.17%
	FP	178	22.00%	2	5.71%	159	21.93%
	HD	99	12.23%	6	17.14%	137	18.90%
	HD ✓	97	97.98%	6	100%	136	99.27%

project in an implicit fashion and should *"never be used directly"*.[8] There are several functions with large sets of mixable arguments resulting from this "type erasure": `cv::rectify3Collinear` takes 8 `InputArray`s, then a numeric type, then 4 `InputArray`s and 7 `OutputArray`s.

In *PostgreSQL*, 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* of functions with ≥ 9 adjacent numeric arguments.

We have seen that there are several functions in the projects which contain lengthy mixable parameter ranges. Contrary to our expectations, there were not many bug reports stemming from argument selection issues found in issue trackers for well-known public repositories. A likely reason for this is that most erroneous function calls are detected through other, potentially more expensive means, such as manual or automated testing systems ahead of or during code review, and apart from a few extreme cases, are stopped in time before making it into a production release.

5.7. Case study: How actionable are the reports?

All analyses, but especially static analysis, always carries the possibility of false positives that are hard to investigate. It has been shown in studies by Peters and Zaidman (2012) and Kovács and Szabados (2016) that developers, in general, are often not concerned about code smells, and code quality only improves when personal motivation is present. Due to this, ensuring that the tool results are useful and actionable is paramount. Some projects produce multiple hundreds or even thousand results even in the strictest analysis, with the report ratio hovering between 20% and 30%, as depicted in Table 2. We have selected three projects from the result set and manually investigated every report produced: Bitcoin, CodeCompass and Xerces. These projects are medium-sized, produce only a few hundred results which can be digested by one or two people, and the projects themselves are comparatively easy to understand, as they deal with very specific purposes that are not extremely abstract or domain-specific either. In the case of CodeCompass (Porkoláb et al., 2018), we are also the developers of the project, so we could gather an inside view of how the reports are useful during a practical evaluation by actual developers receiving them. We do not have an affiliation with Bitcoin and Xerces.

We have investigated the *Strict* and the *CV-Imp* – most relaxed – analysis configurations for all three projects. The evaluation consisted of hand-labelling the reports into three categories. The classification is tallied in Table 4.

- **TP** – true positive – findings are valid reports that indicate that the parameters next to each other are swappable, and this swapping will constitute an argument selection defect.
- **FP** – false positive – findings are parameter ranges which type safety cannot be enhanced via refactoring as the types used unfortunately have to remain the same.
- **HD** – heuristically discard – findings are another kind of false positives, but there is some information available to the compiler, not just the developers, that allow us to silence the superfluous warning automatically. We discuss such heuristics in Section 6.

[8] Quote from the documentation of `cv::_InputArray` (Xperience AI, 2019): *"The class is designed solely for passing parameters. That is, normally you should not declare class members, local and global variables of this type".*

```
int fun(int a, int b, int c, int d) {
  if (a < b)       // a and b in the same expression
    return c + d; // c and d in the same expression
}
```

Listing 4: All 4 parameters of the function share the same type (`int`), and thus a mixable adjacent parameter range of all parameters would be diagnosed. While the diagnostic is valid in a pedantic sense, such a warning is not useful for developers. It is visible from the function's body that parameter pairs (a, b) and (c, d) are "used together", and thus, swapping them *might not* be an issue in the context of type safety. The *relatedness heuristics* remove such pairs, leaving only the range $[b, c]$ to be reported as a mixable.

6. Turning formal interface checks into actionable analysis

The evaluation of the pure interface check shows that such a sterile checking of types via an interface guideline or rule is not good enough to be practically applicable in the general case. We used both the language rules and practical experience to devise some rules that aim to silence a subset of the generated warnings in order to have the more important and perhaps more easily fixable issues highlighted to the developers. These heuristics may take false negative decisions, which is precisely why, in the implementation, all are user-configurable. Our suggested way of actionable refactoring is to start with these heuristics *turned on*. Once all results from the analysis are consumed – either fixed, or deemed a false positive – turn off one of more of the heuristics and analyse their project again.

Definition 3 extends Definition 1 with the heuristic predicate. An example of the heuristics is shown in Listing 4.

Definition 3. Given a function signature $f(T_1\ p_1,\ T_2\ p_2)$, parameters p_1 and p_2 are **mixable after filtering**, if they are mixable – as per Definition 1 –, and a filtering predicate $\mathcal{P}(p_1, p_2)$ does not hold.

The heuristics detailed below have been made part of the implementation, and we analysed how the amount of matched functions and the length of the reports changed. The results are shown in Tables 5 and 6, respectively. On average, around half of the reported functions were discarded using the heuristics. The heuristics' potency at cutting away the excessively long reports follows naturally from the method applied.

Table 5
The breakdown of the effect of applying filtering heuristics on the number of functions producing *at least one* mixable adjacent parameter range diagnostic. The totals of modes *S* and *CV* ∪ *Imp* are the same as in Table 2. The rest of the columns detail the number of functions for which diagnostics disappeared, compared to the number of findings without the filtering. Note that the same function might have multiple separate ranges diagnosed, from which the filtering removes only some, but leaves the function still matching. Thus, the results' cardinality from enabling more heuristics is not trivially equal to subtracting the total of all individual modes'.

Lang.	Project	Strict (S)	CV ∪ Imp	No-`bool` (Section 6.1)		Rel (Section 6.2)				Fil (Section 6.3)				Rel ∩ Fil			
		Total	*Total*	*- vs S.*		*- vs. S*		*- vs. CV ∪ Imp*		*- vs. S*		*- vs. CV ∪ Imp*		*- vs. S*		*- vs. CV ∪ Imp*	
C	curl	134	228	1	0.75%	61	45.52%	80	35.09%	4	2.99%	4	1.75%	64	47.76%	83	36.40%
	git	1 418	1 644	0		826	58.25%	887	53.95%	142	10.01%	119	7.24%	902	63.61%	953	57.97%
	netdata	227	308	0		123	54.19%	134	43.51%	6	2.64%	5	1.62%	124	54.63%	135	43.83%
	PHP	1 272	1 539	6	0.47%	628	49.37%	699	45.42%	165	12.97%	164	10.66%	697	54.80%	766	49.77%
	Postgres	2 696	3 820	157	5.82%	1 449	53.75%	1 730	45.29%	336	12.46%	272	7.12%	1 572	58.31%	1 820	47.64%
	Redis	393	484	0		226	57.51%	251	51.86%	51	12.98%	41	8.47%	238	60.56%	261	53.93%
	TMux	248	306	0		138	55.65%	141	46.08%	81	32.66%	77	25.16%	169	68.15%	170	55.56%
C++	Bitcoin	394	521	28	7.11%	264	67.01%	318	61.04%	29	7.36%	27	5.18%	274	69.54%	328	62.96%
	CodeCompass	27	28	0		12	44.44%	12	42.86%	0		0		12	44.44%	12	42.86%
	guetzli	72	81	0		44	61.11%	45	55.56%	19	26.39%	17	20.99%	47	65.28%	48	59.26%
	LLVM-CTE	6 109	6 898	577	9.45%	4 055	66.38%	4 294	62.25%	818	13.39%	801	11.61%	4 160	68.10%	4 399	63.77%
	OpenCV	3 286	3 714	48	1.46%	2 320	70.60%	2 462	66.29%	533	16.22%	418	11.25%	2 415	73.49%	2 527	68.04%
	ProtoBuf	313	392	21	6.71%	193	61.66%	214	54.59%	57	18.21%	57	14.54%	201	64.22%	222	56.63%
	Tesseract	793	880	26	3.28%	461	58.13%	461	52.39%	106	13.37%	96	10.91%	493	62.17%	490	55.68%
	Xerces	446	532	23	5.16%	303	67.94%	349	65.60%	66	14.80%	64	12.03%	319	71.52%	365	68.61%
	Z3	2 600	2 801	122	4.69%	2 022	77.77%	2 091	74.65%	861	33.12%	841	30.02%	2 134	82.08%	2 194	78.33%

6.1. Ignoring `bool` parameters

A typical cause of lengthy findings is the numerous sequence of `bool` parameters, such as LLVM's function `AnalysisDeclContextManager` that takes **12** toggles. However, booleans are not easily refactored to safer, stronger types (see Section 7) without increasing the source code's verbosity with little help to adding additional semantics to something as simple as a boolean. Enforcing coding conventions, such as requiring the name of the parameter to be typed out when passing a boolean literal or requiring boolean variables to always match the parameter's name, make existing name-based analyses (Rice et al., 2017) excel at catching swaps of booleans.

The implementation allows any type to be ignored based on their name, and we performed an analysis with [`bool`, …] sequences ignored. This is recommended assuming other tools can enforce specific coding conventions around `bool`s, but only if implicit conversions are not considered during the analysis.

6.2. Removal of related – *used together or in a similar fashion* – parameter pairs

At the time of our work and initial paper, the *C++ Core Guidelines* rule (Stroustrup and Sutter, 2017) associated with the basis of our work was called *"Avoid adjacent **unrelated** parameters of the same type"*. However, the concept of *related* parameters was not explained. Since the initial work, through our discussion with the guideline authors (Whisperity, 2021), the rule's title changed to *"Avoid adjacent parameters that can be invoked with the same arguments in either order with different meaning"*. While the latter interpretation of the title is even harder to formalise and check through automated means, the below heuristics are called *relatedness heuristics*, with the namesake being the former title.

We use the concept of *relatedness* to emphasise that a parameter pair's types cannot be changed to a more strongly typed version in any way. This information can be deduced from looking into the function's implementation during the analysis. Consider the example in Listing 4: all of the parameters of the function are of type `int`, an accidental swap of otherwise distinct numeric inputs – e.g. parameters `b` and `c` – is possible. Relatedness – just like the type of a variable – is a *compile-time* property. Neither this property nor our analysis as a whole is checking whether a swapped call between `a` and `b` in Listing 4 is "valid" – such an analysis would require flow-sensitivity and could be infeasible in general.

When two parameters are conceptually "related", they either have to remain the same type indefinitely, or the place in the code where the refactoring should start is in another function. We defined the following criteria for relatedness, with either sufficient to deem a parameter pair not to be warned about. In our implementation, due to the limitations of Clang-Tidy's framework, the criteria are applied to the parameters directly by only looking at the syntax tree representation. More powerful analysis engines in the future may employ flow-sensitive or path-sensitive modelling (Xu et al., 2010), points-to analysis (Ball and Horwitz, 1993; Pearce et al., 2007), and taint analysis (Wang et al., 2008; Grech and Smaragdakis, 2017) to propagate *relatedness* through the usage of parameters, e.g. to local variables created from the parameters and the context these local variables are used in.

Parameters that appear in the same sub-expression tree rooted by a statement are the most common case of direct relatedness. In this context, *expressions* and *statements* are as per the language specification. A function call (`f(a, b)`), an addition (`a + b`), a comparison (`a < b`), are all *expressions*, while variable declarations, conditional branching, loops are *statements*. The decision to consider the condition of an `if` statement and their branches distinct trees explain why not all parameters are deemed related in Listing 4.

Parameters passed in distinct function calls to the same function's same overload as an argument to a parameter on the same index – i.e. `f(a, x)` and `f(b, y)` mark pairs (a, b) and (x, y) – are also related. We dubbed functions containing such constructs *"dispatchers"* – they pass either *a* or *b* to another function, depending on conditions.

Parameters that are returned by the current function in different `return` statements are deemed related. We call such functions *"selectors"* – they select either *a* or *b* as their return value, depending on conditions.

For record types, parameters that have the same data member accessed or member function called inside the function, even if in different expressions, are also considered related. This latter case captures the intent of the two parameters being used in a similar fashion, and this rule is exceptionally capable at – despite the lack of flow-sensitive analysis – silencing operations, such as comparisons, which first take the member of a record into a local variable, and perform something with these local variables.

Table 6
Count of the individual mixable ranges of a particular length reported for the tested projects after applying both filtering heuristics (Sections 6.2 and 6.3). Compare with Table 3. The "*max len.*" column shows the length of the longest finding if more than 5. Column *S* details strict type equality (Section 3), *CV-I* details qualifier mixing and implicit conversions (Sections 3.3 and 4) enabled.

Lang.	Project	2		3		4		5		≥ 6		(max len.)	
		S/RF	*CV-I/RF*	*S/RF*	*CV-I/RF*	*S/RF*	*CV-I/RF*	*S/RF*	*CV-I/RF*	*S/RF*	*CV-I/RF*	*S/RF*	*CV-I/RF*
C	curl	72	145	2	6	1	1	1	1				
	git	512	667	25	55	5	8						
	netdata	95	160	14	24	1	6	1	2	4	4	12	12
	PHP	554	724	42	78	4	10			1	2	8	8
	Postgres	1 117	1 814	114	313	33	94	11	28	4	14	8	15
	Redis	149	215	11	15	1	2						
	TMux	77	128	4	9	1	3						
C++	Bitcoin	113	177	11	22	4	6		2				
	CodeCompass	13	13	1	2	2	2						
	guetzli	25	32	3	5		1						
	LLVM-CTE	1 824	2 299	154	243	34	53	11	15	1	3	13	13
	OpenCV	925	1 179	107	194	23	47	1	15	6	14	10	10
	ProtoBuf	107	146	6	18		8			1	1	7	8
	Tesseract	279	344	41	62	9	18	1	6	7	8	8	8
	Xerces	119	154	7	14	4	5		1				
	Z3	425	539	48	64	6	14	2	2	2	4	6	6

6.3. Removal of named parameters following a pattern

Developers encode semantic information in the symbol names. This information can be used to identify the cases where the similarity of the names indicate that the parameters are designed to be used in a similar fashion, even if such relation cannot be deduced from looking at the body of the function. Rice et al. (2017) noted that even for name-based analysis, certain families of names lack information to be useful. Our predicate here, present in the implementation as another user-configurable option, follows a similar rationale.

Given two parameters (p_1, p_2) and a dissimilarity threshold k, the pair is ignored if their names are each other's prefix or suffix with at most k letters of difference on the non-common end.

Definition 4. Let ℓ_1 and ℓ_2 be the lengths of the strings n and m, respectively. Let n^p be $n_{1,\dots,\ell_1-k}$; n^s be $n_{k+1,\dots,\ell_1}$ – n's prefix and suffix ignoring k letters. m^p and m^s are analogous for m. The string n and m are **prefix/suffix covers** of each other **with at most** k **difference** if $\exists \mathcal{S}_1, \mathcal{S}_2 : |\mathcal{S}_1| \leq k \wedge |\mathcal{S}_2| \leq k$, and from $(n = n^p \cdot \mathcal{S}_1 \wedge m = m^p \cdot \mathcal{S}_2 \wedge n^p = m^p)$ and $(n = \mathcal{S}_1 \cdot n^s \wedge m = \mathcal{S}_2 \cdot m^s \wedge n^s = m^s)$ at least one holds.

Corollary 1. *Selecting $k = 0$ collapses the previous requirements to "$n = m$", effectively turning off the heuristic, as two parameters cannot have the same name in the same function.*

Examples of heuristically removable cases from Section 5.7 include names of the following fashion: `LHS`, `RHS`; `Qmat`, `Rmat`, `Tmat`; `text1`, `text2`. The former two are examples of a suffix cover, while the latter is a prefix cover, with one letter difference. We did our practical evaluation with $k = 1$ threshold, as this covers the vast majority of cases that are beneficial to ignore. From the point of enhancing type safety and guarding against swapping, `text1`, `text2`, . . . carries little information to the developer (Rice et al., 2017), but tells us that diagnosing such constructs would only produce noise.

7. Methods for strengthening function interfaces

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 them. At the same time, some are theoretical for the general situation, with implementations existing for specific use cases.

7.1. 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 solution properly fixes the issue with implicit conversion, generating all possible to-disable overloads for all possibly affected functions is a daunting task. It would also result in severe code bloat by having $\mathcal{O}(n^2) - 1$ disallowing declarations for each pair of mixable parameter pairs identified.

7.2. Explicit type aliases

One possible solution to badly ordered arguments is to replace the adjacent types with such that are 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 run-time 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 semantic typedef instance's size and behaviour are the same as the single variable contained within, with no additional steps to take at the destruction.

This is not the case in Java, where heap allocations are done, and the boxing types cause a performance hit, as shown by Chiba (2007). In the same year automatic, just-in-time inlining of reference-semantic objects inside their parents was proposed by Wimmer and Mössenböck (2007) to speed up the access to member objects that are composited – based on their usage – instead of aggregated. While related, this feature deals with member objects' members, and not specifically the boxing types; however, the notion could be applied for boxing too. We found no discussion on whether this implementation or something alike made it upstream and became generally available. Even though the Java Virtual Machine is continuously evolving, the performance issues with boxing appear to remain to this day.

```
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 = w.value, he1 = h.value;

  // Obtaining values in C++:
  int wi2 = w(), he2 = 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.

There are proposals to change the language's specification to allow the usage of true value semantics (Rose, 2012), or to allow generic programming with the primitive value types of Java (Goetz, 2014). The former proposal's feasibility is actively investigated in *Project Valhalla*, the Java Development Kit's feature incubator project.

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 (Sillito et al., 2008). 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 were proposals (Brown, 2013, 2015) to include *opaque typedefs* for C++ but these have not yet made it into the language. Similarly, Baráth and Porkoláb (2015) discusses a wrapper class over numeric conversions. The LLVM project, in which several functions take multiple boolean parameters adjacent to each other (see Section 6.1) have been using comments to indicate which parameter is assigned a literal value. Community members have suggested implementing wrappers around such instances (Greene, 2019).

Function signatures might commonly repeat identifier-like phrases in the case of strong typedefs over booleans, 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.

7.3. 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 program's business logic, the wrapped type would be used. Strong typing (Meyer, 1992), in which the expressive capabilities of the type system and types used in the program are increased, has been investigated for their effect on language design (Madsen et al., 1990) and as a method to increase type coherence for persistent systems (Kemper and Moerkotte, 1991) and to prevent security vulnerabilities in web applications (Robertson and Vigna, 2009).

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 (Barnett et al., 2004; Sillito et al., 2008; Roehm et al., 2012) 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 (ISO/IEC JTC 1/SC 22, 1994). 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.

7.4. 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* (Schäling, 2014), which was introduced in C++11. Chrono applies strong types and safe conversions regarding 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 were left to the implementation to specify. 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 –, doing so was not mandatory either.

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 – variable representing "the" time, the concepts of hour, minute, etc. was added. User-defined literals allow expressing these concepts in an easily readable way, such as `2021y`.

User-defined literals come from the source code – as opposed to literal suffices defined by the core language itself, such as `f` in `0.5f` – by defining functions of a specific naming pattern. Literals defined in such fashion, even by the *Standard Template Library*, is considered "user-defined" by the language specification. 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.

```cpp
#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 = 2021y / March / 5;
  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(11,59,59,2021,3,5);
  // 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*.)

Building upon the foundations and success of Chrono, various other libraries, such as one for physics dimension calculations (Pusz, 2019), exist. Other libraries, such as the one by Sommerlad (2021), allow a generic way of expressing strong types and the set of allowed operations.

8. Threats to validity

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 (Vandevoorde et al., 2017). We diagnose only *primary templates* and *explicit template specialisations* and provide no warning for cases similar in nature to `template <typename T, typename U> void f(T, U)` instantiated by a call `f(1, 2);`. For this instantiation, `T` and `U` are both `int`, but `T` and `U` are distinct placeholder types for the *primary template*.

Another case not modelled accurately and thus ignored from the analysis is when the adjacent parameters' types' equivalence or convertibility may only be proved through performing the template instantiation. This is costly, and for the representation of the syntax in the compiler, destructive operation. In addition, an instantiation is context-sensitive, and the exact definition for a potential *explicit specialisation* is only known at the point of instantiation, making it impossible to deduce or disprove the possibility of the parameter swap and produce a diagnostic. The function depicted in Listing 7 contains the possibility of mixing up the two parameters in the general case, but this is not diagnosed.

```cpp
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, deducing this depends on how `vector<T>` for the particular `T` is defined, as there could be *explicit specialisations*.

In addition to templates, variadic functions – `int printf( const char* format, ...)`, where the "..." may take an arbitrary amount of parameters – are also not diagnosed in the general case. A call to a variadic function with a given sequence of arguments is conceptually similar to a template instantiation and thus ignored the same way.

However, all these cases are only capable of producing false negatives and do not pose a threat to the already found and discussed results of Section 5. We plan to work with the open-source community to refactor the framework so that diagnosing these cases will be possible and accurate in the future.

We performed our analysis on tagged release versions of the tested projects. Releases are often heavily scrutinised before going public. Thus, the chance of a real mistaken function call being in the project in a way that it causes bogus behaviour is lesser than during development. While the projects' code is frozen in the version control system with the release, changes in third-party library dependencies and the development environment might change which functions are compiled in a project and how, and thus the numbers we reported may not match future analyses exactly.

9. Conclusion

Similarly typed parameters allow for potential misuse at call sites. Mistakenly ordered arguments go unnoticed as the match of the types of arguments to their receiving parameters is the only requirement written in language specifications. While developers use symbol names to encode semantic information, the language and compilers are not required to extract knowledge from the names. Unless extensive testing or analysis tools are employed, a real swapped function call that affects the program's behaviour remains hidden. The proliferation of coarse-grained types requires 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 – through user-toggleable relaxations – 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, even during development, and without the need to discover an actual bad function call. Our analysis implementation does not aim to replace but rather to complement existing literature and works, which mainly focused on the *name-based* analysis between arguments and parameters.

We developed a practical implementation on top of the LLVM Compiler Infrastructure's C-family compiler's, Clang's static analysis framework, and as such, could easily be integrated into a development pipeline. Various integrated developer environments (*IDE*s), such as Eclipse (The Eclipse Foundation, 2001) or CLion (JetBrains Inc., 2015), already integrate, or through the Language Server Protocol (Microsoft Corporation, 2015), allow integrating analysis tools into the same views where code is written.

The pure, formal interface check, however, produces time-consumingly verbose results when applied to existing projects. We devised, implemented, and evaluated several heuristics, most importantly the *relatedness heuristics*, in Section 6.2. These heuristics suppress individual analysis reports about functions that are violating the rule of not having multiple adjacent parameters that can be mixed, but for which the strengthening of interface type safety or refactoring of the function is either impractical, or likely impossible. The implementation of these heuristics allow developers to receive results that offer a path of lesser resistance for type safety enhancement.

While our discussion focused primarily on C and C++ programming languages, the idea can be applied to other multi-paradigm 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 dangerous implicit conversions.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to thank Aaron Ballman (Intel Corporation) and Herb Sutter (Microsoft Corporation) for their invaluable help as C++ experts given during the review of the analysis rule's implementation process in Clang-Tidy. We are grateful to Scott et al. (2020), whose paper was presented in the same session as ours, and the participants of SCAM for the insightful discussion during the session.

References

andreser, 2017. `_sUbborrow_u64 argument order inconsistent`. URL: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=81294. accessed 2019-12-16.

Apache Software Foundation, 1999. Xerces C++. URL: http://xerces.apache.org. version `3.2.2` (`71cc0e8`), accessed 2019-12-30.

Arnold, K., Gosling, J., Holmes, D., 2000. The Java Programming Language, third ed. Addison-Wesley Longman Publishing Co., Inc., USA.

Ball, T., Horwitz, S., 1993. Slicing programs with arbitrary control-flow. In: Fritzson, P.A. (Ed.), Automated and Algorithmic Debugging. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 206–222. http://dx.doi.org/10.1007/BFb0019410, URL: http://link.springer.com/chapter/10.1007/BFb0019410.

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

Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W., 2004. Verification of object-oriented programs with invariants. J. Object Technol. 3 (6), 27–56, URL: http://jot.fm/issues/issue_2004_06/article2.pdf.

Brown, W.E., 2013. Toward Opaque Typedefs for `C++1Y`. Technical Report, ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee (ISOCPP) proposals, URL: http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3515.pdf.

Brown, W.E., 2015. Function aliases + extended inheritance = opaque typedefs. Technical Report, ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee (ISOCPP) proposals, URL: http://open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0109r0.pdf.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2010. Exploring the influence of identifier names on code quality: An empirical study. In: 2010 14th European Conference on Software Maintenance and Reengineering. pp. 156–165. http://dx.doi.org/10.1109/CSMR.2010.27, URL: http://ieeexplore.ieee.org/document/5714430.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2011. Improving the tokenisation of identifier names. In: Mezini, M. (Ed.), ECOOP 2011 – Object-Oriented Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 130–154. http://dx.doi.org/10.1007/978-3-642-22655-7_7, URL: http://link.springer.com/chapter/10.1007/978-3-642-22655-7_7.

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

Caprile, B., Tonella, P., 1999. Nomen est omen: analyzing the language of function identifiers. In: Sixth Working Conference on Reverse Engineering (Cat. No.PR00303). pp. 112–122. http://dx.doi.org/10.1109/WCRE.1999.806952, URL: http://ieeexplore.ieee.org/document/806952.

Caprile, B., Tonella, P., 2000. Restructuring program identifier names. In: Proceedings 2000 International Conference on Software Maintenance. pp. 97–107. http://dx.doi.org/10.1109/ICSM.2000.883022, URL: http://ieeexplore.ieee.org/document/883022.

Carnegie Mellon University Software Engineering Institute, 2016. `INT02-C`: Understand integer conversion rules. URL: http://wiki.sei.cmu.edu/confluence/display/c/INT02-C.+Understand+integer+conversion+rules. accessed 2020-07-13.

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

Dash, S.K., Allamanis, M., Barr, E.T., 2018. 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. In: ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, pp. 107–117. http://dx.doi.org/10.1145/3236024.3236042.

EFPL, 2021. Contextual abstractions – Overview. URL: http://dotty.epfl.ch/docs/reference/contextual/motivation.html. version `3.0.1-rc1`, accessed 2021-05-17.

EFPL, Lightbend Inc., 2006. Tour of Scala: Implicit conversions. URL: http://docs.scala-lang.org/tour/implicit-conversions.html. accessed 2020-01-07.

Ericsson, A.B., 2014. *CodeChecker* – an analyzer tooling, defect database and viewer. URL: http://github.com/Ericsson/CodeChecker. accessed 2020-09-18.

Ericsson, A.B., 2016. CodeCompass. URL: http://github.com/Ericsson/CodeCompass. CodeCompass does not have tagged releases, we used git commit `1796dcb1a2538c541d372e5e673d11438cd9b7b2`, titled *"Change creation/destruction order of* `pHandler(PARSER_PLUGIN_DIR)`*"*, dated 2020-11-27, accessed 2021-01-10.

Goetz, B., 2014. JEP 218: Generics over primitive types. URL: http://openjdk.java.net/jeps/169. last updated 2017-10-17, accessed 2021-05-12.

Google, Inc., 2008. Protocol buffers. URL: http://developers.google.com/protocol-buffers/. version `3.11.2` (`fe1790c`), accessed 2019-12-30.

Google, Inc., 2009. CppLint rule: "runtime/memset". URL: http://github.com/google/styleguide/blob/gh-pages/cpplint/cpplint.py. accessed 2019-12-16.

Google, Inc., 2016. `Guetzli`. URL: http://github.com/google/guetzli. version `1.0.1` (`a0f47a2`), accessed 2019-12-30.

Grech, N., Smaragdakis, Y., 2017. P/taint: Unified points-to and taint analysis. Proc. ACM Program. Lang. 1 (OOPSLA), http://dx.doi.org/10.1145/3133926, URL: http://dl.acm.org/doi/abs/10.1145/3133926.

Greene, D., 2019. Provide a semantic `typedef` class and operators. URL: http://reviews.llvm.org/D66148. accessed 2019-11-15.

Guo, P.J., Perkins, J.H., McCamant, S., Ernst, M.D., 2006. Dynamic inference of abstract types. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. In: ISSTA '06, ACM, New York, NY, USA, pp. 255–265. http://dx.doi.org/10.1145/1146238.1146268, URL: http://doi.acm.org/10.1145/1146238.1146268.

Hangal, S., Lam, M.S., 2009. Automatic dimension inference and checking for object-oriented programs. In: 2009 IEEE 31st International Conference on Software Engineering. pp. 155–165. http://dx.doi.org/10.1109/ICSE.2009.5070517, URL: http://ieeexplore.ieee.org/document/5070517.

Horváth, G., Kovács, R.N., Szécsi, P., 2020. Report on the differential testing of static analyzers. Acta Cybern. http://dx.doi.org/10.14232/actacyb.282831, URL: http://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/4099.

Horváth, G., Szécsi, P., Gera, Z., Krupp, D., Pataki, N., 2018. [Engineering paper] challenges of implementing cross translation unit analysis in clang static analyzer. In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 171–176. http://dx.doi.org/10.1109/SCAM.2018.00027, URL: http://ieeexplore.ieee.org/document/8530731/.

ISO/IEC JTC 1/SC 22, 1978. FORTRAN 77, ANSI X3J3-1978. International Organization for Standardization, URL: http://wg5-fortran.org/ARCHIVE/Fortran77.html.

ISO/IEC JTC 1/SC 22, 1994. 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. In: Draft International Standard, Intermetrics, Inc., Cambridge, MA, USA, URL: http://iso.ch/cate/d22983.html.

ISO/IEC JTC 1/SC 22, 2017. ISO/IEC 14882:2017 Information Technology — Programming Languages — C++, Version 17 (C++17). International Organization for Standardization, Geneva, Switzerland, p. 1605, URL: http://iso.org/standard/68564.html.

JetBrains Inc., 2015. CLion - A cross-platform IDE for C and C++. URL: http://jetbrains.com/clion. accessed 2020-01-06.

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

Kovács, R.N., 2017. Clang-Tidy rule: "`bugprone-suspicious-memset-usage`". URL: http://clang.llvm.org/extra/clang-tidy/checks/bugprone-suspicious-memset-usage.html. accessed 2019-12-16.

Kovács, A., Szabados, K., 2016. Internal quality evolution of a large test system – an industrial study. Acta Univ. Sapientiae Inf. 8 (2), 216–240. http://dx.doi.org/10.1515/ausi-2016-0010.

Lawrie, D., Morrell, C., Feild, H., Binkley, D., 2006. What's in a name? A study of identifiers. In: 14th IEEE International Conference on Program Comprehension (ICPC'06). pp. 3–12. http://dx.doi.org/10.1109/ICPC.2006.51, URL: http://ieeexplore.ieee.org/document/1631100.

Liu, H., Liu, Q., Staicu, C.-A., Pradel, M., Luo, Y., 2016. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). pp. 1063–1073. http://dx.doi.org/10.1145/2884781.2884841, URL: http://ieeexplore.ieee.org/document/7886980.

LLVM Foundation, 2001. Clang: a C language family frontend for the LLVM Compiler Infrastructure. URL: http://clang.llvm.org. version `9.0` (`0399d5a`), accessed 2019-12-30.

Madsen, O.L., Magnusson, B., Mølier-Pedersen, B., 1990. 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. In: OOPSLA/ECOOP '90, Association for Computing Machinery, New York, NY, USA, pp. 140–150. http://dx.doi.org/10.1145/97945.97964.

Marriott, N., et al., 2007. `Tmux`. URL: http://github.com/tmux/tmux. version `3.0` (`bbcb199`), accessed 2019-12-30.

Márton, G., Krupp, D., 2020. Codechecker. In: 9th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis (SOAP). URL: http://pldi20.sigplan.org/details/SOAP-2020-papers/13/Tool-Talk-CodeChecker.

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

Microsoft Corporation, 2015. LSP: Language Server Protocol. URL: http://microsoft.github.io/language-server-protocol. accessed 2020-01-06.

Microsoft Research, 2012. The Z3 theorem prover. URL: http://github.com/Z3Prover/z3. version `4.8.9` (`79734f2`), accessed 2021-01-10.

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

Nagy, G.A., Porkoláb, Z., 2017. Performance issues with implicit resolution in scala. In: Proceeedings of the 10th International Conference on Applied Informatics. In: ICAI '17, Eszterházy Károly University, Eger, Hungary, pp. 211–223. http://dx.doi.org/10.14794/ICAI.10.2017.211, URL: http://icai.uni-eszterhazy.hu/icai2017/uploads/papers/2017/final/ICAI.10.2017.211.pdf.

Nakamoto, S., The Bitcoin Core Developers, et al., 2009. Bitcoin. URL: http://bitcoincore.org. version `0.19.0.1` (`1bc9988`), accessed 2019-12-30.

Netdata Corporation, 2013. Netdata. URL: http://my-netdata.io. version `1.19.0` (`5000257`), accessed 2019-12-30.

Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S., Paradkar, A., 2012. Inferring method specifications from natural language API descriptions. In: Proceedings of the 34th International Conference on Software Engineering. In: ICSE '12, IEEE Press, pp. 815–825. http://dx.doi.org/10.5555/2337223.2337319, URL: http://dl.acm.org/doi/10.5555/2337223.2337319.

Pearce, D.J., Kelly, P.H., Hankin, C., 2007. Efficient field-sensitive pointer analysis of C. ACM Trans. Program. Lang. Syst. 30 (1), 4:1–4:42. http://dx.doi.org/10.1145/1290520.1290524.

Peruma, A., 2019. Towards a model to appraise and suggest identifier names. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 639–643. http://dx.doi.org/10.1109/ICSME.2019.00103, URL: http://ieeexplore.ieee.org/document/8918988.

Peters, R., Zaidman, A., 2012. Evaluating the lifespan of code smells using software repository mining. In: 2012 16th European Conference on Software Maintenance and Reengineering. pp. 411–416. http://dx.doi.org/10.1109/CSMR.2012.79, URL: http://ieeexplore.ieee.org/document/6178888.

PHP Group, 1999. PHP: Hypertext preprocessor. URL: http://php.net. `php-src` version `7.4.1` (`b1a8ab0`), accessed 2019-12-30.

Porkoláb, Z., Brunner, T., Krupp, D., Csordás, M., 2018. Codecompass: An open software comprehension framework for industrial usage. In: Proceedings of the 26th Conference on Program Comprehension. In: ICPC '18, Association for Computing Machinery, New York, NY, USA, pp. 361–369. http://dx.doi.org/10.1145/3196321.3197546.

PostgreSQL Developers, 1996. PostgreSQL. URL: http://postgresql.org. version `12.1` (`578a551`), accessed 2019-12-30.

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

Pradel, M., Gross, T.R., 2013. Name-based analysis of equally typed method arguments. IEEE Trans. Softw. Eng. 39 (8), 1127–1143. http://dx.doi.org/10.1109/TSE.2013.7, URL: http://ieeexplore.ieee.org/document/6419711.

Pusz, M., 2019. Implementing physical units library for C++. URL: http://youtube.com/watch?v=wKchCktZPHU. accessed 2019-12-27.

Rice, A., Aftandilian, E., Jaspan, C., Johnston, E., Pradel, M., Arroyo-Paredes, Y., 2017. Detecting argument selection defects. Proc. ACM Program. Lang. 1 (OOPSLA), 104:1–104:22. http://dx.doi.org/10.1145/3133928, URL: http://doi.acm.org/10.1145/3133928.

Robertson, W., Vigna, G., 2009. Static enforcement of web application integrity through strong typing. In: Proceedings of the 18th Conference on USENIX Security Symposium. In: SSYM'09, USENIX Association, USA, pp. 283–298. http://dx.doi.org/10.5555/1855768.1855786, URL: http://dl.acm.org/doi/10.5555/1855768.1855786.

Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T., 2013. Automated API property inference techniques. IEEE Trans. Softw. Eng. 39 (5), 613–637. http://dx.doi.org/10.1109/TSE.2012.63, URL: http://ieeexplore.ieee.org/document/6311409.

Roehm, T., Tiarks, R., Koschke, R., Maalej, W., 2012. How do professional developers comprehend software? In: Proceedings of the 34th International Conference on Software Engineering. In: ICSE '12, IEEE Press, pp. 255–265. http://dx.doi.org/10.5555/2337223.2337254, URL: http://dl.acm.org/doi/10.5555/2337223.2337254.

Rose, J., 2012. JEP 169: Value objects. URL: http://openjdk.java.net/jeps/169. last updated 2019-10-12, accessed 2021-05-12.

Rust Programming Language, 2004a. Rust by example `§5.1` "casting". URL: http://doc.rust-lang.org/rust-by-example/types/cast.html. accessed 2020-01-07.

Rust Programming Language, 2004b. Rust by example `§6.1` "`from` and `into`". URL: http://doc.rust-lang.org/rust-by-example/conversion/from_into.html. accessed 2020-01-07.

Sanfilippo, S., et al., 2006. Redis. URL: http://redis.io. version `5.0.7` (`4891612`), accessed 2019-12-30.

Schäling, B., 2014. The Boost C++ Libraries. XML Press, URL: http://theboostcpplibraries.com. accessed 2020-03-07.

Scott, R., Ranieri, J., Kot, L., Kashyap, V., 2020. Out of sight, out of place: Detecting and assessing swapped arguments. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 227–237. http://dx.doi.org/10.1109/SCAM51674.2020.00031, URL: http://ieeexplore.ieee.org/document/9252035.

Sillito, J., Murphy, G.C., De Volder, K., 2008. Asking and answering questions during a programming change task. IEEE Trans. Softw. Eng. 34 (4), 434–451. http://dx.doi.org/10.1109/TSE.2008.26, URL: http://ieeexplore.ieee.org/document/4497212.

Smith, R., Google, Inc., et al., 2006. Tesseract OCR engine. URL: http://github.com/tesseract-ocr/tesseract. version `4.1.0` (`5280bbc`), accessed 2019-12-30.

Sommerlad, P., 2021. Simplest strong typing instead of language proposal (`P0109`). URL: http://github.com/PeterSommerlad/PSsst. accessed 2021-05-08.

Stenberg, D., et al., 1996. `Curl`. URL: http://curl.haxx.se. version `7.67.0` (`2e9b725`), accessed 2019-12-30.

Storsjö, M., 2019. Fix accidentally swapped input/output parameters of `string(REPLACE)`. URL: http://reviews.llvm.org/rGe16434a0497bdb2da587390171a496b56f1c41b6. accesssed 2019-12-16.

Stroustrup, B., Sutter, H., 2017. `§I.24` "Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning" in the C++ Core Guidelines. URL: http://github.com/isocpp/CppCoreGuidelines/blob/v0.8/CppCoreGuidelines.md#Ri-unrelated. git commit `a97be2d5` after version `0.8`, accessed 2021-02-15.

Szalay, R., Sinkovics, Á., Porkoláb, Z., 2020. The role of implicit conversions in erroneous function argument swapping in C++. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 203–214. http://dx.doi.org/10.1109/SCAM51674.2020.00028, URL: http://ieeexplore.ieee.org/document/9252022.

The Eclipse Foundation, 2001. Eclipse IDE - C/C++ development tooling (CDT). URL: http://projects.eclipse.org/projects/tools.cdt. accessed 2020-01-06.

Torvalds, L., et al., 2005. `Git`. URL: http://git-scm.org. version `2.24.1 (53a06cf)`, accessed 2019-12-30.

Vandevoorde, D., Josuttis, N.M., Gregor, D., 2017. C++ Templates: The Complete Guide (2nd Edition), second ed. Addison-Wesley Professional, URL: http://tmplbook.com.

Varjú, J., 2016. Suspicious call argument checker. source code. URL: http://reviews.llvm.org/D20689. accessed 2019-12-27.

Varjú, J., 2017. 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.

Wang, X., Jhi, Y.-C., Zhu, S., Liu, P., 2008. STILL: Exploit code detection via static taint and initialization analyses. In: 2008 Annual Computer Security Applications Conference (ACSAC). pp. 289–298. http://dx.doi.org/10.1109/ACSAC.2008.37, URL: http://ieeexplore.ieee.org/document/4721566.

Whisperity, 2019. Add `cppcoreguidelines-avoid-adjacent-parameters-of-the-same-type` check. URL: http://reviews.llvm.org/D69560. accessed 2020-01-07.

Whisperity, 2021. `#1732` adjacent parameters of the same type – issues in implementing proper automated analysis. URL: http://github.com/isocpp/CppCoreGuidelines/issues/1732. discussion thread, accessed 2021-02-15.

Wimmer, C., Mössenböck, H., 2007. Automatic feedback-directed object inlining in the java hotspot™ virtual machine. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. In: VEE '07, Association for Computing Machinery, New York, NY, USA, pp. 12–21. http://dx.doi.org/10.1145/1254810.1254813.

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

Xu, Z., Kremenek, T., Zhang, J., 2010. A memory model for static analysis of c programs. In: Margaria, T., Steffen, B. (Eds.), Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I. In: ISoLA'10, Springer-Verlag, Berlin, Heidelberg, pp. 535—548. http://dx.doi.org/10.5555/1939281.1939332, URL: http://link.springer.com/chapter/10.1007/978-3-642-16558-0_44.

Zhang, C., Yang, J., Zhang, Y., Fan, J., Zhang, X., Zhao, J., Ou, P., 2012. Automatic parameter recommendation for practical API usage. In: Proceedings of the 34th International Conference on Software Engineering. In: ICSE '12, IEEE Press, Piscataway, NJ, USA, pp. 826–836. http://dx.doi.org/10.5555/2337223.2337321, URL: http://dl.acm.org/doi/10.5555/2337223.2337321.

Zhong, H., Zhang, L., Xie, T., Mei, H., 2009. Inferring resource specifications from natural language API documentation. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. In: ASE '09, IEEE Computer Society, USA, pp. 307–318. http://dx.doi.org/10.1109/ASE.2009.94.

Richárd Szalay is second-year Ph.D. student at the Department of Programming Languages and Compilers of Eötvös Loránd University (ELTE). His research topic is the use of language elements and tooling-based assistance in software development processes. His work aims to enhance code comprehension, maintainability, and safety.

Ábel Sinkovics received his doctoral degree in Computer Science from the Eötvös Loránd University (ELTE), Budapest, in 2014. He is the author and maintainer of Metashell[9] and Boost.Metaparse.[10] He is working as a software engineer at Morgan Stanley.

Zoltán Porkoláb received his doctoral degree in Computer Science from the Eötvös Loránd University (ELTE), Budapest, in 2004. He is an Associate Professor of the Department of Programming Languages and Compilers at the Faculty of Informatics of ELTE. At the same time, he holds a Principal C++ Developer position at Ericsson Hungary Ltd.

[9] Metashell (http://metashell.org) is an interactive template metaprogramming REPL (read-evaluate-print-loop) environment.

[10] Boost.Metaparse (https://boost.org/doc/libs/1_76_0/doc/html/metaparse.html) is a parser generator library for C++ template metaprograms.